

Politecnico di Torino

Politecnico di Torino

Master Degree course in Cybersecurity

A.a. 2025/2026

Graduation Session September 2026

Experimental Evaluation of REDIAL:

Latency, Hardware Portability, and DoS Resilience in Linux Firewalls

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

Candidate:
Giovanni Drago

Summary

Packet-filtering firewalls evaluate their rules one by one, so every packet pays a cost that grows with the position of the rule it matches. With rulesets of hundreds of entries this cost is no longer negligible: it appears as forwarding latency and, at high traffic rates, as a hard limit on the sustainable throughput. REDIAL [1] attacks the problem by redistributing the filtering rules along the chain of firewalls crossed by the traffic, so that each device evaluates fewer rules per packet. The original paper proves formal bounds $[\Gamma_m, \Gamma_M]$ on the achievable gain and reports a limited test-bed demonstration. This thesis extends that evidence through a systematic and reproducible experimental campaign.

The validation targets the upstream firewall FW1, emulated with Linux netfilter/iptables inside a network namespace. Measurements rely on a low-cost bench derived from the methodology of [2]: a traffic generator that emits packets at a constant, burst-free rate; a network TAP on the firewall path; hardware receive timestamping on the capture NICs; and a calibration procedure for the residual offset between capture ports. Two quantities are measured in every run. The first is σ, the average number of rules evaluated per packet, read exactly from the iptables per-rule counters. The second is D, the mean forwarding latency, extracted from the packet captures. A linear model links them, $D = D_0 + c \cdot \sigma$, obtained by extending the firewall model of [3] to this scenario: D_0 is the fixed traversal cost, c is the price of each rule evaluated.

The core of the experimental work is a matrix of 27 configurations on a 601-rule ruleset at 100 *kpps*: three traffic mixes, three rule splits between the two protected subnets, and three positions of the matching rules within the ruleset (head, center, tail). Each configuration was measured before and after the redistribution, for a total of 54 runs. A single fit over all runs yields $D_0 = 34.30\,\mu s$ and $c = 73.9\,ns$ per rule. Each measured gain is compared with the theoretical reference selected for its variant: rules at the head attain Γ_M, interleaved rules track Γ_{av}, and rules at the tail remain just above Γ_m. No single bound is applied to all 27 cells. The measured values also confirm the blend property of the model: σ and D under mixed traffic are linear combinations of the per-subnet values.

Two further campaigns probe the limits of these results. The same matrix,

repeated on a Raspberry Pi 5 with two USB Ethernet adapters, returns the same σ and Γ values and preserves the linearity of the model, with $c \approx 102$–$110\,ns$ per rule; the fixed cost D_0 grows by roughly $10\times$, dominated by the USB data path. The structural gain Γ therefore does not depend on the hardware platform; the measured net latency gain remains closely aligned, while its price in nanoseconds changes. A final stress test translates the gain into forwarding capacity. It keeps the matrix's most favorable logical configuration, a 25–75 rule split with net2 at the head and $\rho_2 = 0.75$, while scaling the PRE policy to 7501 rules. At 800 *kpps* for 60 *s*, PRE delivers 17 616 215 of 48 000 000 packets, for 293.604 *kpps* of mean throughput and 63.300% loss. After REDIAL reduces the policy kept on FW1 to 1914 rules, POST delivers all 48 000 000 packets at 800.000 *kpps*, with zero observed loss.

The linear model holds, the measured gains agree with their variant-specific references Γ_M for head, Γ_{av} for interleaved, and Γ_m for tail and the redistribution restores full packet delivery at the tested stress point under a load that saturates the PRE firewall. Within the limits of the emulated single-firewall scenario, the promises of REDIAL are confirmed by measurement.

Acknowledgements

Arrivare alla fine di questo percorso significa fermarsi per un momento e pensare non soltanto al lavoro svolto, ma soprattutto alle persone che mi hanno accompagnato fino a qui.

Il mio primo e più sentito ringraziamento va al professor Luca Durante. Mi ha seguito durante tutta la tesi, dimostrandosi sempre presente e disponibile. Sapere di poter contare sulla sua guida ha fatto davvero la differenza. Ringrazio anche Fulvio Valenza, Riccardo Sisto e Daniele Bringhenti per il loro contributo alla realizzazione di questo lavoro.

A mamma, papà e mia sorella devo uno dei doni più importanti: la libertà di scegliere la mia strada. Grazie per avermi permesso di costruire il mio percorso secondo le mie aspirazioni e per aver rispettato le mie decisioni, anche quando mi hanno portato lontano.

Un pensiero va ai miei nonni: a quelli che hanno potuto accompagnarmi per una parte della mia vita, a quelli che oggi non sono più qui e a quelli che non ho avuto la fortuna di conoscere. Anche senza poter condividere questo momento con tutti loro, ciascuno fa parte delle mie radici e della storia che mi ha portato fin qui.

Un ringraziamento particolare va a Marika, che mi accompagna fin dai tempi delle superiori e ha condiviso con me l'intero cammino che mi ha portato a questo traguardo. Durante tutti questi anni, la sua presenza costante e la certezza di poter sempre contare su di lei hanno avuto per me un valore inestimabile. Ha visto questo percorso iniziare, crescere e arrivare infine a compimento. Spero che questo sia soltanto uno dei tanti traguardi che avremo modo di condividere.

Grazie a tutti gli amici che hanno fatto parte di questi anni: a chi ha affrontato insieme a me la preparazione di un esame dopo l'altro, condividendone ansie e soddisfazioni; a chi, nonostante il tempo e la distanza, continua ad aspettarmi ogni volta che torno in Sicilia; e a chi ha avuto la pazienza di sopportarmi anche tra le mura di casa, condividendo con me l'inizio della mia vita a Torino. Grazie per le risate, per i momenti di leggerezza e per tutti i ricordi che porterò con me.

Questo traguardo porta il mio nome, ma conserva dentro di sé una parte di tutte le persone che mi hanno aiutato a raggiungerlo.

A ciascuna di loro va il mio grazie più sincero.

Table of Contents

List of Tables x

List of Figures xii

1 Introduction 1
- 1.1 Context and motivation . . . 1
- 1.2 Problem statement and goal . . . 3
- 1.3 Contributions . . . 4
- 1.4 Thesis outline . . . 5

2 Background and reference works 6
- 2.1 Packet-filtering firewalls in Linux . . . 6
- 2.2 REDIAL: rule redistribution . . . 7
- 2.3 Low-cost software delay measurement . . . 9
- 2.4 Firewall performance modeling . . . 11
- 2.5 Related work . . . 12

3 Analytical model and observables 15
- 3.1 The observable sigma . . . 15
- 3.2 The observable D . . . 16
- 3.3 The linear model . . . 16
- 3.4 Predicted gain and theoretical references . . . 17

4 Experimental testbed 18
- 4.1 Measurement scenario . . . 18
- 4.2 Testbed hardware . . . 20
- 4.3 Network and measurement instrumentation . . . 21
- 4.4 System software and configuration . . . 23
- 4.5 Emulated topology . . . 25
- 4.6 Reused vs redesigned . . . 27

5 Measurement and experiment toolchain 28
5.1 Workload generation . . . 29
5.2 Experiment execution . . . 29
5.3 Measurement and analysis . . . 31
5.4 Design principles . . . 32

6 Testbed characterization and validation 33
6.1 Offset calibration . . . 33
6.2 Method validation . . . 35
6.3 Baseline firewall cost . . . 37
6.4 The knee . . . 39
6.5 Rate effects . . . 42
6.6 Operating rate . . . 44

7 Experimental campaigns and results 46
7.1 Experimental design . . . 46
7.2 Run protocol . . . 48
7.3 Gain results . . . 48
7.4 Latency results . . . 50
7.5 Theory and blend . . . 52
7.6 Summary . . . 54

8 Generalizability: porting to embedded hardware 56
8.1 Porting the testbed . . . 57
8.2 Characterization . . . 58
8.3 Results . . . 60
8.4 Platform comparison . . . 62

9 DoS experiment: REDIAL under stress 65
9.1 Threat scenario . . . 65
9.2 Experiment design . . . 66
9.2.1 Testbed and measurement points . . . 66
9.2.2 Ruleset and traffic distribution . . . 66
9.2.3 Validity conditions and accounting . . . 67
9.3 Results . . . 67
9.3.1 Aggregate packet accounting . . . 67
9.3.2 Throughput over time . . . 68
9.3.3 Packet loss over time . . . 69
9.4 Interpretation . . . 69
9.5 Scope and limits . . . 70

10 Discussion 71
10.1 The value of REDIAL . . . 71
10.2 Comparison with the literature . . . 72
10.3 Threats to validity . . . 74

11 Conclusions and future work 76
11.1 Summary of results . . . 76
11.2 Future work . . . 77

A Toolchain: usage and source code 79
A.1 gen_fw_rules v6 . . . 79
A.2 gen_packets_v2 . . . 90
A.3 sendpkts (the sender) . . . 95
A.4 run/listen-redial.sh . . . 105
A.5 campagna-redial.sh . . . 110
A.6 The REDIAL orchestrator . . . 114
A.7 sigma collector . . . 131
A.8 compare-redial . . . 141
A.9 compare-redial-win . . . 153

B Topology creation and recovery 159
B.1 TS-P5: creation-namespace.sh . . . 159
B.2 TS-P360: fw1-up.sh . . . 162
B.3 Recovery after a power cycle . . . 165

C Complete result tables 166
C.1 x86 tables . . . 166
C.2 Pi 5 tables . . . 167

Bibliography 170

List of Tables

4.1 Hosts of the testbed. Management addresses belong to the 192.168.100.0/24 network of Figure 4.2; the addresses of the emulated topology exist only inside network namespaces (Section 4.5). 20
4.2 Measurement-relevant system configuration. All settings are reapplied automatically at the start of every run. 25
4.3 Namespaces, interfaces, and addressing of the emulated topology. The capture interfaces carry no test traffic of their own and run in promiscuous mode. 26

6.1 Switch delay as a function of the frame size. Each row aggregates 3.22 million packets, paired with zero loss; the session offset of Section 6.1 is already removed. 36

7.1 Factors and levels of the experimental matrix. Every cell is measured in the PRE and POST states of the same session: $27 \times 2 = 54$ runs, all at 100 *kpps*. 47
7.2 Measured gain $\Gamma = \sigma_{\mathrm{PRE}}/\sigma_{\mathrm{POST}}$ over the 27 cells of the matrix. Each value comes from one PRE/POST pair of six-million-packet runs at 100 *kpps*; σ is read from the iptables counters. 49
7.3 Latency saving $\Delta = D_{\mathrm{PRE}} - D_{\mathrm{POST}}$ and net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$ over the 27 cells of the matrix, with the $D_0 = 34.30\,\mu$s of the global fit (Figure 7.1). Same runs as Table 7.2; the session offset is common to the two states and cancels in the difference. Entries read $\Delta\,[\mu\mathrm{s}]$ ($G_{net}\,[\%]$). 51
7.4 Measured Γ against the reference selected for each variant, for all 27 cells. Each entry reports the reference and, in parentheses, the excess of the measured value of Table 7.2 over it. In the head cells the excess is zero at the three-decimal precision of the tables. 53

8.1 PRE/POST latency difference Δ and net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$ on the Raspberry Pi at 2 *kpps*, with the $D_0 = 367.7\,\mu$s of the sweep fit, for the 27 cells of the matrix. Entries read $\Delta\,[\mu\mathrm{s}]$ ($G_{net}\,[\%]$). The complete per-run tables are in Section C.2 of Appendix C. 61
8.2 The two platforms side by side. The matrix-fit parameters use all 54 runs at the respective operating rates. The pass-all sweep estimates and the D_0 used for G_{net} are listed separately. 63

9.1 DoS stress campaign at 800 *kpps*. PRE and POST are separate 60 *s* acquisitions with 48 million offered packets each. 67

C.1 REDIAL matrix on the x86 firewall at 100 kpps — ruleset with the net2 rules at the *head* ($h = 1$). σ in rules evaluated; times in µs, mean over 6 000 000 packets per run, zero loss. $G_{\mathrm{net}} = \Delta/(D_{\mathrm{pre}} - D_0)$ with $D_0 = 34.30$ µs (global fit): the measured values match the prediction $1 - 1/\Gamma$ within about one point. 166
C.2 REDIAL matrix on the x86 firewall at 100 kpps — *mixed* ruleset (proportional interleaving, $\Gamma \approx \Gamma_{av}$). Same conditions as Table C.1. In the mixed PRE state σ is flat in ρ: Δ and G grow with ρ, the opposite of the head variant. 167
C.3 REDIAL matrix on the x86 firewall at 100 kpps — ruleset with the net2 rules at the *tail* ($\Gamma \approx \Gamma_m$, worst case). Same conditions as Table C.1. At $\rho = 0.25$ the Γ_M formula returns a negative value: a known artifact of the bound at large h. 167
C.4 REDIAL matrix on the Raspberry Pi 5 at 2 kpps — ruleset with the net2 rules at the *head* ($h = 1$). σ in rules evaluated; times in µs, mean over 120 000 packets per run, zero loss. σ and Γ coincide with the x86 matrix (deterministic counters); the times reflect the USB fixed cost $D_0 \approx 370$ µs and $c_{\mathrm{Pi}} \approx 102$–$110$ ns per rule. 168
C.5 REDIAL matrix on the Raspberry Pi 5 at 2 kpps — *mixed* ruleset (proportional interleaving, $\Gamma \approx \Gamma_{av}$). Same conditions as Table C.4. In the mixed PRE state σ is flat in ρ: Δ and G grow with ρ, the opposite of the head variant. 168
C.6 REDIAL matrix on the Raspberry Pi 5 at 2 kpps — ruleset with the net2 rules at the *tail* ($\Gamma \approx \Gamma_m$, worst case). Same conditions as Table C.4. At $\rho = 0.25$ the Γ_M formula returns a negative value: the same artifact already observed on the x86 bench. 169

List of Figures

1.1 Reference topology for the REDIAL model. “Upstream” and “downstream” describe the order in which a chain of firewalls is crossed by one traffic flow, not a fixed physical position: the same firewall can be upstream for traffic to one destination and downstream for traffic arriving from another direction. FW1 denotes the firewall that is upstream for the traffic flow considered in this thesis. 3

2.1 Measurement architecture of the reference methodology [2]. The test manager hosts both the sender and the receiver and uses three ports of the same network card: the sender port (SX), a first capture port (RX_1) that timestamps the TAP copy of every outgoing packet, and a second capture port (RX_2) that timestamps the same packet after it has crossed the device under test. The delay introduced by the device is the difference between the two hardware receive timestamps, once the constant offset between the two capture ports has been calibrated out (Chapter 6). 10

4.1 Logical topology of the measurement path. The TAP delivers a copy of each packet to the first capture port (t_1); the same packet, after crossing FW1, reaches the second capture port (t_2). Both ports belong to the same Intel I350 on TS-P5, which timestamps them in hardware; the firewall delay is the difference between the two timestamps. 19

4.2 Management network of the testbed. All control traffic, SSH orchestration, REDIAL apply and rollback, result collection, flows through this segment and never touches the measured path. ieiitp97 is the only host exposed outside the laboratory. 20

4.3 Physical cabling of the measurement path. Port and interface names as configured on the bench. 23

5.1 Pipeline of the toolchain, from ruleset and traffic generation through campaign execution to the extraction of σ and D. 28

6.1 Topology of the calibration runs. The firewall is out of the path: the forward output of the TAP is cabled directly to the second capture port, so both capture ports receive the same frame at the same instant and any difference between the two timestamps is the clock offset itself. 34
6.2 Linear fit of the switch delay against the Ethernet frame size. The slope, $8.006 \pm 0.005\,ns$ per byte, matches the serialization time of one byte at 1 *Gbps*; the residuals (bottom panel) stay within $\pm 15\,ns$, about twice the resolution of the PHC. 36
6.3 Delay of FW1 against the depth h of the matching rule at 100 *kpps*. Each point is the mean of two independent runs of six million packets each; $\sigma = h$ is verified from the iptables counters at every run. The fit over the 20 positions below saturation ($h = 1001$ excluded) returns $D_0 = 60.7\,\mu$s and $c = 69.5\,ns$ per rule. 39
6.4 Delivered rate against offered rate for FW1 loaded with the 500-rule ruleset, fine ramp from 100 to 850 *kpps*. The curve follows the diagonal up to 400 *kpps*, then flattens at the forwarding capacity of about 440 *kpps*; every packet offered above the plateau is lost. . . . 40
6.5 Mean delay against the depth h of the matching rule at the four sweep rates, logarithmic vertical axis. Cells below the knee sit on the linear branch of Section 6.3; saturated cells collapse onto a rate-independent branch about two orders of magnitude higher, whose slope is the drain time of the full queue. 41
6.6 Mean delay of the empty firewall (no rules, ACCEPT policy) against the offered rate, logarithmic rate axis. Nine-step ramp, 30 million packets paired with zero loss; each point is the per-step mean with the session offset applied. The plateau below 1 *kpps* is the cost of waking the CPU for each packet; the decline above it is the NAPI driver moving from one interrupt per packet to polling. 43

7.1 Mean delay against mean rules evaluated per packet for the 54 runs of the matrix, 27 PRE and 27 POST. A single fit over all the points returns $D_0 = 34.30\,\mu$s and $c = 73.9\,ns$ per rule, with $R^2 = 0.998$: both states of the firewall and all nine rulesets follow the same line. 52

8.1 Fitted fixed cost D_0 of the Raspberry Pi 5 firewall as a function of the offered rate (pass-all sweep over 21 positions of h, two runs per cell). The curve is U-shaped: at low rates every packet pays the full USB polling cycle, batching amortizes it down to the minimum of 367.7 μs at 2 *kpps*, the operating rate of the matrix (dashed line), and queueing inflates it again beyond 10 *kpps*. 59
8.2 Mean delay against σ for the 54 runs of the matrix on the Raspberry Pi at 2 *kpps*. The linear model $D = D_0 + c \cdot \sigma$ holds on this platform as it does on the x86; the slope implied by the runs sits between 102 and 110 *ns* per rule. 62

9.1 Offered and delivered packet rates during the PRE and POST phases. The offered load remains fixed at 800 *kpps*; mean delivered throughput increases from 293.604 *kpps* in PRE to 800.000 *kpps* in POST. . 68
9.2 Packet loss during the PRE and POST phases at an offered rate of 800 *kpps*. Mean loss decreases from 63.300% in PRE to zero observed loss in POST. 69

Chapter 1

Introduction

A packet-filtering firewall decides the fate of every packet by scanning an ordered list of rules and applying the first one that matches. The scan is sequential, so the time spent on a packet depends on where its rule sits in the list: a packet accepted by rule 500 costs the firewall a hundred times more lookups than one accepted by rule 5. On a network path this cost is paid twice. It adds latency to every forwarded packet, and it caps the packet rate the device can sustain before it starts dropping traffic. Rulesets grow over time and rarely shrink, so the problem gets worse as a network ages.

When traffic crosses several firewalls in sequence, part of this work is repeated at every hop: a rule enforced by the last device may be evaluated, without effect, by all the devices before it. REDIAL [1] removes this redundancy by redistributing the filtering rules along the chain, so that each firewall keeps only the rules it must enforce and the average number of rules evaluated per packet decreases. The authors define the model formally, prove bounds $[\Gamma_m, \Gamma_M]$ on the achievable gain, and report a limited two-firewall test-bed demonstration. What remains open is a systematic, reproducible validation across rule distributions, traffic profiles, hardware platforms, and the transition from latency to packet loss under saturation.

This thesis answers that question by measurement. A real Linux firewall is built, instrumented, and measured before and after the redistribution, across a full experimental campaign and on two different hardware platforms.

1.1 Context and motivation

Industrial and critical networks are built around segmentation. Standards such as IEC 62443 prescribe the division of the network into zones, with filtering devices guarding every conduit between them [4]. In this defense-in-depth arrangement a firewall is not an exceptional presence at the network border: it is a recurring

element of the internal topology, and the traffic between two end devices may cross more than one of them. Every crossing is paid in evaluation time.

For much of the traffic that populates these networks, that price matters. Industrial control traffic is cyclic and runs against deadlines: a control loop with a cycle time of a few milliseconds leaves a budget of microseconds to each network element on its path. A firewall that adds tens of microseconds per hop consumes a visible share of that budget, and the share grows with the size of the ruleset. Throughput is affected too. Since rule evaluation is sequential, the product of packet rate and rules evaluated per packet cannot exceed the evaluation capacity of the device; past that point the firewall drops packets. An attacker who knows this does not need to break any rule to cause damage. Flooding the device with traffic that matches late rules, or no rule at all, is enough.

Rulesets, meanwhile, only grow. Segmented networks favor whitelisting, and a whitelist needs roughly one rule per allowed flow, so every new device or service adds entries. Old rules are rarely removed, because nobody wants to be the one who broke the plant by deleting a rule that seemed unused. Rulesets of several hundred entries are therefore a realistic operating condition, not a pathological case.

The cost of a ruleset, however, depends on how the rules are placed. In a chain of firewalls the same rule is often evaluated at several hops while being enforced at one, and the upstream device, which sees all the traffic, pays the highest price. This observation is the starting point of REDIAL: if each firewall in the chain keeps only the rules it actually has to enforce, the redundant evaluations disappear and the average work per packet drops, with no change to the filtering policy as a whole. How the redistribution is computed, and what gain it can promise, is the subject of Chapter 2. How much of that promise survives contact with a real firewall is the subject of this thesis.

1.2 Problem statement and goal

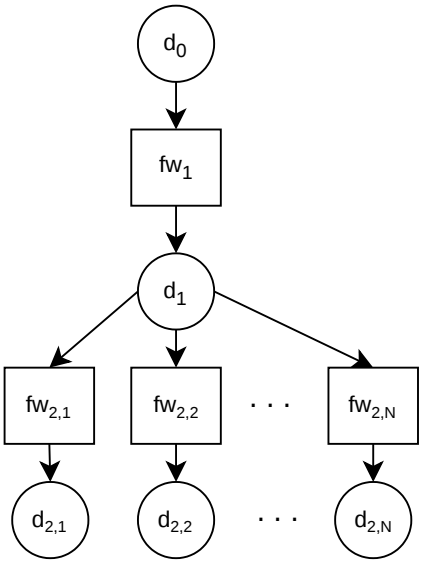


Figure 1.1: Reference topology for the REDIAL model. "Upstream" and "downstream" describe the order in which a chain of firewalls is crossed by one traffic flow, not a fixed physical position: the same firewall can be upstream for traffic to one destination and downstream for traffic arriving from another direction. FW1 denotes the firewall that is upstream for the traffic flow considered in this thesis.

The REDIAL paper answers a theoretical question: given a chain of firewalls and a traffic profile, how should the rules be redistributed, and what reduction of the average evaluation work is achievable. This thesis asks the empirical counterpart of that question. On a real firewall, with a real ruleset and real packets, how large is the gain, and does it match what the theory predicts?

In a REDIAL deployment the largest benefit falls on the upstream firewall, called FW1 in the model. As Figure 1.1 makes explicit, "upstream" and "downstream" are relative to the traffic flow under consideration, not absolute positions fixed by the topology alone: a device that is upstream for one flow can be downstream for another crossing the same network in a different direction. FW1 in this thesis denotes the firewall that is upstream for the traffic flow whose redistribution is measured, the device that sees the whole of that traffic and gives up the most

rules. The validation therefore targets FW1 alone, in what this thesis calls the *FW1-only* scenario. FW1 is a real packet filter, built with Linux netfilter/iptables inside a network namespace; the rest of the chain does not need to be materialized, because the effect of the redistribution on FW1 is entirely captured by the two rulesets the device holds before and after it. Every experiment in this work is a comparison between these two states of the same firewall, labeled PRE and POST, under identical traffic.

Answering the question requires quantities that can be measured, not only defined. Two are used throughout this work. The first is σ, the average number of rules the firewall evaluates per packet; iptables exposes per-rule counters, so σ can be computed exactly, and the theoretical gain Γ is the ratio between its PRE and POST values. The second is D, the mean forwarding latency, which is what the network actually experiences; the two are linked by a linear model, $D = D_0 + c \cdot \sigma$, whose validity is itself one of the claims to verify. The goal of the thesis can then be stated precisely: measure σ and D on FW1 before and after REDIAL across a full grid of operating conditions, verify the measured Γ against the theoretical reference selected for each variant—Γ_M for head, Γ_{av} for interleaved, and Γ_m for tail—and translate the gain into the units that matter in operation, microseconds of latency and packets not dropped. The formal definitions of these quantities are given in Chapter 3.

1.3 Contributions

The work described in this document produced five main contributions.

- A reproducible, low-cost testbed for measuring per-packet firewall latency with nanosecond-scale resolution, built from commodity hardware around the NICs' receive timestamping and a network TAP. The measurement chain was calibrated and then verified on devices of known delay before being trusted (Chapters 4 and 6).
- A toolchain that covers the whole experimental cycle: generation of parametric rulesets and of reproducible, seed-fixed traffic vectors; automated execution of PRE/POST campaign pairs; extraction of σ from the iptables counters and of D from the packet captures (Chapter 5).
- A systematic and reproducible experimental validation of the REDIAL model that extends the limited test-bed evidence in the original paper. A grid of 27 operating conditions, measured before and after the redistribution for a total of 54 runs at 100 *kpps*, confirms the linear latency model with a single global fit and compares each measured gain with the reference selected for its variant: Γ_M for head, Γ_{av} for interleaved, and Γ_m for tail (Chapter 7).

- Evidence that the structural gain Γ does not depend on the platform. The same 27-condition grid, repeated on a Raspberry Pi 5, returns the same σ and Γ values and preserves the linearity of the model; measured G_{net} remains closely aligned, while the fixed cost and the per-rule price change with the hardware (Chapter 8).
- A demonstration that the gain translates into forwarding capacity. In the official stress campaign at 800 *kpps*, reducing the policy kept on FW1 from 7501 to 1914 rules raises mean throughput from 293.604 to 800.000 *kpps* and changes packet loss from 63.300% to zero observed loss (Chapter 9).

1.4 Thesis outline

The rest of this document follows the path from theory to measurement.

Chapters 2 and 3 lay the foundations. Chapter 2 reviews packet filtering in Linux and the three works this thesis builds on: the REDIAL model itself, the low-cost delay measurement methodology adopted by the testbed, and the firewall performance model from which the latency equation descends. Chapter 3 turns the theory into measurable quantities, defining σ, D, the linear model $D = D_0 + c \cdot \sigma$, and the criteria that the experiments must satisfy for the validation to hold.

Chapters 4 to 6 describe the instrument. Chapter 4 presents the testbed: the hosts, the network path with its timestamping hardware, and the emulated topology that hosts FW1. Chapter 5 presents the software developed to run the experiments, from ruleset generation to the extraction of the observables. Chapter 6 establishes how far the instrument can be trusted, by calibrating and verifying the measurement chain on devices of known delay, locating the saturation point of the firewall, and fixing the operating rate of the campaigns.

Chapters 7 to 9 contain the experiments. Chapter 7 presents the core campaign, a grid of 27 operating conditions measured before and after the redistribution, and compares the outcome with the applicable theoretical references. Chapter 8 repeats the grid on a Raspberry Pi 5 to test whether the conclusions depend on the hardware. Chapter 9 pushes the firewall to saturation to show what the gain is worth under a denial-of-service load.

Chapter 10 weighs the results and their limits, and Chapter 11 summarizes the findings and outlines future work. The appendices collect the material needed to repeat the experiments: the toolchain manual, the procedures to recreate the topology, and the complete result tables.

Chapter 2

Background and reference works

Chapter 1 named REDIAL as the model this thesis validates, without yet saying what it is or what it rests on. This chapter fills that gap. Section 2.1 reviews packet filtering in Linux, the mechanics that the firewall of Chapter 4 implements and that the cost model of Chapter 3 describes. Section 2.2 presents REDIAL itself, its formal model, and the three placement-dependent references used in this thesis: Γ_M for head, Γ_{av} for proportional interleaving, and Γ_m for tail. Section 2.3 presents the low-cost delay measurement methodology that the testbed adopts, and Section 2.4 the firewall performance model that the linear latency equation of Chapter 3 extends. Section 2.5 closes the chapter with the alternative approaches REDIAL does not replace.

2.1 Packet-filtering firewalls in Linux

Netfilter is the packet-filtering framework built into the Linux kernel; iptables is the user-space tool that configures it. A ruleset is organized into chains, ordered lists of rules attached to a hook in the packet's path through the kernel; the chain relevant to this thesis is `FORWARD`, traversed by every packet the host routes between two interfaces rather than delivering to a local process or originating itself. FW1 in the REDIAL model (Section 2.2) and FW1 on the testbed (Chapter 4) are both, ultimately, a `FORWARD` chain.

A chain is evaluated sequentially and stops at the first match. Each rule specifies a set of match criteria, source and destination address, protocol, ports, and a target, typically ACCEPT or DROP; iptables walks the chain rule by rule, in the order they were inserted, and applies the target of the first rule whose criteria the packet satisfies. A packet accepted by the tenth rule pays nine failed comparisons before

the one that decides its fate; a packet accepted by the five-hundredth pays four hundred and ninety-nine. If no rule matches, the chain falls back to its default policy; setting that policy to DROP, as this thesis does throughout (Section 4.5), makes the ruleset a whitelist, and a packet that matches nothing pays the full length of the chain before being discarded. This is the mechanism behind σ, the average number of rules evaluated per packet defined in Chapter 3: it is not an abstraction over the firewall's behavior, it is a direct count of comparisons iptables performs, exposed by the per-rule packet counters every chain already maintains.

The consequence that motivates this whole thesis follows directly from the mechanism. Because the chain is scanned front to back, the position of a rule, not only its presence, determines its cost: two rulesets with the same rules in a different order impose a different average number of comparisons on the same traffic, without changing what the firewall accepts or rejects. Rulesets also tend to grow rather than shrink over the life of a network, so the comparisons paid by traffic near the end of the chain accumulate over time. Both effects are properties of the mechanism just described, and both are what REDIAL, presented next, sets out to reduce without touching the filtering policy itself.

2.2 REDIAL: rule redistribution across firewall chains

REDIAL [1] targets exactly the situation described above, extended to a chain of firewalls. The reference topology is a tree rooted in a traffic source d_0: an upstream firewall fw_1 filters everything leaving d_0, a routing device fans the accepted traffic out over N downstream branches, and a downstream firewall $fw_{2,i}$ guards each destination subnetwork $d_{2,i}$. Both fw_1 and every $fw_{2,i}$ are, in the model, an ordered sequence of rules evaluated under the first-matching strategy of Section 2.1, with three possible actions: accept a packet, discard it, or defer the decision to a later rule (the *goto n* action, the abstract counterpart of a jump). The paper singles out $K \leq N$ of the downstream firewalls as candidates to receive rules from fw_1, and calls D_1^K the union of the subnetworks they protect and D_{K+1}^N the union of the subnetworks left untouched.

The redistribution itself, formalized as the REDIAL algorithm, does four things to fw_1 and to each involved $fw_{2,i}$:

- **Copy and removal of redundant rules.** Every rule of fw_1 whose destination field intersects some $d_{2,i}$, $i \leq K$, is copied to $fw_{2,i}$ and removed from consideration in fw_1; rules that also match D_{K+1}^N are kept in fw_1 as well, since that traffic is not being redistributed.
- **Insertion of the aggregate accept rule.** The first time a rule is moved, a

single new rule is inserted into fw_1 at that same position h: accept anything addressed to D_1^K. Every later packet destined for a redistributed subnetwork is decided at position h, and the rules that used to filter it, wherever they sat in fw_1, no longer cost it anything.

- **Unchanged behavior on each $fw_{2,i}$.** Each $fw_{2,i}$ that receives rules must still behave exactly as the pair $(fw_1, fw_{2,i})$ used to: a rule inherited from fw_1 is placed at the head of $fw_{2,i}$, guarded by a check on whether the packet actually came through fw_1 or arrived directly from another subnetwork, and an inherited accept does not terminate the evaluation but jumps back into the original $fw_{2,i}$ rules, exactly as it would have if fw_1 had kept checking it.
- **Guaranteed correctness.** The paper proves formally, and confirms by exhaustive simulation, that this transformation changes nothing about which packets are ultimately accepted or dropped.

What the transformation buys is captured by a single number. Writing σ_1^N for the average number of rules fw_1 evaluates per packet from d_0 before the transformation and $\hat{\sigma}_1^N$ for the same quantity afterward, the paper defines the gain $\Gamma = \sigma_1^N / \hat{\sigma}_1^N$ and expresses both quantities through ρ, the fraction of traffic addressed to D_1^K, and h, the position of the new shortcut rule: $\sigma_1^N = \sigma_1^K \cdot \rho + \sigma_{K+1}^N \cdot (1 - \rho)$ and $\hat{\sigma}_1^N = h \cdot \rho + \hat{\sigma}_{K+1}^N \cdot (1 - \rho)$. The one term that cannot be measured without deploying the transformation, $\hat{\sigma}_{K+1}^N$, is bounded by considering where the removed rules used to sit in fw_1: bunched at the top, evenly spread, or bunched at the bottom, giving respectively the maximum, average, and minimum achievable gain, Γ_M, Γ_{av}, and Γ_m. The lower and upper values bound the theoretical range, while Γ_{av} is an average-position reference. In the experiments, each placement is compared only with its corresponding reference: Γ_M for head, Γ_{av} for proportional interleaving, and Γ_m for tail.

The paper backs the model with a C-language simulator, run over hundreds of randomized rule sets, and with a small hardware test-bed of two cascaded Linux firewalls built with iptables. The simulation reports gains for the upstream firewall alone ranging from about 1.2 to 90 depending on ρ and the rule mix, and gains for the firewall pair as a whole between roughly 1.2 and 2.2; the test-bed run, at $\rho = 75\%$ with 300 rules on fw_1, measures an average filtering latency drop from 1.86 to 0.32 μs, a gain of 5.76, consistent with both the simulation and the σ-based prediction. What this experiment does not do is what Chapters 4 to 7 of this thesis set out to provide: a validation across a full grid of operating conditions, at a fixed, characterized operating rate, with each measured gain checked against the reference selected by its placement Γ_M for head, Γ_{av} for proportional interleaving, and Γ_m for tail and repeated on more than one hardware platform. That gap, not any doubt about the correctness of the model, is the starting point of this thesis.

2.3 Low-cost software-based delay measurement

Measuring the microsecond-scale delay REDIAL is expected to save requires an instrument at least as fine as the effect it is meant to observe, and instruments of that class are usually built around dedicated hardware or licensed test equipment. Cheminod et al. [2] propose a methodology that reaches sub-microsecond resolution with a conventional PC, a commodity multi-port network card, and open-source Linux tools, and it is this methodology, not a specialized measurement appliance, that the testbed of Chapter 4 builds on.

The architecture collapses the sender and the receiver of the test traffic onto a single machine, called the test manager, using two ports of the same network interface card. Consolidating both roles on one host removes the need to synchronize two independent clocks, the usual difficulty in delay measurement, at the cost of a different problem: the two ports still hold two independent hardware timestamp counters, one per port, which start at different values, and the offset between them is unknown and must be handled some other way. The card exploited in the paper, an Intel I350, timestamps every received packet in hardware at 8 ns resolution, tagging it before the frame reaches the kernel's protocol stack, so the timestamp is unaffected by scheduling, interrupt coalescing, or anything else that happens in software afterward; only the receive path is used for this purpose, since the transmit path lacks a timestamp storage mechanism precise enough to associate a value unambiguously with the correct outgoing packet. A network tap placed right after the sender solves the transmit-side half of the problem: it mirrors every outgoing packet back to a third port on the same card, so what would have been a transmit timestamp becomes another hardware receive timestamp, taken with the same guarantees as the one at the far end. The resulting arrangement is sketched in Figure 2.1.

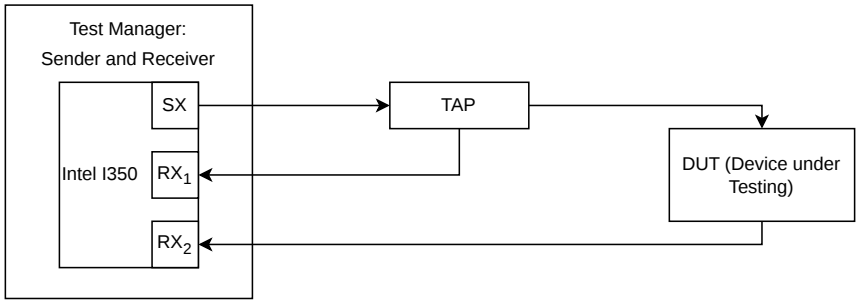


Figure 2.1: Measurement architecture of the reference methodology [2]. The test manager hosts both the sender and the receiver and uses three ports of the same network card: the sender port (SX), a first capture port (RX_1) that timestamps the TAP copy of every outgoing packet, and a second capture port (RX_2) that timestamps the same packet after it has crossed the device under test. The delay introduced by the device is the difference between the two hardware receive timestamps, once the constant offset between the two capture ports has been calibrated out (Chapter 6).

Reconstructing this measurement instrument required care to isolate exactly what the device under test contributes to the recorded delay. Two unknowns remain in any single timestamp difference: the constant offset between the two ports' counters, and any deterministic delay introduced by the measurement setup itself, tap included. The paper's solution is architectural rather than numerical: an intermediate switch is placed between the tap and the two capture points, wired with two configurable VLANs so it can be reprogrammed, without any physical recabling, to route packets either along a short reference path that bypasses the device under test or along a path that crosses it. Every fixed and common component of the delay, tap, switch, and the unknown port offset alike, cancels when the reference measurement is subtracted from the measurement taken with the device in path, leaving only the delay the device under test actually introduces. Chapter 4 adopts the same reasoning without adopting the same architecture: the redistribution studied in this thesis needs the firewall itself to be the element that changes between the reference and the test case, not an element removed from the path, so the offset between the two capture ports is calibrated directly rather than cancelled by construction, as detailed in Chapter 6.

The other half of the methodology concerns what is fed into the device under test. The paper shows that an uneven, bursty arrival process can push some devices, firewalls among them, into a transient state in which they process even fewer packets per second than their steady-state capacity, which would corrupt any measurement that assumes steady conditions. It compares three traffic generators against this

requirement, a widely used general-purpose tool, a high-performance generator that depends on specific network cards and drivers, and a custom-built alternative designed for the paper's own purposes, and finds that the general-purpose one produces markedly bursty inter-packet intervals under demanding conditions, while the custom generator matches the high-performance one in regularity without its stricter hardware requirements, by pacing transmissions with an absolute, kernel-level timer and by skipping rather than bursting after a missed deadline. Applied to a first characterization of its own baseline path and to the evaluation of a commercial switch, the methodology resolves a delay difference of a few microseconds and its variance with confidence intervals tight enough to separate the device from the measurement noise around it, the same standard of resolution the campaigns of Chapter 7 are held to.

2.4 Firewall performance characterization and modeling

Where Section 2.3 supplies the instrument, Cheminod et al. [3] supply the model this thesis extends. The paper studies a commercial industrial firewall able to filter Modbus/TCP traffic, and asks the same question this thesis asks of iptables: how does the delay a firewall introduces depend on the number of rules a packet is checked against, and does that dependence follow a predictable law.

The testbed built to answer it is a smaller version of the one described in Section 2.3: a traffic generator feeds the firewall through a mirroring switch, and a receiver equipped with two identical network cards captures both the mirrored input and the filtered output, so every packet's delay is the difference between two software timestamps taken on the same machine. Before trusting the setup, the authors verify it on a passive tap and again on the switch, both without the firewall in the path, and find the two capture channels agree to within about 1 microsecond, small enough against the tens to hundreds of microseconds the firewall itself introduces; a companion test confirms the traffic generator delivers packets at the nominal rate with negligible jitter. It is a coarser instrument than the hardware-timestamped one of Section 2.3, built around ordinary NICs and software timestamps rather than a hardware clock, but coarse enough only relative to the delays it targets, which sit two to three orders of magnitude above its resolution.

The firewall itself is modeled as a finite-buffer, single-server queue: packets accumulate in an input buffer and are served one at a time, where "service" means walking the rule list under the same first-matching strategy of Section 2.1 until a match is found. Because the rules used in the experiments are all of the same kind, the service time is written as $t_1 = t_1^{dl} + R \cdot t_1^r$, a fixed data-link cost plus a per-rule cost t_1^r multiplied by the position R of the matching rule, exactly the

linear structure that Chapter 3 adopts for D as a function of σ. A queue of this kind has two regimes, set by whether the packet arrival rate is below or above the service rate $1/t_1$: below it, every packet is served as it arrives and the measured delay is the fixed cost plus the position-dependent term with no queueing contribution; above it, the buffer fills, a share of the arriving packets is dropped, and the measured inter-departure time settles to t_1 itself, no longer a function of the offered rate. Sweeping the matching rule from position 1 to 600 and fitting the two regimes separately gives $\hat{\delta}(R) = 170.82 + R \cdot 0.74\,\mu s$ below the threshold and $\hat{\delta}_r(R) = 138.87 + R \cdot 0.75\,\mu s$ above it, in close agreement with each other and with a maximum fitting error under $6\,\mu s$; the buffer size, recovered independently from the number of packets lost every time the queue overflows, comes out at 256 entries. The same reasoning, applied to the application-layer stage that inspects Modbus/TCP payloads once a packet has already cleared the IP rules, isolates a second, smaller queue whose service time turns out not to depend on the position of the matching Modbus rule at all, evidence that the function-code check that stage performs runs in constant time rather than by scanning a list.

Two features of this model carry over directly to this thesis, and one does not. What carries over is the shape: a linear relation between the number of rules evaluated and the delay incurred, and a firewall that behaves as a two-region queue, stable at low load and saturated beyond a threshold set by the service rate. Chapter 3 restates that linear relation as $D = D_0 + c \cdot \sigma$ for an iptables firewall running inside a network namespace, and Chapter 6 locates the same stable-to-saturated boundary on the testbed of this thesis, the "knee" beyond which the measured delay stops reflecting the cost of a single rule. What does not carry over as directly is the measurement precision: the software timestamps used here resolve microseconds, adequate for a device whose per-rule cost is fractions of a microsecond, but iptables rules on commodity hardware cost tens of nanoseconds each (Chapter 7), an order of magnitude finer than this methodology can see, which is exactly the gap the hardware-timestamped instrument of Section 2.3 is built to close.

2.5 Related work

The cost that REDIAL reduces comes from the sequential, first-matching evaluation of Section 2.1, and the literature that attacks the same cost splits into works acting inside a single firewall and works that involve the network architecture around it [1]. The oldest single-firewall idea is to change the order in which rules are consulted. Hamed and Al-Shaer observed that in enterprise networks most of the traffic matches a small and stable subset of the ruleset, so the average filtering time drops when the frequently matched rules move toward the head of the chain; since

the optimal ordering under the precedence constraints between overlapping rules is NP-complete, they proposed a heuristic approximation [5]. Fulp modeled the precedence relations among rules as a directed acyclic graph and reordered them through admissible swaps of adjacent entries [6]. What this family can promise is bounded by construction: the device keeps the same rules, and the best outcome is the best permutation of what is already there.

A second family shrinks the ruleset instead of reordering it. The Firewall Compressor of Liu et al. [7] transforms the rule sequence into a decision diagram and returns an equivalent configuration close to the minimum size; Daly et al. [8] reach the same goal by resolving the differences between adjacent hyperplanes of the rule space. Both run offline, so the administrator keeps managing the uncompressed, readable ruleset. Rule analysis starts from a different concern: the conflict classification of Al-Shaer et al. [9] discovers anomalies such as rule shadowing and redundancy, within one firewall and across connected ones, and the survey of Jabal et al. [10] collects the tools of this kind. Ordering and compression preserve the filtering policy exactly; analysis may change what the firewall accepts, because its purpose is to expose misconfigurations. As the REDIAL paper notes, all three families are compatible with the redistribution and can be profitably combined with it [1].

The works closest to REDIAL in scope involve more than one firewall. Parallel architectures spread the filtering load over devices working side by side, distributing packets or rules across them and reconciling the verdicts [11]; the price is additional hardware and a topology designed for it, both rare in the hierarchically organized industrial networks of Chapter 1. A different approach moves the rules at runtime between devices already deployed. Yan et al. [12] equalize the workload of the firewalls along a critical traffic path by moving rules toward the source of a traffic peak, reporting an average workload reduction of 60%; their mechanism, however, relies on extra checks on otherwise unused IP header fields, so the filtering semantics of the device does not survive unchanged. Bagheri and Shameli-Sendi [13] target a cloud instead: when a distributed controller detects a change in the traffic, a virtual firewall is instantiated at the edge and the relevant rules migrate to it, with packets rerouted accordingly; the claimed preservation of the policy is not proven formally.

REDIAL, presented in Section 2.2, sits between these lines of work and outside them. It shares with rule ordering and compression the preservation of the filtering policy, and with the multi-firewall schemes the idea of acting on the architecture rather than on one device. What sets it apart is the direction and the price of the move: rules travel downstream, toward the destination subnetworks, onto devices already in the field, while routing, packet formats, and firewall semantics stay untouched, and the transformation comes with a correctness proof and with the bounds $[\Gamma_m, \Gamma_M]$ on the achievable gain [1]. Measuring that gain has its own, thinner

literature. The benchmarking methodology of RFC 3511 [14] targets conventional networks and favors throughput-oriented figures over the latency and jitter that matter in industrial scenarios; the Markov model of Salah et al. [15] describes an iptables-based firewall, but stops at IP-layer filtering; and high-speed packet generators such as MoonGen [16] mark the state of the art from which Section 2.3 departs. The two reference works of this thesis break both patterns: the model of Section 2.4 brings the analysis to an industrial firewall, where determinism is the requirement [3], and the methodology of Section 2.3 brings sub-microsecond resolution to commodity hardware [2]. This thesis applies the same standard of measurement to a firewall whose ruleset REDIAL has rewritten, and checks each cell against the placement-specific reference of Section 2.2 on the full grid of operating conditions of Chapter 7.

Chapter 3

Analytical model and observables

Chapter 2 named the quantities this thesis measures without yet defining them precisely enough to compute. Validating REDIAL needs two kinds of quantity, one that the theory predicts and one that a real network actually experiences. The theory speaks in rules evaluated: σ, the average number of rules a packet is checked against before being matched, and the gain Γ built from it. The network speaks in time: D, the latency the firewall adds to every packet that crosses it. This chapter defines both, ties them together with the linear model that turns one into a prediction of the other, and identifies the placement-specific reference used in Chapter 7: Γ_M for head, Γ_{av} for proportional interleaving, and Γ_m for tail.

3.1 The observable σ: average rules evaluated per packet

For a given traffic mix, σ is the average number of rules a packet is checked against before the ruleset returns a verdict for it: a packet matched by the k-th rule contributes k to the average, and a packet that falls through to the default policy contributes the full ruleset length N. This is exactly the quantity the REDIAL bounds of Section 2.2 are stated in terms of, so no reinterpretation is needed to compare theory and measurement.

σ is measurable exactly, not estimated, because iptables keeps a per-rule packet counter. Reading the counters before and after a run and reconstructing, for each rule, how many packets reached it gives σ directly from the ruleset's own bookkeeping, with no need to infer matching order from traffic traces. Wildcard or catch-all rules are counted like any other rule in this sum, consistent with how the

REDIAL paper treats them: a packet that only matches a trailing wildcard still pays for every rule placed ahead of it.

Because traffic in the testbed originates from two subnets with independent rulesets, σ is really two quantities, σ_{net1} and σ_{net2}, one per source. When the two flows are combined in proportion ρ of $net2$ traffic and $1 - \rho$ of $net1$ traffic, the aggregate observable is the linear blend

$$\sigma_{tot}(\rho) = (1 - \rho)\,\sigma_{net1} + \rho\,\sigma_{net2},$$

a property checked repeatedly in Chapter 7 to confirm that mixed-traffic measurements are consistent with the per-subnet ones taken separately.

3.2 The observable D: per-packet forwarding latency

D is the mean additional delay a packet experiences while traversing the firewall, averaged over all packets in a run. It is obtained the same way as in the measurement setup of Section 2.3: hardware timestamps are taken on the TAP copy of the ingress link and on the egress link, the per-port offset is compensated as described in Chapter 4, and the difference between the two timestamps for each packet is averaged over the run to give D.

D is not purely the cost of rule evaluation. It also contains a fixed traversal cost common to every packet, independent of the ruleset: interrupt handling, buffer copies, and whatever baseline the kernel network stack imposes even with an empty chain. Separating this fixed part from the part that grows with the ruleset is exactly the purpose of the linear model introduced next.

3.3 The linear model $D = D_0 + c \cdot \sigma$

Section 2.4 recalled the TII 2018 result that latency in an application-layer firewall grows linearly with the position of the matching rule. The same shape carries over to plain iptables filtering in a network-namespace setup: the mean latency D is modeled as

$$D = D_0 + c \cdot \sigma,$$

with D_0 the fixed traversal cost identified in the previous section and c the marginal cost of evaluating one additional rule. Both parameters are obtained by a linear fit of measured D against measured σ across rulesets of different lengths, with D_0 the intercept and c the slope.

The model is only claimed to hold in the stable region of this relationship, where D grows linearly with σ. Beyond a certain ruleset length the system leaves this

region and latency grows faster than linearly, a knee that Chapter 6 characterizes separately; every use of D_0 and c in this thesis assumes operation below that knee.

3.4 Predicted gain Γ and the placement-specific theoretical references

The predicted gain Γ is the ratio between σ before rule redistribution and σ after it, $\Gamma = \sigma^{PRE}/\sigma^{POST}$, with both terms taken from the exact per-rule counters of Section 3.1 rather than from the closed-form expressions of Section 2.2. This makes Γ a measured quantity rather than a purely theoretical one. In the experimental matrix it is compared with one closed-form reference selected by the placement of the net2 rules, not with the same bound in every cell.

The three experimental variants turn these closed-form values into distinct targets. When the net2 block is at the head ($h = 1$), the measured Γ attains Γ_M by construction. Proportional interleaving reproduces the uniform-position assumption underlying Γ_{av}. When the net2 rules are at the tail, the measured Γ is compared with Γ_m and approaches it from above. Head, interleaved, and tail are therefore not tested against one common bound.

The blend property established for σ in Section 3.1 carries over to D: for a mix of ρ of $net2$ traffic and $1 - \rho$ of $net1$ traffic,

$$D_{tot}(\rho) = (1 - \rho)\, D_{net1} + \rho\, D_{net2},$$

a direct consequence of the linear model of Section 3.3 applied to a blended $\sigma_{tot}(\rho)$.

Operationally, validating REDIAL in Chapter 7 will mean checking three things against the definitions of this chapter: that each placement tracks its selected reference head $\rightarrow \Gamma_M$, interleaved $\rightarrow \Gamma_{av}$, and tail $\rightarrow \Gamma_m$ with no single bound applied to all 27 cells; that the D versus σ relationship stays linear over the tested range; and that the blend properties for both $\sigma_{tot}(\rho)$ and $D_{tot}(\rho)$ hold on real mixed-traffic runs and not only on the two single-subnet extremes.

Chapter 4

Experimental testbed

Every number in this thesis comes from one physical setup. This chapter describes it: the hosts and their roles, the network path with its timestamping hardware, the system software, and the emulated topology that contains FW1. The design is not built from scratch; it adapts the low-cost delay measurement methodology of [2], presented in Section 2.3, to the FW1-only scenario defined in Chapter 3. What the adaptation keeps, and what it had to change, is discussed at the end of the chapter (Section 4.6). How far the resulting instrument can be trusted is not argued here: that is the subject of Chapter 6.

4.1 Measurement scenario

Chapter 3 reduced the validation to a comparison: the same firewall, loaded first with the PRE ruleset and then with the POST one, crossed by identical traffic, with σ and D measured in both states. This section describes the scenario that makes the comparison possible in practice, while the following sections detail the hardware, the instrumentation, and the emulated topology.

The bench must provide three things. The first is a firewall that is real in every respect that matters here: a Linux netfilter/iptables packet filter, forwarding traffic between subnets through a FORWARD chain with a DROP policy. The second is an exact measurement of σ, obtained by reading the iptables per-rule counters before and after each run over a channel that does not touch the data path. The third is a per-packet latency measurement fine enough to resolve the cost of a single rule, which the results of Chapter 7 place below 100 ns per rule. This last requirement is the demanding one, and it dictates the structure of the whole bench.

Latency is measured differentially, following the methodology of [2]. No clock on the path is trusted with an absolute timestamp. Every test packet is captured twice instead: a network TAP inserted upstream of the firewall delivers a copy of

the packet as it enters, and the receiving endpoint captures the same packet after it has crossed the firewall. Both captures are timestamped in hardware by the NIC, at nanosecond resolution, and the firewall delay is the difference Δ between the two timestamps. The TAP emits the monitor copy and the forwarded packet together, so its own delay is common to both branches and cancels in Δ; the only correction left is the constant offset between the clocks of the two capture ports, which is calibrated separately (Chapter 6).

A deliberate simplification makes this scheme robust: sender and collector are the same machine. The traffic source and both capture ports live on TS-P5, on one Intel I350 card, so the measurement never compares timestamps taken by two different hosts and no clock synchronization protocol is needed. The firewall runs on a separate machine, TS-P360, inside a network namespace, and knows nothing about how it is being observed. The roles and specifications of each host are given in Section 4.2.

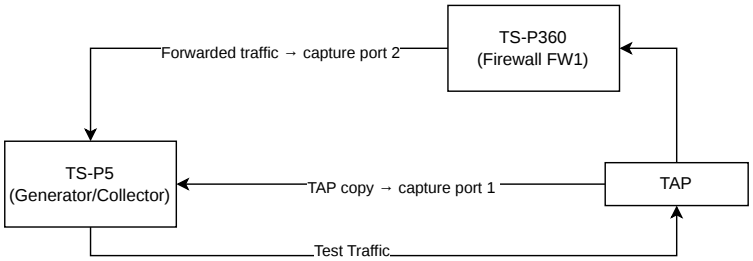


Figure 4.1: Logical topology of the measurement path. The TAP delivers a copy of each packet to the first capture port (t_1); the same packet, after crossing FW1, reaches the second capture port (t_2). Both ports belong to the same Intel I350 on TS-P5, which timestamps them in hardware; the firewall delay is the difference between the two timestamps.

The scenario is completed by a strict separation between two planes. The subnets that carry test packets exist only inside the namespaces and on the cabled path between SX, the TAP, the firewall, and RX. Everything else travels on a management network, a flat 192.168.100.0/24 segment built around a small unmanaged switch: the SSH sessions that drive the runs, the application and rollback of REDIAL, and the collection of counters and captures at the end of each run. No management packet ever crosses the measured path, so orchestration cannot perturb the quantity being measured. The segment has a single gateway to the outside, ieiitp97 (192.168.100.102), which acts both as bastion host and as REDIAL orchestrator; TS-P5 (.101), TS-P360 (.103) and, for the campaigns of Chapter 8, the Raspberry Pi 5 (.104) are reachable only from inside. Figure 4.2 shows this arrangement.

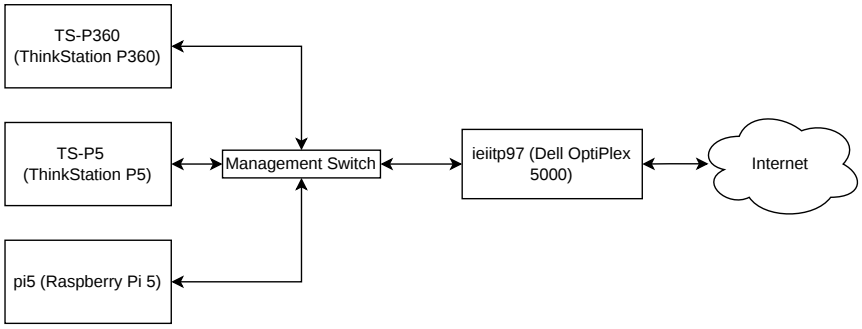


Figure 4.2: Management network of the testbed. All control traffic, SSH orchestration, REDIAL apply and rollback, result collection, flows through this segment and never touches the measured path. ieiitp97 is the only host exposed outside the laboratory.

4.2 Testbed hardware

The bench is built from four machines, all commodity hardware: three desktop workstations and, for the portability study of Chapter 8, one single-board computer. Table 4.1 lists them with their roles and their addresses on the management network; the paragraphs below give the details that matter for the measurements.

Table 4.1: Hosts of the testbed. Management addresses belong to the 192.168.100.0/24 network of Figure 4.2; the addresses of the emulated topology exist only inside network namespaces (Section 4.5).

Host	Machine	Role	Mgmt IP
TS-P5	ThinkStation P5	Traffic sender and collector; hosts the capture NIC	.101
ieiitp97	Dell OptiPlex 5000	REDIAL orchestrator, bastion host	.102
TS-P360	ThinkStation P360	Firewall under test (FW1)	.103
pi5	Raspberry Pi 5	Firewall host for the porting study (Chapter 8)	.104

TS-P5 is the measurement instrument. It generates the test traffic and captures it at both ends of the measured path, and it hosts the component the whole methodology depends on: an Intel I350-T4, a four-port Gigabit Ethernet card whose ports timestamp incoming packets in hardware, on the NIC itself, before any driver or kernel code is involved. Two of its ports are dedicated to capture: one receives the TAP copy of each packet, the other the traffic that has crossed

the firewall (Section 4.3). The machine is a Lenovo ThinkStation P5 with an Intel Xeon w3-2425 (6 cores, 12 threads, up to 4.2 *GHz*) and 32 *GB* of DDR5-4800 ECC memory. Its computing power is not critical: with hardware timestamping, the quality of the measurement does not depend on how fast the host processes packets, only the sustainable send and capture rate does.

TS-P360 is the system under test. FW1 runs here, an iptables packet filter inside a network namespace, and every latency measured in this thesis is paid by this machine's CPU. Its specifications are therefore the ones that shape the numbers: the fixed cost D_0 and the per-rule cost c measured in Chapters 6 and 7 are properties of FW1 running on this hardware. The machine is a Lenovo ThinkStation P360 Tower with an Intel Core i9-12900K and 32 *GB* of DDR5-5600 memory. The processor is a hybrid design, with 8 performance and 8 efficiency cores running up to 5.1 *GHz*: which kind of core serves the network path is one of the factors behind the rate-dependent effects examined in Chapter 6, and one of the reasons the system configuration described in Section 4.4 pins the operating conditions of the measurements.

ieiitp97 is the control plane. A Dell OptiPlex 5000 with an Intel Core i7-12700 (12 cores, 20 threads, up to 4.8 *GHz*) and 16 *GB* of DDR4-3200 memory, it plays two roles: it runs the REDIAL orchestrator, which computes and applies the rule redistributions, and it is the bastion host, the only machine with an address outside the laboratory. Campaigns are driven from here over SSH. The host never touches the measured path, so its hardware has no influence on the results; what it buys is repeatability, since every run of every campaign is launched, supervised, and archived from a single point.

The fourth machine, a Raspberry Pi 5, joins the bench only in Chapter 8, where it replaces TS-P360 as the firewall host to test whether the conclusions of the validation depend on the platform. Its configuration, including the USB Ethernet adapters it requires, is described there.

4.3 Network and measurement instrumentation

The differential scheme of Section 4.1 needs two things from the instrumentation: timestamps that are accurate at the nanosecond scale, and a way to capture the same packet before and after the firewall. The first is provided by the capture NIC, the second by the TAP.

Timestamps are taken by the Intel I350 in hardware. When a frame arrives, the NIC records the time at the MAC layer, against its own physical hardware clock (PHC), and hands the timestamp to the capture stack together with the packet. Nothing that happens later on the host, interrupt coalescing, scheduling, buffering in libpcap, can alter it. This is the core of the methodology of [2]: a

software timestamp taken by the kernel would carry microseconds of jitter, more than the entire effect to be measured, while the PHC of the I350 advances in steps of 8 *ns*. The two capture ports are served by two distinct PHCs, one per port, and the captures are ordinary tcpdump sessions configured to record the hardware timestamps.

Two clocks, however, mean one offset. The PHCs of the two capture ports start from different instants and differ by a constant that is roughly 12 μs on this bench, the same order of magnitude as the delays under measurement, so it cannot be ignored. It is measured with a run in which the firewall is removed from the path and both ports capture the same packets: the arrivals are physically simultaneous, so any difference read by the clocks is the offset itself. The value is stable within a run campaign but changes at every power cycle, which is why it is remeasured whenever the bench is restarted; the calibration procedure and its precision are part of Chapter 6.

The TAP is the split point of the measurement. It sits on the segment between the sender and the firewall, forwards each frame toward FW1, and emits a copy of it on the monitor output; the copy reaches the first capture port and provides t_1, while the forwarded frame crosses the firewall and reaches the second capture port, providing t_2. The TAP is an active device and adds its own forwarding delay, but the copy and the forwarded frame leave it together: whatever time the TAP spends on a frame is spent upstream of the split, shifts t_1 and t_2 by the same amount, and cancels in $\Delta = t_2 - t_1$. For this reason the TAP needs no delay budget and no calibration.

The laboratory also includes an industrial Ethernet switch, a Hirschmann MACH104. It is not part of the REDIAL measurement path: it enters the story only in Chapter 6, as a device of stable, known delay on which the measurement method is put to the test before being trusted on the firewall.

Physically, the data path is four cables and no intermediate device. The sender transmits from a 10 GbE X550 port of TS-P5 into the TAP; the TAP forwards toward one X550 port of TS-P360, where the firewall runs, and delivers the copy of the traffic to the first I350 capture port (t_1); the second X550 port of TS-P360 is cabled directly to the second I350 capture port, which is also the sink (t_2). The Ethernet switch mentioned above therefore never touches the measured path.

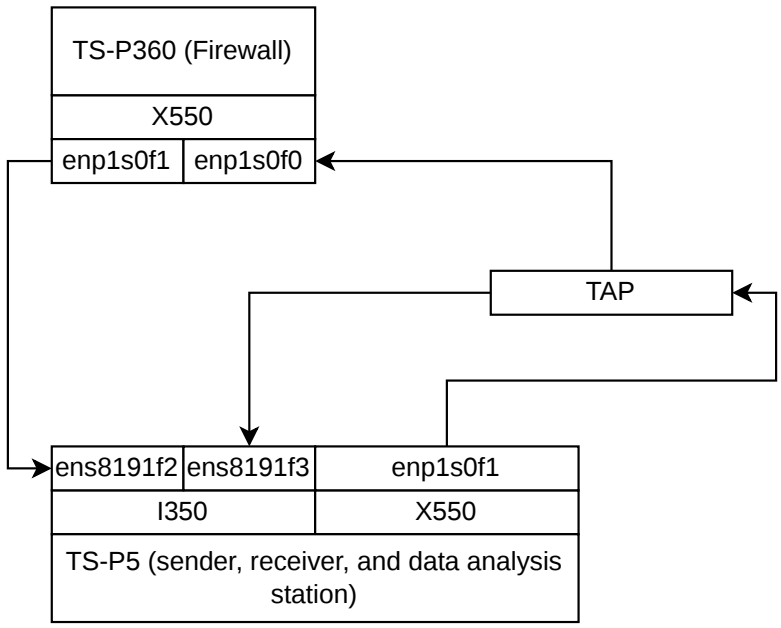


Figure 4.3: Physical cabling of the measurement path. Port and interface names as configured on the bench.

4.4 System software and configuration

The three x86 hosts run Ubuntu 26.04 LTS with kernel 7.0, netfilter/iptables, and tcpdump/libpcap. Nothing in their stack is custom or patched. The Raspberry Pi 5 used in Chapter 8 runs Ubuntu 25.10 with kernel 6.17; its platform-specific configuration is documented in that chapter. What makes the bench a measurement instrument is not the software itself but its configuration: a modern Linux host is tuned for throughput and energy efficiency, and several of those default behaviors trade exactly the property this bench needs, per-packet determinism. This section lists the settings that were changed and why; Table 4.2 collects them for reproducibility.

The first group of settings concerns CPU power management. Left to itself, the CPU scales its frequency with load and parks idle cores in deep sleep states; at the rates of interest here the network path is mostly idle between packets, so each packet risks paying a wake-up whose cost depends on how long the CPU has been sleeping. The consequences are visible in the data and are analyzed in Chapter 6; the countermeasure, applied on every host, is to pin the frequency with

the `performance` governor. Energy-Efficient Ethernet is likewise disabled on the capture ports, so the PHY never enters its low-power idle state during a run.

The second group concerns the capture path. The captures are ordinary tcpdump sessions, but configured so that nothing between the NIC and the pcap file can drop or distort packets. Timestamps are the hardware ones from the I350 (`-j adapter_unsynced`, nanosecond precision), so nothing in this list affects their accuracy; the configuration protects completeness, since a run that loses even one packet in capture confuses loss caused by the firewall with loss caused by the instrument. The NIC receive rings are raised to their maximum of 4096 descriptors, the offload engines (GRO, LRO, TSO, GSO) are disabled so that frames reach the capture socket exactly as they arrived on the wire, one timestamp each, a BPF filter selects the test traffic already in the kernel, and the socket buffer is enlarged so that bursts of arrivals do not overrun it. The tcpdump processes run with real-time priority (`SCHED_FIFO`) on dedicated cores. These settings are not applied once and hoped for: the capture script reapplies the whole set at the start of every run, so a reboot or a driver reload cannot silently return an interface to its defaults.

The sender receives the same treatment. It writes raw frames through an `AF_INET` raw socket with `IP_HDRINCL`, bypassing the kernel UDP stack, and paces them with an integer period in nanoseconds (Chapter 5); it runs under `SCHED_FIFO` with its memory locked (`mlockall`) on a core reserved for it, so neither the scheduler nor the page cache can insert pauses into the transmit timing.

Some noise remains. The cores are not fully isolated from the kernel: the scheduler tick and the softirq machinery still interrupt the sender and the capture processes occasionally, which is visible as isolated spikes in the inter-packet intervals at the microsecond scale. These interruptions do not touch the measured quantity, because the timestamps are taken by the NIC against its own clock regardless of what the host CPUs are doing; their only effect is on the regularity of the offered load, which is verified run by run from the inter-packet intervals recorded in the TAP capture.

Table 4.2: Measurement-relevant system configuration. All settings are reapplied automatically at the start of every run.

Item	Setting
x86 OS / kernel	Ubuntu 26.04 LTS, kernel 7.0.0-27-generic, unmodified
Raspberry Pi OS / kernel	Ubuntu 25.10, kernel 6.17, unmodified
CPU governor	`performance` on all hosts
EEE	disabled on the capture ports
NIC offloads	GRO, LRO, TSO, GSO disabled on all test-path interfaces
RX ring	4096 descriptors (I350 maximum) on the capture ports
Timestamping	hardware, `adapter_unsynced`, nanosecond precision
Capture filter	BPF on the test flow, applied in kernel
Scheduling	`SCHED_FIFO` for sender and tcpdump, dedicated cores
Sender memory	locked with `mlockall`

4.5 Emulated topology

The network that the firewall observes is emulated with Linux network namespaces. A namespace gives a process group its own private network stack: its own interfaces, routing table, ARP cache, and iptables ruleset. This choice has two properties that matter here. First, the firewall under test is not a simulation of netfilter but netfilter itself, running natively in the kernel with no hypervisor or container runtime in the data path; the measured cost per rule is therefore the cost a dedicated middlebox would pay. Second, each physical NIC port is moved into a namespace, so a packet travelling from one namespace to another must leave through a real port, cross a real cable, and re-enter through another real port. The kernel cannot short-circuit the path internally, and the TAP of Section 4.3 sees every packet on the wire.

Four namespaces implement the topology of Figure 4.1. On TS-P5, `SX` holds the sender and one 10 GbE port, `TAP` holds the capture port that receives the TAP copy, and `RX` holds the capture port at the output of the firewall. On TS-P360, `FW` holds both 10 GbE ports and is the only node with packet forwarding enabled; it is the firewall. Table 4.3 summarizes the addressing plan.

Table 4.3: Namespaces, interfaces, and addressing of the emulated topology. The capture interfaces carry no test traffic of their own and run in promiscuous mode.

Namespace (host)	Interface	Address	Role
`SX` (TS-P5)	X550 port	`10.0.0.1/24`	traffic source
`TAP` (TS-P5)	I350 port	none, promiscuous	capture of the TAP copy (t_1)
`RX` (TS-P5)	I350 port	`10.0.1.3/24`, promiscuous	sink and capture at the firewall output (t_2)
`FW` (TS-P360)	X550 port	`10.0.0.3/24`	firewall ingress, default gateway of `SX`
`FW` (TS-P360)	X550 port	`10.0.1.1/24`	firewall egress toward `RX`

The address plan uses three subnets. The source side is `10.0.0.0/24`. On the destination side there are two subnets, `10.0.1.0/24` and `10.0.2.0/24`: two, and not one, because the distinction between them is the dimension along which REDIAL operates. The ruleset partitions its rules between the two destination subnets, and REDIAL's decision is precisely which of those partitions to offload downstream. Neither subnet contains real services. `RX` owns an address in the first subnet and absorbs the second through a static route on the firewall (`onlink`, pointing at `RX`); since its capture interface is promiscuous, it records every forwarded packet regardless of destination address. The firewall is the only router: `ip_forward` is enabled in the `FW` namespace alone, and `SX` and `RX` reach each other only through it.

The ruleset lives in the `FORWARD` chain of the `FW` namespace with a `DROP` policy. The reference ruleset of the characterization campaign holds 601 rules; a packet that matches rule h pays the evaluation of the $h - 1$ rules above it, and a packet that matches nothing pays the full scan and is discarded by the policy. How the rules are generated, how they are split between the two destination subnets, and where the matching rule is placed are experimental factors, described with the toolchain in Chapter 5. The per-rule packet counters that iptables maintains are read over the management network at the end of each run and provide the traffic distribution σ without adding any instrument to the data path.

One property of namespaces required explicit treatment: they are runtime state and do not survive a reboot, while the physical cabling and the scripts on disk do. The topology is therefore never built by hand. Two idempotent scripts, one per machine, create the namespaces, move the ports, assign addresses and routes, and enable forwarding; each supports an `up`, `status`, and `down` action, so the bench can be verified or rebuilt in seconds after a power cycle. Since Linux interface names may change across cold boots, the scripts identify the ports by MAC address rather

than by name. The scripts are listed in Appendix B.

4.6 What was reused from the reference methodology and what was redesigned

The bench is an adaptation of the measurement architecture of [2]; this closing section separates what was taken as published from what had to change. The parts that carry the metrological weight are reused as they were designed. The traffic monitor built around NIC receive timestamping, the TAP as split point of the measured path, and the differential measurement that cancels every common-mode delay are all taken from the paper; Chapter 6 verifies that they behave on this bench as they do in the original work.

The traffic generator is not one of these reused components. What the bench takes from the paper is the principle, pacing every packet against an absolute, kernel-level time reference so that a missed deadline is skipped rather than made up with a burst, not the paper's own software. The sender that actually drives the bench, `sendpkts` (Section 4.3), is a dedicated tool built for this thesis, developed from an earlier in-house generator used for the calibration campaigns of Chapter 6, and the remainder of the toolchain is described in Chapter 5.

The differences come from the different question being asked. The paper measures a device under test in isolation; here the device is one firewall of a REDIAL chain, and the bench must switch it between two rulesets and attribute the work it does to individual rules. Four changes follow. Sender and collector are collapsed onto one machine, so both capture ports share the same card and no clock synchronization between hosts is needed; the price is the constant offset between the two port clocks, which the original setup does not have and which is handled with the calibration of Section 4.3. The firewall runs inside a network namespace rather than on a dedicated box, which lets one machine host the whole emulated topology of Section 4.5 without changing what netfilter executes. The observable σ is added alongside the latency, read from the iptables counters over the management network, because the REDIAL model is stated in rules evaluated per packet, not in microseconds. The whole cycle is automated: runs are driven over SSH from the orchestrator host, and the application and rollback of REDIAL between the PRE and POST halves of each pair is a scripted step of the campaign, not a manual one (Chapter 5).

With the testbed in place, two questions remain: what software runs it, and how far its measurements can be trusted. Chapter 5 answers the first; Chapter 6 answers the second.

Chapter 5

Measurement and experiment toolchain

No off-the-shelf tool covers the workflow this thesis needs: generating a parametric ruleset with a known matching position, generating traffic reproducible enough that σ is predictable before a packet is sent, applying and rolling back a REDIAL redistribution, driving a full PRE/POST campaign around all of that, and recovering σ and D from what each run produces. A dedicated toolchain was developed to fill these roles. This chapter presents it by phase of the pipeline, generation, execution, and measurement, rather than one section per tool, and discusses the design choices behind each phase; the command-line details needed to reproduce a run are collected in Appendix A.

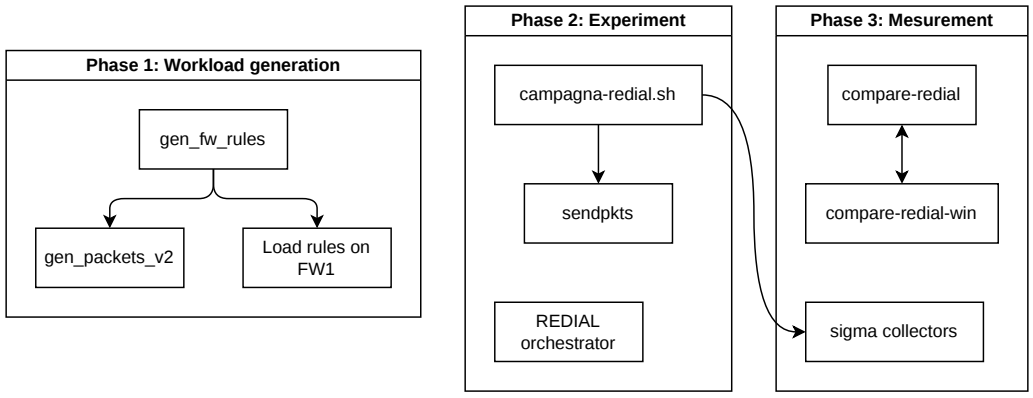


Figure 5.1: Pipeline of the toolchain, from ruleset and traffic generation through campaign execution to the extraction of σ and D.

5.1 Workload generation

The first phase produces the two inputs every run needs: a ruleset with known properties and a packet vector to send against it. `gen_fw_rules` builds the ruleset. It exposes four knobs that map directly onto the experimental factors of Chapter 7: the order in which rules for the two destination subnets are placed (`n2n1`, `n1n2`, or `mixed`), the presence of wildcard rules, the split of rules between `net1` and `net2` of Section 4.5, and the depth h at which the matching rule for the probe flow sits. Every knob is a parameter of the generator, not something adjusted by hand in an iptables script, so a ruleset used in a campaign is fully described by the command line that produced it.

The tool went through two iterations, v5 and v6, and the difference is worth recording because it changed what "mixed" means. In v5, a mixed ordering interleaved rules for the two subnets one-for-one regardless of how many rules each subnet actually held, which is only a faithful mix when the split is close to even. v6 replaced this with proportional interleaving: rules from `net1` and `net2` are interleaved in proportion to their share of the ruleset, so a 75/25 split still produces a chain that looks locally mixed instead of degenerating into long runs of one subnet's rules. This matters because the mixed ordering is meant to sit between the head-heavy and tail-heavy extremes of Section 3.4; an interleaving that ignored the split would not be representative of that middle case for the asymmetric splits used in the campaign.

`gen_packets_v2` produces the second input, the packet vector sent by `sendpkts`, the custom-built sender of Section 4.3. Every vector is generated from a fixed seed, so the same vector can be regenerated exactly for a PRE and a POST run, or years later to repeat an experiment. Reproducibility here is not a convenience but a requirement of the model: σ in Section 3.1 is exact only if the traffic distribution across destinations and match positions is known, and a vector that could not be reproduced exactly would turn every PRE/POST comparison into a comparison between two different, unmeasured traffic mixes rather than the same traffic against two rulesets.

5.2 Experiment execution and orchestration

Three pieces of software turn the generated inputs into a run:

1. **`sendpkts`, the sender.** It is a dedicated tool built for this thesis rather than the traffic generator described in the reference methodology of Section 4.3; it follows the same principle, pacing packets with an integer period in nanoseconds so that a rate has no fractional remainder to distribute unevenly across packets

and the offered traffic is burst-free by construction rather than by post-hoc smoothing. Its design combines four elements:

- **Pre-built packet vectors.** `sendpkts` reads pre-generated packet vectors from the CSV files `gen_packets_v2` produces and builds every IP/UDP frame once in memory before the run starts, so the hot send path performs no per-packet construction.
- **A dedicated real-time engine.** Timing accuracy is reached with a raw socket using `IP_HDRINCL`, a busy-spin loop against `CLOCK_MONOTONIC`, `SCHED_FIFO` scheduling, locked memory (`mlockall`), and a dedicated, isolated CPU core. This engine was developed and validated independently, on the rate ramps used for the calibration campaigns of Chapter 6, before being extended to read the ruleset-derived vectors of Section 5.1.
- **Deterministic split.** A Bresenham-style interleaving between the net1 and net2 vectors realizes the traffic split ρ exactly over the long run, without bunching packets of either subnet together.
- **Independent wrap-around.** Each vector wraps around on its own once its packets are exhausted, so a vector far shorter than a campaign can still drive it to completion.

2. **`campagna-redial.sh`, campaign orchestration.** Sitting above the sender, it automates a full campaign rather than a single run:

   - **PRE/POST pairing.** Drives a PRE and a POST run as a pair, sharing the same session and packet vector, enforcing automatically the pairing required by the fixed-rate constraint of Section 6.5 rather than leaving it to discipline.
   - **Per-run folder layout.** Lays out one folder per run, named from its parameters, so a result can always be traced back to the ruleset and rate that produced it.
   - **Automatic retry on loss.** Retries a run automatically when the capture reports a loss, a fault expected occasionally given the sensitivity of the differential chain, rather than leaving a gap the analysis stage would later have to notice by hand.

3. **The REDIAL orchestrator.** The third piece, it turns a PRE ruleset into a POST ruleset on FW1:

   - **Apply and rollback.** Applies the redistribution computed for a given experimental condition and can roll it back to restore the PRE state, coordinating the three hosts of Chapter 4 over the management network by SSH so that a campaign requires no manual step on any of them.

- **Custom chains instead of numeric offsets.** The REDIAL paper describes the redistribution as a jump from the upstream chain to rule index j of the downstream one, but iptables has no primitive that jumps into an arbitrary rule index of a chain (`-j GOTO` only jumps to the start of a named chain). The orchestrator therefore materializes, for every redistribution target j, a chain named `REDIAL_A_<j>` (prefix configurable via `custom_chain_prefix`), populates it with the rules that would have started at position j, and replaces the numeric jump with a jump rule `-s d0 -j REDIAL_A_<j>` at the head of the `FORWARD` chain of the downstream firewall.
- **Rule-action translation.** An `ACCEPT` rule moved into `REDIAL_A_<j>` becomes a `RETURN`, so control correctly returns to the caller chain after a first-match verdict; `DROP`/`REJECT` rules keep their terminal action unchanged, since denial never needs to hand control back.
- **Configurable log handling.** A `log_strategy` setting (`move`, `keep`, or `duplicate`) governs logging rules caught in the redistributed range, so a redistribution never silently creates or destroys a log entry the operator relied on.

This custom-chain scheme is specific to this implementation and not part of the REDIAL paper; it is recorded here because every ruleset in Chapters 7 through 9 was produced by it. One scoping note matters for the chapters that follow: in the FW1-only scenario measured in this thesis, no downstream firewall exists to inherit rules, so no jump survives on FW1 itself and the redistribution reduces to the aggregate accept rule; the campaigns were therefore driven by the FW1-only variant of the orchestrator, which builds exactly that. The custom-chain machinery above belongs to the general implementation, and both variants are listed in Appendix A.

5.3 Measurement and analysis

The last phase turns raw counters and captures into the two observables of Chapter 3. The sigma collector implements Section 3.1 directly: it reads the iptables per-rule packet counters at the end of a run, separately for the rules of `net1` and `net2`, and reconstructs σ_{net1} and σ_{net2} from them, including the wildcard rules in the count exactly as the REDIAL model requires. Keeping the two subnets separate at collection time, rather than only reporting a combined σ, is what makes the blend property $\sigma_{tot}(\rho)$ checkable after the fact instead of assumed.

`compare-redial` implements Section 3.2: it pairs the TAP and RX pcap files of a run packet by packet, applies the session offset of Section 6.1, and computes the per-packet latency Δ from the corrected timestamps. `compare-redial-win` then

turns the resulting stream of per-packet latencies into the statistics actually reported in Chapter 7: it aggregates them into fixed-size windows and summarizes each window with the median rather than the mean, which is more robust to the isolated interrupt-driven spikes described in Section 4.4, and applies the acceptance criteria used to flag a run as disturbed, whose thresholds are set from the characterization of Chapter 6. The window size is a user-configurable parameter rather than a fixed constant, so the same tool also supports finer analyses on windows chosen by the user. The official DoS campaign of Chapter 9 instead derives one-second delivered rate and packet loss directly from the TAP and RX packet counts, because aggregate packet accounting, not per-packet delay, is the relevant observable above the knee.

5.4 Cross-cutting design principles

Three principles run through every tool of this chapter rather than belonging to just one of them. Reproducibility is built in from the bottom up: fixed seeds in the generators, versioned scripts so a result can be tied to the exact code that produced it, and one folder per run so nothing is overwritten by the next campaign. Robustness is handled the same way throughout: automatic retries on capture loss, the acceptance criteria of Section 5.3, and windowed, median-based statistics all exist to keep an isolated disturbance from silently becoming a wrong conclusion. Finally, the toolchain mirrors the separation of roles set out in Chapter 4: generation and orchestration run from ieiitp97, execution from TS-P5, and the redistribution is applied to FW1 on TS-P360, so no tool ever runs on the machine whose measurement it would bias.

Chapter 6

Testbed characterization and validation

Chapter 4 described the instrument; this chapter earns the right to use it. Before any REDIAL measurement is presented, the clocks of the capture ports are calibrated (Section 6.1), the differential method is verified on devices whose delay is known and stable (Section 6.2), and the cost of the bare firewall is measured (Section 6.3). Two limits of the bench are then located: the load at which the firewall leaves the linear regime (Section 6.4) and the rate effects that make the fixed cost D_0 depend on the packet rate (Section 6.5). The chapter closes with the decision the remainder of the thesis inherits from these results: the operating rate of the campaigns (Section 6.6).

6.1 Calibration of the capture-port offset

The differential measurement rests on one correction, and this section quantifies it. The two capture ports of the I350 are served by two independent hardware clocks, one PHC per port. The clocks are left free-running: no synchronization protocol touches them during a run, because a servo continuously steering a clock would appear in the data as a slowly varying error. The price of that freedom is that each PHC starts counting at power-up, so the two clocks differ by a constant. In every calibration performed for this work the constant fell near $-12\,\mu$s, the same order of magnitude as the delays to be measured, so it must be removed from every Δ.

The offset is measured by making the two arrivals physically simultaneous. For the calibration run the firewall is taken out of the path: the forward output of the TAP is cabled directly to the second capture port, so the TAP copy and the forwarded frame, which leave the TAP together, reach the two ports at the same instant. Whatever difference the two clocks report is then the offset itself. The

same sender and the same capture chain used in the experiments generate and record the calibration traffic, so the procedure exercises exactly the path it corrects. Figure 6.1 shows the arrangement.

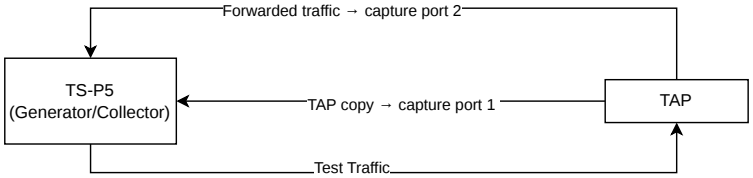


Figure 6.1: Topology of the calibration runs. The firewall is out of the path: the forward output of the TAP is cabled directly to the second capture port, so both capture ports receive the same frame at the same instant and any difference between the two timestamps is the clock offset itself.

The procedure is precise well beyond what the measurements require. As a representative example, one calibration session paired 2.22 million packets, with no loss, and returned an offset of $-11\,764\,ns$ with a standard deviation of $2.6\,ns$, the same order as the $8\,ns$ quantization step of the PHC. Within a session, repeating the run at different rates does not move the value by any measurable amount, so one calibration covers every rate used in the campaigns.

The offset is constant within a session, but not across sessions. The PHCs restart at every power cycle, and the bench was powered down more than once over the course of this work; every restart produced a different value. Two consecutive sessions, for example, returned $-11\,973\,ns$ and $-11\,764\,ns$, a $209\,ns$ jump, small in absolute terms but two orders of magnitude above the calibration noise. There is therefore no such thing as the offset of the bench, only the offset of a session. The rule adopted for all the campaigns follows: the offset is remeasured after every power cycle, stored with the measurement session, and applied only to the captures of that session. A run whose offset cannot be traced to a calibration of the same session is discarded.

6.2 Validating the method on a known device

Section 6.1 established that the two clocks agree to a few nanoseconds; it does not yet prove that the chain measures a real device correctly. For that, the chain was pointed at an object whose behavior is constrained by physics: the Hirschmann MACH104-20TX-F, an industrial Gigabit Ethernet switch. The switch was inserted on the RX branch only, between the forward output of the TAP and the second capture port; the topology, the tools, and the session offset were otherwise exactly those of the calibration runs, so everything the switch adds appears in Δ as it is, and a single run returns its delay directly.

Two properties were verified:

- **Stability:**
  - **Measurements and rate independence:** With minimum-size frames the switch adds 2.734 μs, with a standard deviation of 7.2 *ns*, measured over 30 222 220 packets without a single loss; the value moves by less than ± 3 *ns* between 1 *pps* and 800 *kpps*, six decades of rate.
  - **Jitter and asymmetry argument:** The increase of the jitter with respect to the calibration runs, from 2.6 *ns* to 7.2 *ns*, is itself informative: the switch sits on one branch only, so its fluctuations do not cancel. This is the asymmetry argument of Section 4.3 seen from the other side: what a device adds off the common path appears in Δ in full, and what sits upstream of the split does not appear at all.
- **Accuracy (the stronger test):**
  - **Theoretical basis:** The delay of a store-and-forward switch must grow with the frame length at exactly the serialization rate of the line, 8 *ns* per byte at 1 *Gbps*: a number the measurement chain knows nothing about.
  - **Experimental fit:** A sweep of seven payload sizes, from the minimum to the maximum Ethernet frame, returned the delays of Table 6.1. The delays are fitted with a least-squares line, and the goodness of the fit is reported through the coefficient of determination R^2: the fraction of the variance of the measured points that the fitted line reproduces, equal to 1 when every point lies exactly on the line and decreasing as the scatter around it grows. The fit gives a slope of 8.006 ± 0.005 *ns* per byte and an intercept of 2.219 μs of internal processing, with $R^2 = 0.999998$ and every residual within ± 15 *ns*, about twice the PHC tick (Figure 6.2).
  - **Validation result:** Asked to measure an object with a known physical signature, the chain recovered that signature to within 0.07%, with no free parameter left to tune.

Table 6.1: Switch delay as a function of the frame size. Each row aggregates 3.22 million packets, paired with zero loss; the session offset of Section 6.1 is already removed.

UDP payload (*B*)	**Ethernet frame (*B*)**	D_{switch} **(*µ*s)**	**Std. dev. (*ns*)**
18	64	2.733	7.0
64	110	3.105	7.3
128	174	3.599	7.4
256	302	4.641	7.3
512	558	6.690	7.3
1024	1070	10.786	7.3
1472	1518	14.370	7.3

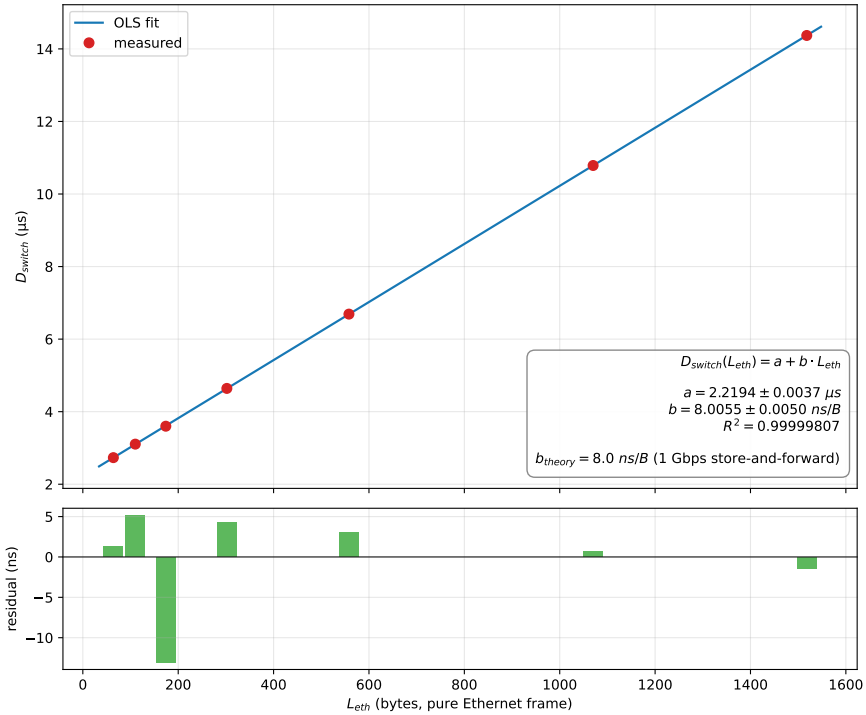


Figure 6.2: Linear fit of the switch delay against the Ethernet frame size. The slope, $8.006 \pm 0.005\,ns$ per byte, matches the serialization time of one byte at 1 *Gbps*; the residuals (bottom panel) stay within $\pm 15\,ns$, about twice the resolution of the PHC.

These numbers enter no delay budget: the switch is not part of the REDIAL data path, and nothing in the following chapters depends on its $2.734\,\mu s$. What

the campaigns inherit from this section is a verdict on the instrument: a chain that resolves a 7 *ns* jitter and recovers the line serialization rate to within 0.07% can be trusted with the per-rule effects that Chapter 7 places below 100 *ns* each.

6.3 Baseline firewall cost: estimating D_0

With the chain validated, the instrument is pointed at the device it was built for. The switch leaves the bench: the TAP sits in series between the sender and the ingress port of FW1, and the egress port of FW1 is cabled directly to the second capture port. Every Δ, corrected with the session offset of Section 6.1, now contains the full cost of crossing the firewall. This section measures that cost in configurations whose σ is known by construction, and produces the first estimates of the two parameters of the model of Chapter 3: the fixed cost D_0 and the per-rule cost c.

The first configuration is the empty firewall: a FORWARD chain with no rules and ACCEPT policy, so $\sigma = 0$ and the measured delay is the pure traversal cost of the path, namespace, routing and netfilter hooks with nothing to evaluate. The same nine-step ramp used for the switch, 30 million packets from 1 *pps* to 800 *kpps*, returned three facts:

- The cost is tens of microseconds: between 1 *kpps* and 400 *kpps* the session measured 64 to 85 μs.
- The noise changes nature: the standard deviation sits at 13 to 25 μs where the switch showed 7.2 *ns*, because the delay is now paid in software, on a CPU that takes interrupts and schedules processes; the chain of Section 6.2 is three orders of magnitude finer than the object it is now measuring.
- The firewall kept forwarding at every rate: no loss up to 400 *kpps*, and 99.96% of the offered traffic delivered even at 800 *kpps*.

One number is deliberately left out of this list: below 1 *kpps* the same empty firewall measured 336 to 351 μs per packet, four to five times its own regime value. That dependence on rate is not noise, and it gets its own section (Section 6.5).

The second configuration prices the scan. Five hundred rules that can never match the test flow, deny entries pointing at destination addresses the probe traffic never uses, are placed in front of the same ACCEPT policy: every packet evaluates all of them and is then forwarded, so $\sigma = 500$. On its own session, the same ramp returned 45 to 94 μs in the stable region.

The obvious next step, subtracting the two sessions to extract the cost of 500 rules, fails, and the failure teaches the rule that the campaigns obey. At 100 *kpps* the two sessions differ by 11 μs, where the slope measured below prices 500 rules at

35; at 200 *kpps* the difference is negative, the loaded firewall apparently faster than the empty one. A small part of the gap is real: a rule that fails its first comparison is abandoned immediately, so ballast rules are the cheapest rules there are. But a negative price has only one reading: D_0 moved between the two sessions by more than the quantity being estimated. Repeated no-rule sessions on this bench scatter over ± 10–$15\,\mu$s with the thermal and frequency state of the machine. What Section 6.1 concluded for the clock offset holds for D_0 as well: there is no D_0 of the bench, only the D_0 of a session. Every estimate of c in this thesis is therefore taken within a single session, and the campaigns of Chapter 7 always pair the PRE and POST runs of a comparison in the same session.

The configuration that isolates c does exactly that: it moves σ within one session and holds everything else still. The ruleset holds 1001 rules: one thousand filler rules that cannot match the probe, plus a single pass-all rule placed at depth h. A probe packet is matched by the pass-all rule and by nothing else, so it evaluates exactly h rules and $\sigma = h$ by construction; the iptables counters verify it at every run, with the pass-all counter holding every packet and every filler counter at zero. Sliding h from 1 to 1001 in steps of 50 sweeps σ across its whole range, with two independent 60-second runs per position. At 100 *kpps*, the one rate at which the sweep stays in the linear regime over almost the whole range of h (the boundary is the subject of Section 6.4), the fit over the 20 positions below saturation returns an intercept $D_0 = 60.7\,\mu$s and a slope $c = 69.5\,ns$ per rule, as Figure 6.3 shows. The points do not sit on a perfect line: above $h \approx 600$ the delay bends upward from the fit, the first sign of the boundary measured in the next section. This is the model of Chapter 3 seen in the data for the first time: the latency of the firewall is linear in the number of rules evaluated per packet, and one rule costs about 70 *ns* at the operating rate, a price the chain validated in Section 6.2 resolves with room to spare.

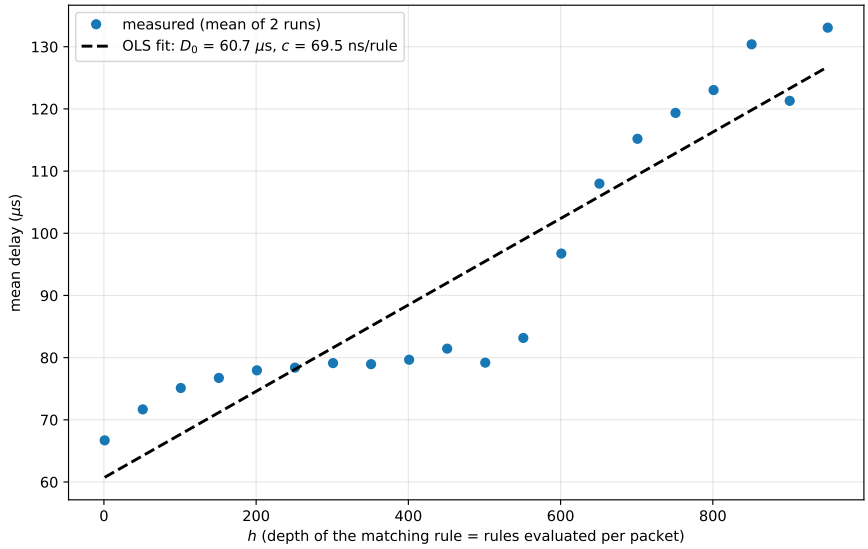


Figure 6.3: Delay of FW1 against the depth h of the matching rule at 100 *kpps*. Each point is the mean of two independent runs of six million packets each; $\sigma = h$ is verified from the iptables counters at every run. The fit over the 20 positions below saturation ($h = 1001$ excluded) returns $D_0 = 60.7\,\mu$s and $c = 69.5\,ns$ per rule.

The sweep also shows where this regime ends. At 200 *kpps* and above the high-h cells leave the linear branch: the delay jumps by more than an order of magnitude, and the iptables counters show that part of the offered traffic no longer reaches the output. Locating that boundary, and measuring what happens beyond it, is the subject of the next section.

6.4 The knee: linear region vs saturation

The measurements of Section 6.3 described the firewall in its linear regime; this section locates the boundary of that regime. The frame is the queue model of [3]: the firewall is a server with a finite buffer, and its service time per packet grows with the number of rules it evaluates. A system like this has two operating regions. As long as packets arrive slower than they are served, the queue stays short, the delay is essentially the service time, and nothing is lost. When the arrival rate exceeds the service rate, the buffer fills: packets that find it full are dropped, and packets that enter it wait for a full queue to drain ahead of them. The boundary between the two regions is the knee. Since the service time grows with σ, the knee is not a packet rate: what saturates the machine is the evaluation load, the product of rate and σ in evaluations per second. The same firewall that delivered 99.96%

of the traffic at 800 *kpps* with an empty chain starts dropping at 450 *kpps* with 500 rules loaded.

The knee was located with the 500-rule ruleset of Section 6.3, driven in one session through a fine ramp of fourteen rates from 100 to 850 *kpps*. Figure 6.4 shows the outcome as an input-output curve. Up to 400 *kpps* the firewall forwards everything: delivered rate equals offered rate and the counters report zero loss. At 450 *kpps* the first packets die, 2.06% of the offered traffic. Beyond that point the delivered rate stops growing: whatever is offered, FW1 delivers about 440 *kpps*, and the loss is the difference, up to 48.2% at 850 *kpps*. The knee of this configuration therefore sits between 400 and 450 *kpps* offered, and the plateau of the curve is the forwarding capacity of FW1 loaded with these 500 rules: about 440 *kpps*, or 2.2×10^8 evaluations per second.

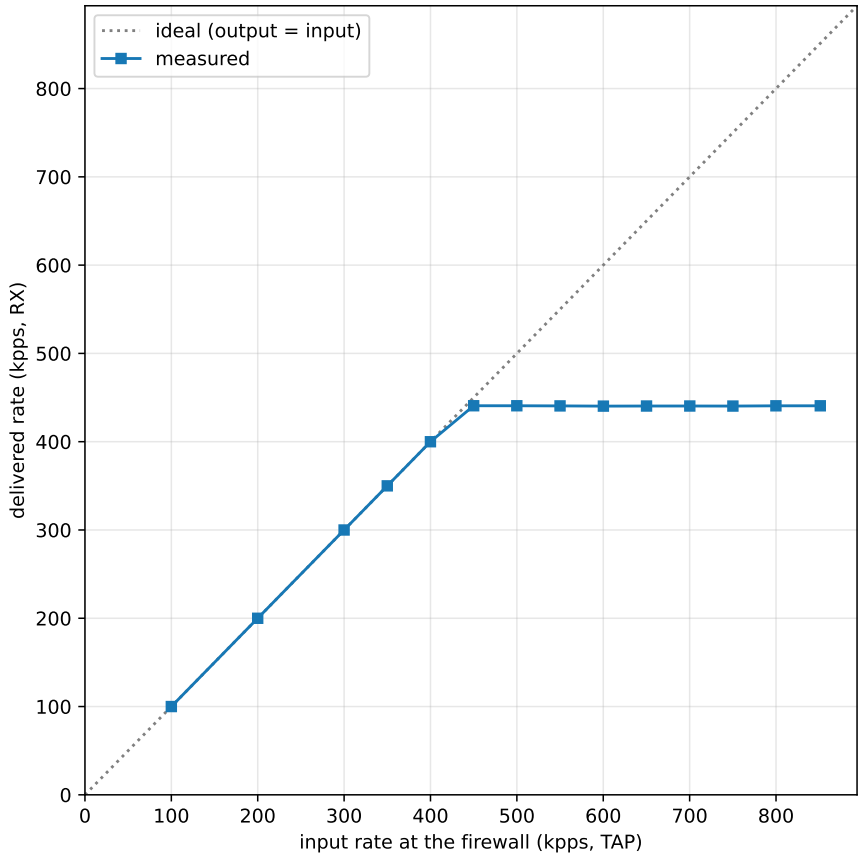


Figure 6.4: Delivered rate against offered rate for FW1 loaded with the 500-rule ruleset, fine ramp from 100 to 850 *kpps*. The curve follows the diagonal up to 400 *kpps*, then flattens at the forwarding capacity of about 440 *kpps*; every packet offered above the plateau is lost.

The delay states the same verdict. Below the knee the ramp measures the tens of microseconds of Section 6.3; above it the delay jumps to 1.14–1.16 *ms* and stays there, whether the offered rate is 450 or 850 *kpps*. The number is no longer a forwarding cost. A packet that survives above the knee enters a buffer that is permanently full, and its delay is the time the firewall needs to drain the queue standing in front of it: a property of the buffer size and of the service time, not of the offered load.

The saturated cells of the pass-all sweep confirm this reading. Their delay does not depend on the rate: the cells saturated at 200, 400, and 800 *kpps* collapse onto a single branch that grows with h at about 5.1 μs per rule, two orders of magnitude above the 69.5 *ns* of the linear branch, because a queued packet pays the scan of every packet standing ahead of it, not only its own (Figure 6.5). The delay is also almost deterministic: at $h = 1001$ and 800 *kpps* the mean is 5 136 μs with a standard deviation of 87 μs, under 2% in relative terms, where the linear regime lives with 20 to 30%. A dedicated session verified where the missing packets die: in that same cell the NIC counters of FW1, the iptables counters, and the RX capture reconcile exactly, 87.75% of the offered traffic is dropped on the ingress ring of FW1 before the first rule is evaluated, and every packet the firewall forwarded reached the capture.

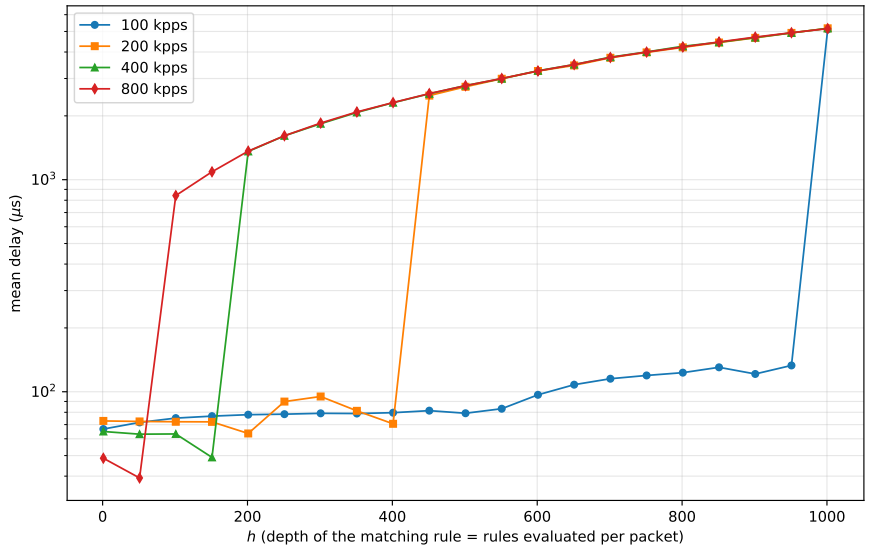


Figure 6.5: Mean delay against the depth h of the matching rule at the four sweep rates, logarithmic vertical axis. Cells below the knee sit on the linear branch of Section 6.3; saturated cells collapse onto a rate-independent branch about two orders of magnitude higher, whose slope is the drain time of the full queue.

The sweep also maps how the knee moves. At 100 *kpps* only the last cell, $h = 1001$, is saturated; at 200 *kpps* the jump appears near $h = 600$, at 400 *kpps* near $h = 300$, at 800 *kpps* near $h = 150$. Doubling the rate halves the depth the firewall can stand: the boundary follows a constant product of rate and h, about 1.15×10^8 evaluations per second across the three saturated rates, and a finer scan around $h = 601$ places its knee between 150 and 175 *kpps*, consistent with the same product. That value is about half of the 2.2×10^8 at which the ballast ramp starts losing packets, and the gap has two components worth recording. The first is the price of an evaluation: rules with different match fields cost differently, and traffic spread over many flows exercises the machine differently than the single probe flow used here, so a capacity in evaluations per second belongs to the machine, the ruleset, and the traffic that crosses them together. The second is that the two numbers measure two different thresholds. The sweep product marks where the delay leaves the linear branch; sustained packet loss needs more, because a queue can inflate the times long before it is permanently full. Between the two thresholds the firewall still delivers everything it receives, but no longer at its linear-regime latency. What generalizes is the shape, a boundary at constant work rate: the two-region behavior that the model of [3] predicts, measured on iptables.

The consequence is a design rule that every campaign in this thesis obeys. Near the knee the measured slope is no longer c: queueing delay mixes into the fit, and Figure 6.3 already showed the bend that appears at 100 *kpps* when h grows past 600. A REDIAL comparison run near the boundary would distort the very quantity it is supposed to measure, so the operating point of the campaigns is chosen with its worst cell far below the delay threshold, the stricter of the two; the margin computation is in Section 6.6. The knee itself returns twice more: as the capacity metric that separates the two platforms in Chapter 8, and as the target of the attack in Chapter 9.

6.5 Rate-dependent effects: NAPI and power management

One number was deliberately left out of Section 6.3: below 1 *kpps*, the same empty firewall that costs 64 to 85 μs in its regime costs 336 to 351 μs per packet. The measurement chain is not the suspect: Section 6.2 showed it rate-invariant within ± 3 *ns* over six decades of rate, so a delay that moves by 280 μs with the rate belongs to the firewall machine. The effect is also counterintuitive in the most useful way: on this bench, offering packets faster makes each of them cross the firewall sooner.

Figure 6.6 shows the effect across the whole ramp of Section 6.3, nine rates from 1 *pps* to 800 *kpps* on the empty firewall. The curve has two regions. From 1 to 100 *pps* the delay sits on a plateau at 336 to 351 μs. Between 100 *pps* and 1 *kpps* it

falls by a factor of four, to 84.9 μs, and from there it keeps declining gently: 76.6 μs at 10 *kpps*, 65.4 at 100 *kpps*, down to 55.7 μs at 800 *kpps*.

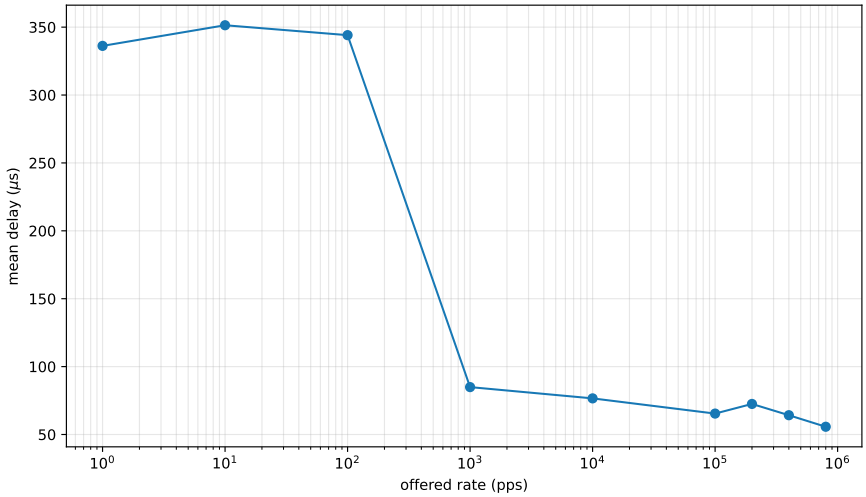


Figure 6.6: Mean delay of the empty firewall (no rules, ACCEPT policy) against the offered rate, logarithmic rate axis. Nine-step ramp, 30 million packets paired with zero loss; each point is the per-step mean with the session offset applied. The plateau below 1 *kpps* is the cost of waking the CPU for each packet; the decline above it is the NAPI driver moving from one interrupt per packet to polling.

Two mechanisms of the Linux network stack produce this shape, and they dominate in different regions. The plateau is the price of waking up. At 100 *pps* the gap between two consecutive packets is 10 *ms*, long enough for the CPU of TS-P360 to descend into its deepest idle states; the next packet then pays the interrupt and the wake-up from those states in full. The governor setting of Section 4.4 does not prevent this: pinning the frequency governs a running core, not a sleeping one. The sharp step between 100 *pps* and 1 *kpps* marks the point where the inter-packet gap drops below the residency threshold of the deep idle states and the core stops reaching them.

The decline that follows is the driver changing strategy. At low rate every packet is delivered by its own interrupt. As the rate grows, NAPI, the interrupt-mitigation model of the Linux network stack, keeps the receive path in polling mode: packets are drained from the ring in batches, and the fixed cost of entering the kernel is shared by the batch instead of being paid by each packet alone. The fine-rate scan of the pass-all sweep puts numbers on this transition: the fitted intercept D_0 falls from 80.4 μs at 10 *kpps* to 52.7 μs at 90 *kpps*, and between 60 and 100 *kpps* the per-position curves settle on a floor of 68 to 75 μs. Past that floor the curves of the deep positions turn upward again, but that rise is the approach of the knee of

Section 6.4, an effect of queueing, not of the driver.

Part of this behavior can be configured away and part cannot. The settings of Section 4.4 already remove the avoidable share on the measurement side. What remains is structural: the interrupt path and the NAPI transition are the normal operation of the stack, and the idle states of the CPU belong to the machine being characterized, so a firewall in production would pay these same costs at these same rates. The bench therefore does not fight the effect; it measures it, and then operates where the effect is exhausted.

The consequence is a reading rule for every number in this thesis. The fixed cost is not a constant of the firewall: D_0 is a function of the offered rate, on top of being a property of the session (Section 6.3). The linear model $D = D_0 + c \cdot \sigma$ therefore holds at fixed rate: two delays measured at different rates differ by driver and power effects that have nothing to do with the ruleset, and no subtraction can separate them afterwards. Every comparison in this work is built to respect this constraint. The PRE and POST runs of each pair share rate and session, so D_0 is common to both and drops out of the difference, and the whole matrix of Chapter 7 runs at one rate. Choosing that rate is the last calibration decision left, and it is the subject of the next section.

6.6 Choice of the operating rate

Section 6.5 ended with a constraint: the whole matrix of Chapter 7 must run at one fixed rate. This section chooses it. The characterization gives the choice the form of a window. The rate must sit above the region where the delay is still falling with the rate, and below the knee of the worst cell the matrix will produce. The value adopted is 100 *kpps*, and both margins can be read from the measurements of the previous sections.

The floor of the window is where the driver reaches its steady state. The fitted intercept D_0 keeps falling up to about 90 *kpps*, and between 60 and 100 *kpps* the per-position curves of the fine-rate scan settle on their floor of 68 to 75 μs (Section 6.5). An operating point below 60 *kpps* would sit on the descending slope of the NAPI transition, where the measured delay still depends on the exact rate offered. At 100 *kpps* the inter-packet gap is 10 μs, three orders of magnitude shorter than the 10 *ms* that lets the CPU reach its deep idle states, and the receive path stays in polling for the whole run.

The ceiling of the window is the knee, and the margin is computed on the worst cell of the matrix. The campaign ruleset holds 601 rules behind a DROP policy, so the deepest scan a packet can pay is 601 evaluations; at 100 *kpps* the worst-case evaluation load is $100\,000 \times 601 = 6.0 \times 10^7$ evaluations per second, about half of the 1.15×10^8 at which the delay leaves the linear branch and less than a third of

the 2.2×10^8 at which packets start dying (Section 6.4). The direct measurement agrees: the finer scan around $h = 601$ placed the knee of that cell between 150 and 175 *kpps*, so 100 *kpps* holds the worst cell below its measured boundary with a margin of at least 1.5. The margin is not larger than it needs to be: at 200 *kpps*, the next rate of the sweep, the $h = 601$ cell is already saturated, its mean delay at 3.3 *ms* in place of 97 μs.

One practical constraint completes the choice. The sender of Chapter 5 paces packets with an integer period in nanoseconds, so a rate is delivered exactly at its nominal value only when its period has no fractional part. At 100 *kpps* the period is 10 000 *ns*, with no truncation, and a 60-second run carries six million packets, enough to push the standard error of the mean far below the per-rule effects being compared.

Every PRE/POST comparison of the campaigns therefore runs at 100 *kpps*. The experiments that leave this operating point do so on purpose: the campaign of Chapter 9 drives the firewall across the knee to measure what the redistribution is worth when the device is overloaded.

The characterization ends here. What the next chapters inherit is an instrument with calibrated clocks, a differential chain verified on a device of known delay, a measured D_0 and c, a mapped saturation boundary, and one operating rate. Chapter 7 puts it to work on REDIAL.

Chapter 7

Experimental campaigns and results

Everything is now in place. Chapter 3 stated the question in measurable terms: does the gain Γ measured on a real firewall agree with the theoretical reference selected for each variant, and does the linear model $D = D_0 + c \cdot \sigma$ hold when the rulesets are the ones REDIAL actually produces? Chapters 4 and 5 built the bench and the software that drives it, and Chapter 6 calibrated the instrument and fixed its operating point at 100 *kpps*. This chapter spends that preparation on the core experiment of the thesis: a matrix of 27 operating conditions, each measured before and after the redistribution, 54 runs in total. All of them were executed on FW1, the firewall of the testbed of Chapter 4, which is a conventional x86 server: throughout this thesis the *x86 firewall* (or *x86 bench*) denotes this machine, as opposed to the Raspberry Pi 5 of Chapter 8. Section 7.1 presents the design of the matrix and Section 7.2 the protocol of a run; Sections 7.3 and 7.4 report the results on σ and on D, Section 7.5 compares them with the theory, and Section 7.6 collects what the campaign established.

7.1 Experimental design: the 27-cell matrix

The unit of the campaign is the cell: one operating condition of FW1, measured twice under identical traffic, in the PRE state and in the POST state. In the PRE state the firewall holds the full 601-rule ruleset of Chapter 4. In the POST state REDIAL has removed the block of rules that FW1 no longer needs to enforce, the rules guarding the destination subnet net2, and left in its place a single shortcut rule; the rules of the other subnet, net1, stay where they were. Each run lasts 60 seconds at 100 *kpps*, six million packets, and the two runs of a cell always share the measurement session, so the clock offset and the fixed cost D_0 are common to both

states and cancel in the PRE/POST difference, as Chapter 6 requires.

Three factors define a cell, and Table 7.1 collects them. The traffic mix ρ is the share of the probe traffic addressed to net2, the offloaded subnet, with levels 0.25, 0.50, and 0.75. The rule split divides the subnet-specific rules between net1 and net2 in the ratios 25–75, 50–50, and 75–25, producing three base rulesets of identical size and different ownership. The third factor is the position of the net2 block inside the PRE order: at the head of the chain, interleaved proportionally with the net1 rules (the mixed variant), or at the tail. Three levels per factor give $3 \times 3 \times 3 = 27$ cells, hence 54 runs.

Table 7.1: Factors and levels of the experimental matrix. Every cell is measured in the PRE and POST states of the same session: $27 \times 2 = 54$ runs, all at 100 *kpps*.

Factor	**Levels**	**Role**
Traffic mix ρ	0.25, 0.50, 0.75	share of the traffic addressed to net2, the offloaded subnet
Rule split (net1–net2)	25–75, 50–50, 75–25	share of the subnet-specific rules each subnet owns
Position of the net2 block	head, mixed, tail	placement in the PRE order; targets Γ_M, Γ_{av}, Γ_m

The first two factors are the parameters of the REDIAL model: ρ and the split enter the equations of Chapter 3 that predict the gain, so sweeping them probes the prediction where it changes. The third factor is the quantity the model deliberately does not fix. How much work the redistribution removes depends on where the offloaded rules sat in the PRE order, and the paper summarizes this degree of freedom with lower, average, and upper references: the three positions of the matrix turn those formulas into targets. With the net2 block at the head, the offloaded rules start at depth 1 of the PRE chain, the $h = 1$ case for which the model computes its upper bound, and in the POST state the shortcut rule at that depth matches every net2 packet; the measured Γ must therefore equal Γ_M by construction. With the block at the tail, every net2 packet pays the full net1 block before reaching its rule, and the gain approaches its floor Γ_m. The proportional interleaving of the mixed variant reproduces the uniform rule distribution for which the model computes Γ_{av}. One factor therefore spans the whole interval $[\Gamma_m, \Gamma_M]$, which is exactly the sweep that the validation of Section 7.5 needs.

Everything else is held still. The nine PRE rulesets are generated from the same seed, so the rule bodies are identical across variants and only their order and ownership change; the traffic vectors are regenerated from each ruleset with the same seed and are reproducible bit by bit. Every run offers the operating rate chosen in Section 6.6, and the design was checked against the knee before the first packet was sent. The heaviest cell of the matrix, the 25–75 split with net2 at

the head at $\rho = 0.25$, evaluates 420.7 rules per packet on average, an evaluation load of 4.2×10^7 evaluations per second at 100 *kpps*: a factor of 2.7 below the 1.15×10^8 at which the delay leaves the linear branch (Section 6.4). The whole matrix therefore runs in the linear regime, where the model of Chapter 3 applies and zero loss is expected. In practice the 27 cells were executed as nine campaigns, one per combination of split and position, each measuring its three values of ρ as three PRE/POST pairs; how a single pair runs, from ruleset load to counter collection, is the protocol of the next section.

7.2 Run protocol

A cell starts from a clean firewall and ends with a rollback, and every step in between is scripted (Appendix A); no run of the matrix was driven by hand. The orchestrator loads the PRE ruleset of the cell into the FORWARD chain, zeroes the per-rule counters, and reads the chain back: the run does not start unless exactly 601 rules are in place with every counter at zero. The traffic vector is generated from the same ruleset with a fixed seed, so the six million packets of a run are a deterministic function of the cell, and the PRE and POST runs replay the identical vector packet by packet.

The pair then runs back to back. The PRE run offers the vector at 100 *kpps* for 60 seconds while both capture ports record (Section A.4); at the end the iptables counters are collected over the management network, which gives σ exactly, with no sampling and no model. The orchestrator applies REDIAL, replacing the net2 block with its shortcut rule (Section A.6), and the POST run repeats the identical transmission; a rollback then restores the full ruleset before the next cell. Both captures of the pair are corrected with the offset of the session calibration (Section 6.1), and since the two runs share session and rate, the fixed cost D_0 is common to both and what their difference contains is the effect of the redistribution.

A run is accepted only if its accounting closes. The packets sent, the packets counted by the rules, and the packets captured at RX must reconcile exactly: a missing packet is loss, and a packet counted by a rule of the wrong subnet is a leak that would falsify σ. A run that fails any check is discarded and repeated within the same session; the campaign script performs the retry automatically (Section A.5). The 54 runs reported in this chapter all close their accounting, with zero loss and zero leaked packets over six million packets each.

7.3 Rule-evaluation results: σ and measured Γ

The first observable of the campaign is σ, and it carries no measurement uncertainty worth discussing: every value in this section is computed from the iptables counters

of its run, and the accounting of Section 7.2 guarantees that every packet is counted once, by the right rule. One cell makes the arithmetic concrete. In the 50–50 split with the net2 block at the head, at $\rho = 0.25$, the PRE firewall evaluates 354.900 rules per packet on average; after the redistribution the same six million packets cost 109.909 rules each, and the measured gain of the cell is $\Gamma = 354.900/109.909 = 3.229$. In the POST state the shortcut rule takes the place where the net2 block began, so the redistribution never reorders what it does not remove. The complete tables, with the per-class σ of each subnet in both states, are collected in Section C.1 of Appendix C; Table 7.2 condenses them into the quantity the model predicts, the measured Γ of each of the 27 cells.

Table 7.2: Measured gain $\Gamma = \sigma_{\mathrm{PRE}}/\sigma_{\mathrm{POST}}$ over the 27 cells of the matrix. Each value comes from one PRE/POST pair of six-million-packet runs at 100 *kpps*; σ is read from the iptables counters.

Position of net2 block	**Split (net1–net2)**	$\rho = 0.25$	$\rho = 0.50$	$\rho = 0.75$
head	25–75	7.436	9.338	14.848
	50–50	3.229	3.934	6.012
	75–25	1.768	2.050	2.887
mixed	25–75	5.115	7.531	14.302
	50–50	2.618	3.895	7.592
	75–25	1.760	2.620	5.107
tail	25–75	1.770	2.113	2.365
	50–50	1.260	1.375	1.459
	75–25	1.088	1.127	1.155

Two trends hold across the whole matrix. The gain grows with ρ in all nine variants: the more traffic is addressed to the offloaded subnet, the more packets benefit from the shortcut. The gain also grows with the share of rules that net2 owns, because that share is what the redistribution removes: at equal ρ and position, the 25–75 split beats the 50–50, which beats the 75–25, in all nine comparisons. The combined range is wide. The same 601-rule policy, on the same firewall, yields $\Gamma = 1.088$ in the least favorable cell (75–25 split, block at the tail, $\rho = 0.25$) and $\Gamma = 14.848$ in the most favorable one (25–75 split, block at the head, $\rho = 0.75$): a factor of 13.6 between two cells that enforce the same policy, decided entirely by where the rules sit and where the traffic goes.

The position of the net2 block behaves as Section 7.1 designed it to. In the nine head cells the measured Γ reproduces Γ_M exactly, at the three-decimal precision of the tables. This identity is a construction check rather than a discovery: the head cells realize precisely the configuration for which the model computes its upper bound, and the counters return the numbers the formula predicts. The nine tail

cells land just above Γ_m, and the nine mixed cells just above Γ_{av}: the floor and the average of the model are approached from the right side in every cell. The cell-by-cell comparison with the selected references is the subject of Section 7.5.

One detail of Table 7.2 rewards a second look: the head position does not always win. At $\rho = 0.25$ the 50–50 head cell gains more than its mixed counterpart, 3.229 against 2.618, but at $\rho = 0.75$ the order flips, 6.012 against 7.592. The mechanism is visible in the PRE costs. With the block at the head, the net2 packets already match at the top of the chain, so the PRE cost falls as ρ grows, from 354.900 to 224.275 rules per packet, and less work is left for the redistribution to remove; in the mixed variant the PRE cost is flat in ρ, between 288 and 291 rules per packet in all nine cells, so the removable work grows with the traffic share of net2. There is no contradiction with the bounds, which refer each PRE ruleset to itself: they bracket the gain achievable from that starting order, and do not rank different orders against each other.

σ counts rules; what a rule costs in microseconds is the subject of the next section.

7.4 Latency results: D and the global fit

The second observable is paid in time. For every run, D is the mean delay of its six million packets, computed from the paired captures and corrected with the offset of the session calibration (Section 6.1). The dispersion around that mean is the software noise already met in Section 6.3: across the 54 runs the per-packet standard deviation sits between 10 and 25 μs, with the PRE runs between about 17 and 24 μs and the POST runs between about 10 and 17 μs. The ordering is consistent: a packet that evaluates fewer rules spends less time inside the firewall and offers a smaller window to the interruptions of the host. Averaging over six million packets pushes the standard error of each mean below 10 *ns*, so the price of a single rule, about 74 *ns*, stands seven standard errors above the noise of a mean.

Table 7.3 reports, for every cell, the saving $\Delta = D_{\mathrm{PRE}} - D_{\mathrm{POST}}$ and the metric this thesis quotes as the gain: the net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$, the saved fraction of the part of the delay the rules are responsible for. The complete delays, state by state, are in Section C.1 of Appendix C. Across the matrix the PRE firewall costs between 46.15 and 66.36 μs per packet and the POST firewall between 35.88 and 60.54 μs. A saving quoted in microseconds alone says little, because its weight depends on the delay it is subtracted from, and the raw ratio Δ/D_{PRE} dilutes the effect with the fixed cost D_0 that no ruleset change can touch; G_{net} avoids both problems, and Section 7.5 derives it from the counters alone. The redistribution recovers from 10.8% of the variable cost in the least favorable cell (75–25 split, block at the tail, $\rho = 0.25$) to 92.8% in the most favorable one (25–75

split, block at the head, $\rho = 0.75$). The patterns of Section 7.3 return here: G_{net} grows with ρ in all nine variants, as Γ does, while the absolute Δ falls with ρ in the head cells and grows in the mixed and tail ones, following in every variant the number of evaluations the redistribution removes.

Table 7.3: Latency saving $\Delta = D_{\mathrm{PRE}} - D_{\mathrm{POST}}$ and net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$ over the 27 cells of the matrix, with the $D_0 = 34.30\,\mu$s of the global fit (Figure 7.1). Same runs as Table 7.2; the session offset is common to the two states and cancels in the difference. Entries read $\Delta\,[\mu\mathrm{s}]$ ($G_{net}\,[\%]$).

Position of net2 block	**Split (net1–net2)**	$\rho = 0.25$	$\rho = 0.50$	$\rho = 0.75$
head	25–75	27.61 (86.1)	23.81 (89.1)	20.22 (92.8)
	50–50	18.38 (68.8)	16.36 (75.3)	13.97 (82.6)
	75–25	9.28 (43.5)	8.41 (52.4)	7.82 (66.0)
mixed	25–75	17.46 (80.2)	18.80 (86.8)	20.01 (92.6)
	50–50	13.18 (61.1)	16.22 (74.8)	18.72 (85.9)
	75–25	9.27 (43.6)	13.43 (63.6)	16.99 (80.4)
tail	25–75	5.24 (43.6)	8.56 (52.1)	12.50 (58.0)
	50–50	3.50 (20.9)	6.34 (29.5)	9.19 (34.3)
	75–25	2.28 (10.8)	3.22 (12.6)	4.79 (15.4)

The strongest result of the campaign is that one line describes all of it. Figure 7.1 plots the 54 runs, 27 PRE and 27 POST, in the plane of D against σ. A single least-squares fit over all the points returns an intercept $D_0 = 34.30\,\mu$s and a slope $c = 73.9\,ns$ per rule, with $R^2 = 0.998$: both states of the firewall and all nine rulesets sit on the same line. The redistribution does not change the fixed cost of the path, nor the price of a rule; it only changes how many rules each packet evaluates, which is exactly the claim of the model of Chapter 3. The slope is 6% above the $69.5\,ns$ per rule that the synthetic sweep of Section 6.3 returned in a different session, on rules with simpler match fields: the price of an evaluation belongs to the machine, the ruleset, and the traffic together, as Section 6.4 already noted, and the two estimates agree as closely as that caveat allows. The intercept, as Section 6.5 prescribes, belongs to the sessions and the rate of the matrix, not to the bench.

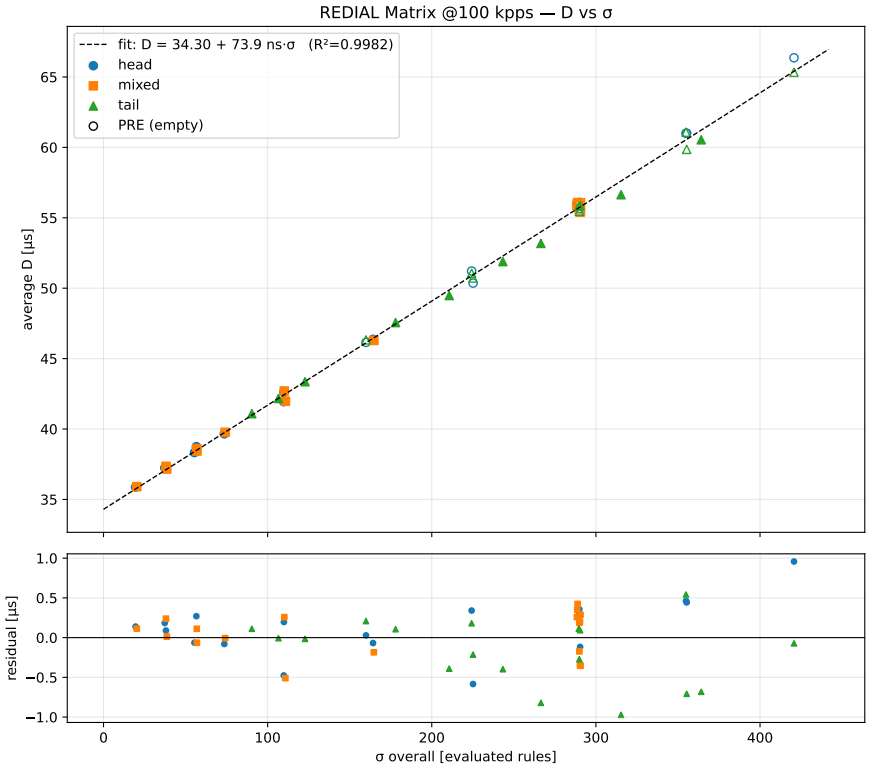


Figure 7.1: Mean delay against mean rules evaluated per packet for the 54 runs of the matrix, 27 PRE and 27 POST. A single fit over all the points returns $D_0 = 34.30\,\mu$s and $c = 73.9\,ns$ per rule, with $R^2 = 0.998$: both states of the firewall and all nine rulesets follow the same line.

The fit also turns σ into a predictor. Once the D_0 and c of a session are known, the latency effect of a redistribution can be computed from the iptables counters alone, before any capture is taken. What remains to be checked is that the measured gains agree with the applicable theoretical references and that the per-subnet quantities combine as the model prescribes. That comparison is the next section.

7.5 Validation against the theory: references and blend

The question of Chapter 3 can now be answered with the numbers of the last two sections. The model makes three falsifiable claims about the matrix: each traffic variant must track its selected theoretical reference Γ_M for head, Γ_{av} for interleaved,

and Γ_m for tail; the cost of a mixed-traffic run must be the blend of its per-subnet costs, weighted by the traffic; and the net latency gain at the working point must follow the measured Γ as $1 - 1/\Gamma$. The matrix confirms all three. No single bound is tested against all 27 cells.

The theoretical references first. Table 7.4 compares each cell only with the reference selected for its variant: the nine head cells with Γ_M, the nine mixed cells with Γ_{av}, and the nine tail cells with Γ_m. The nine head cells reproduce Γ_M exactly, as Section 7.3 established; the nine tail cells exceed Γ_m by 0.5% to 17.9%, and the nine mixed cells exceed Γ_{av} by 2.2% to 22.3%. The margin has one sign and clear drivers: it grows with the share of rules that net2 owns, and in the tail cells it shrinks as ρ grows. The model computes these references from three numbers alone: the ruleset size, the size of the offloaded block, and the depth at which the block begins. In the mixed and tail variants the measured gain lies above the selected reference; in the head variant the upper reference is reached exactly.

Table 7.4: Measured Γ against the reference selected for each variant, for all 27 cells. Each entry reports the reference and, in parentheses, the excess of the measured value of Table 7.2 over it. In the head cells the excess is zero at the three-decimal precision of the tables.

Variant (reference)	**Split**	$\rho = 0.25$	$\rho = 0.50$	$\rho = 0.75$
head (Γ_M)	25–75	7.436 (+0.0%)	9.338 (+0.0%)	14.848 (+0.0%)
	50–50	3.229 (+0.0%)	3.934 (+0.0%)	6.012 (+0.0%)
	75–25	1.768 (+0.0%)	2.050 (+0.0%)	2.887 (+0.0%)
mixed (Γ_{av})	25–75	4.182 (+22.3%)	6.177 (+21.9%)	11.837 (+20.8%)
	50–50	2.442 (+7.2%)	3.636 (+7.1%)	7.098 (+7.0%)
	75–25	1.721 (+2.3%)	2.562 (+2.3%)	4.998 (+2.2%)
tail (Γ_m)	25–75	1.501 (+17.9%)	1.919 (+10.1%)	2.266 (+4.4%)
	50–50	1.187 (+6.1%)	1.328 (+3.5%)	1.437 (+1.5%)
	75–25	1.064 (+2.3%)	1.113 (+1.3%)	1.149 (+0.5%)

The blend is the second claim. The model treats a mixed-traffic run as a weighted average of its classes: $\sigma_{tot}(\rho) = (1 - \rho)\,\sigma_{net1} + \rho\,\sigma_{net2}$. The counters verify this on every run of the matrix: the per-class costs of a variant stay put while ρ moves from 0.25 to 0.75, and each total σ is the blend of the two per-class values of its run (the per-class tables are in Section C.1 of Appendix C). The flat PRE cost of the mixed variants, between 288 and 291 rules per packet in all nine cells, is this property seen from the special case in which the two classes cost the same. Through the fit of Section 7.4 the delay inherits the property: $D_{tot}(\rho)$ is the same blend of the per-class delays, which is what allows a single line to carry all 54 runs.

The third claim prices the gain at the working point. The raw saving Δ/D_{PRE}

understates what the redistribution does, because it dilutes the effect with the fixed cost D_0 that no ruleset change can touch. The honest metric is the net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$, the saved fraction of the part of the delay the rules are responsible for. The prediction is not an extra assumption but algebra on the linear model, and the derivation is three steps long: writing both states as $D = D_0 + c \cdot \sigma$, the fixed cost cancels in the difference at the numerator and the per-rule price cancels in the ratio,

$$G_{net} = \frac{D_{\mathrm{PRE}} - D_{\mathrm{POST}}}{D_{\mathrm{PRE}} - D_0} = \frac{c\,(\sigma_{\mathrm{PRE}} - \sigma_{\mathrm{POST}})}{c\,\sigma_{\mathrm{PRE}}} = 1 - \frac{\sigma_{\mathrm{POST}}}{\sigma_{\mathrm{PRE}}} = 1 - \frac{1}{\Gamma}\,. \tag{7.1}$$

The net gain is therefore written by the counters alone. With the well-conditioned $D_0 = 34.30\,\mu$s of the global fit, the prediction holds across the whole matrix within 3 percentage points; the largest absolute difference is about 2.9 points. In the most favorable cell, the 25–75 split with the block at the head at $\rho = 0.75$, the measured net gain is 92.8% against a predicted $1 - 1/14.848 = 93.3\%$; the same agreement, at smaller gains, repeats over the 27 cells.

The three checks return the same verdict: the model describes the firewall it was written for, with agreement against the reference selected for each variant, a blend that the counters verify run by run, and a measured Γ that prices the latency gain. The next section collects what the campaign established and hands its open questions to the chapters that follow.

7.6 Summary of findings

The campaign closes the question it was designed to answer, and it is worth stating plainly what the 54 runs established.

The linear model holds on real rulesets. A single pair (D_0, c) carries the whole matrix, both states of the firewall and all nine rule orders, with $R^2 = 0.998$; the price of a rule at the operating point is 73.9 *ns*.

The variant-specific references behave as the model predicts. The nine head cells attain Γ_M exactly, the nine mixed cells approach Γ_{av} from above, and the nine tail cells approach Γ_m from above. These are three separate comparisons; no single bound is applied to all 27 cells.

The blend holds. The per-class costs of a ruleset do not depend on the traffic mix, so the effect of any mix is a weighted average that the counters predict.

The gains are large, and they are predictable. The same 601-rule policy yields Γ from 1.088 to 14.848 depending only on where the rules sit and where the traffic goes; the redistribution recovers from 10.8% to 92.8% of the delay the rules are responsible for; and Equation 7.1 prices every cell from the iptables counters alone, before a single packet is captured.

Two questions remain that the matrix cannot answer. Every number in this chapter was measured on one x86 machine; whether the model, the variant-specific references, and the blend survive a change of hardware is the subject of Chapter 8. The matrix also stays below the knee. Chapter 9 retains its most favorable logical configuration, the 25–75 split with net2 at the head and $\rho_2 = 0.75$, while scaling the ruleset and offered rate until the PRE firewall operates beyond the knee.

Chapter 8

Generalizability: porting to embedded hardware

Every number in Chapter 7 was measured on one machine. The linear model, the variant-specific theory comparison, and the blend all held, but they held on an x86 workstation whose processor evaluates a rule in 73.9 ns and whose network path costs 34.30 μs before the first rule is touched. The validation therefore leaves one question open: how much of it belongs to REDIAL, and how much to the machine.

The model itself says where the platform may and may not enter. σ and Γ are computed from the iptables counters, and the counters depend only on the ruleset and on the packets that cross it: any platform that runs netfilter, fed the same rulesets and the same traffic vectors, must return the same σ and the same Γ to the last decimal. What the platform owns is the price of the work, the pair (D_0, c) of the linear model. If the model is right, changing the hardware moves those two numbers and nothing else: the same variant-specific theoretical references are still tracked, the blend still holds, and the gain of the redistribution is still written by the counters.

This chapter tests that prediction at the opposite end of the hardware spectrum. The firewall leaves the Core i9 workstation and moves to a Raspberry Pi 5 that forwards through two USB Ethernet adapters: an embedded board of the class that guards industrial conduits far more often than a server does, with a network path that could hardly be more different from the PCIe ports of TS-P360. The 27-cell matrix of Chapter 7 is then repeated there in full, with the same rulesets, the same traffic vectors, and the same measurement chain on TS-P5. Section 8.1 describes the platform and what the porting did and did not touch; Section 8.2 characterizes the new firewall as Chapter 6 characterized the old one, and fixes its operating rate; Section 8.3 presents the matrix; Section 8.4 separates what the platform changed from what it could not change, and closes with the verdict.

8.1 Porting the testbed

The porting replaces exactly one block of the bench: the firewall host. TS-P360 leaves the measured path and a Raspberry Pi 5 takes its place, a single-board computer with a quad-core ARM Cortex-A76 at 2.4 *GHz*, running Ubuntu 25.10 with kernel 6.17 for the aarch64 architecture. The board joins the management network at 192.168.100.104 and is driven over SSH like every other host. As a firewall platform it sits at the opposite end of the spectrum from the i9-12900K of TS-P360: a different instruction set, a fraction of the clock rate, and a network path that never touches PCIe.

That network path is the real novelty. The Pi has one native Gigabit Ethernet port, and the bench reserves it for the management network, so orchestration and result collection stay off the measured path as they always have. The test path runs through two USB Ethernet adapters instead, one for the ingress of FW1 and one for the egress: Eminent EM1010 units, 10/100 *Mbps* devices that `lsusb` identifies as Realtek RTL8152 controllers (`0bda:8152`), served by the in-kernel `r8152` driver. The two adapters are attached to two separate USB buses of the board, verified with `lsusb -t`, so ingress and egress never contend for the bandwidth of a single bus. The cabling of Figure 4.1 is otherwise unchanged: the forward output of the TAP feeds the ingress adapter, and the egress adapter is cabled directly to the second capture port.

Nothing on the measurement side moves. The sender, the TAP, and both capture ports stay on TS-P5; every timestamp is still taken in hardware by the PHCs of the I350; and the session offset of Section 6.1 remains what it was, a property of the two capture ports that does not know which firewall the packets crossed. This is the part of the methodology that pays off most in the porting: the Pi is never asked to timestamp anything. It could not do it well even if asked, because its USB host schedules transfers in 125 μs microframes, a quantization three orders of magnitude above the per-rule effects of interest; a methodology that relied on the clock of the device under test would not survive this platform. The differential chain does not care what sits between the TAP and the second capture port.

The software of the experiments crosses the porting unchanged. The rulesets are the same 601-rule sets of Chapter 7, produced by the same generator with the same seeds; the traffic vectors are the same seed-fixed sequences; the PRE/POST campaign automation and the REDIAL orchestrator drive the Pi exactly as they drove TS-P360; and the σ collector reads the iptables counters over the management network, since netfilter exposes the same counters on aarch64 as on x86. The firewall configuration of the `FW` namespace is reproduced on the two adapters: FORWARD chain with DROP policy, forwarding enabled, the reverse-path filter relaxed on the test interfaces, the static route that absorbs the second destination subnet, and static neighbor entries for the test addresses. The system configuration

follows Section 4.4 where the platform allows it: `performance` governor, wireless interfaces down, and USB autosuspend disabled, so the host controller never idles between packets.

One parameter cannot cross the porting unchanged: the rate. A platform that forwards through 100 *Mbps* adapters and evaluates rules on a Cortex-A76 does not hold the operating point chosen in Section 6.6, and offering it 100 *kpps* would only measure a permanently full queue. Where its operating point must sit is a question for measurement, and the next section answers it with the same procedure that answered it for TS-P360.

8.2 Characterizing the new platform

A new firewall is a new instrument, and it earns trust the same way the old one did. Before any REDIAL comparison, the Pi receives the characterization of Chapter 6 in miniature: the empty firewall for the fixed cost, the pass-all sweep for the per-rule cost, and the same sweep driven across the rates to locate the knee. Reusing the method is not only economy. Since the procedure is the same, the numbers it returns are comparable one to one with those of TS-P360, and the comparison of Section 8.4 rests on measurements taken the same way on both platforms. The only adaptation is the range: the sweep drives 21 positions of h across the 1001-rule ruleset at eight rates from 100 *pps* to 40 *kpps*, two runs per cell, 336 runs in all.

The fixed cost alone announces a different platform. The empty FORWARD chain, which cost tens of microseconds on TS-P360, costs 420.9 μs per packet on the Pi at 5 *kpps* with the default governor; pinning the CPU with the `performance` governor lowers it to about 414 μs, a 1.5% change. That number is the first diagnostic of the chapter: if the fixed cost lived in the CPU, pinning the frequency would move it visibly, and since it barely moves, D_0 lives in the USB path, in the transfers between the host controller and the two adapters. Its scale, roughly twelve times the 34.30 μs intercept of the x86 matrix fit, prices the difference between a PCIe NIC and a 100 *Mbps* adapter on USB.

The pass-all sweep then prices a rule. At 2 *kpps* the fit over the positions below saturation returns $D_0 = 367.7\,\mu$s and $c = 125.6\,ns$ per rule; at 5 *kpps* it returns $D_0 = 429.0\,\mu$s and $c = 122.7\,ns$ per rule. The per-rule cost of the platform is therefore 123 to 126 *ns*, about 1.8 times the 69.5 *ns* of TS-P360: the Cortex-A76 scans the same chain the Core i9 machine scans, only slower. Not every rate returns this number. At 500 and 1000 *pps* the fitted slope drops to 54.8 *ns* per rule, and that value is an artifact, not a price: at those rates the delay moves with the transfer schedule of the USB host more than with the scan, part of the per-rule cost disappears under that quantization, and a fit taken there underestimates c. No estimate from that region is used in this thesis.

The fixed cost also behaves differently with the rate. On TS-P360, D_0 fell as the rate grew and settled on a floor once NAPI kept the receive path in polling (Section 6.5). On the Pi the curve is U-shaped (Figure 8.1): 599 μs at 500 to 1000 *pps*, a minimum of 368 μs at 2 *kpps*, and a rise beyond it. The left flank is the same wake-up and interrupt economics measured on the x86; the right flank is new, and it belongs to the USB path, whose polling and batching grow more expensive as the offered rate grows. The minimum of the U is the natural operating region of the platform.

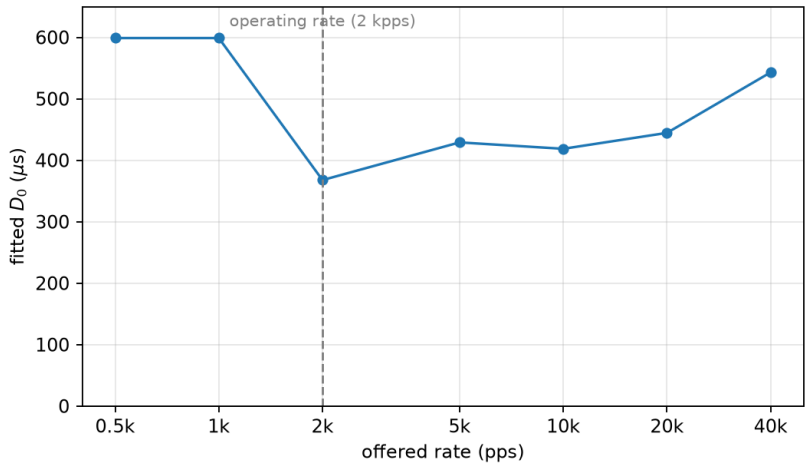


Figure 8.1: Fitted fixed cost D_0 of the Raspberry Pi 5 firewall as a function of the offered rate (pass-all sweep over 21 positions of h, two runs per cell). The curve is U-shaped: at low rates every packet pays the full USB polling cycle, batching amortizes it down to the minimum of 367.7 μs at 2 *kpps*, the operating rate of the matrix (dashed line), and queueing inflates it again beyond 10 *kpps*.

The knee states how much smaller this machine is. The deepest position of the sweep, $h = 1001$, leaves the linear branch between 10 and 20 *kpps*, and at 40 *kpps* every position beyond $h \approx 600$ is saturated; the two readings place the boundary between 1.0 and 2.4×10^7 evaluations per second. The shape found in Section 6.4 generalizes, a boundary at constant work rate, but the capacity behind it is about five times below the 1.15×10^8 of TS-P360.

The operating rate follows from these numbers as 100 *kpps* followed from Chapter 6. The matrix runs on the Pi at 2 *kpps*. The rate sits at the minimum of the D_0 curve, so both flanks of the U are avoided; its period is an integer 500 000 *ns*, so the sender delivers it exactly; and the margin on the knee is wide, since the worst cell of the matrix evaluates about 420 rules per packet (Chapter 7) and loads the firewall with 8.4×10^5 evaluations per second, more than an order of

magnitude below even the pessimistic reading of the boundary. A 60-second run carries 120 000 packets, far fewer than the six million of the x86 campaigns but enough to hold the standard error of the mean below one microsecond, well under the PRE/POST differences the matrix produces.

8.3 Results: the matrix on the Pi

The matrix of Chapter 7 runs on the Pi as it ran on TS-P360: the same 27 cells, three traffic mixes, three rule splits, and three positions of h, each cell measured PRE and POST in the same session, 54 runs in all at 2 *kpps* with 120 000 packets per run. Every run passed the acceptance checks that survive the platform: zero loss, zero leak, and a TAP capture that reconciles with the RX capture packet by packet. The dispersion thresholds of the x86 protocol do not survive it: the per-packet standard deviation sits near 245 μs on the Pi, inflated by the 125 μs USB microframe quantization, so no threshold of a few tens of microseconds is meaningful here. What remains meaningful is the mean, which 120 000 packets pin down to better than one microsecond (Section 8.2).

The counters return the strongest result of the chapter. In every one of the 27 cells, σ_{PRE}, σ_{POST}, and therefore Γ are identical to the x86 values of Chapter 7 to the third decimal digit. This is the prediction made in the opening of the chapter, verified at the finest granularity available: the counters are deterministic functions of the ruleset and of the traffic vector, and neither of them changed in the porting. The consequence saves this chapter an entire analysis. The comparison of Γ with the reference selected for each variant, carried out cell by cell in Section 7.5, transfers to the Pi verbatim: the head cells still attain Γ_M exactly, the mixed cells still track Γ_{av}, and the tail cells still sit just above Γ_m. The redistribution does not see the platform.

The latency sees it instead. Table 8.1 collects, for the 27 cells, the PRE/POST difference Δ and the net gain G_{net} of Equation 7.1, computed with the $D_0 = 367.7\,\mu$s of the sweep fit. Every Δ is positive, and every cell exceeds its x86 counterpart, as a platform with a higher per-rule cost must return at equal $\Delta\sigma$. The gain this thesis quotes is G_{net}: the redistribution recovers from 10.3% of the variable cost in the cheapest corner of the matrix (75–25 split, tail, $\rho = 0.25$) to 91.2% in the most favorable one (25–75 split, head, $\rho = 0.75$), matching the x86 range within a few points.

Table 8.1: PRE/POST latency difference Δ and net gain $G_{net} = \Delta/(D_{\mathrm{PRE}} - D_0)$ on the Raspberry Pi at 2 *kpps*, with the $D_0 = 367.7\,\mu$s of the sweep fit, for the 27 cells of the matrix. Entries read $\Delta\,[\mu\mathrm{s}]$ ($G_{net}\,[\%]$). The complete per-run tables are in Section C.2 of Appendix C.

Position	**Split**	$\rho = 0.25$	$\rho = 0.5$	$\rho = 0.75$
head	25–75	37.92 (83.1)	32.88 (83.9)	31.34 (91.2)
	50–50	24.95 (64.9)	23.75 (72.0)	20.11 (74.9)
	75–25	13.77 (40.8)	12.04 (46.0)	11.41 (61.2)
mixed	25–75	25.37 (76.0)	29.03 (85.9)	27.60 (85.6)
	50–50	20.19 (59.1)	23.21 (70.4)	27.45 (84.8)
	75–25	12.13 (38.4)	18.15 (56.0)	22.83 (71.5)
tail	25–75	8.03 (44.7)	13.85 (54.2)	18.24 (56.6)
	50–50	6.43 (23.5)	9.92 (29.4)	12.60 (32.5)
	75–25	3.36 (10.3)	6.10 (15.2)	7.60 (16.3)

The trends inside the table are the x86 trends. In the head group Δ falls as ρ grows, in the mixed and tail groups it grows, for the reason established in Section 7.4: the sign of the trend is written by how $\Delta\sigma$ moves with the traffic mix, not by the hardware. One point breaks the monotonicity, the mixed 25–75 cell at $\rho = 0.75$ (27.60 μs after 29.03), and the break stays within what the dispersion of the platform allows on a difference of microseconds. Read together, the 27 values imply a per-rule price of 102 to 110 *ns*, slightly below the 123 to 126 *ns* of the sweep; the gap is expected for differences of tens of microseconds taken under a 245 μs per-packet dispersion, and both estimates say the same thing at the scale that matters: a rule costs 1.4 to 1.8 times on this platform what it costs on the x86. Figure 8.2 shows the 54 runs against σ: the linear model of Chapter 3 holds on the Pi as it held on TS-P360.

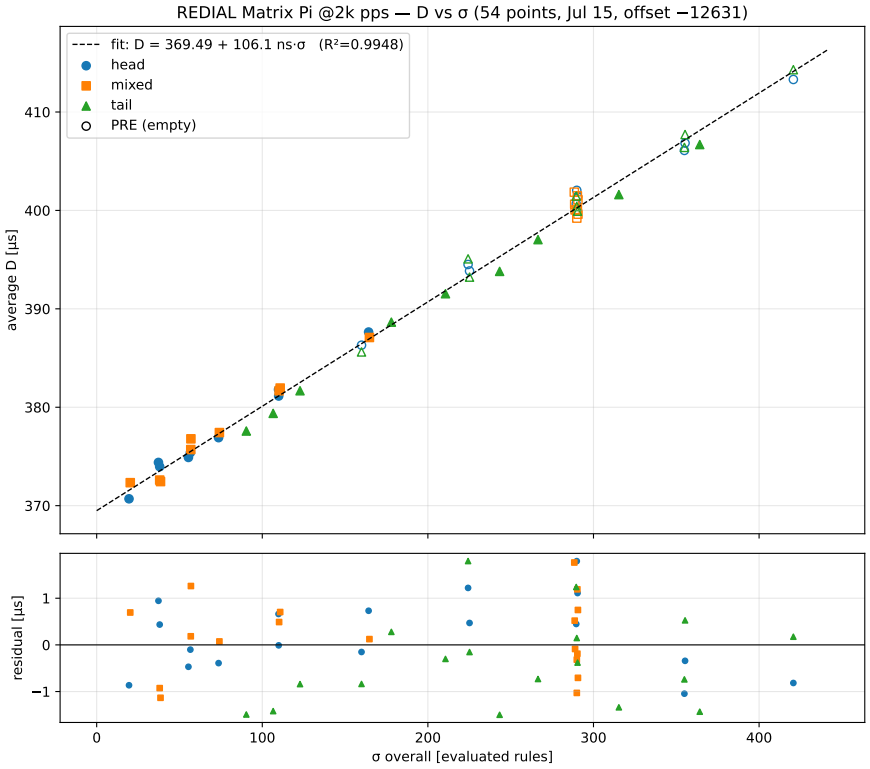


Figure 8.2: Mean delay against σ for the 54 runs of the matrix on the Raspberry Pi at 2 *kpps*. The linear model $D = D_0 + c \cdot \sigma$ holds on this platform as it does on the x86; the slope implied by the runs sits between 102 and 110 *ns* per rule.

8.4 What changes across platforms

Two complete campaigns, taken with the same method on two machines, allow a comparison free of new measurements: every number in Table 8.2 appeared earlier in this thesis, and the table only puts the two columns side by side.

Table 8.2: The two platforms side by side. The matrix-fit parameters use all 54 runs at the respective operating rates. The pass-all sweep estimates and the D_0 used for G_{net} are listed separately.

	TS-P360 (x86)	**Raspberry Pi 5**
Operating rate	100 *kpps*	2 *kpps*
Matrix-fit D_0	34.30 μs	369.49 μs
Matrix-fit c	73.9 *ns*/rule	106.1 *ns*/rule
Matrix-fit R^2	0.9982	0.9948
c, pass-all sweep	69.5 *ns*/rule	122.7–125.6 *ns*/rule
D_0 used for G_{net}	34.30 μs	367.7 μs
Evaluation capacity (evals/s)	1.15×10^8	1.0–2.4×10^7
Δ across the matrix	2.28–27.61 μs	3.36–37.92 μs
G_{net} across the matrix	10.8–92.8%	10.3–91.2%
σ, Γ, variant-specific verdicts	identical	

The invariants are the upper half of the story. σ and Γ did not move by one count over the 6.48 million packets of the campaign; the variant-specific verdicts of Section 7.5 transferred cell by cell; the blend holds because it is a property of the counters; and the latency stayed linear in σ (Figure 8.2). Everything the REDIAL model claims for the redistribution survived a change of instruction set, of NIC technology, and of an operating rate fifty times lower.

What changed is the price list. The fixed cost grew by an order of magnitude and moved from the CPU to the USB path; the per-rule cost roughly doubled; the capacity fell by a factor of five to ten. These numbers carry one practical lesson on how the gain should be quoted. The relative gain $G = \Delta / D_{PRE}$, which reached 42% on the x86 in the best cell, compresses to 0.8 to 9.2% on the Pi even though every Δ is larger: the denominator is dominated by a fixed cost of about 370 μs that no redistribution can touch, so G is a property of the pair platform plus redistribution, not of REDIAL alone. The net gain of Equation 7.1, which removes D_0 before dividing, is the quantity this thesis quotes as the gain, and it repairs the comparison: on the Pi it spans 10.3 to 91.2% against the 10.8 to 92.8% of the x86, cell by cell compatible with $1 - 1/\Gamma$ on both platforms. Expressed as the saved fraction of the variable cost, the benefit of the redistribution does not know the platform either. One caution applies to the Pi values: the denominator $D_{PRE} - D_0$ is 27 to 46 μs, a small difference between two numbers near 400 μs, so an error of one microsecond on the estimate of D_0 moves G_{net} by a few points. The caution widens the error bar; it does not move the verdict.

The verdict of the chapter fits in two sentences. The gain of REDIAL is written by the counters, and the counters do not know the platform: same σ, same Γ, same variant-specific verdicts, on two machines that share nothing below iptables.

The platform sets the price of the work, D_0, c, and above all the capacity, the number that decides how much traffic a device can filter before it starts dropping or delaying: 1.15×10^8 evaluations per second on one machine, between a fifth and a tenth of that on the other. That last number is also where the redistribution has something left to prove. Lowering σ multiplies the packet rate a firewall can sustain at a given capacity, and Chapter 9 measures exactly that, driving the firewall past its knee to see what REDIAL is worth when the traffic is an attack.

Chapter 9

DoS experiment: REDIAL under stress

Chapters 7 and 8 evaluated REDIAL below the forwarding knee, where reducing the number of rule evaluations mainly appears as a latency improvement. This chapter moves beyond that operating region and studies the same mechanism while the firewall is overloaded. The experiment holds the offered load at 800 *kpps* and compares the same traffic before and after REDIAL changes the portion of the policy kept on FW1.

The selected configuration deliberately creates a large separation between the two states. PRE contains 7501 rules and cannot sustain the complete input rate. POST contains 1914 rules and forwards the entire offered load. Over two 60 *s* acquisitions, PRE loses 63.300% of the packets, whereas POST records zero observed loss. The result shows that the processing saved by REDIAL can appear not only as lower delay, but also as additional useful throughput and restored packet delivery under stress.

9.1 Threat scenario and rationale

A denial-of-service attack against a packet filter does not need to violate the filtering policy. A firewall has finite processing capacity and finite queues. Its load depends on both the arrival rate and the number of rules that must be evaluated for each packet. An attacker controls the first factor through the packet rate and can influence the second by selecting traffic that traverses a costly part of the policy. Once the offered work exceeds the sustainable service rate, the input queue remains occupied and the excess traffic is discarded. At that point, attack and legitimate packets can be affected alike.

REDIAL changes this operating point by reducing the policy evaluated on a

congested firewall. If the resulting processing cost moves the firewall below its loss threshold, the same hardware can forward a larger packet rate without changing the filtering decision. The availability benefit should therefore be visible as an increase in delivered packets and a corresponding decrease in packet loss.

The experiment tests this hypothesis at a fixed offered rate. It does not attempt to classify packets as benign or malicious. Instead, it measures the aggregate difference between the packets observed at the TAP before FW1 and the packets observed at RX after it. This choice isolates the forwarding capacity of the firewall from any application-specific interpretation of the traffic.

9.2 Experiment design

9.2.1 Testbed and measurement points

The sender runs on TS-P5. Traffic crosses the FW namespace on TS-P360, where the FORWARD chain uses a default DROP policy. Two capture points delimit the system under test. TAP records the complete stream immediately before the firewall, while RX records the packets delivered after forwarding. The sender is pinned to CPU 9, and the TAP and RX listeners use dedicated cores.

Each acquisition lasts 60 s. The sender offers 800 $kpps$, corresponding to 48 000 000 packets per state. The capture interval is extended to 82 s, including 2 s before transmission and 20 s after it. This margin ensures that the listeners remain active until the sender has completed the configured packet count.

PRE and POST are separate acquisitions. For presentation, the first 60 s of the figures report PRE and the following 60 s report POST on a common time axis. The boundary at $t = 60\,s$ therefore separates the two measured states; it is not an in-flight policy replacement.

9.2.2 Ruleset and traffic distribution

The PRE policy contains 7501 rules. Of these, 1862 belong to net1, 5588 belong to net2, 50 are wildcard rules, and the final rule closes the chain. The generated order places the net2 block before the net1 block. The policy therefore preserves the 25/75 distribution used in the selected REDIAL test case, with net2 representing the dominant destination domain.

The traffic follows the same asymmetric distribution. The sender directs 25% of the packets to net1 and 75% to net2, hence $\rho_2 = 0.75$. The packet vectors are generated from the 7501-rule policy with seed 42 and contain 10 000 templates per network. The same vectors, packet order, CPU affinity, offered rate, and run duration are reused in PRE and POST. The ruleset held by FW1 is the only experimental variable.

REDIAL removes the net2 block from the policy evaluated by FW1 and installs a shortcut for its destination prefix. The POST policy on FW1 consequently contains 1914 rules: the 1862 net1 rules, the 50 wildcard rules, the closing rule, and the REDIAL shortcut. Relative to PRE, this is a reduction of 5587 rules, or 74.483% of the original FW1 policy.

9.2.3 Validity conditions and accounting

A run is accepted only if the sender completes all 48 000 000 packets with zero transmission errors, TAP records the same 48 000 000 packets, and both capture processes report zero packets dropped by the kernel. These conditions hold for the two acquisitions used below.

The NIC byte counters provide an independent consistency check. In PRE, TAP records a delta of 3 072 000 000 bytes and RX records 1 127 437 760 bytes. These values correspond exactly to 48 000 000 and 17 616 215 packets in the NIC accounting. In POST, both TAP and RX record 3 072 000 000 bytes, corresponding to the complete 48-million-packet stream. The agreement between sender totals, packet captures, kernel-drop counters, and NIC counters attributes the missing PRE packets to the forwarding path rather than to the measurement system.

9.3 Results

9.3.1 Aggregate packet accounting

Table 9.1 reports the aggregate result. PRE receives all 48 000 000 offered packets but delivers only 17 616 215. The remaining 30 383 785 packets are lost, corresponding to 63.300% of the input. Its mean useful throughput is therefore 293.604 *kpps*.

POST receives the same 48 000 000 packets and delivers all of them. The observed loss is 0.000%, and the useful throughput reaches the complete 800.000 *kpps* offered load.

Table 9.1: DoS stress campaign at 800 *kpps*. PRE and POST are separate 60 *s* acquisitions with 48 million offered packets each.

State	Rules on FW1	Offered	Delivered	Lost	Loss	Throughput
PRE	7501	48 000 000	17 616 215	30 383 785	63.300 %	293.604 *kpps*
POST	1914	48 000 000	48 000 000	0	0.000 %	800.000 *kpps*

REDIAL therefore restores 30 383 785 packet deliveries under an unchanged offered load. The delivered fraction rises from 36.700% to 100%. The mean

throughput increases by 506.396 *kpps*, and the ratio between POST and PRE throughput is

$$\frac{800.000}{293.604} = 2.725. \tag{9.1}$$

Equivalently, POST provides a 172.476% throughput increase relative to PRE. Packet loss decreases by 63.300 percentage points.

9.3.2 Throughput over time

Figure 9.1 shows the offered and delivered packet rates in 1 *s* windows. The offered rate remains fixed at 800 *kpps* in both phases. During PRE, the delivered rate starts near 330 *kpps*, decreases gradually, and then settles near 270 *kpps* in the latter part of the acquisition. Its average over the complete phase is 293.604 *kpps*. This evolution is consistent with a firewall that remains persistently above its sustainable service rate under the 7501-rule policy.

At the beginning of the POST phase, the delivered rate reaches the offered rate and remains at 800 *kpps*. The two curves overlap because every packet captured at TAP is also captured at RX. The numeric annotations in the figure report the two mean rates and the 2.725-fold throughput gain.

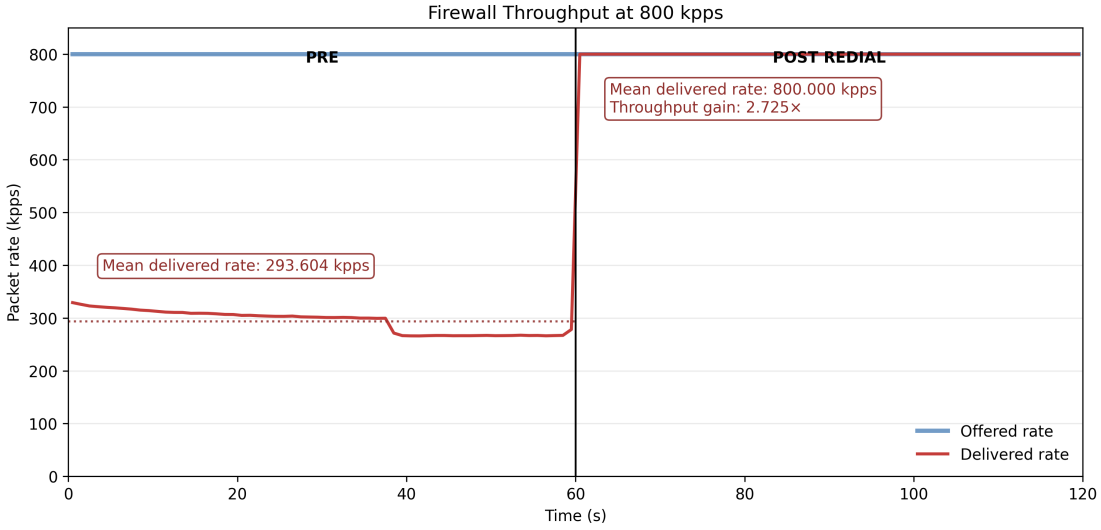


Figure 9.1: Offered and delivered packet rates during the PRE and POST phases. The offered load remains fixed at 800 *kpps*; mean delivered throughput increases from 293.604 *kpps* in PRE to 800.000 *kpps* in POST.

9.3.3 Packet loss over time

The same transition is visible in Figure 9.2. PRE loss starts close to 59% and increases as the delivered rate decreases, reaching approximately 67% in the latter portion of the phase. The aggregate mean is 63.300%. The time profile confirms that the headline value is not caused by a short capture failure: loss persists throughout the complete PRE acquisition while the TAP and kernel-drop checks remain valid.

POST loss is zero in every aggregate accounting result and zero over the complete 48-million-packet acquisition. The figure therefore falls from the PRE regime to 0.000% in POST. The two annotations report the aggregate mean loss of each state: 63.300% in PRE and 0.000% in POST.

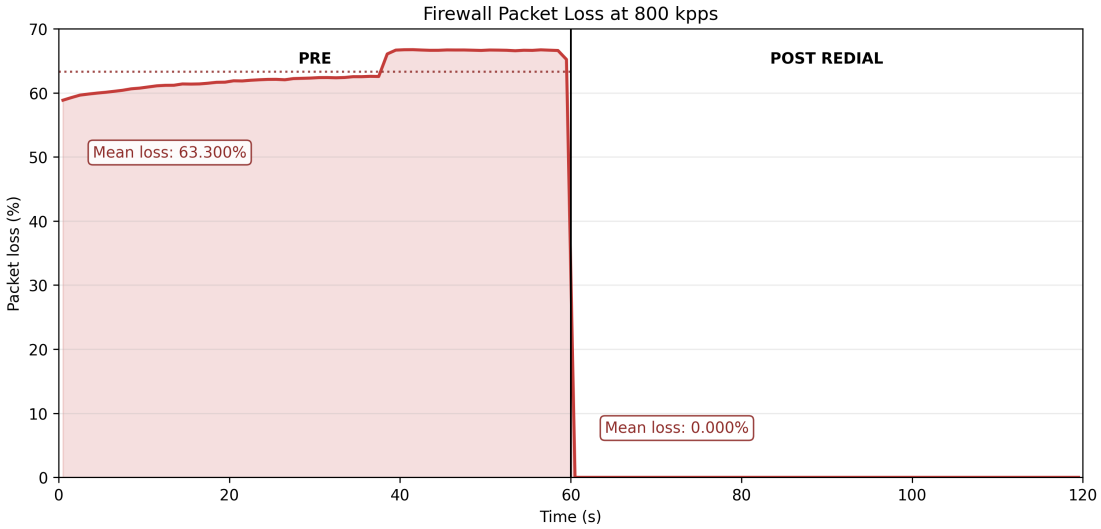


Figure 9.2: Packet loss during the PRE and POST phases at an offered rate of 800 *kpps*. Mean loss decreases from 63.300% in PRE to zero observed loss in POST.

9.4 Interpretation

The two states receive the same packet count, use the same traffic vectors, and run on the same hardware. Their difference is the policy that FW1 must evaluate. With 7501 rules, the firewall delivers only 36.700% of the input. After REDIAL reduces the local policy to 1914 rules and inserts the net2 shortcut, FW1 delivers the complete stream.

The 25/75 configuration amplifies the benefit in a controlled way. Net2 accounts for both 75% of the generated policy rules and 75% of the offered traffic. It is also the domain handled by the REDIAL shortcut. Consequently, the dominant packet

class no longer crosses the original net2 rule block on FW1, while the policy for net1 remains locally available. The reduction in local rule evaluation moves the POST firewall from a deeply overloaded state to a lossless state at the tested rate.

This result extends the interpretation of REDIAL beyond latency. Below the knee, avoiding rule evaluations saves forwarding time. Above the knee, the same saving becomes additional service capacity. Here it restores more than 30 million packet deliveries in 60 s and increases useful throughput by a factor of 2.725.

9.5 Scope and limits

The result does not make the POST firewall immune to denial of service. Zero loss is observed at 800 *kpps* for this ruleset, packet size, traffic mix, and hardware platform. A higher offered rate or a different remaining policy could move POST above its loss threshold again. The experiment does not locate that new threshold.

The campaign uses generated traffic, one rule ordering, one asymmetric traffic distribution, and one acquisition per state. It therefore provides a controlled case study rather than a statistical characterization of every possible workload. The traffic is evaluated in aggregate and does not contain a separately labelled population of legitimate application flows. The result quantifies forwarding availability, not application-level impact.

Finally, the figures place two separate 60 s acquisitions on a common 120 s axis. They compare stable PRE and POST regimes but do not measure the transient duration of an in-flight REDIAL application. That would require a separate run in which the policy is replaced while one continuous sender and capture session remains active.

Within these limits, the conclusion is precise. At the same 800 *kpps* offered load, REDIAL reduces the policy on FW1 from 7501 to 1914 rules, raises useful throughput from 293.604 to 800.000 *kpps*, and changes packet loss from 63.300% to zero observed loss. The experiment demonstrates that reducing rule evaluations can restore complete packet delivery even when the original firewall is deeply overloaded.

Chapter 10

Discussion

Chapters 7, 8 and 9 produced three answers to three different questions: the size of the gain at the working point, its portability across hardware, and its effect when the load exceeds the capacity. Each answer came with its own numbers and its own caveats. This chapter puts the answers side by side, weighs what the numbers support against what they do not, and positions the results with respect to the works this thesis builds on.

10.1 How much is REDIAL worth in practice

At the working point, the answer of Chapter 7 is a fraction, not a number of microseconds: the net gain G_{net} of Equation 7.1, the share of the variable cost of the firewall, the part of its delay the rules are responsible for, that the redistribution recovers. An absolute saving says little without the delay it is subtracted from, and the raw ratio to the PRE delay dilutes the effect with the fixed cost D_0 of the platform, which no redistribution can touch; G_{net} avoids both traps. Across the 27 cells of the matrix at 100 *kpps*, the redistribution recovers 43.5 to 92.8% of the variable cost when the traffic matches at the head of its partition, 43.6 to 92.6% on the mixed profile, and 10.8 to 58.0% when it matches at the tail. The rules cost each packet 12 to 32 μs across the matrix, so recovering half of that cost returns to a control loop with a millisecond cycle a visible share of the per-hop budget that Chapter 1 placed at tens of microseconds.

The per-rule price makes the number predictive rather than merely descriptive. The whole matrix collapses onto one line, $\Delta = c \cdot \sigma_{saved}$ with $c = 73.9\,ns$ per rule on the x86 firewall ($R^2 = 0.998$), so the latency gain of a planned redistribution can be estimated before applying it: count the rules a packet will no longer evaluate, multiply by the c of the platform. The price of a rule is a property of the hardware,

not of REDIAL: on the Raspberry Pi the matrix implies a per-rule price of 102–110 ns, so the same Γ buys more microseconds where the hardware is slower, and the redistribution helps most exactly where the capacity is scarcest.

Chapter 8 shows that the structural gain Γ does not depend on the platform. On the Raspberry Pi the 27 cells return a measured G_{net} of 10.3 to 91.2%, close to, but not numerically identical to, the 10.8 to 92.8% of the x86. The counter-based prediction $G_{net,pred} = 1 - 1/\Gamma$ is identical on the two platforms because Γ comes from the iptables counters and depends only on the ruleset and traffic. The latency-derived G_{net} remains close but differs slightly because it is inferred from measured delays and an estimated D_0. What the Pi does not share with the x86 is the fixed cost D_0: the USB path adds about 370 μs that no redistribution can touch, and a gain quoted as Δ/D_{PRE} ranges from 0.8 to 9.2% on the Pi, not because REDIAL works less, but because the denominator grew. Removing D_0 before dividing is what keeps the number a property of the redistribution rather than of the network path upstream of it.

The placement-specific references are the operational lesson of the matrix. The same ruleset and the same redistribution returned every value of Γ between 1.09 and 14.85: the position of the matching rules, the split of the ruleset, and the traffic balance ρ decide where in $[\Gamma_m, \Gamma_M]$ a deployment lands, and each traffic variant tracks its applicable reference (head $\to \Gamma_M$, mixed $\to \Gamma_{av}$, tail $\to \Gamma_m$) in every cell. The practical advice follows. The rulesets alone do not determine the benefit; an operator who wants a forecast needs a traffic profile. With one, each measured cell can be compared with the reference selected for its variant: Γ_M for head, Γ_{av} for mixed, and Γ_m for tail. No single bound is applied to the whole matrix.

The working point is only half of the value. Chapter 9 reads the same mechanism in a different unit. The stress campaign retains the most favorable logical configuration of the matrix, a 25–75 rule split with the net2 block at the head and $\rho_2 = 0.75$, but scales the PRE policy to 7501 rules and offers 800 *kpps* for 60 *s*. PRE delivers 17 616 215 of 48 000 000 packets, for a mean throughput of 293.604 *kpps* and 63.300% loss. After REDIAL reduces the policy kept on FW1 to 1914 rules, POST delivers all 48 000 000 packets at 800.000 *kpps*, with zero observed loss. Useful throughput therefore increases by a factor of 2.725. At this controlled operating point, the reduction in local rule evaluation appears as additional forwarding capacity, not only as lower latency.

10.2 Comparison with the reference works

REDIAL was published with a formal model, a redistribution algorithm, and the bounds $[\Gamma_m, \Gamma_M]$, backed by simulations and by one small-scale test-bed demonstration [1]. That demonstration is the single experimental point the paper offers,

and it is the one this thesis can be compared with directly: two cascaded iptables firewalls, 300 rules on the upstream one and 100 on the downstream one, and at $\rho = 75\%$ the transformation moves the measured latency of the upstream firewall from 1.86 μs to 0.32 μs while σ falls from 217.34 to 37.73 rules per packet — a measured $\Gamma \approx 5.76$, in agreement with both the latency ratio $l_1/l_2 \approx 5.81$ and the 5.25 predicted by the paper's own simulation. This point sits comfortably inside the interval mapped by Chapter 7: at the same traffic share the head cells of this thesis return $\Gamma = 6.012$ for the 50–50 split and $\Gamma = 14.848$ for the 25–75 one, and across the matrix the measured gain spans 1.088 to 14.848 depending only on rule placement and traffic mix. The comparison also shows what a grid adds to a single point: the paper's number says REDIAL works on that ruleset, while the matrix maps how the three controlled placements land between Γ_m and Γ_M. Two method differences are worth recording. The paper quotes a raw latency ratio, which folds the fixed traversal cost into the gain; the metric of this thesis, G_{net}, removes D_0 before dividing (Equation 7.1). And the paper's trace exposes the transient cost of the live ruleset swap, a peak of 2.77 *ms* over a 6.11 *ms* reconfiguration interval, which the campaigns of this thesis keep out of the data by applying the redistribution between the runs of a pair, not during a run. On the side of the combinatorics the verification is clean: σ measured from the iptables counters behaves exactly as the model prescribes, the three traffic variants tracked their selected references as designed: Γ_M for head, Γ_{av} for mixed, and Γ_m for tail. What the theory could not say, and the testbed adds, is the exchange rate into time: a rule not evaluated is worth 73.9 *ns* on the x86 firewall and 102–110 *ns* on the Pi, and, net of the fixed cost, the redistribution recovers 10.8 to 92.8% of the delay the rules are responsible for, depending on the cell. The nuance is in the width of the interval: with Γ ranging from 1.09 to 14.85 on one and the same ruleset, the theoretical bounds are honest but loose, and a forecast needs the traffic profile that selects the point between them.

The latency model descends from the firewall characterization of [3], built on a finite-buffer queue whose service time grows with the position of the matched rule. Both of its qualitative predictions survived the transfer to iptables-in-namespace: the latency is linear in the rules evaluated, with $R^2 = 0.9982$ on x86 and $R^2 = 0.9948$ on the Pi, and the system has two operating regions, with an almost deterministic delay above the boundary, where the measurement reads the drain time of a full buffer rather than the cost of forwarding. Three adaptations were needed. The fixed term D_0 is not a constant of the device: it moves with the offered rate through the NAPI transition and the idle states of the CPU, and it moves between sessions, which is why every comparison in this work is paired at fixed rate within one session. The boundary is not a packet rate but a work rate, constant in evaluations per second, so the same firewall that forwards 800 *kpps* with an empty chain starts dropping at 450 *kpps* with 500 rules loaded. And the capacity depends on the

traffic shape: multi-flow traffic spreads over the receive queues and the cores of the machine, which the single-flow probe of the characterization does not exercise (Chapter 9).

The measurement methodology of [2] promised nanosecond-scale delay measurements from commodity hardware, and on this bench it kept the promise: the offset calibration repeats within 2.6 *ns*, the same order as the 8 *ns* tick of the PHC, and the method, pointed at an industrial switch, recovered the serialization rate of the line with a nominal relative deviation of 0.075%, with residuals within ± 15 *ns*. The adaptation to the FW1-only scenario collapsed sender and collector onto one host, which removed the need for clock synchronization between machines but introduced a constant offset between the two capture ports that the original setup does not have. The experience distilled one rule the paper does not need to state: the offset survives a session, not a power cycle. Absolute delays are comparable only within a session, differences are immune, and the historical absolute values re-read across sessions in this work proved unreliable until recomputed against their own calibration. The precision is real, but it is rented, and the rent is a calibration at every restart of the bench.

10.3 Limits of the testbed and of the measurements

The claims of this thesis are only as good as the bench that produced them, so this section lists what the bench cannot do, from the narrowest limit to the broadest.

The narrowest limits belong to the measurement itself. The delays of the firewall live with a dispersion of 20 to 30% in the linear regime, fed by interrupts, scheduling, and the thermal state of the machine; the campaigns absorb it with windowed statistics and the median, and the residual dispersion is reported with every result rather than hidden. The clock offset adds a discipline rather than an uncertainty: absolute delays are comparable only within a calibration session. Neither limit touches the conclusions, because every quantity this thesis rests on is a paired PRE/POST difference taken at fixed rate within one session, and σ is read exactly from the counters, immune to both effects.

The second limit is the operating region. Everything Chapters 7 and 8 claim holds below the knee, by design: the working point keeps the worst cell below its measured boundary with a margin of at least 1.5, and the rate invariance of Δ, c, and Γ was verified there. Near and above the knee the fit is no longer trustworthy, because queueing delay compresses the measured c; the thesis does not extrapolate across the boundary, and Chapter 9, which operates beyond it on purpose, switches the observable from the fit to the packet loss for exactly this reason. The limit is therefore a fence, not a crack: the claims stop where the linear regime stops.

The broadest limits belong to the scenario. The chain is emulated FW1-only: the firewall that benefits from REDIAL is measured, the downstream firewalls that receive its rules are not, so the price they pay is asserted by the model, not by this bench. The traffic is synthetic, constant in rate and packet size, with no mix of legitimate and background flows; the rulesets are generated families, with 601 rules in the matrix and a 7501-rule PRE policy reduced to 1914 rules in the DoS campaign, not policies taken from a production network; each platform is one machine, and the DoS attack rate is one value, the ceiling of the sender. These limits bound the generality of the numbers, not their correctness: within the measured envelope the results are reconciled to the packet, and the porting campaign showed that the part of the outcome REDIAL actually promises, σ and Γ, does not depend on the machine at all.

Each scenario limit marks a direction of future work: a materialized multi-firewall chain to measure the downstream price, traffic profiles and rulesets taken from a real network, and a sender fast enough to find the loss threshold of the optimized firewall. Chapter 11 collects them.

Chapter 11

Conclusions and future work

Chapter 1 posed a question the REDIAL paper leaves open: how large is the gain the redistribution of rules delivers on a real firewall, and does it agree with the theoretical reference selected for each traffic variant. Built on the FW1-only emulation of Chapter 3 and measured across a full experimental campaign, the answer is yes: the linear model $D = D_0 + c \cdot \sigma$ holds on real hardware, the measured gains agree with their variant-specific references, and the gain survives both a change of platform and a stress test the REDIAL paper never considers. The goal stated in Section 1.2 is achieved.

11.1 Summary of results

The five contributions promised in Section 1.3 are each closed by a result.

The testbed of Chapter 4 is not merely built but verified: Chapter 6 calibrated its clocks, checked the differential method against a switch whose measured serialization slope had a nominal relative deviation of 0.075% from theory, and located the boundaries, in rate and in ruleset depth, within which the rest of the thesis operates. The toolchain of Chapter 5 turned that instrument into 54 runs of the core campaign, and dozens more for the porting and DoS studies, without a manual step between generation and result.

The linear model of Chapter 3 is not an approximation that degrades with scale: the full 27-cell matrix of Chapter 7 fits a single global relationship, $D = 34.30\,\mu s + 73.9\,ns \cdot \sigma$, with $R^2 = 0.998$. Within that same matrix each measured gain Γ was compared with the reference selected for its variant. The head cells attained Γ_M, the mixed positions landed close to Γ_{av}, and the tail cells remained just above Γ_m, exactly the map Section 3.4 predicted. No single bound was applied to all 27 cells. The blend properties $\sigma_{tot}(\rho)$ and $D_{tot}(\rho)$ held on the mixed-traffic runs, not only at the two single-subnet extremes.

The gain does not belong to one machine. Repeating the 27-condition matrix on a Raspberry Pi 5 (Chapter 8) returned the same σ and the same Γ, and preserved the linearity of the model; what changed with the platform was the price of a rule, $c_{Pi} \approx 102$–$110\,ns$ against roughly $74\,ns$ on the x86 bench, and the fixed cost D_0. The gain REDIAL predicts is a property of the ruleset and the traffic, not of the CPU that evaluates it; c is.

Finally, the DoS experiment of Chapter 9 showed that this gain is not only a matter of microseconds. The campaign retains the most favorable logical configuration of the matrix, a 25–75 rule split with the net2 block at the head and $\rho_2 = 0.75$, while scaling the PRE policy to 7501 rules. At an offered rate of 800 *kpps* for 60 *s*, PRE delivers 17 616 215 of 48 000 000 packets, for 293.604 *kpps* of mean throughput and 63.300% loss. After REDIAL reduces the policy kept on FW1 to 1914 rules, POST delivers all 48 000 000 packets at 800.000 *kpps*, with zero observed loss. The useful throughput ratio is 2.725. The reduction in rule-evaluation work therefore becomes additional forwarding capacity under stress.

11.2 Future work

The most direct extension is the one this thesis deliberately left out: a full multi-firewall scenario, with FW1 through FWn measured jointly rather than FW1 in isolation. The FW1-only emulation of Chapter 3 captures the largest share of the gain by design, but only an end-to-end chain can verify the full model the REDIAL paper states, including how the gains at each hop compose. Two questions are specific to that setting and cannot be answered from the FW1-only results of this thesis. The first is the cost on the downstream side: REDIAL relieves FW1 by handing rules to the firewalls behind it, and a topology with at least one downstream firewall would measure how much its own σ and D grow in exchange for FW1's gain, rather than assuming the trade is favorable. The second is scale: with more than two firewalls sharing a larger ruleset across more subnets, the redistribution problem itself gets harder, and it is an open question whether the same linear model and the same bounds $[\Gamma_m, \Gamma_M]$ still describe every firewall in the chain, or only the upstream one studied here.

A second extension broadens the ruleset families beyond the orderings and splits of Chapter 7. The matrix used three positions of h and three splits, chosen to map the placement-specific theoretical references; a realistic ruleset extracted from a production network would test whether the same linear model and the same bounds hold when the matching position is not a controlled parameter but whatever a real policy happens to produce.

A third line follows from the porting study of Chapter 8. The Raspberry Pi 5 preserved the gain and the linearity of the model; repeating the porting exercise

on other embedded platforms and other NIC paths would say more about which parts of the characterization of Chapter 6 are properties of iptables and which are properties of one machine.

The last is the DoS study of Chapter 9, which this thesis treats as a demonstration rather than a full characterization. One ruleset, one traffic profile, and one offered rate are enough to show zero observed loss after REDIAL at the tested 800 *kpps* point, but not to establish general DoS protection or the maximum sustainable POST rate. A deeper study should sweep multiple rulesets, packet sizes, traffic profiles, and rate regions, including the POST loss threshold.

The promise of a formal model is only as good as its experimental confirmation. This thesis provides that confirmation for REDIAL, on real hardware, from the firewall's own counters to the packets it forwards, end to end.

Appendix A

Toolchain: usage and source code

This appendix collects the software developed for this thesis, one section per tool. Each section reports the command lines used in the campaigns and the full listing of the final version; earlier iterations, where they existed, are discussed in Chapter 5 and are not listed here. With these listings every artifact of the experimental work can be regenerated exactly: the rulesets and the traffic vectors, the runs themselves, and the values of σ and D extracted from them.

A.1 gen_fw_rules: the ruleset generator

The generator builds the FW1 rulesets of every campaign (Section 5.1): a catalog of realistic UDP services with unique ports, so that no rule can shadow another, mirrored between the two destination subnets, with order, split, and wildcard placement set from the command line. The version listed here is v6, the one used for all the results of this thesis: its mixed order interleaves the two subnets proportionally to their share of the ruleset, and on a symmetric split its output is bit-per-bit identical to v5 (Section 5.1). The nine PRE rulesets of the matrix of Chapter 7 differ only in `-net-ratio` and `-order`:

```
python3 gen_fw_rules_v6.py -o fw1_r2575.rules --seed 42 \
    --limit 601 --net-ratio 25,75 --wildcard-rules 50 \
    --wildcard-style drop --order n2n1
python3 gen_fw_rules_v6.py -o fw1_5050_coda.rules --seed 42 \
    --limit 601 --net-ratio 50,50 --wildcard-rules 50 \
    --wildcard-style drop --order n1n2
python3 gen_fw_rules_v6.py -o fw1_7525_mixed.rules --seed 42 \
    --limit 601 --net-ratio 75,25 --wildcard-rules 50 \
    --wildcard-style drop --order mixed
```

```
#!/usr/bin/env python3
import argparse
import ipaddress
import random
import sys
from datetime import datetime

SRC = "10.0.0.0/24"
NET1 = "10.0.1.0/24"
NET1_PREFIX = "10.0.1."
NET2_PREFIX = "10.0.2."
RESERVED_OCTETS = {0, 1, 2, 3, 255}


SERVICES = [
    ("modbus",      502,   "modbus-udp",           "hosts", "ics"),
    ("dnp3",        20000, "dnp3-udp",             "hosts", "ics"),
    ("iec104",      2404,  "iec60870-5-104-udp",   "hosts", "ics"),
    ("enip_io",     2222,  "ethernetip-io",        "hosts", "ics"),
    ("enip_exp",    44818, "ethernetip-explicit",  "hosts", "ics"),
    ("profinet_rt", 34962, "profinet-rt",          "hosts", "ics"),
    ("profinet_al", 34964, "profinet-alarm",       "hosts", "ics"),
    ("opcua",       4840,  "opcua-udp",            "block", "ics"),
    ("fins",        9600,  "omron-fins-udp",       "hosts", "ics"),
    ("melsec",      5007,  "mitsubishi-melsec",    "hosts", "ics"),
    ("ethercat",    34980, "ethercat-udp",         "hosts", "ics"),
    ("bacnet",      47809, "bacnet-ot",            "hosts", "ics"),
    ("coap",        5683,  "coap-udp",             "hosts", "ics"),
    ("coaps",       5684,  "coaps-dtls",           "hosts", "ics"),
    ("dns",         53,    "dns-udp",              "wide",  "srv"),
    ("ntp",         123,   "ntp",                  "wide",  "srv"),
    ("snmp_get",    161,   "snmp-get",             "wide",  "srv"),
    ("snmp_trap",   162,   "snmp-trap",            "hosts", "srv"),
    ("syslog",      514,   "syslog",               "wide",  "srv"),
    ("radius_auth", 1812,  "radius-auth",          "hosts", "srv"),
    ("radius_acct", 1813,  "radius-acct",          "hosts", "srv"),
    ("tacacs",      49,    "tacacs-udp",           "hosts", "srv"),
    ("kerberos",    88,    "kerberos-udp",         "hosts", "srv"),
    ("cldap",       389,   "cldap-udp",            "hosts", "srv"),
    ("http3",       80,    "http3-cleartext",      "hosts", "srv"),
    ("https_quic",  443,   "https-quic",           "hosts", "srv"),
    ("https_alt",   8443,  "https-quic-alt",       "hosts", "srv"),
    ("ipsec_ike",   500,   "ipsec-ike",            "hosts", "srv"),
    ("ipsec_natt",  4500,  "ipsec-nat-t",          "hosts", "srv"),
    ("openvpn",     1194,  "openvpn",              "hosts", "srv"),
    ("sip",         5060,  "sip-udp",              "hosts", "srv"),
```

```
    ("netflow",     2055,  "netflow-v9",          "hosts", "srv"),
    ("sflow",       6343,  "sflow-v5",            "hosts", "srv"),
    ("zbx_agent",   10050, "zabbix-agent-udp",    "hosts", "mon"),
    ("zbx_srv",     10051, "zabbix-server-udp",   "hosts", "mon"),
    ("prometheus",  9090,  "prom-quic",           "hosts", "mon"),
    ("node_exp",    9100,  "node-exporter-udp",   "hosts", "mon"),
    ("influx",      8089,  "influx-line-udp",     "hosts", "mon"),
    ("grafana",     3000,  "grafana-quic",        "hosts", "mon"),
    ("fluentd",     24224, "fluentd-udp",         "hosts", "mon"),
    ("nfs",         2049,  "nfs-udp",             "hosts", "stor"),
    ("veeam",       6160,  "veeam-udp",           "hosts", "stor"),
    ("bacula_dir",  9101,  "bacula-dir-udp",      "hosts", "stor"),
    ("bacula_sd",   9103,  "bacula-sd-udp",       "hosts", "stor"),
    ("isns",        3205,  "isns-udp",            "hosts", "stor"),
]

BLOCKS = {
    "opcua": ["10.0.1.16/28", "10.0.1.32/28"],
}

INSECURE_WILDCARD = [
    ("udp", "19",    "chargen-amp"),
    ("udp", "67",    "dhcp-server-rogue"),
    ("udp", "68",    "dhcp-client-rogue"),
    ("udp", "69",    "tftp"),
    ("udp", "111",   "rpcbind-udp"),
    ("udp", "137",   "netbios-ns"),
    ("udp", "138",   "netbios-dgm"),
    ("udp", "177",   "xdmcp"),
    ("udp", "520",   "rip-cleartext"),
    ("udp", "623",   "ipmi-rmcp"),
    ("udp", "1645",  "radius-legacy-auth"),
    ("udp", "1646",  "radius-legacy-acct"),
    ("udp", "1701",  "l2tp-uncontrolled"),
    ("udp", "1883",  "mqtt-cleartext"),
    ("udp", "1900",  "ssdp"),
    ("udp", "4789",  "vxlan-cross-segment"),
    ("udp", "5353",  "mdns"),
    ("udp", "5355",  "llmnr"),
    ("udp", "6000",  "x11-udp"),
    ("udp", "11211", "memcached-udp"),
    ("udp", "17185", "vxworks-debug"),
    ("udp", "27015", "steam-srcds"),
    ("udp", "30718", "lantronix-discovery"),
    ("udp", "47808", "bacnet-scan"),
]

EXTRA_WILDCARD = [
    ("udp", "7",     "echo-amp"),
```

```
    ("udp", "9",     "discard"),
    ("udp", "13",    "daytime-amp"),
    ("udp", "17",    "qotd-amp"),
    ("udp", "37",    "time-amp"),
    ("udp", "42",    "wins"),
    ("udp", "113",   "auth-udp"),
    ("udp", "135",   "msrpc-udp"),
    ("udp", "427",   "slp-amp"),
    ("udp", "464",   "kpasswd"),
    ("udp", "517",   "talk"),
    ("udp", "518",   "ntalk"),
    ("udp", "593",   "http-rpc-epmap"),
    ("udp", "749",   "kerberos-adm"),
    ("udp", "1434",  "mssql-monitor-slammer"),
    ("udp", "1604",  "citrix-ica"),
    ("udp", "2302",  "gamesrv-probe"),
    ("udp", "3283",  "apple-remote-desktop"),
    ("udp", "3389",  "rdp-udp"),
    ("udp", "3702",  "ws-discovery-amp"),
    ("udp", "5093",  "sentinel-lm"),
    ("udp", "5351",  "nat-pmp"),
    ("udp", "10001", "ubiquiti-discovery"),
    ("udp", "10080", "amanda"),
    ("udp", "37810", "dahua-discovery"),
    ("udp", "49152", "ephemeral-probe"),
]


def mirror(dst):
    return dst.replace(NET1_PREFIX, NET2_PREFIX, 1)


def to_net(cidr):
    return ipaddress.ip_network(cidr, strict=False)


def covers(broader, narrower):
    return to_net(narrower).subnet_of(to_net(broader))


def host_pool():
    return [o for o in range(4, 255) if o not in RESERVED_OCTETS]


def build_net1(limit, seed, variants, n_wild=None, n1_target=None):
    pool = host_pool()
    wide = [s for s in SERVICES if s[3] == "wide"]
    block = [s for s in SERVICES if s[3] == "block"]
    hosts = [s for s in SERVICES if s[3] == "hosts"]
```

```
145
146     if n_wild is None:
147         n_wild = 2 * len(INSECURE_WILDCARD)
148     if n1_target is None:
149         n1_target = max(1, (limit - n_wild - 1) // 2)
150     fixed = len(wide) + sum(len(BLOCKS[s[0]]) for s in block)
151     remaining = max(0, n1_target - fixed)
152
153     if variants is not None:
154         F = max(1, variants)
155     else:
156         F = max(1, -(-remaining // max(1, len(hosts))))
157     F = min(F, len(pool))
158
159     entries = []
160     for idx, (name, port, descr, mode, zone) in enumerate(SERVICES):
161         if mode == "wide":
162             dsts = [NET1]
163         elif mode == "block":
164             dsts = list(BLOCKS[name])
165         else:
166             offset = (seed * 131 + idx * 37) % len(pool)
167             octets = [pool[(offset + k) % len(pool)] for k in range(F)]
168             dsts = [f"{NET1_PREFIX}{o}/32" for o in octets]
169         for dst in dsts:
170             entries.append({"name": name, "port": str(port), "descr": descr,
171                             "mode": mode, "zone": zone, "dst": dst})
172     if len(entries) > n1_target:
173         entries = entries[:n1_target]
174     return entries, F
175
176
177 def assert_no_shadow(rules):
178     seen = {}
179     conflicts = []
180     for (src, dst, proto, port) in rules:
181         prev = seen.setdefault((proto, port), [])
182         for (s_old, d_old) in prev:
183             if covers(d_old, dst):
184                 conflicts.append(((s_old, d_old), (src, dst), proto, port))
185         prev.append((src, dst))
186     return conflicts
187
188
189 def emit_line(src, dst, proto, port, target="ACCEPT", extra=""):
190     base = f"-A FORWARD -s {src} -d {dst} -p {proto} -m {proto} --dport {
    port}"
191     if extra:
```

```
192         base += f" {extra}"
193     return f"{base} -j {target}"
194 
195 
196 def main():
197     ap = argparse.ArgumentParser(description="Generatore ruleset FW1 v6 (
    REDIAL)")
198     ap.add_argument("-o", "--output", default="-")
199     ap.add_argument("--limit", type=int, default=1000,
200                     help="numero totale di regole desiderato (~50/50 net1/
    net2)")
201     ap.add_argument("--seed", type=int, default=42,
202                     help="seme per la scelta degli host destinazione")
203     ap.add_argument("--variants", type=int, default=None,
204                     help="forza il numero di host /32 per servizio (default:
     auto)")
205     ap.add_argument("--order",
206                     choices=["grouped", "mixed", "n1n2", "n2n1", "random"],
207                     default="grouped",
208                     help="ordine degli ACCEPT: grouped (per servizio net1->
    net2), "
209                          "mixed (net1/net2 interleaved in modo
    PROPORZIONALE al "
210                          "rapporto |net1|:|net2|; con liste uguali =
    alternanza "
211                          "classica), n1n2 (tutte net1 poi net2), n2n1 (
    tutte "
212                          "net2 poi net1), random (shuffle riproducibile "
213                          "pilotato da --seed)")
214     ap.add_argument("--wildcard-pos", choices=["tail", "head", "mixed"],
215                     default="tail",
216                     help="posizione delle wildcard /22: tail (in coda),
    head "
217                          "(in testa), mixed (mescolate a caso fra gli
    ACCEPT). "
218                          "Il log di default-drop resta comunque sempre
    ultimo.")
219     ap.add_argument("--wildcard-action", choices=["deny", "accept"],
220                     default="deny",
221                     help="azione delle regole wildcard /22: deny (DROP,
    storico) "
222                          "oppure accept (ACCEPT, per osservare cosa passa
    davvero "
223                          "attraverso FW1). In stile pairs il LOG resta
    sempre "
224                          "presente; con accept il prefisso diventa 'FW-WILD-
    OK '.")
225     ap.add_argument("--net-split", default=None,
226                     help="regole ACCEPT per net: 'N1,N2' (es. 100,450). "
```

```
227                         "Se impostato, --limit non governa piu' gli ACCEPT
    ")
228     ap.add_argument("--net-ratio", default=None,
229                     help="percentuale net1,net2 (es. 25,75): usa --limit
    come "
230                         "TOTALE desiderato e calcola da solo lo split
    ACCEPT. "
231                         "Mutuamente esclusivo con --net-split")
232     ap.add_argument("--wildcard-rules", type=int, default=None,
233                     help="numero di regole wildcard /22 (default: 48 = 24
    porte "
234                         "LOG+azione). Con --wildcard-style drop ogni
    regola e' "
235                         "un'azione secca su una porta diversa (max 50
    porte)")
236     ap.add_argument("--wildcard-style", choices=["pairs", "drop"],
237                     default="pairs",
238                     help="pairs = coppie LOG+azione (storico); drop = solo
    azione "
239                         "(DROP o ACCEPT a seconda di --wildcard-action)")
240     ap.add_argument("--no-final-log", action="store_true",
241                     help="omette la riga finale di LOG default-drop "
242                         "('FW-DROP-DEF'): la policy FORWARD DROP scarta "
243                         "comunque, ma senza log. Il totale scende di 1")
244     args = ap.parse_args()
245
246     rng = random.Random(args.seed)
247
248     if args.net_split and args.net_ratio:
249         ap.error("usa --net-split OPPURE --net-ratio, non entrambi")
250
251     if args.wildcard_rules is not None:
252         n_wild = args.wildcard_rules
253     elif args.wildcard_style == "drop":
254         n_wild = len(INSECURE_WILDCARD)
255     else:
256         n_wild = 2 * len(INSECURE_WILDCARD)
257
258     n_final = 0 if args.no_final_log else 1
259
260     net_split = None
261     if args.net_split:
262         try:
263             p1, p2 = (int(x) for x in args.net_split.split(","))
264             net_split = (p1, p2)
265         except ValueError:
266             ap.error("--net-split formato N1,N2 (es. 100,450)")
267     elif args.net_ratio:
268         try:
```

```
269             r1, r2 = (float(x) for x in args.net_ratio.split(","))
270         except ValueError:
271             ap.error("--net-ratio formato R1,R2 (es. 25,75)")
272         if r1 < 0 or r2 < 0 or (r1 + r2) == 0:
273             ap.error("--net-ratio: proporzioni non valide (somma > 0)")
274         accept_budget = max(2, args.limit - n_wild - n_final)
275         n1 = round(accept_budget * r1 / (r1 + r2))
276         n2 = accept_budget - n1
277         net_split = (n1, n2)
278         print(f"[i] --net-ratio {args.net_ratio} su totale ~{args.limit}: "
279               f"net-split={n1},{n2} (ACCEPT={accept_budget}, "
280               f"wildcard={n_wild}, +{n_final} final)", file=sys.stderr)
281 
282     cand_target = max(net_split) if net_split else None
283     net1_entries, F = build_net1(args.limit, args.seed, args.variants,
284                                  n_wild=n_wild, n1_target=cand_target)
285 
286     if net_split is None:
287         list1, list2 = net1_entries, net1_entries
288     else:
289         if len(net1_entries) < max(net_split):
290             print(f"[!] candidati insufficienti ({len(net1_entries)}) per "
291                   f"--net-split {args.net_split}: alza --variants",
292                   file=sys.stderr)
293         list1 = net1_entries[:net_split[0]]
294         list2 = net1_entries[:net_split[1]]
295 
296     accept_rules = []
297     accept_out = []
298 
299     def add_accept(e, tag):
300         dst = e["dst"] if tag == "net1" else mirror(e["dst"])
301         accept_rules.append((SRC, dst, "udp", e["port"]))
302         accept_out.append((tag, emit_line(SRC, dst, "udp", e["port"]),
303                            f"{e['descr']}-{e['zone']}-to-{dst}"))
304 
305     if args.order == "grouped":
306         for name, port, descr, mode, zone in SERVICES:
307             for e in [x for x in list1 if x["name"] == name]:
308                 add_accept(e, "net1")
309             for e in [x for x in list2 if x["name"] == name]:
310                 add_accept(e, "net2")
311     elif args.order == "mixed":
312         n1_len, n2_len = len(list1), len(list2)
313         if n1_len > 0:
314             add_accept(list1[0], "net1")
315         i, j = (1 if n1_len > 0 else 0), 0
316         while i < n1_len or j < n2_len:
```

```
317             if j >= n2_len or (i < n1_len and (i + 1) * n2_len <= (j + 1) *
     n1_len):
318                 add_accept(list1[i], "net1")
319                 i += 1
320             else:
321                 add_accept(list2[j], "net2")
322                 j += 1
323     elif args.order == "n1n2":
324         for e in list1:
325             add_accept(e, "net1")
326         for e in list2:
327             add_accept(e, "net2")
328     elif args.order == "n2n1":
329         for e in list2:
330             add_accept(e, "net2")
331         for e in list1:
332             add_accept(e, "net1")
333     else:
334         todo = [(e, "net1") for e in list1] + [(e, "net2") for e in list2]
335         rng.shuffle(todo)
336         for e, tag in todo:
337             add_accept(e, tag)
338 
339     conflicts = assert_no_shadow(accept_rules)
340     if conflicts:
341         print(f"[!] ATTENZIONE: {len(conflicts)} shadowed rules rilevate!",
342               file=sys.stderr)
343         for c in conflicts[:10]:
344             print("    ", c, file=sys.stderr)
345 
346     WILDCARD_DST = "10.0.0.0/22"
347     all_wild = INSECURE_WILDCARD + EXTRA_WILDCARD
348 
349     if args.wildcard_action == "accept":
350         wild_target = "ACCEPT"
351         wild_log_prefix = "FW-WILD-OK "
352         action_tag = "accept"
353     else:
354         wild_target = "DROP"
355         wild_log_prefix = "FW-INSEC "
356         action_tag = "drop"
357 
358     if args.wildcard_style == "drop":
359         n_ports = n_wild
360     else:
361         if n_wild % 2:
362             print(f"[!] --wildcard-rules {n_wild} dispari con style=pairs:
    "
363                   f"emetto {n_wild - 1} regole", file=sys.stderr)
```

```
364         n_ports = n_wild // 2
365
366     if n_ports > len(all_wild):
367         print(f"[!] richieste {n_ports} porte wildcard, disponibili "
368               f"{len(all_wild)}: tronco", file=sys.stderr)
369         n_ports = len(all_wild)
370
371     wild_units = []
372     for proto, port, name in all_wild[:n_ports]:
373         if args.wildcard_style == "pairs":
374             wild_units.append([
375                 ("wildcard",
376                  f'-A FORWARD -d {WILDCARD_DST} -p {proto} -m {proto} --
    dport {port} '
377                  f'-m limit --limit 5/min -j LOG --log-prefix "{
    wild_log_prefix}"',
378                  f"log-{name}"),
379                 ("wildcard",
380                  f'-A FORWARD -d {WILDCARD_DST} -p {proto} -m {proto} --
    dport {port} '
381                  f'-j {wild_target}',
382                  f"{action_tag}-{name}"),
383             ])
384         else:
385             wild_units.append([
386                 ("wildcard",
387                  f'-A FORWARD -d {WILDCARD_DST} -p {proto} -m {proto} --
    dport {port} '
388                  f'-j {wild_target}',
389                  f"{action_tag}-{name}"),
390             ])
391
392     wildcard_out = [row for u in wild_units for row in u]
393     final_out = []
394     if not args.no_final_log:
395         final_out = [("final",
396             '-A FORWARD -m limit --limit 10/min -j LOG --log-prefix "FW-
    DROP-DEF "',
397             "log-default-drop")]
398
399     if args.wildcard_pos == "mixed":
400         mix = [[row] for row in accept_out] + wild_units
401         rng.shuffle(mix)
402         all_entries = [row for u in mix for row in u] + final_out
403     elif args.wildcard_pos == "head":
404         all_entries = wildcard_out + accept_out + final_out
405     else:  # tail
406         all_entries = accept_out + wildcard_out + final_out
407
```

```
408     split_desc = f"{net_split[0]},{net_split[1]}" if net_split else "mirror
        "
409
410     now = datetime.now().strftime("%a %b %d %H:%M:%S %Y")
411     header = [
412         f"# Generated by gen_fw_rules_v6.py on {now}",
413         "# REDIAL ruleset v6 - src=10.0.0.0/24, dst net1(10.0.1)/net2
        (10.0.2),",
414         f"# solo UDP, nessuna shadowed rule. order={args.order} "
415         f"wildcard={args.wildcard_pos}/{args.wildcard_style}/{args.
        wildcard_action} "
416         f"net-split={split_desc}",
417         "*filter",
418         ":INPUT ACCEPT [0:0]",
419         ":FORWARD DROP [0:0]",
420         ":OUTPUT ACCEPT [0:0]",
421     ]
422
423     out = []
424     counts = {}
425     for i, (tag, line, descr) in enumerate(all_entries, start=1):
426         rid = f"R{i:04d}"
427         out.append(f'{line} -m comment --comment "{rid} [REDIAL:{tag}] {
        descr}"')
428         counts[tag] = counts.get(tag, 0) + 1
429
430     text = "\n".join(header + out + ["COMMIT", f"# Completed on {now}", ""
        ])
431     if args.output == "-":
432         print(text)
433     else:
434         with open(args.output, "w") as fh:
435             fh.write(text)
436         breakdown = ", ".join(f"{k}={v}" for k, v in sorted(counts.items())
        )
437         print(f"[+] {len(out)} regole scritte in {args.output} "
438               f"(order={args.order}, wildcard={args.wildcard_pos}/"
439               f"{args.wildcard_style}/{args.wildcard_action}, "
440               f"net-split={split_desc}, "
441               f"limit={args.limit}, seed={args.seed}, "
442               f"host/servizio={F}; {breakdown}; shadowed={len(conflicts)})",
443               flush=True)
444
445
446 if __name__ == "__main__":
447     main()
```

Listing A.1: The ruleset generator `gen_fw_rules_v6.py`, final version used in every campaign of this thesis.

A.2 gen_packets_v2: the packet-vector generator

The second generator turns a ruleset into the two traffic vectors sent by `sendpkts` (Section 5.1). It parses the ruleset file directly, so it stays aligned with whatever parameters produced it, and emits one vector per destination subnet: every packet matches exactly one rule, the load is spread uniformly over all the ACCEPT rules of the subnet, and a 32-bit sequence number in the payload makes every packet unique. The vectors are deterministic functions of the ruleset and of `-seed`: the same command line regenerates them bit by bit, which is what allows the PRE and POST runs of a pair to replay the identical traffic (Section 7.2). The version listed here is v2, the one used in the campaigns: it reads the wildcard rules of the v5/v6 rulesets in both their DROP and ACCEPT forms.

```
python3 gen_packets_v2.py -r fw1_r2575.rules --count 10000 --seed 42
python3 gen_packets_v2.py -r fw1.rules \
    --payload-bytes 18,64,128,256,512,1024,1472
python3 gen_packets_v2.py -r fw1.rules --wildcard-frac 0
```

```
#!/usr/bin/env python3
import argparse
import ipaddress
import json
import random
import re
import sys
from collections import Counter

SRC_NET = "10.0.0.0/24"
NET1 = "10.0.1.0/24"
NET2 = "10.0.2.0/24"
RESERVED_OCTETS = {0, 1, 2, 3, 255}

RE_ACCEPT = re.compile(
    r'-A FORWARD -s (?P<src>\S+) -d (?P<dst>\S+) -p udp -m udp --dport (?P<dport>\d+) -j ACCEPT'
)
RE_WILD = re.compile(
```

```
    r'-A FORWARD -d (?P<dst>\S+) -p udp -m udp --dport (?P<dport>\d+) -j (?P<target>ACCEPT|DROP)\b'
)
RE_COMMENT = re.compile(r'--comment "(?P<rid>R\d+) \[REDIAL:(?P<tag>\w+)\] (?P<label>[^"]*)"')


class Rule:
    __slots__ = ("rid", "tag", "dport", "label", "dst_cidr", "hosts")

    def __init__(self, rid, tag, dport, label, dst_cidr):
        self.rid = rid
        self.tag = tag
        self.dport = dport
        self.label = label
        self.dst_cidr = dst_cidr
        net = ipaddress.ip_network(dst_cidr, strict=False)
        if net.prefixlen == 32:
            self.hosts = [str(net.network_address)]
        else:
            self.hosts = [
                str(h) for h in net.hosts()
                if int(str(h).split(".")[-1]) not in RESERVED_OCTETS
            ]
            if not self.hosts:
                self.hosts = [str(h) for h in net.hosts()]


def parse_rules(path):
    accepts = {"net1": [], "net2": []}
    wildcards = {}
    with open(path) as fh:
        for line in fh:
            line = line.rstrip("\n")
            cm = RE_COMMENT.search(line)
            if not cm:
                continue
            tag = cm.group("tag")
            if tag in accepts:
                ma = RE_ACCEPT.search(line)
                if ma:
                    accepts[tag].append(Rule(
                        cm.group("rid"), tag, int(ma.group("dport")),
                        cm.group("label"), ma.group("dst"),
                    ))
            elif tag == "wildcard":
                mw = RE_WILD.search(line)
                if mw:
                    wildcards[int(mw.group("dport"))] = mw.group("target")
```

```
66      return accepts, dict(sorted(wildcards.items()))
67
68
69  def even_counts(total, n_buckets):
70      if n_buckets == 0:
71          return []
72      base, rem = divmod(total, n_buckets)
73      return [base + (1 if i < rem else 0) for i in range(n_buckets)]
74
75
76  def host_pool(cidr):
77      net = ipaddress.ip_network(cidr, strict=False)
78      return [str(h) for h in net.hosts()
79              if int(str(h).split(".")[-1]) not in RESERVED_OCTETS]
80
81
82  def make_payload(seq, size, fixed_hex):
83      if fixed_hex is not None:
84          return fixed_hex
85      seqb = (seq & 0xFFFFFFFF).to_bytes(4, "big")
86      if size <= 4:
87          return seqb[:size].hex()
88      filler = bytes((seq + k) & 0xFF for k in range(size - 4))
89      return (seqb + filler).hex()
90
91
92  def build_vector(net_tag, dst_net, rules, wildcards, args, rng):
93      if not rules:
94          sys.exit(f"[err] nessuna regola ACCEPT trovata per {net_tag}")
95
96      n_wild = round(args.count * args.wildcard_frac) if wildcards else 0
97      n_accept = args.count - n_wild
98
99      if args.src == "sweep":
100         srcs = host_pool(SRC_NET)
101     else:
102         srcs = [args.src]
103
104     payload_sizes = [int(x) for x in str(args.payload_bytes).split(",") if
        x != ""]
105     fixed_hex = args.payload_hex.replace(" ", "") if args.payload_hex else
        None
106
107     targets = []
108     for cnt, rule in zip(even_counts(n_accept, len(rules)), rules):
109         targets.extend([("accept", rule)] * cnt)
110     if n_wild:
111         ports = list(wildcards.items())
112         for cnt, item in zip(even_counts(n_wild, len(ports)), ports):
```

```
            targets.extend([("wild", item)] * cnt)

    rng.shuffle(targets)

    packets = []
    seen = set()
    whosts = host_pool(dst_net)
    for i, (kind, obj) in enumerate(targets, start=1):
        if kind == "accept":
            rule = obj
            dport = rule.dport
            dst = rng.choice(rule.hosts)
            verdict = "ACCEPT"
            rid = rule.rid
            label = rule.label
        else:
            dport, action = obj
            dst = rng.choice(whosts)
            verdict = action
            rid = ""
            label = f"wildcard-{dport}"

        src = rng.choice(srcs)
        for _ in range(4):
            sport = rng.randint(args.sport_min, args.sport_max)
            key = (src, dst, sport, dport)
            if key not in seen:
                break
        seen.add(key)

        size = payload_sizes[(i - 1) % len(payload_sizes)]
        payload_hex = make_payload(i, size, fixed_hex)
        plen = len(payload_hex) // 2

        packets.append({
            "seq": i, "net": net_tag, "proto": "udp",
            "src": src, "sport": sport, "dst": dst, "dport": dport,
            "plen": plen, "verdict": verdict, "rule_id": rid,
            "label": label, "payload_hex": payload_hex,
        })
    return packets


CSV_FIELDS = ["seq", "net", "proto", "src", "sport", "dst", "dport",
              "plen", "verdict", "rule_id", "label", "payload_hex"]


def write_csv(path, packets):
    with open(path, "w") as fh:
```

```
162         fh.write(",".join(CSV_FIELDS) + "\n")
163         for p in packets:
164             fh.write(",".join(str(p[k]) for k in CSV_FIELDS) + "\n")
165
166
167 def write_jsonl(path, packets):
168     with open(path, "w") as fh:
169         for p in packets:
170             fh.write(json.dumps(p, separators=(",", ":")) + "\n")
171
172
173 def summarize(net_tag, packets):
174     legit = sum(1 for p in packets if p["rule_id"])
175     wild = Counter(p["verdict"] for p in packets if not p["rule_id"])
176     rules_hit = len({p["rule_id"] for p in packets if p["rule_id"]})
177     uniq5 = len({(p["src"], p["dst"], p["sport"], p["dport"]) for p in
    packets})
178     uniqpkt = len({(p["src"], p["dst"], p["sport"], p["dport"], p["
    payload_hex"]) for p in packets})
179     wild_str = " ".join(f"{k}={v}" for k, v in sorted(wild.items())) if
    wild else "0"
180     return (f"  {net_tag}: {len(packets)} pkt | legit(ACCEPT)={legit} "
181             f"wildcard[{wild_str}] | regole coperte={rules_hit} | "
182             f"5-tuple uniche={uniq5} ({100*uniq5/len(packets):.2f}%) | "
183             f"pacchetti unici={uniqpkt} ({100*uniqpkt/len(packets):.2f}%)")
184
185
186 def main():
187     ap = argparse.ArgumentParser(description="Generatore vettori di
    pacchetti REDIAL v2 (net1/net2).")
188     ap.add_argument("-r", "--rules", required=True)
189     ap.add_argument("--out-net1", default="packets_net1.csv")
190     ap.add_argument("--out-net2", default="packets_net2.csv")
191     ap.add_argument("--count", type=int, default=10000)
192     ap.add_argument("--seed", type=int, default=42)
193     ap.add_argument("--wildcard-frac", type=float, default=0.05)
194     ap.add_argument("--payload-bytes", default="4")
195     ap.add_argument("--payload-hex", default=None)
196     ap.add_argument("--src", default="10.0.0.1")
197     ap.add_argument("--sport-min", type=int, default=1024)
198     ap.add_argument("--sport-max", type=int, default=65535)
199     ap.add_argument("--format", choices=["csv", "jsonl"], default="csv")
200     args = ap.parse_args()
201
202     accepts, wildcards = parse_rules(args.rules)
203     rng = random.Random(args.seed)
204
205     if not wildcards and args.wildcard_frac > 0:
```

```
206         print("[!] nessuna porta wildcard trovata nel ruleset: vettori
    SENZA rumore "
207               "(controlla i tag [REDIAL:wildcard] in fw1.rules)", file=sys.
    stderr)
208
209     pkt1 = build_vector("net1", NET1, accepts["net1"], wildcards, args, rng
    )
210     pkt2 = build_vector("net2", NET2, accepts["net2"], wildcards, args, rng
    )
211
212     ext = ".jsonl" if args.format == "jsonl" else None
213     out1 = args.out_net1 if ext is None else args.out_net1.rsplit(".", 1)
    [0] + ext
214     out2 = args.out_net2 if ext is None else args.out_net2.rsplit(".", 1)
    [0] + ext
215     writer = write_jsonl if args.format == "jsonl" else write_csv
216     writer(out1, pkt1)
217     writer(out2, pkt2)
218
219     actions = ", ".join(sorted(set(wildcards.values()))) if wildcards else
    "n/d"
220     print(f"[+] regole lette da {args.rules}: "
221           f"net1={len(accepts['net1'])} ACCEPT, net2={len(accepts['net2'])}
     ACCEPT, "
222           f"{len(wildcards)} porte wildcard (azione: {actions})")
223     print(f"[+] scritti {out1} e {out2} (seed={args.seed}, formato={args.
    format})")
224     print(summarize("net1", pkt1))
225     print(summarize("net2", pkt2))
226
227
228 if __name__ == "__main__":
229     main()
```

Listing A.2: `gen_packets_v2.py`, the packet-vector generator used in every campaign of this thesis.

A.3 sendpkts: the sender

`sendpkts` is the traffic sender of the bench (Section 5.2). It reads the two CSV vectors produced by `gen_packets_v2` and builds every IP/UDP frame once in memory before the run starts, so the hot send path is a single `sendto` per packet on a raw socket with `IP_HDRINCL`. Each step of a ramp is paced by busy-spinning on `CLOCK_MONOTONIC` against an integer period in nanoseconds, so a rate has no fractional remainder to distribute unevenly across packets; the process runs under `SCHED_FIFO` with locked memory on an isolated core. The split between the two

subnets is realized by a Bresenham-style interleaving that attains ρ exactly over the long run, and each vector wraps around independently once exhausted. Build and typical invocations:

```
gcc -O3 -Wall -o sendpkts sendpkts.c -lrt
sudo ip netns exec SX ./sendpkts --split 50,50 --cpu 9 \
    --ramp 60:100000 --epoch-file ramp_start.epoch
sudo ip netns exec SX ./sendpkts --split 25,75 --cpu 9 --ramp 10:800000
./sendpkts --split 50,50 --ramp 2:1000,10000 --dry-run
```

```
#define _GNU_SOURCE
#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <sched.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define LIKELY(x)    __builtin_expect(!!(x), 1)
#define UNLIKELY(x)  __builtin_expect(!!(x), 0)

#define MAX_LINE   16384
#define MAX_FIELDS 16

static volatile sig_atomic_t running = 1;
static void sig_handler(int s) { (void)s; running = 0; }

static void timespec_add_ns(struct timespec *ts, long ns) {
    ts->tv_nsec += ns;
    while (ts->tv_nsec >= 1000000000L) { ts->tv_sec++; ts->tv_nsec -=
    1000000000L; }
    while (ts->tv_nsec <  0)           { ts->tv_sec--; ts->tv_nsec +=
    1000000000L; }
}

static int timespec_before(const struct timespec *a, const struct timespec
    *b) {
```

```
    return a->tv_sec < b->tv_sec ||
           (a->tv_sec == b->tv_sec && a->tv_nsec < b->tv_nsec);
}

static void wait_until(const struct timespec *target) {
    struct timespec now;
    do { clock_gettime(CLOCK_MONOTONIC, &now); }
    while (timespec_before(&now, target));
}

static uint16_t ip_checksum(const void *data, int len) {
    const uint16_t *p = (const uint16_t *)data;
    unsigned long sum = 0;
    while (len > 1) { sum += *p++; len -= 2; }
    if (len) sum += *(const uint8_t *)p;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (uint16_t)(~sum);
}

typedef struct {
    uint8_t *buf;
    int      len;
    uint32_t dst_ip;
    uint16_t dport;
} packet_t;

typedef struct {
    packet_t *pkts;
    size_t    n;
    size_t    idx;
    char      tag[8];
} vector_t;

static int split_csv(char *line, char **out, int maxf) {
    int n = 0;
    char *p = line;
    out[n++] = p;
    for (; *p; p++) {
        if (*p == ',') {
            *p = '\0';
            if (n < maxf) out[n++] = p + 1;
        }
    }
    return n;
}

static int hex2bin(const char *hex, uint8_t *out, int maxlen) {
    int n = 0;
```

```
84      while (hex[0] && hex[1] && n < maxlen) {
85          if (!isxdigit((unsigned char)hex[0]) || !isxdigit((unsigned char)
        hex[1])) break;
86          int hi = (hex[0] <= '9') ? hex[0]-'0' : (tolower(hex[0])-'a'+10);
87          int lo = (hex[1] <= '9') ? hex[1]-'0' : (tolower(hex[1])-'a'+10);
88          out[n++] = (uint8_t)((hi << 4) | lo);
89          hex += 2;
90      }
91      return n;
92  }
93
94  static void build_packet(packet_t *pk, uint32_t src_ip, uint16_t sport,
95                           uint32_t dst_ip, uint16_t dport,
96                           const uint8_t *payload, int plen, uint16_t ip_id)
        {
97      int total = sizeof(struct iphdr) + sizeof(struct udphdr) + plen;
98      uint8_t *buf = calloc(1, total);
99      if (!buf) { perror("calloc"); exit(1); }
100     struct iphdr  *iph  = (struct iphdr  *)buf;
101     struct udphdr *udph = (struct udphdr *)(buf + sizeof(struct iphdr));
102     uint8_t       *pl   = buf + sizeof(struct iphdr) + sizeof(struct udphdr
        );
103
104     iph->ihl      = 5;
105     iph->version  = 4;
106     iph->tos      = 0;
107     iph->tot_len  = htons(total);
108     iph->id       = htons(ip_id);
109     iph->frag_off = 0;
110     iph->ttl      = 64;
111     iph->protocol = IPPROTO_UDP;
112     iph->check    = 0;
113     iph->saddr    = src_ip;
114     iph->daddr    = dst_ip;
115
116     udph->source = htons(sport);
117     udph->dest   = htons(dport);
118     udph->len    = htons(sizeof(struct udphdr) + plen);
119     udph->check  = 0;
120
121     if (plen > 0) memcpy(pl, payload, plen);
122
123     iph->check = ip_checksum(iph, sizeof(struct iphdr));
124
125     pk->buf    = buf;
126     pk->len    = total;
127     pk->dst_ip = dst_ip;
128     pk->dport  = dport;
129 }
```

```
130
131 static void load_vector(const char *path, const char *tag, vector_t *v) {
132     FILE *f = fopen(path, "r");
133     if (!f) { fprintf(stderr, "[ERR] apertura %s: %s\n", path, strerror(
    errno)); exit(1); }
134
135     size_t cap = 4096;
136     v->pkts = malloc(cap * sizeof(packet_t));
137     if (!v->pkts) { perror("malloc"); exit(1); }
138     v->n = 0; v->idx = 0;
139     snprintf(v->tag, sizeof(v->tag), "%s", tag);
140
141     char line[MAX_LINE];
142     char *fields[MAX_FIELDS];
143     uint8_t payload[2048];
144     uint16_t ip_id = 0;
145     int first = 1;
146
147     while (fgets(line, sizeof(line), f)) {
148         size_t L = strlen(line);
149         while (L && (line[L-1] == '\n' || line[L-1] == '\r')) line[--L] = '
    \0';
150         if (L == 0) continue;
151         if (first) { first = 0; if (!strncmp(line, "seq,", 4)) continue; }
152
153         int nf = split_csv(line, fields, MAX_FIELDS);
154         if (nf < 7) continue;
155
156         uint32_t src_ip = inet_addr(fields[3]);
157         uint16_t sport  = (uint16_t)atoi(fields[4]);
158         uint32_t dst_ip = inet_addr(fields[5]);
159         uint16_t dport  = (uint16_t)atoi(fields[6]);
160         const char *phex = fields[nf - 1];
161         int plen = hex2bin(phex, payload, sizeof(payload));
162
163         if (v->n == cap) {
164             cap *= 2;
165             v->pkts = realloc(v->pkts, cap * sizeof(packet_t));
166             if (!v->pkts) { perror("realloc"); exit(1); }
167         }
168         build_packet(&v->pkts[v->n], src_ip, sport, dst_ip, dport,
169                      payload, plen, ip_id++);
170         v->n++;
171     }
172     fclose(f);
173     if (v->n == 0) { fprintf(stderr, "[ERR] %s: nessun pacchetto caricato\n
    ", path); exit(1); }
174 }
175
```

```
176 static void enable_realtime(int cpu) {
177     if (mlockall(MCL_CURRENT | MCL_FUTURE) == -1) {
178         perror("[ERR] mlockall (serve root)"); exit(1);
179     }
180     struct sched_param p;
181     p.sched_priority = sched_get_priority_max(SCHED_FIFO);
182     if (sched_setscheduler(0, SCHED_FIFO, &p) == -1) {
183         perror("[ERR] sched_setscheduler"); exit(1);
184     }
185     if (cpu >= 0) {
186         cpu_set_t s; CPU_ZERO(&s); CPU_SET(cpu, &s);
187         if (sched_setaffinity(0, sizeof(s), &s) == -1)
188             perror("[WARN] sched_setaffinity");
189     }
190     fprintf(stderr, "[+] Real-time engine attivo (SCHED_FIFO, mlockall, CPU
    =%d)\n", cpu);
191 }
192
193 typedef struct { int duration_sec; int rate; } ramp_step_t;
194
195 static ramp_step_t *parse_ramp(const char *spec, int *n_out) {
196     if (!spec || !*spec) return NULL;
197     char *copy = strdup(spec);
198     if (!copy) return NULL;
199     char *colon = strchr(copy, ':');
200     if (!colon || colon == copy) { free(copy); return NULL; }
201     *colon = '\0';
202     int dur = atoi(copy);
203     if (dur <= 0) { free(copy); return NULL; }
204     char *rates = colon + 1;
205     if (!*rates) { free(copy); return NULL; }
206     int n = 1;
207     for (char *p = rates; *p; p++) if (*p == ',') n++;
208     ramp_step_t *steps = calloc(n, sizeof(*steps));
209     if (!steps) { free(copy); return NULL; }
210     int i = 0;
211     char *tok = strtok(rates, ",");
212     while (tok && i < n) {
213         int r = atoi(tok);
214         if (r <= 0) {
215             fprintf(stderr, "[ERR] rate '%s' invalido\n", tok);
216             free(steps); free(copy);
217             return NULL;
218         }
219         steps[i].duration_sec = dur;
220         steps[i].rate         = r;
221         i++;
222         tok = strtok(NULL, ",");
223     }
```

```
224     free(copy);
225     *n_out = i;
226     return steps;
227 }
228 
229 static void usage(const char *p) {
230     fprintf(stderr,
231         "Usage: sudo ip netns exec SX %s [opts] --ramp DUR:R1,...,RN\n"
232         "  --net1 FILE     CSV vettore net1            (default
    packets_net1.csv)\n"
233         "  --net2 FILE     CSV vettore net2            (default
    packets_net2.csv)\n"
234         "  --split P1,P2   ripartizione %% net1/net2    (default 50,50)\n"
235         "  --cpu  N        pin a core N                (default 2, -1 = no
    pin)\n"
236         "  --epoch-file F  scrive l'epoch REALTIME di start (allineamento
    listener)\n"
237         "  --dry-run       non invia (test logica/conteggi senza root/raw
    socket)\n"
238         "  --ramp DUR:R1,...,RN\n"
239         "                  N step di DUR secondi, ognuno al rate TOTALE Ri
    pps\n"
240         "                  (lo split ripartisce Ri fra net1 e net2)\n"
241         "Esempio (50/50 a 800k tot = 400k+400k):\n"
242         "  sudo ip netns exec SX %s --net1 packets_net1.csv --net2
    packets_net2.csv \\\n"
243         "          --split 50,50 --cpu 2 --ramp
    20:1,10,100,1000,10000,100000,400000,800000\n",
244         p, p);
245 }
246 
247 int main(int argc, char **argv) {
248     const char *net1_path = "packets_net1.csv";
249     const char *net2_path = "packets_net2.csv";
250     const char *split_s   = "50,50";
251     const char *ramp_spec = NULL;
252     const char *epoch_file = NULL;
253     int cpu = 2;
254     int dry_run = 0;
255 
256     for (int i = 1; i < argc; i++) {
257         if      (!strcmp(argv[i], "--net1")       && i+1<argc) net1_path  =
     argv[++i];
258         else if (!strcmp(argv[i], "--net2")       && i+1<argc) net2_path  =
     argv[++i];
259         else if (!strcmp(argv[i], "--split")      && i+1<argc) split_s    =
     argv[++i];
260         else if (!strcmp(argv[i], "--cpu")        && i+1<argc) cpu        =
     atoi(argv[++i]);
```

```
261         else if (!strcmp(argv[i], "--ramp")       && i+1<argc) ramp_spec  =
     argv[++i];
262         else if (!strcmp(argv[i], "--epoch-file") && i+1<argc) epoch_file =
     argv[++i];
263         else if (!strcmp(argv[i], "--dry-run")) dry_run = 1;
264         else if (!strcmp(argv[i], "-h") || !strcmp(argv[i], "--help")) {
    usage(argv[0]); return 0; }
265         else { fprintf(stderr, "[ERR] opzione sconosciuta: %s\n", argv[i]);
     usage(argv[0]); return 1; }
266     }
267
268     if (!ramp_spec) { usage(argv[0]); return 1; }
269
270     double w1 = 0, w2 = 0;
271     {
272         char *copy = strdup(split_s);
273         char *c = strchr(copy, ',');
274         if (!c) { fprintf(stderr, "[ERR] --split formato P1,P2\n"); return
    1; }
275         *c = '\0';
276         w1 = atof(copy); w2 = atof(c + 1);
277         free(copy);
278         if (w1 < 0 || w2 < 0 || (w1 + w2) <= 0) {
279             fprintf(stderr, "[ERR] --split valori non validi\n"); return 1;
280         }
281     }
282     double frac1 = w1 / (w1 + w2);
283
284     int n_steps = 0;
285     ramp_step_t *steps = parse_ramp(ramp_spec, &n_steps);
286     if (!steps || n_steps <= 0) {
287         fprintf(stderr, "[ERR] --ramp invalido (formato: DUR:R1,R2,...)\n")
    ; return 1;
288     }
289
290     vector_t v1 = {0}, v2 = {0};
291     if (frac1 > 0.0) load_vector(net1_path, "net1", &v1);
292     if (frac1 < 1.0) load_vector(net2_path, "net2", &v2);
293
294     int sock = -1;
295     if (!dry_run) {
296         enable_realtime(cpu);
297         sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
298         if (sock < 0) { perror("socket (richiede root o CAP_NET_RAW)");
    return 1; }
299         int one = 1;
300         if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one)) <
    0) {
301             perror("setsockopt IP_HDRINCL"); close(sock); return 1;
```

```
302         }
303         int sndbuf = 4 * 1024 * 1024;
304         setsockopt(sock, SOL_SOCKET, SO_SNDBUF, &sndbuf, sizeof(sndbuf));
305     } else {
306         fprintf(stderr, "[+] DRY-RUN: nessun invio reale (test logica/
        conteggi)\n");
307     }
308 
309     signal(SIGINT,  sig_handler);
310     signal(SIGTERM, sig_handler);
311 
312     long total_sec = 0;
313     for (int i = 0; i < n_steps; i++) total_sec += steps[i].duration_sec;
314 
315     fprintf(stderr, "=======================================\n");
316     fprintf(stderr, "  sendpkts  net1=%s (%zu pkt)  net2=%s (%zu pkt)\n",
317             net1_path, v1.n, net2_path, v2.n);
318     fprintf(stderr, "  split       : %.4g%% net1 / %.4g%% net2  (frac1=%.4f
        )\n",
319             100*frac1, 100*(1-frac1), frac1);
320     fprintf(stderr, "  ramp        : %d step, totale %lds (rate = pps
        TOTALE/step)\n",
321             n_steps, total_sec);
322     fprintf(stderr, "  rates (pps) :");
323     for (int i = 0; i < n_steps; i++) fprintf(stderr, " %d", steps[i].rate)
        ;
324     fprintf(stderr, "\n=======================================\n");
325 
326     struct sockaddr_in dst = {0};
327     dst.sin_family = AF_INET;
328     dst.sin_port   = 0;
329 
330     struct timespec sync_ts;
331     clock_gettime(CLOCK_MONOTONIC, &sync_ts);
332     sync_ts.tv_nsec = 0;
333     sync_ts.tv_sec += 1;
334     wait_until(&sync_ts);
335 
336     if (epoch_file) {
337         struct timespec rt; clock_gettime(CLOCK_REALTIME, &rt);
338         FILE *ef = fopen(epoch_file, "w");
339         if (ef) { fprintf(ef, "%ld.%09ld\n", (long)rt.tv_sec, rt.tv_nsec);
        fclose(ef); }
340     }
341 
342     unsigned long sent1 = 0, sent2 = 0, errs = 0;
343     double acc = 0.0;
344     struct timespec next = sync_ts;
345 
```

```
346     for (int i = 0; i < n_steps && running; i++) {
347         long interval_ns = 1000000000L / steps[i].rate;
348         struct timespec step_end = next;
349         step_end.tv_sec += steps[i].duration_sec;
350 
351         unsigned long p1 = sent1, p2 = sent2, pe = errs;
352 
353         time_t t = time(NULL); struct tm tmv; localtime_r(&t, &tmv);
354         char tb[16]; strftime(tb, sizeof(tb), "%H:%M:%S", &tmv);
355         fprintf(stderr, "[%s] STEP %d/%d  rate_tot=%d pps  dur=%ds
    interval=%ldns\n",
356                 tb, i + 1, n_steps, steps[i].rate, steps[i].duration_sec,
    interval_ns);
357 
358         while (LIKELY(running)) {
359             if (UNLIKELY(!timespec_before(&next, &step_end))) break;
360             wait_until(&next);
361 
362             acc += frac1;
363             int use1;
364             if (acc >= 1.0) { use1 = 1; acc -= 1.0; } else { use1 = 0; }
365             vector_t *v = use1 ? &v1 : &v2;
366             packet_t *pk = &v->pkts[v->idx];
367             v->idx++; if (v->idx >= v->n) v->idx = 0;
368 
369             if (LIKELY(!dry_run)) {
370                 dst.sin_addr.s_addr = pk->dst_ip;
371                 ssize_t r = sendto(sock, pk->buf, pk->len, 0,
372                                    (const struct sockaddr *)&dst, sizeof(
    dst));
373                 if (UNLIKELY(r < 0)) {
374                     if (errno == ENOBUFS || errno == EAGAIN) { errs++; }
375                     else { perror("sendto"); running = 0; break; }
376                 } else { if (use1) sent1++; else sent2++; }
377             } else {
378                 if (use1) sent1++; else sent2++;
379             }
380 
381             timespec_add_ns(&next, interval_ns);
382         }
383 
384         t = time(NULL); localtime_r(&t, &tmv);
385         strftime(tb, sizeof(tb), "%H:%M:%S", &tmv);
386         fprintf(stderr, "[%s]   end  net1=%lu net2=%lu (tot=%lu)  errs=%lu\
    n",
387                 tb, sent1 - p1, sent2 - p2, (sent1 - p1) + (sent2 - p2),
    errs - pe);
388 
389         next = step_end;
```

```
390     }
391
392     if (sock >= 0) close(sock);
393     free(steps);
394     fprintf(stderr, "========================================\n");
395     fprintf(stderr, "  inviati net1 : %lu\n", sent1);
396     fprintf(stderr, "  inviati net2 : %lu\n", sent2);
397     fprintf(stderr, "  totale       : %lu\n", sent1 + sent2);
398     fprintf(stderr, "  errori       : %lu\n", errs);
399     fprintf(stderr, "========================================\n");
400     return 0;
401 }
```

Listing A.3: `sendpkts.c`, the sender of the testbed (final version used in all the campaigns).

A.4 run-redial.sh and listen-redial.sh: run orchestration

Two shell scripts drive a single run of the bench (Section 5.2). `listen-redial.sh` arms the captures: one tcpdump per capture port with the hardware timestamps of the NIC (`-j adapter_unsynced`, nanosecond precision), the receive rings at their maximum, the offload engines off, real-time scheduling on dedicated cores, and a snapshot of the NIC counters before and after the run, so a packet lost by the instrument is told apart from one lost by the firewall. `run-redial.sh` orchestrates the run: it starts the listener in the background, waits for tcpdump to be capturing, then launches `sendpkts` inside the `SX` namespace with the requested split and ramp, and derives the capture length from the ramp itself with a grace margin on both sides.

```
1 sudo ./run-redial.sh --split 50,50 --cpu 9 --ramp 60:100000
```

```
1  #!/usr/bin/env bash
2  set -euo pipefail
3
4  # default
5  OUTDIR="./captures"
6  NET1="packets_net1.csv"
7  NET2="packets_net2.csv"
8  SPLIT="50,50"
9  CPU=9
10 RAMP=""
11 BPF="udp"
```

```
SENDPKTS="${SENDPKTS:-./sendpkts}"
LISTEN_SH="${LISTEN_SH:-./listen-redial.sh}"
GRACE_BEFORE=2
GRACE_AFTER=3

usage() {
  cat <<EOF
Usage: sudo $0 [options] --ramp DUR:R1,R2,...,RN
  --net1 FILE     CSV vettore net1                  (default $NET1)
  --net2 FILE     CSV vettore net2                  (default $NET2)
  --split P1,P2   ripartizione % net1/net2          (default $SPLIT)
  --cpu  N        core isolato per sendpkts         (default $CPU)
  --out  DIR      cartella pcap + epoch             (default $OUTDIR)
  --bpf  FILTER   filtro tcpdump del listener       (default "$BPF")
  --ramp DUR:R1,...   spec rampa, Ri = pps TOTALE/step  (OBBLIGATORIO)
Env:
  SENDPKTS, LISTEN_SH = percorso a binario/script
  CAP_DST=1 IF_DST=<iface>  = cattura anche il sink d2 (net2)
Esempio (50/50, rampa fino a 800k tot = 400k+400k):
  sudo $0 --split 50,50 --cpu 9 \\
          --ramp 20:1,10,100,1000,10000,100000,400000,800000
EOF
}

while [[ $# -gt 0 ]]; do
  case "$1" in
    --net1)   NET1="$2";   shift 2 ;;
    --net2)   NET2="$2";   shift 2 ;;
    --split)  SPLIT="$2";  shift 2 ;;
    --cpu)    CPU="$2";    shift 2 ;;
    --out)    OUTDIR="$2"; shift 2 ;;
    --bpf)    BPF="$2";    shift 2 ;;
    --ramp)   RAMP="$2";   shift 2 ;;
    -h|--help) usage; exit 0 ;;
    *) echo "[ERR] opzione sconosciuta: $1"; usage; exit 1 ;;
  esac
done

[[ -z "$RAMP" ]]      && { echo "[ERR] --ramp obbligatorio"; usage; exit 1; }
[[ -x "$SENDPKTS"  ]] || { echo "[ERR] $SENDPKTS non eseguibile"; exit 1; }
[[ -x "$LISTEN_SH" ]] || { echo "[ERR] $LISTEN_SH non eseguibile"; exit 1; }

[[ -f "$NET1" ]]     || { echo "[ERR] $NET1 mancante"; exit 1; }
[[ -f "$NET2" ]]     || { echo "[ERR] $NET2 mancante"; exit 1; }

DUR_STEP="${RAMP%%:*}"
N_RATES=$(awk -F, '{print NF}' <<<"${RAMP#*:}")
GEN_SEC=$((DUR_STEP * N_RATES))
```

```
LISTEN_SEC=$((GEN_SEC + GRACE_BEFORE + GRACE_AFTER))

mkdir -p "$OUTDIR"
STAMP=$(date +%Y%m%d_%H%M%S)
EPOCH_FILE="$OUTDIR/ramp_start_${STAMP}.epoch"

echo "========================================"
echo "  vettori          : $NET1 / $NET2"
echo "  split            : $SPLIT (% net1/net2, sul rate TOTALE)"
echo "  rampa            : $RAMP  (Ri = pps totale/step)"
echo "  durata generator : ${GEN_SEC}s"
echo "  durata listener  : ${LISTEN_SEC}s"
echo "  filtro listener  : $BPF"
echo "  cartella out     : $OUTDIR"
echo "  epoch file       : $EPOCH_FILE"
echo "  CPU pin (sender) : $CPU"
echo "========================================"

echo "[1/2] avvio listener (${LISTEN_SEC}s) ..."
"$LISTEN_SH" "$LISTEN_SEC" "$BPF" "$OUTDIR" &
LISTEN_PID=$!

trap 'echo "[!] interrotto, killo listener"; kill -INT $LISTEN_PID 2>/dev/
    null || true; wait 2>/dev/null || true; exit 130' INT TERM

sleep "$GRACE_BEFORE"

echo "[2/2] avvio sendpkts ..."
ip netns exec SX "$SENDPKTS" \
    --net1 "$NET1" --net2 "$NET2" \
    --split "$SPLIT" --cpu "$CPU" \
    --epoch-file "$EPOCH_FILE" \
    --ramp "$RAMP"
GEN_RC=$?

echo "[*] attendo il listener (PID=$LISTEN_PID) ..."
wait "$LISTEN_PID" || true

echo
echo "[done] generatore exit=$GEN_RC"
echo "[done] epoch di start: $(cat "$EPOCH_FILE" 2>/dev/null || echo 'N/D')
    "
echo "[done] pcap prodotti (questa run):"
ls -lht "$OUTDIR"/*.pcap 2>/dev/null | head -n 3
```

Listing A.4: `run-redial.sh`, the single-run orchestrator of the testbed.

```
#!/usr/bin/env bash
set -euo pipefail
```

```
3
4  DURATION="${1:-30}"
5  BPF="${2:-udp}"
6  OUTDIR="${3:-./captures}"
7
8  NS_TAP="${NS_TAP:-TAP}";  IF_TAP="${IF_TAP:-ens8191f3}"
9  NS_RX="${NS_RX:-RX}";     IF_RX="${IF_RX:-ens8191f2}"
10 NS_DST="${NS_DST:-DST}";  IF_DST="${IF_DST:-}"
11 CAP_DST="${CAP_DST:-0}"
12
13 CPU_TAP="${CPU_TAP:-4}"
14 CPU_RX="${CPU_RX:-6}"
15 CPU_DST="${CPU_DST:-8}"
16
17 TCPDUMP_BUF_KB="${TCPDUMP_BUF_KB:-524288}"
18 NIC_RING_RX="${NIC_RING_RX:-4096}"
19
20 if [[ "$CAP_DST" == "1" && -z "$IF_DST" ]]; then
21     echo "[ERR] CAP_DST=1 ma IF_DST non impostata (interfaccia del sink d2)
       "; exit 1
22 fi
23
24 mkdir -p "$OUTDIR"
25 STAMP=$(date +%Y%m%d_%H%M%S)
26 TAP_PCAP="$OUTDIR/tap_${STAMP}.pcap"
27 RX_PCAP="$OUTDIR/rx_${STAMP}.pcap"
28 DST_PCAP="$OUTDIR/dst_${STAMP}.pcap"
29
30 preflight() {
31     local ns="$1" iface="$2"
32     sudo ip netns exec "$ns" ethtool -G "$iface" rx "$NIC_RING_RX" 2>/dev/
       null || true
33     sudo ip netns exec "$ns" ethtool -K "$iface" gro off lro off tso off
       gso off 2>/dev/null || true
34 }
35 preflight "$NS_TAP" "$IF_TAP"
36 preflight "$NS_RX"  "$IF_RX"
37 [[ "$CAP_DST" == "1" ]] && preflight "$NS_DST" "$IF_DST"
38
39 snap() {
40     sudo ip netns exec "$1" ethtool -S "$2" 2>/dev/null | \
41         awk '/rx_(missed_errors|fifo_errors|no_buff_count|no_dma_resources|
       over_errors|dropped|long_byte_count)/{
42             gsub(":",""); print $1, $2
43         }'
44 }
45 snap "$NS_TAP" "$IF_TAP" > /tmp/et_pre_tap.txt
46 snap "$NS_RX"  "$IF_RX"  > /tmp/et_pre_rx.txt
47 [[ "$CAP_DST" == "1" ]] && snap "$NS_DST" "$IF_DST" > /tmp/et_pre_dst.txt
```

```
48
49 echo "Cattura ${DURATION}s   filtro='${BPF}'"
50 echo "  TAP=$IF_TAP(ns $NS_TAP, CPU$CPU_TAP)  -> $TAP_PCAP"
51 echo "  RX =$IF_RX(ns $NS_RX, CPU$CPU_RX)  -> $RX_PCAP"
52 [[ "$CAP_DST" == "1" ]] && echo "  DST=$IF_DST(ns $NS_DST, CPU$CPU_DST)  ->
       $DST_PCAP"
53 echo "  buffer tcpdump=${TCPDUMP_BUF_KB} KB   ring RX=${NIC_RING_RX}"
54
55 LAST_PID=""
56 start_tcpdump() {
57     local ns="$1" iface="$2" cpu="$3" out="$4" log="$5"
58     sudo ip netns exec "$ns" \
59         chrt -f 50 taskset -c "$cpu" \
60         tcpdump -i "$iface" -j adapter_unsynced \
61             --time-stamp-precision=nano -nn \
62             -B "$TCPDUMP_BUF_KB" \
63             -w "$out" "$BPF" \
64             2>"$log" &
65     LAST_PID=$!
66 }
67
68 start_tcpdump "$NS_TAP" "$IF_TAP" "$CPU_TAP" "$TAP_PCAP" /tmp/tcpdump_tap.
      log; PID_TAP=$LAST_PID
69 start_tcpdump "$NS_RX"  "$IF_RX"  "$CPU_RX"  "$RX_PCAP"  /tmp/tcpdump_rx.
      log;  PID_RX=$LAST_PID
70 PID_DST=""
71 if [[ "$CAP_DST" == "1" ]]; then
72     start_tcpdump "$NS_DST" "$IF_DST" "$CPU_DST" "$DST_PCAP" /tmp/
      tcpdump_dst.log; PID_DST=$LAST_PID
73 fi
74
75 trap 'sudo kill -INT $PID_TAP $PID_RX $PID_DST 2>/dev/null; wait 2>/dev/
      null; exit 130' INT
76
77 sleep "$DURATION"
78
79 sudo kill -INT $PID_TAP $PID_RX $PID_DST 2>/dev/null || true
80 wait $PID_TAP $PID_RX $PID_DST 2>/dev/null || true
81
82 snap "$NS_TAP" "$IF_TAP" > /tmp/et_post_tap.txt
83 snap "$NS_RX"  "$IF_RX"  > /tmp/et_post_rx.txt
84 [[ "$CAP_DST" == "1" ]] && snap "$NS_DST" "$IF_DST" > /tmp/et_post_dst.txt
85
86 delta() {
87     awk 'NR==FNR { a[$1]=$2; next }
88          { if ($1 in a) {
89              d = $2 - a[$1];
90              if (d != 0) printf "  %-26s %12d -> %12d  (delta %+d)\n", $1,
      a[$1], $2, d
```

```
91            }}' \
92          "$1" "$2"
93 }
94
95 echo
96 echo "--- tcpdump TAP (kernel) ---"; grep -E "received|dropped" /tmp/
       tcpdump_tap.log || true
97 echo "--- tcpdump RX  (kernel) ---"; grep -E "received|dropped" /tmp/
       tcpdump_rx.log  || true
98 if [[ "$CAP_DST" == "1" ]]; then
99     echo "--- tcpdump DST (kernel) ---"; grep -E "received|dropped" /tmp/
       tcpdump_dst.log || true
100 fi
101
102 echo "--- NIC TAP delta (hw/driver) ---"; delta /tmp/et_pre_tap.txt /tmp/
       et_post_tap.txt | grep . || echo "  (nessun delta)"
103 echo "--- NIC RX  delta (hw/driver) ---"; delta /tmp/et_pre_rx.txt  /tmp/
       et_post_rx.txt  | grep . || echo "  (nessun delta)"
104 if [[ "$CAP_DST" == "1" ]]; then
105     echo "--- NIC DST delta (hw/driver) ---"; delta /tmp/et_pre_dst.txt /
       tmp/et_post_dst.txt | grep . || echo "  (nessun delta)"
106 fi
107
108 echo
109 cnt() { awk '/received by filter/{print $1; exit}' "$1" 2>/dev/null; }
110 sz()  { du -h "$1" 2>/dev/null | cut -f1; }
111 echo "Pacchetti catturati (received by filter) + dimensione file:"
112 echo "  TAP: $(cnt /tmp/tcpdump_tap.log) pkt   ($(sz "$TAP_PCAP"))"
113 echo "  RX:  $(cnt /tmp/tcpdump_rx.log) pkt   ($(sz "$RX_PCAP"))"
114 [[ "$CAP_DST" == "1" ]] && echo "  DST: $(cnt /tmp/tcpdump_dst.log) pkt   (
       $(sz "$DST_PCAP"))"
115 echo "[i] conteggio esatto on-demand (lento su file grandi): capinfos -c <
       file.pcap>"
```

Listing A.5: `listen-redial.sh`, the capture listener: one tcpdump per capture port with the measurement-safe configuration of Section 4.4.

A.5 campagna-redial.sh: the campaign wrapper

The campaign script automates a full campaign for one ruleset (Section 5.2): the three PRE runs at $\rho = 0.25, 0.5, 0.75$, a pause to apply the redistribution, and the three POST runs with the same splits, one folder per run with its captures, epoch file, counter dump, and latency analysis. Every run is guarded: the firewall must hold exactly the expected number of rules with all counters at zero before the start; the captures are accepted only if TAP and RX record the same packet count, otherwise the run is deleted and repeated (up to three attempts); the evaluation

load rate $\times$ σ is checked against the knee at every run; and the per-run dispersion is checked against the acceptance thresholds of Section 5.3. The final summary prints the measured delays at the three values of ρ and, below them, the same quantities derived from the per-subnet values of the 50/50 run alone: the blend property of Section 3.1, verified at every campaign for free. The script is never launched with `sudo`: the counter collector logs into the firewall over SSH as the bench user, and the internal `sudo` calls keep their own timestamp alive.

```
sudo -v && ./campagna-redial.sh --tag r2575-testa --rate 100000 \
    --rules fw1_r2575.rules --post-rules 190
sudo -v && ./campagna-redial.sh --tag r5050-coda --rate 100000 \
    --rules fw1_5050_coda.rules --post-rules 327
```

```
#!/usr/bin/env bash
set -euo pipefail

TAG=""; RATE=400000; RULES="fw1.rules"; OFFSET=-11973
PRE_N=601; POST_N=""; STD_PRE=40; STD_POST=20; KNEE_M=130; MAX_RETRY=3

while [ $# -gt 0 ]; do
  case "$1" in
    --tag)        TAG="$2"; shift 2;;
    --rate)       RATE="$2"; shift 2;;
    --rules)      RULES="$2"; shift 2;;
    --pre-rules)  PRE_N="$2"; shift 2;;
    --post-rules) POST_N="$2"; shift 2;;
    --offset)     OFFSET="$2"; shift 2;;
    *) echo "[ERR] flag sconosciuto: $1"; exit 1;;
  esac
done
[ -n "$TAG" ]   || { echo "[ERR] serve --tag"; exit 1; }
[ -f "$RULES" ] || { echo "[ERR] manca il file regole: $RULES"; exit 1; }

RAMP="60:${RATE}"
BASE="campagne/${TAG}"
RHOS="0.25 0.5 0.75"

if [ "$(id -u)" -eq 0 ]; then echo "[ERR] lancialo da utente normale: il
    sudo serve solo all'interno"; exit 1; fi

mkdir -p "$BASE"
mkdir -p captures; chmod 1777 captures 2>/dev/null || true
[ -w captures ] || { echo "[ERR] captures/ non scrivibile: sudo chown $USER:
     captures && chmod 1777 captures"; exit 1; }

avail_gb=$(df -BG --output=avail . | tail -1 | tr -dc '0-9')
```

```
[ "$avail_gb" -ge 25 ] || { echo "[ERR] servono ~25 GB liberi (disponibili ${avail_gb}G)"; exit 1; }

split_of() { case "$1" in 0.25) echo 75,25;; 0.5) echo 50,50;; 0.75) echo 25,75;; esac; }
dir_of()   { printf '%s/rho%s' "$BASE" "${1/./}"; }
newest_ts() { ls -t captures/tap_*.pcap | head -1 | sed 's#.*tap_\(.*\)\.pcap#\1#'; }

check_zero() {
  local out n
  out=$(python3 sigma_collector_subnet.py zero)
  echo "$out"
  n=$(echo "$out" | grep -o '[0-9]\+ regole' | grep -o '[0-9]\+' | head -1)
  if [ -n "${1:-}" ] && [ "$n" != "$1" ]; then
    echo "[ERR] sul FW ci sono $n regole, attese $1 - stato PRE/POST sbagliato?"
    exit 1
  fi
}

check_std() {
  awk -v lim="$2" '$1=="tot"{ s=$11/1000
    if (s>lim) printf "[WARN] std tot %.1f us > criterio %s us - tempi da scartare/ripetere\n", s, lim
    else       printf "[OK]   std tot %.1f us (criterio %s us)\n", s, lim }' "$1"
}

do_run() {
  local phase=$1 rho=$2 extra="" ts dst split out sig evals
  dst="$(dir_of "$rho")/${phase}"; mkdir -p "$dst"
  split=$(split_of "$rho")
  [ "$phase" = post ] && extra="--post --rho $rho"
  echo; echo "======== ${phase^^}  rho=$rho  (split $split, ramp $RAMP) ========"
  local attempt=1 tap_n rx_n
  while :; do
    if [ "$phase" = pre ]; then check_zero "$PRE_N"; else check_zero "${POST_N:-}"; fi
    sudo ./run-redial.sh --split "$split" --cpu 2 --ramp "$RAMP" 2>&1 | tee "$dst/run.log"
    ts=$(newest_ts)
    tap_n=$(awk '$1=="TAP:"{print $2; exit}' "$dst/run.log")
    rx_n=$(awk  '$1=="RX:"{print $2; exit}'  "$dst/run.log")
    if [ -n "$tap_n" ] && [ "$tap_n" = "$rx_n" ]; then
      echo "[OK]   cattura completa: TAP=$tap_n RX=$rx_n"
      break
    fi
```

```
72      echo "[WARN] perdita in cattura (TAP=$tap_n RX=$rx_n) - cancello i file
         della run e ripeto (tentativo $attempt/$MAX_RETRY)"
73      sudo rm -f "captures/tap_${ts}.pcap" "captures/rx_${ts}.pcap" "captures
        /ramp_start_${ts}.epoch" "$dst/run.log"
74      attempt=$((attempt+1))
75      [ "$attempt" -le "$MAX_RETRY" ] || { echo "[ERR] perdite per $MAX_RETRY
         run di fila - controlla banco/ring buffer, non e' sfortuna"; exit 1; }
76    done
77    sudo mv "captures/tap_${ts}.pcap" "captures/rx_${ts}.pcap" "$dst/"
78    sudo mv "captures/ramp_start_${ts}.epoch" "$dst/" 2>/dev/null || true
79    sudo chown -R "$USER": "$dst"
80    out=$(python3 sigma_collector_subnet.py collect -o "$dst/sigma_${phase}.
        json" $extra)
81    echo "$out"
82    sig=$(echo "$out" | awk -F'= *' '/sigma_overall/{print $2; exit}')
83    if [ -n "$sig" ]; then
84      evals=$(awk -v s="$sig" -v r="$RATE" 'BEGIN{printf "%d", s*r/1000000}')
85      if [ "$evals" -gt "$KNEE_M" ]; then
86        echo "[WARN] ${evals}M valutazioni/s > ginocchio ~${KNEE_M}M - TEMPI
        non affidabili a questo rate (i sigma restano validi)"
87      else
88        echo "[OK]   carico ${evals}M valutazioni/s (ginocchio ~${KNEE_M}M)"
89      fi
90    fi
91    ./compare-redial.py --tap "$dst/tap_${ts}.pcap" --rx "$dst/rx_${ts}.pcap"
         \
92        --ramp "$RAMP" --rx-offset-ns "$OFFSET" --rules "$RULES" \
93        --csv "$dst/delta.csv" | tee "$dst/delta.txt"
94    if [ "$phase" = pre ]; then check_std "$dst/delta.txt" "$STD_PRE"; else
        check_std "$dst/delta.txt" "$STD_POST"; fi
95    echo "[i] salvato tutto in $dst/ (pcap $ts)"
96  }
97
98  echo "[i] tag=$TAG  rate=$RATE  rules=$RULES  -> 6 run in $BASE/ (~20 GB,
        ~20 min + pausa REDIAL)"
99  echo "[i] promemoria: timer silenziati? run solo di giorno!"
100
101 for RHO in $RHOS; do do_run pre "$RHO"; done
102
103 echo
104 echo "================================================================="
105 echo " Applica REDIAL da ieiitp97:"
106 echo "   cd ~/redial-single-fw && unset WORKDIR && ./run-fw1-only.sh"
107 echo "================================================================="
108 read -rp "Premi INVIO quando il FW e' in POST... "
109
110 for RHO in $RHOS; do do_run post "$RHO"; done
111
112 for RHO in $RHOS; do
```

```
  d=$(dir_of "$RHO")
  echo; echo "----------- compare sigma  rho=$RHO -----------"
  python3 sigma_collector_subnet.py compare "$d/pre/sigma_pre.json" "$d/
    post/sigma_post.json" \
      --rho "$RHO" | tee "$d/compare_sigma.txt"
done

{
  printf '%6s %12s %12s %12s   (ns, misurati)\n' rho D_pre D_post Delta
  for RHO in $RHOS; do
    d=$(dir_of "$RHO")
    awk -v rho="$RHO" '
      FNR==NR && $1=="tot" {dp=$10}
      FNR!=NR && $1=="tot" {dq=$10}
      END {printf "%6s %12.1f %12.1f %12.1f\n", rho, dp, dq, dp-dq}
    ' "$d/pre/delta.txt" "$d/post/delta.txt"
  done
  echo
  echo "check incrociato - derivati dai per-net della sola run 50/50 (ns):"
  awk '
    FNR==NR && ($1=="1"||$1=="2") {pre[$1]=$10}
    FNR!=NR && ($1=="1"||$1=="2") {post[$1]=$10}
    END {
      n=split("0.25 0.5 0.75", r, " ")
      for (i=1;i<=n;i++) { rho=r[i]+0
        dp=(1-rho)*pre["1"]+rho*pre["2"]; dq=(1-rho)*post["1"]+rho*post
    ["2"]
        printf "%6s %12.1f %12.1f %12.1f\n", r[i], dp, dq, dp-dq }
    }' "$(dir_of 0.5)/pre/delta.txt" "$(dir_of 0.5)/post/delta.txt"
} | tee "$BASE/summary.txt"

echo; echo "[done] campagna $TAG completata - vedi $BASE/summary.txt e
    riporta nel tracker."
echo "[i] rollback (ieiitp97): export WORKDIR=<dir della run> && ./start.sh
     pre   (o ./start.sh pre-last)"
```

Listing A.6: `campagna-redial.sh`, the campaign wrapper driving the three PRE/POST pairs of one ruleset, with automatic retry on capture loss. (Comment and echo strings transliterated to ASCII for this listing.)

A.6 The REDIAL orchestrator (redial-single-fw)

The third piece of the execution phase turns the PRE ruleset into the POST one on FW1 (Section 5.2). The orchestrator runs on the management host and drives the firewall over SSH: it saves the current ruleset, converts it to XML, computes the redistributed ruleset, validates it with `iptables-restore -test`, applies it, and

can roll back to the saved PRE state at any time. The redistribution itself lives in `redial_fw1_only.py` (Listing A.7): the rules of FW1 are scanned in order, the first rule whose destination intersects the offloaded domain triggers the insertion of the shortcut rule at that position, rules that still serve the remaining domains are kept, and rules that only serve the offloaded domain are removed. Non-terminating targets follow the configurable `log_strategy` (`move`, `keep`, `duplicate`). Parsing and serialization of the ruleset use `RedialUtils`, the support module of the reference REDIAL implementation, not listed here. The shell scripts around it (Listings A.8 to A.11) automate the cycle and archive every application in a timestamped run folder, so any POST state can be rolled back to its own PRE backup.

```
cd ~/redial-single-fw && unset WORKDIR && ./run-fw1-only.sh
./start.sh pre
./start.sh pre-last
```

```
#!/usr/bin/env python3

import argparse
import ipaddress
from copy import deepcopy
from typing import List, Literal, Optional

from RedialUtils import (
    Firewall,
    Table,
    Rule,
    parse_iptables_xml_to_firewall,
    firewall_to_iptables_save,
)


_NON_TERMINATING = {
    "LOG", "NFLOG", "AUDIT", "MARK", "CONNMARK", "TRACE", "TEE", "CT",
}


def _norm(action: Optional[str]) -> str:
    return (action or "").strip().upper()


def _is_nonterminating(action: Optional[str]) -> bool:
    return _norm(action) in _NON_TERMINATING


def _touches_network(rule: Rule, net: ipaddress.IPv4Network) -> bool:
    return rule.dst is None or rule.dst.overlaps(net)
```

```
32
33
34  def _touches_any(rule: Rule, nets: List[ipaddress.IPv4Network]) -> bool:
35      if rule.dst is None:
36          return True
37      return any(rule.dst.overlaps(n) for n in nets)
38
39
40  def redial_fw1_only(
41      fw1: Firewall,
42      d2_list: List[ipaddress.IPv4Network],
43      dn: List[ipaddress.IPv4Network],
44      log_strategy: Literal["move", "keep", "duplicate"] = "move",
45  ) -> Firewall:
46      assert log_strategy in ("move", "keep", "duplicate")
47
48      fw1_hat = Firewall()
49
50      if not fw1.tables:
51          return fw1_hat
52
53      orig_table = fw1.tables[0]
54
55      fw1_new = Table(star="*", table_name=orig_table.table_name)
56      fw1_new.chain_defs = deepcopy(orig_table.chain_defs)
57
58      shortcut_inserted = False
59
60      for rule in orig_table.rules:
61          is_nonterm = _is_nonterminating(rule.action)
62
63          touches_d2 = any(_touches_network(rule, d2) for d2 in d2_list)
64
65          if touches_d2 and not shortcut_inserted:
66              shortcut_inserted = True
67
68              for d2 in d2_list:
69                  fw1_new.rules.append(Rule(
70                      chain="FORWARD",
71                      src=None,
72                      dst=d2,
73                      proto=None,
74                      sport=None,
75                      dport=None,
76                      icmp_type=None,
77                      action="ACCEPT",
78                  ))
79
80          keep_on_fw1 = _touches_any(rule, dn)
```

```
81
82          # Target non terminanti: preserva la politica della versione
        completa.
83          if is_nonterm and log_strategy in ("keep", "duplicate"):
84              keep_on_fw1 = True
85
86          if keep_on_fw1:
87              fw1_new.rules.append(deepcopy(rule))
88
89      fw1_hat.tables.append(fw1_new)
90      return fw1_hat
91
92
93  def parse_net(value: str) -> ipaddress.IPv4Network:
94      return ipaddress.IPv4Network(value, strict=False)
95
96
97  def main() -> None:
98      ap = argparse.ArgumentParser(
99          description="Genera solo FW1^, ignorando FW2, per misure FW1-only
        post-REDIAL."
100     )
101
102     ap.add_argument("--fw1-xml", required=True)
103
104     ap.add_argument(
105         "--d2",
106         action="append",
107         required=True,
108         type=parse_net,
109         help="Dominio ridistribuito verso FW2. Ripetibile.",
110     )
111
112     ap.add_argument(
113         "--dn",
114         action="append",
115         required=True,
116         type=parse_net,
117         help="Dominio non ridistribuito, che resta filtrato da FW1.
        Ripetibile.",
118     )
119
120     ap.add_argument(
121         "--log-strategy",
122         default="move",
123         choices=("move", "keep", "duplicate"),
124     )
125
126     ap.add_argument("--out-fw1", required=True)
```

```
127
128     args = ap.parse_args()
129
130     fw1 = parse_iptables_xml_to_firewall(args.fw1_xml)
131
132     fw1_hat = redial_fw1_only(
133         fw1=fw1,
134         d2_list=args.d2,
135         dn=args.dn,
136         log_strategy=args.log_strategy,
137     )
138
139     with open(args.out_fw1, "w") as f:
140         f.write(firewall_to_iptables_save(fw1_hat))
141
142     print(f"[OK] FW1^ generated -> {args.out_fw1}")
143     print("[INFO] FW2 ignored by design")
144
145
146 if __name__ == "__main__":
147     main()
```

Listing A.7: `redial_fw1_only.py`, the FW1-only redistribution: builds the POST ruleset of FW1 from the PRE one. (Comment strings transliterated to ASCII for this listing.)

```
1  #!/usr/bin/env bash
2  set -euo pipefail
3
4  export D0_CIDR="${D0_CIDR:-10.0.0.0/24}"
5  export DN_CIDR="${DN_CIDR:-10.0.1.0/24}"
6  export D2_CIDR="${D2_CIDR:-10.0.2.0/24}"
7
8  export FW1_MGMT="${FW1_MGMT:-192.168.100.103}"
9  export FW1_NETNS="${FW1_NETNS:-FW}"
10 export SSH_USER="${SSH_USER:-drago}"
11
12 export PASS_FILE="${PASS_FILE:-$HOME/redial/.fwpass}"
13
14 if [ -z "${WORKDIR:-}" ]; then
15   export WORKDIR="$PWD/runs/$(date +%Y%m%d-%H%M%S)"
16 fi
17
18 mkdir -p "$WORKDIR"
19
20 echo "[inventory] WORKDIR=$WORKDIR"
21 echo "[inventory] FW1=$SSH_USER@$FW1_MGMT netns=$FW1_NETNS"
22 echo "[inventory] d0=$D0_CIDR dn=$DN_CIDR d2=$D2_CIDR"
23
```

```
#!/usr/bin/env bash

source ./inventory.sh

die() {
  echo "[ERR] $*" >&2
  exit 1
}

fw1() {
  sshpass -f "$PASS_FILE" ssh \
    -o StrictHostKeyChecking=accept-new \
    -o ConnectTimeout=10 \
    "$SSH_USER@$FW1_MGMT" "$@"
}

fw1_sudo() {
  sshpass -f "$PASS_FILE" ssh \
    -o StrictHostKeyChecking=accept-new \
    -o ConnectTimeout=10 \
    "$SSH_USER@$FW1_MGMT" \
    "sudo -S -p '' $*" < "$PASS_FILE"
}

fw1_put() {
  local src="$1"
  local dst="$2"

  sshpass -f "$PASS_FILE" scp \
    -o StrictHostKeyChecking=accept-new \
    -o ConnectTimeout=10 \
    "$src" "$SSH_USER@$FW1_MGMT:$dst"
}
```

Listing A.8: Orchestrator configuration (`inventory.sh`) and SSH helpers (`lib.sh`).

```
#!/usr/bin/env bash
set -euo pipefail

cd "$(dirname "$0")"
source ./lib.sh

echo "[collect] salvo ruleset originale FW1"

fw1_sudo "ip netns exec $FW1_NETNS iptables-save" \
  > "$WORKDIR/fw1.backup.rules"

echo "[collect] scritto: $WORKDIR/fw1.backup.rules"
```

```
13
14 echo "[collect] converto in XML"
15 command -v iptables-xml >/dev/null 2>&1 || die "iptables-xml non trovato
       sul nodo di gestione"
16
17 iptables-xml < "$WORKDIR/fw1.backup.rules" > "$WORKDIR/fw1.xml"
18
19 echo "[collect] scritto: $WORKDIR/fw1.xml"
20 echo "[collect] OK"
21 ls -lh "$WORKDIR/fw1.backup.rules" "$WORKDIR/fw1.xml"
22
23 #!/usr/bin/env bash
24 set -euo pipefail
25
26 cd "$(dirname "$0")"
27 source ./inventory.sh
28
29 [ -f "$WORKDIR/fw1.xml" ] || {
30   echo "[ERR] manca $WORKDIR/fw1.xml"
31   echo "      Esegui prima: ./collect-fw1.sh oppure ./start.sh prepare"
32   exit 1
33 }
34
35 echo "[generate] genero FW1^ REDIAL-only"
36 echo "[generate] d2=$D2_CIDR"
37 echo "[generate] dn=$DN_CIDR"
38
39 ./redial_fw1_only.py \
40   --fw1-xml "$WORKDIR/fw1.xml" \
41   --d2 "$D2_CIDR" \
42   --dn "$DN_CIDR" \
43   --out-fw1 "$WORKDIR/fw1.redial-only.rules"
44
45 echo "[generate] scritto: $WORKDIR/fw1.redial-only.rules"
46
47 echo
48 echo "=== shortcut attesa ==="
49 grep -- "-d $D2_CIDR -j ACCEPT" "$WORKDIR/fw1.redial-only.rules" || {
50   echo "[WARN] shortcut non trovata con grep esatto"
51 }
52
53 echo
54 echo "=== conteggio regole FORWARD ==="
55 echo -n "originale: "
56 grep -c "^-A FORWARD" "$WORKDIR/fw1.backup.rules" || true
57
58 echo -n "fw1-only:  "
59 grep -c "^-A FORWARD" "$WORKDIR/fw1.redial-only.rules" || true
60
```

```
61 echo
62 echo "=== prime 80 righe ruleset generato ==="
63 sed -n '1,80p' "$WORKDIR/fw1.redial-only.rules"
64
65 #!/usr/bin/env bash
66 set -euo pipefail
67
68 cd "$(dirname "$0")"
69 source ./lib.sh
70
71 RULES="$WORKDIR/fw1.redial-only.rules"
72
73 [ -f "$RULES" ] || die "manca $RULES; esegui prima generate"
74
75 echo "[validate] copio ruleset su FW1"
76 fw1_put "$RULES" /tmp/fw1.redial-only.rules
77
78 echo "[validate] iptables-restore --test"
79 fw1_sudo "ip netns exec $FW1_NETNS iptables-restore --test /tmp/fw1.redial-
       only.rules"
80
81 echo "[validate] OK"
82
83 #!/usr/bin/env bash
84 set -euo pipefail
85
86 cd "$(dirname "$0")"
87 source ./lib.sh
88
89 RULES="$WORKDIR/fw1.redial-only.rules"
90
91 [ -f "$RULES" ] || die "manca $RULES; esegui prima generate"
92
93 echo "[apply] copio ruleset su FW1"
94 fw1_put "$RULES" /tmp/fw1.redial-only.rules
95
96 echo "[apply] dry-run prima dell'applicazione"
97 fw1_sudo "ip netns exec $FW1_NETNS iptables-restore --test /tmp/fw1.redial-
       only.rules"
98
99 echo "[apply] applico FW1^"
100 fw1_sudo "ip netns exec $FW1_NETNS iptables-restore /tmp/fw1.redial-only.
       rules"
101
102 echo "[apply] OK"
103
104 echo
105 echo "=== FW1 FORWARD prime 30 regole ==="
106 fw1_sudo "ip netns exec $FW1_NETNS iptables -S FORWARD | head -30"
```

```
107
108 echo
109 echo "[INFO] FW2 non e' stato toccato."
110
111 #!/usr/bin/env bash
112 set -euo pipefail
113
114 cd "$(dirname "$0")"
115 source ./lib.sh
116
117 BACKUP="$WORKDIR/fw1.backup.rules"
118
119 [ -f "$BACKUP" ] || die "manca $BACKUP; devi usare la WORKDIR corretta"
120
121 echo "[rollback] ripristino ruleset originale FW1"
122 fw1_put "$BACKUP" /tmp/fw1.backup.rules
123
124 fw1_sudo "ip netns exec $FW1_NETNS iptables-restore /tmp/fw1.backup.rules"
125
126 echo "[rollback] OK"
127
128 echo
129 echo "=== FW1 FORWARD prime 30 regole ==="
130 fw1_sudo "ip netns exec $FW1_NETNS iptables -S FORWARD | head -30"
131
132 #!/usr/bin/env bash
133 set -euo pipefail
134
135 cd "$(dirname "$0")"
136
137 echo "[status] carico lib.sh"
138 source ./lib.sh
139
140 echo "[status] WORKDIR=$WORKDIR"
141 echo "[status] FW1=$SSH_USER@$FW1_MGMT"
142 echo "[status] FW1_NETNS=$FW1_NETNS"
143 echo "[status] D2=$D2_CIDR"
144 echo "[status] PASS_FILE=$PASS_FILE"
145
146 [ -f "$PASS_FILE" ] || die "PASS_FILE non trovato: $PASS_FILE"
147
148 echo
149 echo "[status] test SSH semplice"
150 fw1 "hostname; ip netns list"
151
152 echo
153 echo "[status] test sudo semplice"
154 fw1_sudo "true"
155
```

```
156 echo
157 echo "[status] FW1 FORWARD prime 50 regole"
158 fw1_sudo "ip netns exec $FW1_NETNS iptables -S FORWARD | head -50"
159
160 echo
161 echo "[status] cerca shortcut verso $D2_CIDR"
162 fw1_sudo "ip netns exec $FW1_NETNS iptables -S FORWARD | grep -- '-d
        $D2_CIDR -j ACCEPT' || true"
163
164 echo
165 echo "[status] OK"
```

Listing A.9: The phase scripts of the orchestrator: collect, generate, validate, apply, rollback, and status.

```
1  #!/usr/bin/env bash
2  set -euo pipefail
3
4  cd "$(dirname "$0")"
5
6  cmd="${1:-help}"
7
8  case "$cmd" in
9  prepare)
10 echo "[start] prepare = collect + generate + validate"
11 ./collect-fw1.sh
12 ./generate-fw1-only.sh
13 ./validate-fw1-only.sh
14 ;;
15
16 post)
17 echo "[start] post = applica FW1^ post-REDIAL-only"
18 ./apply-fw1-only.sh
19 ;;
20
21 pre)
22 echo "[start] pre = rollback al ruleset originale"
23 ./rollback-fw1.sh
24 ;;
25
26 status)
27 ./status-fw1.sh
28 ;;
29
30 collect)
31 ./collect-fw1.sh
32 ;;
33
34 generate)
```

```
./generate-fw1-only.sh
;;

validate)
./validate-fw1-only.sh
;;

apply)
./apply-fw1-only.sh
;;

rollback)
./rollback-fw1.sh
;;

help|*)
cat <<USAGE
Uso:
./start.sh prepare    collect + generate + validate, non applica
./start.sh post       applica FW1^ post-REDIAL-only
./start.sh pre        ripristina FW1 originale
./start.sh status     mostra stato FW1

Comandi singoli:
./start.sh collect
./start.sh generate
./start.sh validate
./start.sh apply
./start.sh rollback

Flusso tipico:
source ./inventory.sh
./start.sh prepare
./start.sh post
./start.sh status
./start.sh pre
USAGE
;;
esac
```

Listing A.10: `start.sh`, the multi-command entry point of the orchestrator.

```
#!/usr/bin/env bash
set -euo pipefail

cd "$(dirname "$0")"

if [ -z "${WORKDIR:-}" ]; then
  export WORKDIR="$PWD/runs/$(date +%Y%m%d-%H%M%S)"
```

```
8  fi
9
10 source ./inventory.sh
11
12 echo
13 echo "
       ================================================================================"
14 echo "[run] REDIAL FW1-only"
15 echo "
       ================================================================================"
16 echo "[run] WORKDIR=$WORKDIR"
17 echo "[run] FW1=$SSH_USER@$FW1_MGMT netns=$FW1_NETNS"
18 echo "[run] d0=$D0_CIDR"
19 echo "[run] dn=$DN_CIDR"
20 echo "[run] d2=$D2_CIDR"
21 echo "
       ================================================================================"
22 echo
23
24 echo "[run] Step 1/4: collect FW1 originale"
25 ./collect-fw1.sh
26
27 echo
28 echo "[run] Step 2/4: genera FW1^ post-REDIAL-only"
29 ./generate-fw1-only.sh
30
31 echo
32 echo "[run] Step 3/4: valida ruleset con iptables-restore --test"
33 ./validate-fw1-only.sh
34
35 echo
36 echo "[run] Step 4/4: applica FW1^ su FW1"
37 ./apply-fw1-only.sh
38
39 echo
40 echo "
       ================================================================================"
41 echo "[run] COMPLETATO"
42 echo "
       ================================================================================"
43 echo "[run] FW1 e' ora in configurazione POST REDIAL FW1-only"
44 echo "[run] Backup PRE salvato in:"
45 echo "      $WORKDIR/fw1.backup.rules"
46 echo "[run] Ruleset generato salvato in:"
47 echo "      $WORKDIR/fw1.redial-only.rules"
48 echo
49 echo "[run] Per tornare al PRE:"
50 echo "      export WORKDIR=$WORKDIR"
51 echo "      ./start.sh pre"
```

```
echo "
    ================================================================================"
```

Listing A.11: `run-fw1-only.sh`, the one-command apply: collect, generate, validate, apply. This is the command invoked during the REDIAL pause of every campaign.

The listings above are what the campaigns ran. Listed below for reference is the full implementation of the redistribution instead: it covers the general multi-firewall case of Section 2.2, in which each downstream firewall inherits the rules FW1 gives up, and the paper's `goto` into the middle of a chain, which iptables cannot express, is realized with one custom chain per redistribution target (`REDIAL_A_<j>`), with the inherited `ACCEPT` rules turned into `RETURN` so that control falls back to the caller chain — exactly the mechanism of Section 5.2. The jump that feeds each chain matches the ingress interface of the downstream firewall (`-i`) rather than the source address (`-s d0`), so it also captures traffic from other subnets that crosses FW1 toward the offloaded domain. One design point of the final version is worth recording: earlier iterations moved only the rule classes not already covered by the default policy of the downstream firewall, while the version listed here always moves both classes, because the relative order of `ACCEPT` and `DROP` rules in FW1 is significant under first-match-wins. None of this machinery survives on the measured path of the FW1-only scenario: with no downstream firewall to inherit rules, no jump exists, and the redistribution reduces to the shortcut of Listing A.7. The counters of Chapter 7 confirm it: in every POST run $\hat{\sigma}_1^K$ equals h exactly, which a custom chain on the net2 path would make impossible. Rule parsing and serialization are handled by `RedialUtils` (v4.2, which adds the `in_iface` field emitted as `-i` by the jump rule), not listed here.

```
import ipaddress
from copy import deepcopy
from typing import List, Literal, Optional, Tuple

from redialUtils import (
    Firewall, Table, Rule, ChainDef, Counters,
    parse_iptables_xml_to_firewall, firewall_to_iptables_save,
    convert_rules_txt_to_xml,
)

_NON_TERMINATING = {
    "LOG", "NFLOG", "AUDIT", "MARK", "CONNMARK", "TRACE", "TEE", "CT",
}


def _norm(action: Optional[str]) -> str:
    return (action or "").strip().upper()

```

```
19
20 def _is_allow(action: Optional[str]) -> bool:
21     return _norm(action) in ("ACCEPT", "ALLOW")
22
23
24 def _is_deny(action: Optional[str]) -> bool:
25     return _norm(action) in ("DROP", "DENY", "REJECT")
26
27
28 def _is_nonterminating(action: Optional[str]) -> bool:
29     return _norm(action) in _NON_TERMINATING
30
31
32 def get_forward_policy(fw: Firewall) -> str:
33     for table in fw.tables:
34         if table.table_name != "filter":
35             continue
36         for cd in table.chain_defs:
37             if cd.chain_id == "FORWARD":
38                 return _norm(cd.ext_chain_policy) or "DROP"
39     return "DROP"
40
41
42 def decide_inheritance(fw1_policy: str, fw2_policy: str) -> Tuple[bool,
    bool]:
43     return (True, True)
44
45
46 def _make_custom_chain(name: str) -> ChainDef:
47     return ChainDef(
48         colon=":",
49         chain_id=name,
50         ext_chain_policy="-",
51         counters=Counters(packets=0, bytes=0),
52     )
53
54 def redial(
55     fw1: Firewall,
56     fw2_list: List[Firewall],
57     d2_list: List[ipaddress.IPv4Network],
58     d0: ipaddress.IPv4Network,
59     dn: List[ipaddress.IPv4Network],
60     custom_chain_prefix: str = "REDIAL_A_",
61     log_strategy: Literal["move", "keep", "duplicate"] = "move",
62     fw2_upstream_ifaces: Optional[List[Optional[str]]] = None,
63 ):
64     K = len(d2_list)
65     assert len(fw2_list) == K, "fw2_list e d2_list devono avere la stessa
    lunghezza"
```

```
66      assert log_strategy in ("move", "keep", "duplicate"), \
67          f"log_strategy invalido: {log_strategy!r}"
68
69      if fw2_upstream_ifaces is None:
70          fw2_upstream_ifaces = [None] * K
71      assert len(fw2_upstream_ifaces) == K, \
72          "fw2_upstream_ifaces deve avere la stessa lunghezza di fw2_list"
73
74      fw1_hat = Firewall()
75      fw2_hat: List[Firewall] = [Firewall() for _ in range(K)]
76
77      if not fw1.tables:
78          return fw1_hat, fw2_hat
79
80      orig_table = fw1.tables[0]
81      fw1_policy = get_forward_policy(fw1)
82
83      fw1_new = Table(star="*", table_name=orig_table.table_name)
84      fw1_new.chain_defs = deepcopy(orig_table.chain_defs)
85
86      inherited: List[List[Rule]] = [[] for _ in range(K)]
87      shortcut_inserted = False
88
89      for r in orig_table.rules:
90          dst = r.dst
91          moved = False
92          is_nonterm = _is_nonterminating(r.action)
93          move_this = (not is_nonterm) or (log_strategy in ("move", "
    duplicate"))
94
95          for j in range(K):
96              if dst is None or dst.overlaps(d2_list[j]):
97                  if move_this:
98                      inherited[j].append(deepcopy(r))
99                  moved = True
100
101         if moved and not shortcut_inserted:
102             shortcut_inserted = True
103             for d in d2_list:
104                 fw1_new.rules.append(Rule(
105                     chain="FORWARD", src=None, dst=d,
106                     proto=None, sport=None, dport=None,
107                     icmp_type=None, action="ACCEPT",
108                 ))
109
110         keeps_in_fw1 = (dst is None) or any(dst.overlaps(x) for x in dn)
111         if is_nonterm and log_strategy in ("keep", "duplicate"):
112             keeps_in_fw1 = True
113         if keeps_in_fw1:
```

```
114             fw1_new.rules.append(deepcopy(r))
115
116     fw1_hat.tables.append(fw1_new)
117
118     for j in range(K):
119         orig_fw2 = fw2_list[j]
120         orig_fw2_tbl = orig_fw2.tables[0] if orig_fw2.tables else None
121         fw2_policy = get_forward_policy(orig_fw2)
122         move_accept, move_deny = decide_inheritance(fw1_policy, fw2_policy)
123         upstream_iface = (fw2_upstream_ifaces[j] or "").strip() or None
124
125         new_tbl = Table(star="*", table_name=orig_table.table_name)
126
127         if orig_fw2_tbl is not None:
128             new_tbl.chain_defs = deepcopy(orig_fw2_tbl.chain_defs)
129         chain_name = f"{custom_chain_prefix}{j}"
130         new_tbl.chain_defs.append(_make_custom_chain(chain_name))
131
132         for r in inherited[j]:
133             if _is_nonterminating(r.action):
134                 new_r = deepcopy(r)
135                 new_r.chain = chain_name
136                 new_tbl.rules.append(new_r)
137                 continue
138
139             if _is_allow(r.action):
140                 if not move_accept:
141                     continue
142                 new_r = deepcopy(r)
143                 new_r.chain = chain_name
144                 new_r.action = "RETURN"
145                 new_r.action_params = {}
146                 new_tbl.rules.append(new_r)
147                 continue
148
149             if _is_deny(r.action):
150                 if not move_deny:
151                     continue
152                 new_r = deepcopy(r)
153                 new_r.chain = chain_name
154                 a = _norm(r.action)
155                 new_r.action = a if a in ("DROP", "REJECT") else "DROP"
156                 if new_r.action == "DROP":
157                     new_r.action_params = {}
158                 new_tbl.rules.append(new_r)
159                 continue
160
161         if upstream_iface is not None:
162             jump_rule = Rule(
```

```
                chain="FORWARD",
                src=None, dst=None,
                in_iface=upstream_iface,
                proto=None, sport=None, dport=None,
                icmp_type=None,
                action=chain_name,
            )
        else:
            jump_rule = Rule(
                chain="FORWARD",
                src=d0, dst=None,
                proto=None, sport=None, dport=None,
                icmp_type=None,
                action=chain_name,
            )
        new_tbl.rules.append(jump_rule)

        if orig_fw2_tbl is not None:
            for r in orig_fw2_tbl.rules:
                new_tbl.rules.append(deepcopy(r))

        fw2_hat[j].tables.append(new_tbl)

    return fw1_hat, fw2_hat

if __name__ == "__main__":
    convert_rules_txt_to_xml(rules_txt="rules20250716.rules", rules_xml="
    rules.xml")
    fw1 = parse_iptables_xml_to_firewall("rules.xml")

    fw2_list = [
        parse_iptables_xml_to_firewall("fw2_1.xml"),
    ]

    d0 = ipaddress.IPv4Network("10.0.0.0/24", strict=False)
    d2_list = [ipaddress.IPv4Network("10.0.2.0/24", strict=False)]
    dn = [ipaddress.IPv4Network("10.0.1.0/24", strict=False)]

    fw1_hat, fw2_hat = redial(
        fw1, fw2_list, d2_list, d0, dn,
        log_strategy="move",
        fw2_upstream_ifaces=["veth-fw2-up"],
    )

    with open("iptables_fw1.rules", "w") as f:
        f.write(firewall_to_iptables_save(fw1_hat))
    for i, fw in enumerate(fw2_hat):
        with open(f"iptables_fw2_{i}.rules", "w") as f:
            f.write(firewall_to_iptables_save(fw))
```

Listing A.12: `redial_v5.2.py`, the full REDIAL implementation: the multi-firewall redistribution with the custom-chain emulation of the paper's `goto`; in this final version the jump feeding each custom chain matches the ingress interface of the downstream firewall, not the source address. Listed for reference; the campaigns of this thesis ran the FW1-only variant of Listing A.7. (Comment and docstring strings transliterated to ASCII for this listing.)

A.7 sigma collector: σ from the iptables counters

The sigma collector implements Section 3.1: it reads the per-rule packet counters of the FORWARD chain over SSH at the end of a run and reconstructs σ_{net1} and σ_{net2} exactly, wildcard rules included, as the REDIAL model requires. Two details of the bench make the per-subnet split non-trivial, and the tool handles both: the wildcard rules cover both destination subnets with a single shared counter, which in the PRE state is split between the subnets according to ρ; and in the POST state the shortcut rule captures also the wildcard traffic addressed to net2, so the tail wildcard counter is attributed to net1 alone. The `compare` command takes the PRE and POST dumps of a pair and computes the measured Γ together with the theoretical bounds $[\Gamma_m, \Gamma_M]$ of Section 3.4. The version listed here is v2.1, the one used for the results of Chapters 7 and 8.

```
./sigma_collector_subnet.py zero
./sigma_collector_subnet.py collect --rho 0.5 -o pre.json
./sigma_collector_subnet.py collect --post --rho 0.5 -o post.json
./sigma_collector_subnet.py compare pre.json post.json
```

```
#!/usr/bin/env python3

import argparse
import getpass
import json
import re
import subprocess
import sys
from datetime import datetime
from ipaddress import ip_network

TERMINATING = {"ACCEPT", "DROP", "REJECT", "RETURN"}
NON_TERMINATING = {"LOG", "NFLOG", "AUDIT", "MARK", "CONNMARK", "TRACE", "
    TEE", "CT"}

```

```
15
16 def parse_cli():
17     p = argparse.ArgumentParser(
18         description="Misura i sigma per subnet (con wildcard) pre/post
    REDIAL su FW1.")
19     p.add_argument("--host", default="192.168.100.103",
20                    help="IP management del FW (default: %(default)s = TS-
    P360/FW1)")
21     p.add_argument("--user", default=getpass.getuser(), help="utente SSH")
22     p.add_argument("-i", "--identity", default=None, help="chiave privata
    SSH")
23     p.add_argument("--netns", default="FW", help="network namespace del
    firewall")
24     p.add_argument("--chain", default="FORWARD", help="chain iptables")
25     p.add_argument("--net1", default="10.0.1.0/24", help="subnet dn (non
    redistribuita)")
26     p.add_argument("--net2", default="10.0.2.0/24", help="subnet d2 (
    redistribuita da REDIAL)")
27     p.add_argument("--no-sudo", action="store_true", help="non anteporre '
    sudo'")
28     p.add_argument("--local", action="store_true", help="esegui iptables in
     locale (sul FW)")
29     sub = p.add_subparsers(dest="cmd", required=True)
30     sub.add_parser("zero", help="azzera i contatori della chain (PRIMA di
    ogni run)")
31     c = sub.add_parser("collect", help="legge i contatori e calcola i sigma
     per subnet")
32     c.add_argument("--post", action="store_true",
33                    help="run post-REDIAL: cerca lo shortcut, ricava h,
    gestisce il leak")
34     c.add_argument("--rho", type=float, default=None,
35                    help="frazione di pacchetti verso net2 impostata sul
    sender; "
36                         "se assente usa il rho misurato dai contatori legit
    ")
37     c.add_argument("--packets-net1", default=None,
38                    help="vettore CSV net1 (per leggere la frazione wildcard
     reale f1)")
39     c.add_argument("--packets-net2", default=None,
40                    help="vettore CSV net2 (per leggere la frazione wildcard
     reale f2)")
41     c.add_argument("-o", "--output", default=None, help="salva il risultato
     in JSON")
42     k = sub.add_parser("compare", help="Gamma effettivo vs Gamma_M/Gamma_av
    /Gamma_m")
43     k.add_argument("pre", help="JSON prodotto da 'collect' (run PRE)")
44     k.add_argument("post", help="JSON prodotto da 'collect --post' (run
    POST)")
45     k.add_argument("--rho", type=float, default=None,
```

```
46                      help="forza rho (default: rho misurato nella run PRE)")
47      return p.parse_args()
48
49
50  def run_ssh(a, remote_cmd):
51      if a.local:
52          cmd = ["sh", "-c", remote_cmd]
53      else:
54          cmd = ["ssh"]
55          if a.identity:
56              cmd += ["-i", a.identity]
57          cmd += [f"{a.user}@{a.host}", remote_cmd]
58      r = subprocess.run(cmd, capture_output=True, text=True, timeout=60)
59      if r.returncode != 0:
60          where = "comando locale" if a.local else "ssh"
61          sys.exit(f"[ERRORE] {where} rc={r.returncode}: {r.stderr.strip()}")
62      return r.stdout
63
64
65  def ipt(a, args_str):
66      sudo = "" if a.no_sudo else "sudo "
67      return run_ssh(a, f"{sudo}ip netns exec {a.netns} iptables {args_str}")
68
69
70  def parse_chain(text, chain):
71      policy, policy_pkts, rules = None, 0, []
72      for line in text.splitlines():
73          m = re.match(r"Chain\s+(\S+)\s+\(policy\s+(\S+)\s+(\d+)\s+packets",
       line)
74          if m:
75              if m.group(1) != chain:
76                  sys.exit(f"[ERRORE] chain inattesa nell'output: {m.group(1)}
      ")
77              policy, policy_pkts = m.group(2), int(m.group(3))
78              continue
79          t = line.split(None, 10)
80          if len(t) < 10 or not t[0].isdigit():
81              continue
82          rest = t[10] if len(t) > 10 else ""
83          cm = re.search(r"/\*\s*(.*?)\s*\*/", rest)
84          comment = cm.group(1) if cm else ""
85          tg = re.search(r"\[REDIAL:(\w+)\]", comment)
86          rules.append(dict(pos=int(t[0]), pkts=int(t[1]), target=t[3],
87                            dst=t[9], comment=comment,
88                            tag=tg.group(1) if tg else None))
89      if policy is None:
90          sys.exit("[ERRORE] header della chain non trovato")
91      return policy, policy_pkts, rules
92
```

```
93
94  def classify(rule, net1, net2):
95      if rule["tag"] == "net1":
96          return "net1"
97      if rule["tag"] == "net2":
98          return "net2"
99      if rule["tag"] in ("wildcard", "final"):
100         return "wildcard"
101     try:
102         d = ip_network(rule["dst"], strict=False)
103     except ValueError:
104         return "wildcard"
105     if d.subnet_of(net2):
106         return "net2"
107     if d.subnet_of(net1):
108         return "net1"
109     return "wildcard"
110
111
112 def fmt(x):
113     return f"{x:.3f}" if x is not None else "n/d"
114
115
116 def wildcard_fraction(path):
117     total = wild = 0
118     with open(path) as fh:
119         header = fh.readline().rstrip("\n").split(",")
120         try:
121             i_rid = header.index("rule_id")
122             i_lbl = header.index("label")
123         except ValueError:
124             sys.exit(f"[ERRORE] {path}: header CSV senza rule_id/label")
125         for line in fh:
126             cols = line.rstrip("\n").split(",")
127             if len(cols) <= max(i_rid, i_lbl):
128                 continue
129             total += 1
130             if cols[i_rid].strip() == "" or cols[i_lbl].strip().startswith(
    "wildcard"):
131                 wild += 1
132     return (wild / total) if total else 0.0
133
134
135 def do_zero(a):
136     ipt(a, f"-Z {a.chain}")
137     policy, _, rules = parse_chain(ipt(a, f"-L {a.chain} -nvx --line-
    numbers"), a.chain)
138     print(f"[OK] contatori azzerati su {a.host} ns={a.netns} chain={a.chain}
     "
```

```
139           f"({len(rules)} regole, policy {policy}). Pronto per la run.")
140
141
142 def do_collect(a):
143     policy, policy_pkts, rules = parse_chain(
144         ipt(a, f"-L {a.chain} -nvx --line-numbers"), a.chain)
145     net1, net2 = ip_network(a.net1), ip_network(a.net2)
146     n_rules = len(rules)
147     stats = {k: dict(pkts=0, weight=0, pos_sum=0, n=0)
148              for k in ("net1", "net2", "wildcard")}
149     for r in rules:
150         if r["target"] in NON_TERMINATING:
151             continue
152         if r["target"] not in TERMINATING and r["pkts"] > 0:
153             print(f"[WARN] target '{r['target']}' (pos {r['pos']}): non
    gestito, "
154                   f"trattato come terminante", file=sys.stderr)
155         c = classify(r, net1, net2)
156         stats[c]["pkts"] += r["pkts"]
157         stats[c]["weight"] += r["pkts"] * r["pos"]
158         stats[c]["pos_sum"] += r["pos"]
159         stats[c]["n"] += 1
160
161     tot_pkts = sum(s["pkts"] for s in stats.values()) + policy_pkts
162     tot_weight = sum(s["weight"] for s in stats.values()) + policy_pkts *
    n_rules
163     sigma_overall = tot_weight / tot_pkts if tot_pkts else None
164
165     def avg(s):
166         return s["weight"] / s["pkts"] if s["pkts"] else None
167
168     def expected(s):
169         return s["pos_sum"] / s["n"] if s["n"] else None
170
171     sigma_net1_legit = avg(stats["net1"])
172     sigma_net2_legit = avg(stats["net2"])
173
174     cw = stats["wildcard"]["pkts"]
175     ww = stats["wildcard"]["weight"]
176
177     unamb = stats["net1"]["pkts"] + stats["net2"]["pkts"]
178     rho_meas = stats["net2"]["pkts"] / unamb if unamb else None
179     rho = a.rho if a.rho is not None else rho_meas
180
181     f1 = f2 = None
182     if a.packets_net1 and a.packets_net2:
183         f1 = wildcard_fraction(a.packets_net1)
184         f2 = wildcard_fraction(a.packets_net2)
185
```

```
186     h = None
187     if a.post:
188         for r in rules:
189             try:
190                 if r["target"] == "ACCEPT" and ip_network(r["dst"], strict=
    False) == net2:
191                     h = r["pos"]
192                     break
193             except ValueError:
194                 continue
195         if h is None:
196             print("[WARN] shortcut verso net2 non trovato: REDIAL e' stato
    applicato?",
197                   file=sys.stderr)
198 
199     if a.post:
200         share2 = 0.0
201     elif rho is None:
202         share2 = None
203     elif f1 is not None and f2 is not None:
204         den = (1 - rho) * f1 + rho * f2
205         share2 = (rho * f2 / den) if den else None
206     else:
207         share2 = rho
208 
209     sigma_net1_incl = sigma_net2_incl = None
210     cw1 = cw2 = None
211     if share2 is not None:
212         cw2 = cw * share2
213         cw1 = cw * (1 - share2)
214         ww2 = ww * share2
215         ww1 = ww * (1 - share2)
216         c1 = stats["net1"]["pkts"] + cw1
217         c2 = stats["net2"]["pkts"] + cw2
218         w1 = stats["net1"]["weight"] + ww1
219         w2 = stats["net2"]["weight"] + ww2
220         sigma_net1_incl = w1 / c1 if c1 else None
221         sigma_net2_incl = w2 / c2 if c2 else None
222 
223     res = dict(
224         timestamp=datetime.now().astimezone().isoformat(timespec="seconds"),
225         host=a.host, netns=a.netns, chain=a.chain,
226         mode="post" if a.post else "pre",
227         n_rules=n_rules, policy=policy, policy_pkts=policy_pkts,
228         pkts=dict(net1=stats["net1"]["pkts"], net2=stats["net2"]["pkts"],
229                   wildcard=cw, total=tot_pkts),
230         rho_measured=rho_meas, rho_used=rho, wildcard_share_net2=share2,
231         h=h,
```

```
232         sigma_net1=sigma_net1_legit, sigma_net2=sigma_net2_legit,
233         sigma_net1_incl=sigma_net1_incl, sigma_net2_incl=sigma_net2_incl,
234         sigma_overall=sigma_overall,
235         expected_net1=expected(stats["net1"]), expected_net2=expected(stats
    ["net2"]),
236         rules_net1=stats["net1"]["n"], rules_net2=stats["net2"]["n"],
237         rules_wildcard=stats["wildcard"]["n"],
238     )
239 
240     lbl = "POST" if a.post else "PRE"
241     print(f"=== sigma_collector_subnet v2.1 [{lbl}]  {a.host}  ns={a.netns}
      chain={a.chain} ===")
242     print(f"|fw|={n_rules} regole   policy {policy} ({policy_pkts} pkt)")
243     print(f"pkt terminati: net1={stats['net1']['pkts']}  net2={stats['net2
    ']['pkts']}  "
244           f"wildcard={cw}  policy={policy_pkts}  tot={tot_pkts}")
245     if tot_pkts == 0:
246         sys.exit("[ERRORE] nessun pacchetto contato: contatori azzerati o
    run non partita?")
247 
248     def line(label, meas, exp_v):
249         extra = ""
250         if meas is not None and exp_v is not None:
251             extra = f"   [atteso {exp_v:.3f}  delta {meas - exp_v:+.3f}]"
252         print(f"   {label} = {fmt(meas)}{extra}")
253 
254     print("\n-- (1) Metodo attuale: sigma per subnet, wildcard ESCLUSE --")
255     line("sigma_net1 (legit)", sigma_net1_legit, res["expected_net1"])
256     line("sigma_net2 (legit)", sigma_net2_legit, res["expected_net2"])
257 
258     print("\n-- (2) Sigma per subnet, wildcard INCLUSE (= sigma del paper,
    eq. 3-12) --")
259     if sigma_net1_incl is None and sigma_net2_incl is None:
260         print("   [n/d] manca rho per ripartire le wildcard del PRE: passa
    --rho")
261     else:
262         if rho is not None:
263             src = "forzato --rho" if a.rho is not None else "misurato dai
    contatori legit"
264             print(f"   rho = {fmt(rho)} ({src})")
265         if a.post:
266             print(f"   POST: le wildcard verso net2 cadono nello shortcut (
    pos h={h}) e "
267                   f"sono gia' dentro sigma_net2; la wildcard di coda = sole
     wildcard di net1.")
268         else:
269             if f1 is not None:
270                 print(f"   frazioni wildcard dai vettori: net1={f1:.4f}
    net2={f2:.4f}")
```

```
271             print(f"   contatore wildcard ripartito: net1={cw1:.0f}
    ({100*(1-share2):.1f}%)  "
272                   f"net2={cw2:.0f} ({100*share2:.1f}%)")
273         print(f"   sigma_net1 (incl. wildcard) = {fmt(sigma_net1_incl)}")
274         print(f"   sigma_net2 (incl. wildcard) = {fmt(sigma_net2_incl)}")
275
276     print("\n-- (3) Totale firewall --")
277     print(f"   sigma_overall (tutti i pkt, esatto) = {fmt(sigma_overall)}")
278     if a.post and h is not None:
279         ok = sigma_net2_legit is not None and abs(sigma_net2_legit - h) <
    1.0
280         print(f"   shortcut net2 in posizione h = {h}  "
281               f"({'OK: sigma_net2 ~ h' if ok else 'controlla: sigma_net2 !=
     h'})")
282
283     if a.output:
284         with open(a.output, "w") as f:
285             json.dump(res, f, indent=2)
286         print(f"[OK] risultato salvato in {a.output}")
287
288
289 def do_compare(a):
290     with open(a.pre) as f:
291         pre = json.load(f)
292     with open(a.post) as f:
293         post = json.load(f)
294     if pre.get("mode") != "pre" or post.get("mode") != "post":
295         print("[WARN] i file non sembrano una coppia pre/post coerente",
    file=sys.stderr)
296     rho = a.rho if a.rho is not None else pre.get("rho_used")
297     s1K, sK1N, s1N = pre.get("sigma_net2"), pre.get("sigma_net1"), pre.get(
    "sigma_overall")
298     s1N_post, sK1N_post = post.get("sigma_overall"), post.get("sigma_net1")
299     s1K_i, sK1N_i = pre.get("sigma_net2_incl"), pre.get("sigma_net1_incl")
300     s1K_i_post, sK1N_i_post = post.get("sigma_net2_incl"), post.get("
    sigma_net1_incl")
301     N, Nh = pre["n_rules"], post["n_rules"]
302     h = post.get("h") or post.get("sigma_net2")
303     missing = [n for n, v in [("rho", rho), ("sigma_1^N pre", s1N),
304                               ("sigma_1^K pre", s1K), ("sigma_K+1^N pre",
    sK1N),
305                               ("sigma_1^N post", s1N_post), ("h", h)] if v
    is None]
306     if missing:
307         sys.exit(f"[ERRORE] valori mancanti per il confronto: {', '.join(
    missing)}")
308
309     rho_paper = rho
310     share2 = pre.get("wildcard_share_net2")
```

```
311     pk = pre.get("pkts") or {}
312     if a.rho is None and share2 is not None:
313         denom = pk.get("net1", 0) + pk.get("net2", 0) + pk.get("wildcard",
        0)
314         if denom:
315             rho_paper = (pk.get("net2", 0) + pk.get("wildcard", 0) * share2
        ) / denom
316 
317     def bounds_and_gammas(sk1n_val, rho_val, s1n_pre_val):
318         best = sk1n_val - (N - Nh)
319         unif = sk1n_val * Nh / N
320         worst = sk1n_val + 1
321 
322         def g(shat):
323             return s1n_pre_val / (h * rho_val + shat * (1 - rho_val))
324 
325         return g(best), g(unif), g(worst)
326 
327     print("=== confronto PRE vs POST (FW1) ===")
328     src = "(forzato da --rho)" if a.rho is not None else "(misurato nella
        run PRE)"
329     print(f"rho legit = {fmt(rho)} {src}   rho paper incl. wildcard = {fmt(
        rho_paper)}")
330     print(f"|fw1| = {N}   |fw1^| = {Nh}   h = {h}")
331 
332     print("\n-- (A) Sigma del paper (per destinazione, wildcard INCLUSE) --
        ")
333     if None in (s1K_i, sK1N_i, sK1N_i_post):
334         print("   [n/d] JSON senza sigma incl. wildcard: rigenera i file
        con collect v2.x")
335     else:
336         s1N_i = s1K_i * rho_paper + sK1N_i * (1 - rho_paper)
337         s1N_i_post = h * rho_paper + sK1N_i_post * (1 - rho_paper)
338         gamma_paper = s1N_i / s1N_i_post
339         gM, gav, gm = bounds_and_gammas(sK1N_i, rho_paper, s1N_i)
340         print(f"PRE :  sigma_1^K={fmt(s1K_i)}   sigma_K+1^N={fmt(sK1N_i)}
             "
341               f"sigma_1^N={fmt(s1N_i)}  (eq.3)")
342         print(f"POST:  sigma^_1^K={fmt(s1K_i_post)} (atteso = h = {h})   "
343               f"sigma^_K+1^N={fmt(sK1N_i_post)}   sigma^_1^N={fmt(
        s1N_i_post)}  (eq.4)")
344         print(f"Gamma paper (eq.5) = {gamma_paper:8.3f}   <-- confronto coi
             limiti")
345         print(f"Gamma_M  (eq.10) = {gM:8.3f}   [migliore: regole ereditate
        in testa]")
346         print(f"Gamma_av (eq.11) = {gav:8.3f}   [distribuzione uniforme]")
347         print(f"Gamma_m  (eq.12) = {gm:8.3f}   [peggiore: regole ereditate
        in coda]")
348         if gm - 1e-6 <= gamma_paper <= gM + 1e-6:
```

```
349             print("[OK] Gamma paper dentro l'intervallo teorico [Gamma_m,
    Gamma_M]")
350         else:
351             print("[ATTENZIONE] Gamma paper FUORI dall'intervallo teorico!"
    )
352
353     print("\n-- (B) Scomposizione legit-only (classi net1/net2, wildcard
    ESCLUSE) --")
354     s1N_legit = s1K * rho + sK1N * (1 - rho)
355     s1N_post_pred = h * rho + (sK1N_post or 0) * (1 - rho)
356     gamma_legit = s1N_legit / s1N_post_pred
357     gM_l, gav_l, gm_l = bounds_and_gammas(sK1N, rho, s1N_legit)
358     print(f"PRE :  sigma_1^K={fmt(s1K)}   sigma_K+1^N={fmt(sK1N)}   sigma_1
    ^N={fmt(s1N_legit)}")
359     print(f"POST:  sigma^_1^K={fmt(post.get('sigma_net2'))}   "
360           f"sigma^_K+1^N={fmt(sK1N_post)}   sigma^_1^N={fmt(s1N_post_pred)}
    ")
361     print(f"Gamma legit = {gamma_legit:8.3f}   [range legit: {gm_l:.3f} ..
    {gM_l:.3f}, "
362           f"Gamma_av {gav_l:.3f}]")
363
364     print("\n-- (C) Totale misurato (tutti i pkt, incl. policy) --")
365     print(f"sigma_1^N PRE = {fmt(s1N)}   sigma^_1^N POST = {fmt(s1N_post)}
      "
366           f"Gamma totale = {s1N / s1N_post:8.3f}")
367
368
369 def main():
370     a = parse_cli()
371     if a.cmd == "zero":
372         do_zero(a)
373     elif a.cmd == "collect":
374         do_collect(a)
375     elif a.cmd == "compare":
376         do_compare(a)
377
378
379 if __name__ == "__main__":
380     main()
```

Listing A.13: `sigma_collector_subnet.py` v2.1, the σ collector: per-subnet counters with wildcard attribution, and the measured Γ against its theoretical bounds.

A.8 compare-redial: per-packet latency from the captures

`compare-redial` implements Section 3.2: it pairs the TAP and RX captures packet by packet and computes the per-packet delay Δ, corrected with the session offset of Section 6.1. The pairing copes with the wrap-around of the packet vectors: the sequence number written in the payload repeats once per vector lap, so for every reception at RX the tool takes the latest transmission seen at the TAP within a window (`-max-delay-ms`) that is orders of magnitude longer than the firewall delay and orders of magnitude shorter than the lap period; a lost copy does not desynchronize the others. Packets addressed to the wildcard DROP ports never reach RX by construction: with `-rules` they are counted apart as expected drops and excluded from the loss figure, while a wildcard packet seen at RX is reported as a leak, which the acceptance checks of Section 7.2 require to be zero. The output is a per-step, per-subnet table of counts and delay statistics, plus the aggregated `tot` row that the campaigns quote as D.

```
./compare-redial.py --tap tap_X.pcap --rx rx_X.pcap \
    --ramp 60:100000 --rx-offset-ns -11764 --rules fw1.rules \
    --csv delta.csv
./compare-redial.py --selftest
```

```
#!/usr/bin/env python3

import argparse
import ipaddress
import os
import re
import statistics
import struct
import sys
from collections import defaultdict

RE_WILD = re.compile(r'--dport (\d+) -j DROP')


def load_wildcard_ports(path):
    ports = set()
    with open(path) as fh:
        for line in fh:
            if '[REDIAL:wildcard]' in line:
                m = RE_WILD.search(line)
                if m:
                    ports.add(int(m.group(1)))
```

```
23      return ports
24
25  MAGIC_US = 0xa1b2c3d4
26  MAGIC_NS = 0xa1b23c4d
27
28
29  def _l3_offset(pkt, linktype):
30      if linktype == 1:
31          if len(pkt) < 14:
32              return None
33          eth = struct.unpack('>H', pkt[12:14])[0]
34          o = 14
35          while eth in (0x8100, 0x88a8):
36              if len(pkt) < o + 4:
37                  return None
38              eth = struct.unpack('>H', pkt[o + 2:o + 4])[0]
39              o += 4
40          return o if eth == 0x0800 else None
41      if linktype == 101:
42          return 0
43      if linktype == 113:
44          if len(pkt) < 16:
45              return None
46          return 16 if struct.unpack('>H', pkt[14:16])[0] == 0x0800 else None
47      if linktype == 276:
48          if len(pkt) < 20:
49              return None
50          return 20 if struct.unpack('>H', pkt[0:2])[0] == 0x0800 else None
51      return None
52
53
54  def _classify(dst_int, nets):
55      for net_id, net in nets:
56          if int(net.network_address) <= dst_int <= int(net.broadcast_address
        ):
57              return net_id
58      return None
59
60
61  def parse_packet(pkt, linktype, seq_off, seq_len, nets):
62      o = _l3_offset(pkt, linktype)
63      if o is None or len(pkt) < o + 20:
64          return None
65      vihl = pkt[o]
66      if (vihl >> 4) != 4:
67          return None
68      ihl = (vihl & 0x0f) * 4
69      if pkt[o + 9] != 17:
70          return None
```

```
    dst_int = struct.unpack('>I', pkt[o + 16:o + 20])[0]
    net_id = _classify(dst_int, nets)
    if net_id is None:
        return None
    uo = o + ihl
    if len(pkt) < uo + 8:
        return None
    dport = struct.unpack('>H', pkt[uo + 2:uo + 4])[0]
    payload = pkt[uo + 8:]
    if len(payload) < seq_off + seq_len:
        return None
    seq = int.from_bytes(payload[seq_off:seq_off + seq_len], 'big')
    return net_id, seq, dport

def read_pcap(path, seq_off, seq_len, nets, wildcard_ports):
    with open(path, 'rb') as f:
        gh = f.read(24)
        if len(gh) < 24:
            raise ValueError('%s: header pcap troncato' % path)
        magic = struct.unpack('<I', gh[:4])[0]
        if magic in (MAGIC_US, MAGIC_NS):
            endian, nano = '<', (magic == MAGIC_NS)
        else:
            magic = struct.unpack('>I', gh[:4])[0]
            if magic in (MAGIC_US, MAGIC_NS):
                endian, nano = '>', (magic == MAGIC_NS)
            else:
                raise ValueError('%s: non e\' un pcap classico' % path)
        linktype = struct.unpack(endian + 'I', gh[20:24])[0]
        rec = struct.Struct(endian + 'IIII')
        data = f.read()
    out = defaultdict(lambda: {'acc': [], 'deny': []})
    div = 1e9 if nano else 1e6
    off, n = 0, len(data)
    while off + 16 <= n:
        ts_sec, ts_frac, incl, _orig = rec.unpack_from(data, off)
        off += 16
        if off + incl > n:
            break
        res = parse_packet(data[off:off + incl], linktype, seq_off, seq_len,
 nets)
        off += incl
        if res is not None:
            net_id, seq, dport = res
            kind = 'deny' if dport in wildcard_ports else 'acc'
            out[net_id][kind].append((seq, ts_sec + ts_frac / div))
    for k in out:
        out[k]['acc'].sort(key=lambda x: x[1])
        out[k]['deny'].sort(key=lambda x: x[1])
```

```
119     return out
120
121 def pair_stream(tap, rx, max_delay_s, rx_offset_s):
122     tap_by_seq = defaultdict(list)
123     for seq, ts in tap:
124         tap_by_seq[seq].append(ts)
125     rx_by_seq = defaultdict(list)
126     for seq, ts in rx:
127         rx_by_seq[seq].append(ts)
128     pairs = []
129     for seq, rxts in rx_by_seq.items():
130         tts = tap_by_seq.get(seq)
131         if not tts:
132             continue
133         tts = sorted(tts)
134         rxts = sorted(rxts)
135         used = [False] * len(tts)
136         i = 0
137         for tr in rxts:
138             tr_adj = tr - rx_offset_s
139             while i + 1 < len(tts) and tts[i + 1] <= tr_adj:
140                 i += 1
141             if (not used[i]) and tts[i] <= tr_adj and (tr_adj - tts[i]) <=
    max_delay_s:
142                 pairs.append((seq, tts[i], tr, tr_adj - tts[i]))
143                 used[i] = True
144     return pairs
145
146 def parse_ramp(spec):
147     dur_s, rates_s = spec.split(':', 1)
148     dur = int(dur_s)
149     rates = [int(x) for x in rates_s.split(',') if x != '']
150     return dur, rates
151
152 def step_of(ts, t0, dur, n_steps):
153     if dur <= 0 or n_steps <= 0:
154         return 0
155     s = int((ts - t0) // dur)
156     return max(0, min(n_steps - 1, s))
157
158 def pct(sorted_vals, q):
159     if not sorted_vals:
160         return None
161     idx = min(len(sorted_vals) - 1, int(q * (len(sorted_vals) - 1)))
162     return sorted_vals[idx]
163
164 def analyze(tap_by_net, rx_by_net, dur, rates, max_delay_s, rx_offset_s):
165     n_steps = len(rates) if rates else 1
166     all_tap_ts = [ts for d in tap_by_net.values()
```

```
                for lst in (d['acc'], d['deny']) for _s, ts in lst]
    t0 = min(all_tap_ts) if all_tap_ts else 0.0
    rows = []
    per_pkt = []
    empty = {'acc': [], 'deny': []}
    tot_acc_cnt = [0] * n_steps
    tot_deny_cnt = [0] * n_steps
    tot_leak_cnt = [0] * n_steps
    tot_matched = [0] * n_steps
    tot_deltas = [[] for _ in range(n_steps)]
    for net_id in sorted(set(list(tap_by_net.keys()) + list(rx_by_net.keys
    ()))):
        tap = tap_by_net.get(net_id, empty)
        rx = rx_by_net.get(net_id, empty)
        pairs = pair_stream(tap['acc'], rx['acc'], max_delay_s, rx_offset_s
    )
        acc_cnt = [0] * n_steps
        deny_cnt = [0] * n_steps
        leak_cnt = [0] * n_steps
        for _seq, ts in tap['acc']:
            acc_cnt[step_of(ts, t0, dur, n_steps)] += 1
        for _seq, ts in tap['deny']:
            deny_cnt[step_of(ts, t0, dur, n_steps)] += 1
        for _seq, ts in rx['deny']:
            leak_cnt[step_of(ts, t0, dur, n_steps)] += 1
        deltas = [[] for _ in range(n_steps)]
        matched = [0] * n_steps
        for seq, t_tap, t_rx, d in pairs:
            s = step_of(t_tap, t0, dur, n_steps)
            deltas[s].append(d)
            matched[s] += 1
            per_pkt.append((net_id, s, seq, '%.9f' % t_tap, '%.9f' % t_rx, '
    %.1f' % (d * 1e9)))
        for s in range(n_steps):
            tot_acc_cnt[s] += acc_cnt[s]
            tot_deny_cnt[s] += deny_cnt[s]
            tot_leak_cnt[s] += leak_cnt[s]
            tot_matched[s] += matched[s]
            tot_deltas[s].extend(deltas[s])
            ds = sorted(deltas[s])
            cnt = len(ds)
            mean = statistics.fmean(ds) if cnt else None
            std = statistics.pstdev(ds) if cnt > 1 else 0.0
            med = statistics.median(ds) if cnt else None
            p99 = pct(ds, 0.99)
            rate = rates[s] if rates else 0
            ta = acc_cnt[s]
            loss = ta - matched[s]
            loss_pct = (100.0 * loss / ta) if ta else 0.0
```

```
213             rows.append({
214                 'net': net_id, 'step': s + 1, 'rate_pps': rate,
215                 'tap_acc': ta, 'deny_fw': deny_cnt[s],
216                 'rx_matched': matched[s], 'leak': leak_cnt[s],
217                 'loss': loss, 'loss_pct': round(loss_pct, 4),
218                 'delta_mean_ns': None if mean is None else round(mean * 1e9,
     1),
219                 'delta_std_ns': round(std * 1e9, 1),
220                 'delta_median_ns': None if med is None else round(med * 1e9,
     1),
221                 'delta_p99_ns': None if p99 is None else round(p99 * 1e9,
    1),
222             })
223     for s in range(n_steps):
224         ds = sorted(tot_deltas[s])
225         cnt = len(ds)
226         mean = statistics.fmean(ds) if cnt else None
227         std = statistics.pstdev(ds) if cnt > 1 else 0.0
228         med = statistics.median(ds) if cnt else None
229         p99 = pct(ds, 0.99)
230         rate = rates[s] if rates else 0
231         ta = tot_acc_cnt[s]
232         loss = ta - tot_matched[s]
233         loss_pct = (100.0 * loss / ta) if ta else 0.0
234         rows.append({
235             'net': 0, 'step': s + 1, 'rate_pps': rate,
236             'tap_acc': ta, 'deny_fw': tot_deny_cnt[s],
237             'rx_matched': tot_matched[s], 'leak': tot_leak_cnt[s],
238             'loss': loss, 'loss_pct': round(loss_pct, 4),
239             'delta_mean_ns': None if mean is None else round(mean * 1e9, 1),

240             'delta_std_ns': round(std * 1e9, 1),
241             'delta_median_ns': None if med is None else round(med * 1e9, 1),

242             'delta_p99_ns': None if p99 is None else round(p99 * 1e9, 1),
243         })
244     return rows, per_pkt
245
246 def write_csv(rows, path):
247     import csv
248     cols = ['net', 'step', 'rate_pps', 'tap_acc', 'deny_fw', 'rx_matched',
249             'leak', 'loss', 'loss_pct', 'delta_mean_ns', 'delta_std_ns',
250             'delta_median_ns', 'delta_p99_ns']
251     with open(path, 'w', newline='') as f:
252         w = csv.DictWriter(f, fieldnames=cols)
253         w.writeheader()
254         for r in rows:
255             w.writerow(r)
256
```

```
257 def write_per_pkt(per_pkt, path):
258     import csv
259     with open(path, 'w', newline='') as f:
260         w = csv.writer(f)
261         w.writerow(['net', 'step', 'seq', 't_tap', 't_rx', 'delta_ns'])
262         w.writerows(per_pkt)
263 
264 def make_plots(rows, tap_by_net, rx_by_net, dur, rates, max_delay_s,
265                rx_offset_s, outdir, log_scale):
266     import matplotlib
267     matplotlib.use('Agg')
268     import matplotlib.pyplot as plt
269     os.makedirs(outdir, exist_ok=True)
270     nets = sorted(set(r['net'] for r in rows))
271     lbl = lambda n: 'tot' if n == 0 else 'net%d' % n
272 
273     fig, ax = plt.subplots(figsize=(8, 5))
274     for net_id in nets:
275         rr = [r for r in rows if r['net'] == net_id and r['delta_mean_ns']
    is not None]
276         if not rr:
277             continue
278         x = [r['rate_pps'] for r in rr]
279         y = [r['delta_mean_ns'] for r in rr]
280         e = [r['delta_std_ns'] for r in rr]
281         ax.errorbar(x, y, yerr=e, marker='o', capsize=3, label=lbl(net_id))
282     if log_scale:
283         ax.set_xscale('log')
284     ax.set_xlabel('rate per step (pps, TOTALE)')
285     ax.set_ylabel('Delta = t_rx - t_tap  (ns)')
286     ax.set_title('Ritardo FW TAP->RX per step')
287     ax.grid(True, which='both', alpha=0.3)
288     ax.legend()
289     fig.tight_layout()
290     fig.savefig(os.path.join(outdir, 'delta_vs_step.png'), dpi=130)
291     plt.close(fig)
292 
293     fig, ax = plt.subplots(figsize=(8, 5))
294     for net_id in nets:
295         rr = [r for r in rows if r['net'] == net_id]
296         x = [r['rate_pps'] for r in rr]
297         y = [r['loss_pct'] for r in rr]
298         ax.plot(x, y, marker='s', label=lbl(net_id) + ' loss reale')
299         tot = [r['tap_acc'] + r['deny_fw'] for r in rr]
300         yd = [100.0 * r['deny_fw'] / t if t else 0.0 for r, t in zip(rr,
    tot)]
301         ax.plot(x, yd, marker='v', linestyle='--', label=lbl(net_id) + '
    deny attese')
302     if log_scale:
```

```
303         ax.set_xscale('log')
304     ax.set_xlabel('rate per step (pps, TOTALE)')
305     ax.set_ylabel('loss reale / deny attese (%)')
306     ax.set_title('Perdita TAP->RX per step')
307     ax.grid(True, which='both', alpha=0.3)
308     ax.legend()
309     fig.tight_layout()
310     fig.savefig(os.path.join(outdir, 'loss_vs_step.png'), dpi=130)
311     plt.close(fig)
312 
313     fig, ax = plt.subplots(figsize=(8, 5))
314     for net_id in nets:
315         rr = [r for r in rows if r['net'] == net_id]
316         x = [r['rate_pps'] for r in rr]
317         yin = [r['tap_acc'] / dur if dur else r['tap_acc'] for r in rr]
318         yout = [r['rx_matched'] / dur if dur else r['rx_matched'] for r in
    rr]
319         ax.plot(x, yin, marker='o', linestyle='--', label=lbl(net_id) + '
    in (TAP)')
320         ax.plot(x, yout, marker='x', label=lbl(net_id) + ' out (RX)')
321     if log_scale:
322         ax.set_xscale('log')
323         ax.set_yscale('log')
324     ax.set_xlabel('rate per step (pps, TOTALE)')
325     ax.set_ylabel('throughput (pps)')
326     ax.set_title('Throughput I/O per step')
327     ax.grid(True, which='both', alpha=0.3)
328     ax.legend()
329     fig.tight_layout()
330     fig.savefig(os.path.join(outdir, 'throughput_vs_step.png'), dpi=130)
331     plt.close(fig)
332 
333     fig, ax = plt.subplots(figsize=(8, 5))
334     empty = {'acc': [], 'deny': []}
335     for net_id in nets:
336         pairs = pair_stream(tap_by_net.get(net_id, empty)['acc'],
337                             rx_by_net.get(net_id, empty)['acc'],
338                             max_delay_s, rx_offset_s)
339         ds = sorted(d * 1e9 for _s, _tt, _tr, d in pairs)
340         if not ds:
341             continue
342         ys = [(i + 1) / len(ds) for i in range(len(ds))]
343         ax.plot(ds, ys, label=lbl(net_id))
344     ax.set_xlabel('Delta (ns)')
345     ax.set_ylabel('CDF')
346     ax.set_title('CDF del ritardo TAP->RX')
347     ax.grid(True, alpha=0.3)
348     ax.legend(loc='lower right')
349     fig.tight_layout()
```

```
    fig.savefig(os.path.join(outdir, 'delta_cdf.png'), dpi=130)
    plt.close(fig)


def _write_min_pcap(path, records, linktype=101, nano=True):
    magic = MAGIC_NS if nano else MAGIC_US
    with open(path, 'wb') as f:
        f.write(struct.pack('<IHHiIII', magic, 2, 4, 0, 0, 65535, linktype)
    )
        div = 1e9 if nano else 1e6
        for ts, raw in records:
            sec = int(ts)
            frac = int(round((ts - sec) * div))
            f.write(struct.pack('<IIII', sec, frac, len(raw), len(raw)))
            f.write(raw)


def _mk_udp_ip(dst, seq, sport=1234, dport=5000, payload_len=4):
    src = bytes([10, 0, 0, 1])
    dstb = bytes(int(x) for x in dst.split('.'))
    pl = seq.to_bytes(4, 'big') + b'\x00' * max(0, payload_len - 4)
    udp = struct.pack('>HHHH', sport, dport, 8 + len(pl), 0) + pl
    tot = 20 + len(udp)
    iph = struct.pack('>BBHHHBBH', 0x45, 0, tot, 0, 0, 64, 17, 0) + src +
    dstb
    return iph + udp


def selftest():
    import tempfile
    tmp = tempfile.mkdtemp(prefix='credial_')
    tap_path = os.path.join(tmp, 'tap.pcap')
    rx_path = os.path.join(tmp, 'rx.pcap')
    N, WRAPS = 100, 3
    ACC_PORT, WILD_PORT = 5000, 6666
    delay = 5e-6          # 5 us
    tap_recs, rx_recs = [], []
    t = 1000.0
    lost = denied = 0
    k = 0
    leak_rec = None
    for w in range(WRAPS):
        for seq in range(N):
            for net, dst in ((1, '10.0.1.50'), (2, '10.0.2.50')):
                wild = (seq % 10 == 3)
                raw = _mk_udp_ip(dst, seq, dport=WILD_PORT if wild else
    ACC_PORT)
                tap_recs.append((t, raw))
                if wild:
```

```
                    denied += 1
                    if leak_rec is None and w == 1:
                        leak_rec = (t + delay, raw)
                elif k % 50 == 7:
                    lost += 1
                else:
                    rx_recs.append((t + delay, raw))
                k += 1
                t += 1e-4
    rx_recs.append(leak_rec)
    rx_recs.sort(key=lambda r: r[0])
    _write_min_pcap(tap_path, tap_recs)
    _write_min_pcap(rx_path, rx_recs)

    nets = [(1, ipaddress.ip_network('10.0.1.0/24')),
            (2, ipaddress.ip_network('10.0.2.0/24'))]
    wild_ports = {WILD_PORT}
    tap_by_net = read_pcap(tap_path, 0, 4, nets, wild_ports)
    rx_by_net = read_pcap(rx_path, 0, 4, nets, wild_ports)

    tot_acc = sum(len(v['acc']) for v in tap_by_net.values())
    tot_deny = sum(len(v['deny']) for v in tap_by_net.values())
    assert tot_acc + tot_deny == N * WRAPS * 2, (tot_acc, tot_deny)
    assert tot_deny == denied, (tot_deny, denied)

    ok = True
    matched_total = 0
    for net_id, _ in nets:
        pairs = pair_stream(tap_by_net[net_id]['acc'], rx_by_net[net_id]['
    acc'],
                            1e-3, 0.0)
        matched_total += len(pairs)
        for _seq, _tt, _tr, d in pairs:
            if abs(d - delay) > 1e-9:
                ok = False
    assert matched_total == tot_acc - lost, (matched_total, tot_acc - lost)
    assert ok, 'delta non corrisponde al ritardo iniettato'

    rows, _per_pkt = analyze(tap_by_net, rx_by_net, dur=1, rates=[1],
                             max_delay_s=1e-3, rx_offset_s=0.0)
    per_net_rows = [r for r in rows if r['net'] != 0]
    agg_rows = [r for r in rows if r['net'] == 0]
    tot_loss = sum(r['loss'] for r in per_net_rows)
    tot_leak = sum(r['leak'] for r in per_net_rows)
    tot_deny_rows = sum(r['deny_fw'] for r in per_net_rows)
    assert agg_rows and agg_rows[0]['tap_acc'] == tot_acc, agg_rows
    assert agg_rows[0]['loss'] == tot_loss and agg_rows[0]['deny_fw'] ==
    tot_deny_rows
    assert tot_loss == lost, (tot_loss, lost)
```

```
443     assert tot_deny_rows == denied, (tot_deny_rows, denied)
444     assert tot_leak == 1, tot_leak
445     print('[selftest] OK  acc=%d deny=%d matched=%d loss=%d leak=%d  delta
        =%.0fns'
446           % (tot_acc, tot_deny, matched_total, lost, tot_leak, delay * 1e9)
        )
447     return 0
448
449 def main():
450     ap = argparse.ArgumentParser(description='Differenza timestamp TAP<->RX
         (REDIAL/sendpkts).')
451     ap.add_argument('--tap', help='pcap del TAP')
452     ap.add_argument('--rx', help='pcap del RX')
453     ap.add_argument('--ramp', help='spec rampa DUR:R1,...,RN (per stat per-
        step)')
454     ap.add_argument('--net1-cidr', default='10.0.1.0/24')
455     ap.add_argument('--net2-cidr', default='10.0.2.0/24')
456     ap.add_argument('--seq-offset', type=int, default=0, help='offset del
        seq nel payload (byte)')
457     ap.add_argument('--seq-len', type=int, default=4, help='lunghezza del
        seq (byte, BE)')
458     ap.add_argument('--rx-offset-ns', type=float, default=0.0,
459                     help='offset di clock RX da sottrarre (calibrazione,
        vedi measure_offset)')
460     ap.add_argument('--max-delay-ms', type=float, default=200.0,
461                     help='finestra massima di accoppiamento (deve restare <
         periodo di wrap)')
462     ap.add_argument('--csv', help='output CSV per-step')
463     ap.add_argument('--per-packet-csv', help='output CSV per-pacchetto (
        opzionale)')
464     ap.add_argument('--plot', help='cartella di output per i grafici')
465     ap.add_argument('--log-scale', action='store_true', help='asse x (rate)
         in scala log')
466     ap.add_argument('--rules', help='fw1.rules: dport wildcard -> DENY
        attese (escluse dalla loss)')
467     ap.add_argument('--selftest', action='store_true')
468     args = ap.parse_args()
469
470     if args.selftest:
471         return selftest()
472     if not args.tap or not args.rx:
473         ap.error('--tap e --rx sono obbligatori (oppure usa --selftest)')
474
475     nets = [(1, ipaddress.ip_network(args.net1_cidr)),
476             (2, ipaddress.ip_network(args.net2_cidr))]
477     dur, rates = (0, [])
478     if args.ramp:
479         dur, rates = parse_ramp(args.ramp)
480
```

```
481     wild = load_wildcard_ports(args.rules) if args.rules else set()
482     if args.rules:
483         print('[rules] %d porte wildcard (DENY attese) da %s' % (len(wild),
         args.rules))
484     tap_by_net = read_pcap(args.tap, args.seq_offset, args.seq_len, nets,
        wild)
485     rx_by_net = read_pcap(args.rx, args.seq_offset, args.seq_len, nets,
        wild)
486 
487     max_delay_s = args.max_delay_ms / 1e3
488     rx_offset_s = args.rx_offset_ns / 1e9
489     rows, per_pkt = analyze(tap_by_net, rx_by_net, dur, rates, max_delay_s,
         rx_offset_s)
490 
491     hdr = ('net', 'step', 'rate', 'tap_acc', 'deny', 'rx', 'loss', 'loss%',
492            'leak', 'mean(ns)', 'std(ns)', 'med(ns)', 'p99(ns)')
493     print('%-3s %-4s %-8s %-9s %-8s %-9s %-8s %-7s %-5s %-10s %-9s %-9s %-9
        s' % hdr)
494     for r in rows:
495         print('%-3s %-4d %-8d %-9d %-8d %-9d %-8d %-7.3f %-5d %-10s %-9s
        %-9s %-9s' % (
496             'tot' if r['net'] == 0 else r['net'], r['step'], r['rate_pps'],
         r['tap_acc'], r['deny_fw'],
497             r['rx_matched'], r['loss'], r['loss_pct'], r['leak'],
498             r['delta_mean_ns'], r['delta_std_ns'],
499             r['delta_median_ns'], r['delta_p99_ns']))
500     tot_leak = sum(r['leak'] for r in rows)
501     if tot_leak:
502         print("[WARN] %d pacchetti DENY attesi visti all'RX (leak)!" %
        tot_leak)
503 
504     if args.csv:
505         write_csv(rows, args.csv)
506         print('[csv] ', args.csv)
507     if args.per_packet_csv:
508         write_per_pkt(per_pkt, args.per_packet_csv)
509         print('[csv] ', args.per_packet_csv)
510     if args.plot:
511         make_plots(rows, tap_by_net, rx_by_net, dur if dur else 1, rates,
512                    max_delay_s, rx_offset_s, args.plot, args.log_scale)
513         print('[plot]', args.plot)
514     return 0
515 
516 
517 if __name__ == '__main__':
518     sys.exit(main())
```

Listing A.14: `compare-redial.py`, the latency analyzer: wrap-aware pairing of the TAP/RX captures, per-step and per-subnet delay statistics, expected-drop and leak accounting.

A.9 compare-redial-win: windowed statistics

`compare-redial-win` turns the per-packet delays into the statistics reported in the thesis (Section 5.3). It imports `compare-redial` unchanged, reusing its parser and its pairing, and splits every step of the ramp into fixed-size windows, summarizing each window by its median: an isolated disturbance inflates the global standard deviation of a run while leaving the median of the window means intact, so that median is the robust estimate of D, and a run is flagged as disturbed only when more than a quarter of its windows exceed the dispersion threshold. The per-window CSV is also what Chapter 9 uses to track delay and loss over time during the DoS runs.

```
./compare-redial-win.py --tap tap_X.pcap --rx rx_X.pcap \
    --ramp 60:100000 --rx-offset-ns -11764 --rules fw1.rules \
    --std-warn-us 25 --csv win.csv
./compare-redial-win.py --selftest
```

```
#!/usr/bin/env python3


import argparse
import importlib.util
import ipaddress
import os
import statistics
import sys


def load_base():
    here = os.path.dirname(os.path.abspath(__file__))
    path = os.path.join(here, 'compare-redial.py')
    if not os.path.exists(path):
        sys.exit('[ERRORE] compare-redial.py non trovato accanto a questo script')
    spec = importlib.util.spec_from_file_location('compare_redial', path)
    mod = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(mod)
    return mod
```

```
21
22
23 def window_stats(deltas_by_win, std_warn_ns):
24     rows, means, stds, all_ds = [], [], [], []
25     for w, ds in enumerate(deltas_by_win):
26         ds = sorted(ds)
27         cnt = len(ds)
28         if cnt == 0:
29             rows.append({'win': w + 1, 'count': 0, 'mean_ns': None,
30                          'std_ns': None, 'med_ns': None, 'p99_ns': None,
31                          'flag': ''})
32             continue
33         mean = statistics.fmean(ds) * 1e9
34         std = (statistics.pstdev(ds) if cnt > 1 else 0.0) * 1e9
35         med = statistics.median(ds) * 1e9
36         p99 = ds[min(cnt - 1, int(0.99 * (cnt - 1)))] * 1e9
37         rows.append({'win': w + 1, 'count': cnt, 'mean_ns': round(mean, 1),
38                      'std_ns': round(std, 1), 'med_ns': round(med, 1),
39                      'p99_ns': round(p99, 1), 'flag': ''})
40         means.append(mean)
41         stds.append(std)
42         all_ds.extend(ds)
43     if not means:
44         return rows, None
45     soglia = std_warn_ns if std_warn_ns else 2.0 * statistics.median(stds)
46     soglia = max(soglia, 25000.0)
47     dirty = 0
48     for r in rows:
49         if r['count'] and r['std_ns'] > soglia:
50             r['flag'] = 'SPORCA'
51             dirty += 1
52     summary = {'n_win': len(means),
53                'med_mean_ns': round(statistics.median(means), 1),
54                'med_std_ns': round(statistics.median(stds), 1),
55                'glob_mean_ns': round(statistics.fmean(all_ds) * 1e9, 1),
56                'glob_std_ns': round(statistics.pstdev(all_ds) * 1e9, 1)
57                if len(all_ds) > 1 else 0.0,
58                'dirty': dirty, 'soglia_ns': round(soglia, 1)}
59     return rows, summary
60
61
62 def analyze_windows(cr, tap_by_net, rx_by_net, dur, rates, win_dur,
63                     max_delay_s, rx_offset_s, std_warn_ns):
64     n_steps = len(rates) if rates else 1
65     all_tap_ts = [ts for d in tap_by_net.values()
66                   for lst in (d['acc'], d['deny']) for _s, ts in lst]
67     t0 = min(all_tap_ts) if all_tap_ts else 0.0
68     if dur <= 0:
69         dur = (max(all_tap_ts) - t0) if all_tap_ts else 1.0
```

```
70      n_win = max(1, int(round(dur / win_dur)))
71      empty = {'acc': [], 'deny': []}
72      net_ids = sorted(set(list(tap_by_net.keys()) + list(rx_by_net.keys())))
73      buckets = {n: [[] for _ in range(n_win)] for _ in range(n_steps)]
74                 for n in net_ids + [0]}
75      for net_id in net_ids:
76          pairs = cr.pair_stream(tap_by_net.get(net_id, empty)['acc'],
77                                 rx_by_net.get(net_id, empty)['acc'],
78                                 max_delay_s, rx_offset_s)
79          for _seq, t_tap, _t_rx, d in pairs:
80              s = cr.step_of(t_tap, t0, dur, n_steps)
81              w = int((t_tap - t0 - s * dur) // win_dur)
82              w = max(0, min(n_win - 1, w))
83              buckets[net_id][s][w].append(d)
84              buckets[0][s][w].append(d)
85      results = []
86      for net_id in net_ids + [0]:
87          for s in range(n_steps):
88              rows, summary = window_stats(buckets[net_id][s], std_warn_ns)
89              results.append((net_id, s + 1, rows, summary))
90      return results, n_win
91
92
93  def print_results(results, win_dur):
94      lbl = lambda n: 'tot' if n == 0 else 'net%d' % n
95      hdr = ('net', 'step', 'win', 't(s)', 'count', 'mean(ns)', 'std(ns)',
96             'med(ns)', 'p99(ns)', 'flag')
97      print('%-4s %-4s %-4s %-6s %-9s %-10s %-10s %-10s %-10s %s' % hdr)
98      for net_id, step, rows, _summary in results:
99          for r in rows:
100             print('%-4s %-4d %-4d %-6.1f %-9d %-10s %-10s %-10s %-10s %s' %
     (
101                 lbl(net_id), step, r['win'], (r['win'] - 1) * win_dur,
102                 r['count'], r['mean_ns'], r['std_ns'], r['med_ns'],
103                 r['p99_ns'], r['flag']))
104     print()
105     for net_id, step, _rows, s in results:
106         if s is None:
107             continue
108         print('[win] %s step %d: mediana-finestre mean=%s ns  std=%s ns | '
109               'globale mean=%s ns std=%s ns | finestre sporche %d/%d '
110               '(soglia std %s ns)'
111               % (lbl(net_id), step, s['med_mean_ns'], s['med_std_ns'],
112                  s['glob_mean_ns'], s['glob_std_ns'], s['dirty'],
113                  s['n_win'], s['soglia_ns']))
114         if net_id == 0:
115             if s['dirty'] * 4 <= s['n_win']:
116                 print('[OK]   run utilizzabile: D = mediana delle medie di '
```

```
                      'finestra (%s ns), dispersione = mediana delle std (%
    s ns)'
                      % (s['med_mean_ns'], s['med_std_ns']))
            else:
                print('[WARN] disturbo su piu\' del 25%% delle finestre '
                      '(%d/%d) - run da ripetere' % (s['dirty'], s['n_win'
    ]))


def write_win_csv(results, path, rates, win_dur):
    import csv
    with open(path, 'w', newline='') as f:
        w = csv.writer(f)
        w.writerow(['net', 'step', 'rate_pps', 'win', 't_start_s', 'count',
                    'mean_ns', 'std_ns', 'med_ns', 'p99_ns', 'flag'])
        for net_id, step, rows, _summary in results:
            rate = rates[step - 1] if rates else 0
            for r in rows:
                w.writerow([net_id, step, rate, r['win'],
                            round((r['win'] - 1) * win_dur, 1), r['count'],
                            r['mean_ns'], r['std_ns'], r['med_ns'],
                            r['p99_ns'], r['flag']])


def selftest():
    cr = load_base()
    import tempfile
    tmp = tempfile.mkdtemp(prefix='crwin_')
    tap_path = os.path.join(tmp, 'tap.pcap')
    rx_path = os.path.join(tmp, 'rx.pcap')
    tap_recs, rx_recs = [], []
    t0, k = 1000.0, 0
    for i in range(2000):
        t = t0 + i * 0.01
        dirty = 8.0 <= (t - t0) < 10.0
        for dst in ('10.0.1.50', '10.0.2.50'):
            raw = cr._mk_udp_ip(dst, k)
            tap_recs.append((t, raw))
            jit = (i % 5) * (50e-6 if dirty else 1e-7)
            rx_recs.append((t + 5e-6 + jit, raw))
            k += 1
    cr._write_min_pcap(tap_path, tap_recs)
    cr._write_min_pcap(rx_path, rx_recs)
    nets = [(1, ipaddress.ip_network('10.0.1.0/24')),
            (2, ipaddress.ip_network('10.0.2.0/24'))]
    tap_by_net = cr.read_pcap(tap_path, 0, 4, nets, set())
    rx_by_net = cr.read_pcap(rx_path, 0, 4, nets, set())
    results, n_win = analyze_windows(cr, tap_by_net, rx_by_net, 20, [200],
                                     1.0, 1e-3, 0.0, None)
```

```
164     s = [r[3] for r in results if r[0] == 0][0]
165     assert n_win == 20, n_win
166     assert abs(s['med_mean_ns'] - 5200.0) < 5.0, s
167     assert s['glob_mean_ns'] > 10000.0, s
168     assert s['dirty'] == 2, s
169     print('[selftest] OK  mediana-finestre=%.1f ns (atteso ~5200)  '
170           'globale=%.1f ns (gonfiata dal disturbo)  sporche=%d/20'
171           % (s['med_mean_ns'], s['glob_mean_ns'], s['dirty']))
172     return 0
173
174
175 def main():
176     ap = argparse.ArgumentParser(
177         description='Delta TAP<->RX con statistiche a finestre (REDIAL).')
178     ap.add_argument('--tap', help='pcap del TAP')
179     ap.add_argument('--rx', help='pcap del RX')
180     ap.add_argument('--ramp', help='spec rampa DUR:R1,...,RN (come compare-
    redial.py)')
181     ap.add_argument('--win', type=float, default=5.0,
182                     help='durata finestra in secondi (default 5)')
183     ap.add_argument('--net1-cidr', default='10.0.1.0/24')
184     ap.add_argument('--net2-cidr', default='10.0.2.0/24')
185     ap.add_argument('--seq-offset', type=int, default=0)
186     ap.add_argument('--seq-len', type=int, default=4)
187     ap.add_argument('--rx-offset-ns', type=float, default=0.0)
188     ap.add_argument('--max-delay-ms', type=float, default=200.0)
189     ap.add_argument('--rules', help='fw.rules: dport wildcard -> DENY
    attese')
190     ap.add_argument('--std-warn-us', type=float, default=None,
191                     help='soglia std per finestra sporca in us (40 PRE / 25
     POST); '
192                          'default: 2x la mediana delle std di finestra; '
193                          'pavimento 25 us in ogni caso')
194     ap.add_argument('--csv', help='CSV con una riga per finestra')
195     ap.add_argument('--selftest', action='store_true')
196     args = ap.parse_args()
197
198     if args.selftest:
199         return selftest()
200     if not args.tap or not args.rx:
201         ap.error('--tap e --rx sono obbligatori (oppure usa --selftest)')
202
203     cr = load_base()
204     nets = [(1, ipaddress.ip_network(args.net1_cidr)),
205             (2, ipaddress.ip_network(args.net2_cidr))]
206     dur, rates = (0, [])
207     if args.ramp:
208         dur, rates = cr.parse_ramp(args.ramp)
209     wild = cr.load_wildcard_ports(args.rules) if args.rules else set()
```

```
210     if args.rules:
211         print('[rules] %d porte wildcard (DENY attese) da %s'
212               % (len(wild), args.rules))
213     tap_by_net = cr.read_pcap(args.tap, args.seq_offset, args.seq_len, nets,
         wild)
214     rx_by_net = cr.read_pcap(args.rx, args.seq_offset, args.seq_len, nets,
        wild)
215     std_warn_ns = args.std_warn_us * 1000.0 if args.std_warn_us else None
216     results, _n_win = analyze_windows(cr, tap_by_net, rx_by_net, dur, rates,

217                                       args.win, args.max_delay_ms / 1e3,
218                                       args.rx_offset_ns / 1e9, std_warn_ns)
219     print_results(results, args.win)
220     if args.csv:
221         write_win_csv(results, args.csv, rates, args.win)
222         print('[csv] ', args.csv)
223     return 0
224
225
226 if __name__ == '__main__':
227     sys.exit(main())
```

Listing A.15: `compare-redial-win.py`, the windowed analyzer: per-window statistics and the median-based robust estimate of D.

Appendix B

Topology creation and recovery

This appendix lists the two idempotent scripts that build, verify, and tear down the emulated topology of Section 4.5, one per machine, together with the configurations they apply.

B.1 TS-P5: the measurement host

On the measurement host, `creation-namespace.sh` builds the three namespaces of the measurement path: `SX` with the sender port, `TAP` with the capture port that receives the TAP copy, and `RX` with the port that terminates the path (Section 4.5). The capture ports run in promiscuous mode, RX holds the only address of the destination side, and both default routes point at FW1. The ports are identified by MAC address: interface names may change across cold boots, the MACs cannot.

```
#!/usr/bin/env bash

set -euo pipefail

SX_MAC="<MAC_SX>"
TAP_MAC="<MAC_TAP>"
RX_MAC="e8:ea:6a:0c:74:22"

IP_SX="10.0.0.1/24"
GW_SX="10.0.0.3"
IP_RX="10.0.1.3/24"
GW_RX="10.0.1.1"

need_root() { [ "$(id -u)" -eq 0 ] || { echo "Serve root (sudo)." >&2; exit
    1; }; }
```

```
15
16 pick() {
17     local name=""
18     if [ -z "${2:-}" ]; then
19         name=$(ip -br link | awk -v m="$1" 'tolower($3)==tolower(m){print
       $1; exit}')
20     elif ip netns list | grep -qw "$2"; then
21         name=$(ip netns exec "$2" ip -br link | awk -v m="$1" 'tolower($3)
       ==tolower(m){print $1; exit}')
22     fi
23     [ -n "$name" ] || { echo "[ERR] nessuna interfaccia con MAC $1" >&2;
       exit 1; }
24     echo "$name"
25 }
26
27 move_in() {
28     if ip netns list | grep -qw "$2" && \
29        ip netns exec "$2" ip -br link | awk -v m="$1" 'tolower($3)==tolower
       (m){f=1} END{exit !f}'; then
30         echo "[=] $1 gia' in $2"
31     else
32         local if_root
33         if_root=$(pick "$1")
34         ip link set "$if_root" down 2>/dev/null || true
35         ip addr flush dev "$if_root" 2>/dev/null || true
36         ip link set "$if_root" netns "$2"
37         echo "[+] $if_root ($1) -> $2"
38     fi
39 }
40
41 cmd_up() {
42     need_root
43
44     for ns in SX TAP RX; do
45         if ip netns list | grep -qw "$ns"; then
46             echo "[=] netns $ns esiste gia'"
47         else
48             ip netns add "$ns"; echo "[+] netns $ns creato"
49         fi
50     done
51
52     move_in "$SX_MAC"  SX
53     move_in "$TAP_MAC" TAP
54     move_in "$RX_MAC"  RX
55
56     local if_sx if_tap if_rx
57     if_sx=$(pick "$SX_MAC" SX)
58     if_tap=$(pick "$TAP_MAC" TAP)
59     if_rx=$(pick "$RX_MAC" RX)
```

```
60
61      for ns in SX TAP RX; do ip netns exec "$ns" ip link set lo up; done
62      ip netns exec SX  ip link set "$if_sx" up
63      ip netns exec TAP ip link set "$if_tap" up promisc on
64      ip netns exec RX  ip link set "$if_rx" up promisc on
65
66      ip netns exec SX ip addr flush dev "$if_sx"
67      ip netns exec SX ip addr add "$IP_SX" dev "$if_sx"
68      ip netns exec RX ip addr flush dev "$if_rx"
69      ip netns exec RX ip addr add "$IP_RX" dev "$if_rx"
70
71      ip netns exec SX ip route replace default via "$GW_SX"
72      ip netns exec RX ip route replace default via "$GW_RX"
73
74      echo "[OK] SX/TAP/RX pronti"
75      cmd_status
76  }
77
78  cmd_down() {
79      need_root
80      for ns in SX TAP RX; do
81          if ip netns list | grep -qw "$ns"; then
82              ip netns exec "$ns" ip -br link 2>/dev/null | awk '$1!="lo"{
    print $1}' | \
83              while read -r ifn; do
84                  ip netns exec "$ns" ip link set "$ifn" down 2>/dev/null ||
    true
85                  ip netns exec "$ns" ip link set "$ifn" netns 1
86                  echo "[-] $ifn -> root ns"
87              done
88              ip netns del "$ns"
89              echo "[-] netns $ns rimosso"
90          fi
91      done
92  }
93
94  cmd_status() {
95      for ns in SX TAP RX; do
96          if ip netns list | grep -qw "$ns"; then
97              echo "----- $ns: ip -br addr -----"; ip netns exec "$ns" ip -br
     addr
98              echo "----- $ns: ip route -----";    ip netns exec "$ns" ip
    route
99          else
100             echo "[=] $ns non esiste"
101         fi
102     done
103 }
104
```

```
case "${1:-up}" in
    up)     cmd_up ;;
    down)   cmd_down ;;
    status) cmd_status ;;
    *)      echo "Uso: $0 {up|down|status}" >&2; exit 1 ;;
esac
```

Listing B.1: `creation-namespace.sh` (TS-P5): builds the SX/TAP/RX namespaces, moving the ports by MAC address. (MACs of the SX and TAP ports to be filled in from the bench.)

B.2 TS-P360: the firewall host

On the firewall host, `fw1-up.sh` builds the `FW` namespace: the two 10 GbE ports are moved in, addressed, and given the routes of the emulated topology, with forwarding enabled and the reverse-path filter relaxed on the test interfaces. Both destination subnets are routed toward RX (`via 10.0.1.3 onlink`): in the FW1-only scenario no FW2 exists, and RX absorbs net2 through its promiscuous capture port (Section 4.5). The two ports are identified by MAC address, because on this bench the name-to-role mapping is the reverse of the nominal one and only the MACs are stable: the port toward SX (`10.0.0.3`) is `ec:e7:a7:03:47:cd`, the port toward RX (`10.0.1.1`) is `ec:e7:a7:03:47:cc`. One check matters before the first run: the management address of the host must not live on either of the two ports, or moving them into the namespace cuts the SSH session that drives the script.

```
#!/bin/sh

set -eu

NS=FW

DO_MAC="ec:e7:a7:03:47:cd"
DN_MAC="ec:e7:a7:03:47:cc"

IP_DO=10.0.0.3/24
IP_DN=10.0.1.1/24

NH_RX=10.0.1.3
D2_CIDR=10.0.2.0/24
DN_CIDR=10.0.1.0/24

ns_exec() { ip netns exec "$NS" "$@"; }
need_root() { [ "$(id -u)" -eq 0 ] || { echo "Serve root (sudo)." >&2; exit
    1; }; }

```

```
pick() {
    name=$(ip -br link | awk -v m="$1" 'tolower($3)==tolower(m){print $1; exit}')
    if [ -z "$name" ] && ip netns list | grep -qw "$NS"; then
        name=$(ns_exec ip -br link | awk -v m="$1" 'tolower($3)==tolower(m){print $1; exit}')
    fi
    if [ -z "$name" ]; then
        echo "[ERR] nessuna interfaccia con MAC $1" >&2; exit 1
    fi
    echo "$name"
}

cmd_up() {
    need_root

    # 1. netns
    if ip netns list | grep -qw "$NS"; then
        echo "[=] netns $NS esiste gia'"
    else
        ip netns add "$NS"; echo "[+] netns $NS creato"
    fi

    for MAC in "$DO_MAC" "$DN_MAC"; do
        IF=$(pick "$MAC")
        if ip link show "$IF" >/dev/null 2>&1; then
            ip link set "$IF" down 2>/dev/null || true
            ip addr flush dev "$IF" 2>/dev/null || true
            ip link set "$IF" netns "$NS"
            echo "[+] $IF ($MAC) -> $NS"
        elif ns_exec ip link show "$IF" >/dev/null 2>&1; then
            echo "[=] $IF ($MAC) gia' in $NS"
        else
            echo "[!] interfaccia $IF non trovata" >&2; exit 1
        fi
    done

    IF_DO=$(pick "$DO_MAC")
    IF_DN=$(pick "$DN_MAC")

    ns_exec ip addr flush dev "$IF_DO"
    ns_exec ip addr flush dev "$IF_DN"
    ns_exec ip addr add "$IP_DO" dev "$IF_DO"
    ns_exec ip addr add "$IP_DN" dev "$IF_DN"
    ns_exec ip link set lo up
    ns_exec ip link set "$IF_DO" up
    ns_exec ip link set "$IF_DN" up

    ns_exec sysctl -qw net.ipv4.ip_forward=1
```

```
67      ns_exec sysctl -qw net.ipv4.conf.all.rp_filter=0
68      ns_exec sysctl -qw net.ipv4.conf.default.rp_filter=0
69      ns_exec sysctl -qw "net.ipv4.conf.$IF_D0.rp_filter=0"
70      ns_exec sysctl -qw "net.ipv4.conf.$IF_DN.rp_filter=0"
71
72      ns_exec ip route replace "$NH_RX/32" dev "$IF_DN"
73      ns_exec ip route replace "$DN_CIDR" via "$NH_RX" dev "$IF_DN" onlink
74      ns_exec ip route replace "$D2_CIDR" via "$NH_RX" dev "$IF_DN" onlink
75
76      ns_exec iptables -F
77      ns_exec iptables -t nat    -F 2>/dev/null || true
78      ns_exec iptables -t mangle -F 2>/dev/null || true
79      ns_exec iptables -P FORWARD ACCEPT
80      ns_exec iptables -P INPUT   ACCEPT
81      ns_exec iptables -P OUTPUT  ACCEPT
82
83      echo "[OK] netns $NS pronto"
84      cmd_status
85  }
86
87  cmd_down() {
88      need_root
89      if ip netns list | grep -qw "$NS"; then
90          IF_D0=$(pick "$D0_MAC")
91          IF_DN=$(pick "$DN_MAC")
92          for IF in "$IF_D0" "$IF_DN"; do
93              if ns_exec ip link show "$IF" >/dev/null 2>&1; then
94                  ns_exec ip link set "$IF" down 2>/dev/null || true
95                  ns_exec ip link set "$IF" netns 1
96                  echo "[-] $IF -> root ns"
97              fi
98          done
99          ip netns del "$NS"
100         echo "[-] netns $NS rimosso"
101     else
102         echo "[=] netns $NS non esiste"
103     fi
104 }
105
106 cmd_status() {
107     if ! ip netns list | grep -qw "$NS"; then
108         echo "[=] $NS non esiste"; return 0
109     fi
110     echo "----- ip -br addr -----"; ns_exec ip -br addr
111     echo "----- ip route -----";    ns_exec ip route
112     echo "----- forwarding -----";  ns_exec sysctl -n net.ipv4.ip_forward
113     echo "----- iptables FORWARD -----"; ns_exec iptables -nvL FORWARD
114 }
115
```

```
case "${1:-up}" in
    up)     cmd_up ;;
    down)   cmd_down ;;
    status) cmd_status ;;
    *)      echo "Uso: $0 {up|down|status}" >&2; exit 1 ;;
esac
```

Listing B.2: `fw1-up.sh` (TS-P360): builds the FW namespace with both destination subnets routed toward RX; ports identified by MAC.

B.3 Recovery after a power cycle

Namespaces are runtime state: they do not survive a reboot, while the scripts on disk and the cabling do. The recovery of the whole bench is therefore the two scripts above, one per machine, followed by the smoke test below: net1 must answer three for three with `TTL=63`, the route to net2 must resolve `via 10.0.1.3`, no PTP daemon may be active, and forwarding must be on with an empty FORWARD baseline. The capture-port offset is then remeasured, because the PHCs restart at power-up (Section 6.1).

```
sudo ip netns exec SX ping -c3 10.0.1.3
sudo ip netns exec FW ip route get 10.0.2.55
systemctl is-active ptp4l phc2sys
sudo ip netns exec FW iptables -nvL FORWARD
sudo ip netns exec FW cat /proc/sys/net/ipv4/ip_forward
```

Listing B.3: Smoke test of the recovered topology and pre-measurement hygiene.

Appendix C

Complete result tables

This appendix collects the complete result tables of the matrix campaign: for each of the 27 cells, the per-class σ of the two subnets in the PRE and POST states, the measured delays, and the gains, on the x86 firewall (Chapter 7) and on the Raspberry Pi 5 (Chapter 8).

C.1 x86 firewall at 100 kpps

Tables C.1 to C.3 report the 27 cells of the matrix on the x86 firewall, one table per position of the net2 block. Every cell is a PRE/POST pair of the same session at 100 kpps, six million packets per run, zero loss. G_{net} uses the $D_0 = 34.30\,\mu$s of the global fit (Section 7.4).

Table C.1: REDIAL matrix on the x86 firewall at 100 kpps — ruleset with the net2 rules at the *head* ($h = 1$). σ in rules evaluated; times in µs, mean over 6 000 000 packets per run, zero loss. $G_{\text{net}} = \Delta/(D_{\text{pre}} - D_0)$ with $D_0 = 34.30$ µs (global fit): the measured values match the prediction $1 - 1/\Gamma$ within about one point.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_M	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			354.900			109.909	3.229	3.229	61.00	42.62	18.38	30.1	68.8
`fw1` (50–50)	0.50	158.963	420.212	289.587	1.000	146.213	73.606	3.934	3.934	56.03	39.66	16.36	29.2	75.3
	0.75			224.275			37.303	6.012	6.012	51.22	37.24	13.97	27.3	82.6
	0.25			420.700			56.573	7.436	7.436	66.36	38.75	27.61	41.6	86.1
`fw1_r2575` (25–75)	0.50	224.509	486.097	355.303	1.000	75.097	38.049	9.338	9.338	61.01	37.20	23.81	39.0	89.1
	0.75			289.906			19.524	14.848	14.848	56.09	35.88	20.22	36.1	92.8
	0.25			290.381			164.207	1.768	1.768	55.65	46.37	9.28	16.7	43.5
`fw1_r7525` (75–25)	0.50	94.697	355.609	225.153	1.000	218.609	109.805	2.050	2.050	50.36	41.94	8.41	16.7	52.4
	0.75			159.925			55.402	2.887	2.887	46.15	38.33	7.82	16.9	66.0

Table C.2: REDIAL matrix on the x86 firewall at 100 kpps — *mixed* ruleset (proportional interleaving, $\Gamma \approx \Gamma_{av}$). Same conditions as Table C.1. In the mixed PRE state σ is flat in ρ: Δ and G grow with ρ, the opposite of the head variant.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_{av}	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			288.413			110.157	2.618	2.442	55.88	42.70	13.18	23.6	61.1
`fw1_5050_mixed` (50–50)	0.50	289.125	288.175	288.650	2.000	146.209	74.105	3.895	3.636	55.99	39.77	16.22	29.0	74.8
	0.75			288.887			38.052	7.592	7.098	56.08	37.35	18.72	33.4	85.9
	0.25			290.602			56.818	5.115	4.182	56.07	38.61	17.46	31.1	80.2
`fw1_2575_mixed` (25–75)	0.50	289.595	290.938	290.267	2.000	75.091	38.545	7.531	6.177	55.96	37.16	18.80	33.6	86.8
	0.75			289.931			20.273	14.302	11.837	55.92	35.91	20.01	35.8	92.6
	0.25			289.931			164.703	1.760	1.721	55.56	46.29	9.27	16.7	43.6
`fw1_7525_mixed` (75–25)	0.50	290.966	289.586	290.276	3.000	218.605	110.802	2.620	2.562	55.41	41.98	13.43	24.2	63.6
	0.75			290.621			56.901	5.107	4.998	55.43	38.44	16.99	30.7	80.4

Table C.3: REDIAL matrix on the x86 firewall at 100 kpps — ruleset with the net2 rules at the *tail* ($\Gamma \approx \Gamma_m$, worst case). Same conditions as Table C.1. At $\rho = 0.25$ the Γ_M formula returns a negative value: a known artifact of the bound at large h.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_m	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			224.275			177.947	1.260	1.187	51.06	47.56	3.50	6.9	20.9
`fw1_5050_coda` (50–50)	0.50	420.212	158.963	289.587	276.000	145.262	210.631	1.375	1.328	55.82	49.48	6.34	11.4	29.5
	0.75			354.900			243.316	1.459	1.437	61.08	51.89	9.19	15.0	34.3
	0.25			159.925			90.361	1.770	1.501	46.33	41.09	5.24	11.3	43.6
`fw1_2575_coda` (25–75)	0.50	355.609	94.697	225.153	139.000	74.147	106.574	2.113	1.919	50.73	42.17	8.56	16.9	52.1
	0.75			290.381			122.787	2.365	2.266	55.86	43.36	12.50	22.4	58.0
	0.25			289.906			266.495	1.088	1.064	55.46	53.18	2.28	4.1	10.8
`fw1_7525_coda` (75–25)	0.50	486.097	224.509	355.303	413.000	217.659	315.330	1.127	1.113	59.86	56.64	3.22	5.4	12.6
	0.75			420.700			364.165	1.155	1.149	65.33	60.54	4.79	7.3	15.4

C.2 Raspberry Pi 5 at 2 kpps

Tables C.4 to C.6 report the same 27 cells measured on the Raspberry Pi 5 at 2 kpps, 120 000 packets per run, zero loss (Chapter 8). σ and Γ are identical to the x86 tables: the counters depend only on the rulesets and the traffic vectors, which the porting did not change. G_{net} uses $D_0 = 367.7\,\mu$s from the sweep fit (Section 8.2); with a variable cost of only 27–46 μs, an error of one microsecond on D_0 moves G_{net} by a few points.

Table C.4: REDIAL matrix on the Raspberry Pi 5 at 2 kpps — ruleset with the net2 rules at the *head* ($h = 1$). σ in rules evaluated; times in µs, mean over 120 000 packets per run, zero loss. σ and Γ coincide with the x86 matrix (deterministic counters); the times reflect the USB fixed cost $D_0 \approx 370$ µs and $c_{\mathrm{Pi}} \approx 102$–$110$ ns per rule.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_M	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			354.900			109.909	3.229	3.229	406.10	381.14	24.95	6.1	64.9
`fw1` (50–50)	0.50	158.963	420.212	289.587	1.000	146.213	73.606	3.934	3.934	400.66	376.91	23.75	5.9	72.0
	0.75			224.275			37.303	6.012	6.012	394.51	374.39	20.11	5.1	74.9
	0.25			420.700			56.573	7.436	7.436	413.31	375.39	37.92	9.2	83.1
`fw1_r2575` (25–75)	0.50	224.509	486.097	355.303	1.000	75.097	38.049	9.338	9.338	406.84	373.96	32.88	8.1	83.9
	0.75			289.906			19.524	14.848	14.848	402.04	370.70	31.34	7.8	91.2
	0.25			290.381			164.207	1.768	1.768	401.41	387.64	13.77	3.4	40.8
`fw1_r7525` (75–25)	0.50	94.697	355.609	225.153	1.000	218.609	109.805	2.050	2.050	393.85	381.80	12.04	3.1	46.0
	0.75			159.925			55.402	2.887	2.887	386.30	374.90	11.41	3.0	61.2

Table C.5: REDIAL matrix on the Raspberry Pi 5 at 2 kpps — *mixed* ruleset (proportional interleaving, $\Gamma \approx \Gamma_{av}$). Same conditions as Table C.4. In the mixed PRE state σ is flat in ρ: Δ and G grow with ρ, the opposite of the head variant.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_{av}	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			288.413			110.157	2.618	2.442	401.85	381.67	20.19	5.0	59.1
`fw1_5050_mixed` (50–50)	0.50	289.125	288.175	288.650	2.000	146.209	74.105	3.895	3.636	400.63	377.43	23.21	5.8	70.4
	0.75			288.887			38.052	7.592	7.098	400.05	372.60	27.45	6.9	84.8
	0.25			290.602			56.818	5.115	4.182	401.07	375.70	25.37	6.3	76.0
`fw1_2575_mixed` (25–75)	0.50	289.595	290.938	290.267	2.000	75.091	38.545	7.531	6.177	401.48	372.45	29.03	7.2	85.9
	0.75			289.931			20.273	14.302	11.837	399.93	372.34	27.60	6.9	85.6
	0.25			289.931			164.703	1.760	1.721	399.22	387.09	12.13	3.0	38.4
`fw1_7525_mixed` (75–25)	0.50	290.966	289.586	290.276	3.000	218.605	110.802	2.620	2.562	400.10	381.95	18.15	4.5	56.0
	0.75			290.621			56.901	5.107	4.998	399.62	376.79	22.83	5.7	71.5

Table C.6: REDIAL matrix on the Raspberry Pi 5 at 2 kpps — ruleset with the net2 rules at the *tail* ($\Gamma \approx \Gamma_m$, worst case). Same conditions as Table C.4. At $\rho = 0.25$ the Γ_M formula returns a negative value: the same artifact already observed on the x86 bench.

		PRE			POST					FW latency [µs]				
Ruleset	ρ	σ_1^K	σ_{K+1}^N	σ_1^N	$\hat{\sigma}_1^K$	$\hat{\sigma}_{K+1}^N$	$\hat{\sigma}_1^N$	Γ	Γ_m	D_{pre}	D_{post}	Δ	G [%]	G_{net} [%]
	0.25			224.275			177.947	1.260	1.187	395.08	388.65	6.43	1.6	23.5
`fw1_5050_coda` (50–50)	0.50	420.212	158.963	289.587	276.000	145.262	210.631	1.375	1.328	401.45	391.54	9.92	2.5	29.4
	0.75			354.900			243.316	1.459	1.437	406.40	393.81	12.60	3.1	32.5
	0.25			159.925			90.361	1.770	1.501	385.62	377.59	8.03	2.1	44.7
`fw1_2575_coda` (25–75)	0.50	355.609	94.697	225.153	139.000	74.147	106.574	2.113	1.919	393.22	379.38	13.85	3.5	54.2
	0.75			290.381			122.787	2.365	2.266	399.92	381.68	18.24	4.6	56.6
	0.25			289.906			266.495	1.088	1.064	400.39	397.03	3.36	0.8	10.3
`fw1_7525_coda` (75–25)	0.50	486.097	224.509	355.303	413.000	217.659	315.330	1.127	1.113	407.71	401.61	6.10	1.5	15.2
	0.75			420.700			364.165	1.155	1.149	414.30	406.69	7.60	1.8	16.3

Bibliography

[1] L. Durante, L. Seno, and A. Valenzano. «A Formal Model and Technique to Redistribute the Packet Filtering Load in Multiple Firewall Networks». In: *IEEE Trans. Inf. Forensics Security* 16 (2021), pp. 2637–2651 (cit. on pp. ii, 1, 7, 12, 13, 72).

[2] M. Cheminod, I. Cibrario Bertolotti, A. Valenzano, and L. Durante. «Low-Cost Software Evaluation of Communication Delays in Networked Industrial Systems». In: *Future Internet* 17.2 (2025), p. 78. DOI: 10.3390/fi17020078 (cit. on pp. ii, 9, 10, 14, 18, 21, 27, 74).

[3] M. Cheminod, L. Durante, L. Seno, and A. Valenzano. «Performance Evaluation and Modeling of an Industrial Application-Layer Firewall». In: *IEEE Trans. Ind. Informat.* 14.5 (May 2018), pp. 2159–2170 (cit. on pp. ii, 11, 14, 39, 42, 73).

[4] *Security for Industrial Automation and Control Systems*. Geneva, Switzerland: International Electrotechnical Commission, 2018 (cit. on p. 1).

[5] H. Hamed and E. Al-Shaer. «Dynamic Rule-Ordering Optimization for High-Speed Firewall Filtering». In: *Proc. ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*. 2006, pp. 332–342 (cit. on p. 13).

[6] E. W. Fulp. «Optimization of Network Firewall Policies Using Directed Acyclical Graphs». In: *Proc. IEEE Internet Manage. Conf. (IM)*. Jan. 2005, pp. 1–4 (cit. on p. 13).

[7] A. X. Liu, E. Torng, and C. R. Meiners. «Firewall Compressor: An Algorithm for Minimizing Firewall Policies». In: *Proc. 27th Conf. Comput. Commun. (INFOCOM)*. Apr. 2008, pp. 176–180 (cit. on p. 13).

[8] J. Daly, A. X. Liu, and E. Torng. «A Difference Resolution Approach to Compressing Access Control Lists». In: *IEEE/ACM Trans. Netw.* 24.1 (Feb. 2016), pp. 610–623 (cit. on p. 13).

[9] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. «Conflict Classification and Analysis of Distributed Firewall Policies». In: *IEEE J. Sel. Areas Commun.* 23.10 (Oct. 2005), pp. 2069–2084 (cit. on p. 13).

[10] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, A. Russo, and C. Williams. «Methods and Tools for Policy Analysis». In: *ACM Comput. Surv.* 51.6 (Feb. 2019), pp. 1–35 (cit. on p. 13).

[11] E. W. Fulp. «Parallel Firewall Designs for High-Speed Networks». In: *Proc. IEEE 25th Int. Conf. Comput. Commun. (INFOCOM)*. Apr. 2006, pp. 1–4 (cit. on p. 13).

[12] G. Yan, S. Chen, and S. Eidenbenz. «Dynamic Balancing of Packet Filtering Workloads on Distributed Firewalls». In: *Proc. 16th Int. Workshop Quality of Service (IWQoS)*. June 2008, pp. 209–218 (cit. on p. 13).

[13] S. Bagheri and A. Shameli-Sendi. «Dynamic Firewall Decomposition and Composition in the Cloud». In: *IEEE Trans. Inf. Forensics Security* 15 (Apr. 2020), pp. 3526–3539 (cit. on p. 13).

[14] B. Hickman, D. Newman, S. Tadjudin, and T. Martin. *Benchmarking Methodology for Firewall Performance*. Tech. rep. RFC 3511. Available: `https://www.rfc-editor.org/rfc/rfc3511.txt`. RFC Editor, Apr. 2003 (cit. on p. 14).

[15] K. Salah, K. Elbadawi, and R. Boutaba. «Performance Modeling and Analysis of Network Firewalls». In: *IEEE Trans. Netw. Service Manage.* 9.1 (Mar. 2012), pp. 12–21 (cit. on p. 14).

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. «MoonGen: A Scriptable High-Speed Packet Generator». In: *Proc. ACM Internet Measurement Conf. (IMC)*. Tokyo, Japan, Oct. 2015, pp. 275–287 (cit. on p. 14).