Scripting Netfilter with Lua: A Cooperative Kernel-Userspace Pipeline

Lourival Vieira Neto
Ring Zero Networks
Rio de Janeiro, Brazil
lourival.neto@ringzero.com.br

Md. Shehar Yaar Tausif
Independent Researcher
Bengaluru, India
sheharyaar48@gmail.com

Firas Shaari
802.11 Networks Corp
Plymouth Meeting, USA
firas@80211networks.com

Marcel S. A. de Moura
Ring Zero Networks
Rio de Janeiro, Brazil
marcel.moura@ringzero.com.br

Arif Alam
NetExperience
Ottawa, Canada
arif.alam@netexperience.com

Abstract

Secure Web Gateways protect outbound HTTPS traffic, but their deep packet inspection intercepts TLS by terminating each connection through a CA installed on every client, an approach that the commodity hardware of Wi-Fi access points typically cannot sustain. This paper presents a kernel-userspace datapath pipeline that filters L7 traffic by inspecting DNS queries, HTTP requests, and TLS handshake metadata rather than intercepting the connection. Both halves run in Lua: a userspace agent generates an nftables bridge ruleset that dispatches each new flow to a Netfilter hook for classification. Then, nftables caches the verdict in a set, and subsequent packets bypass Lua entirely. The pipeline builds on Lunatik, our Linux kernel-scripting framework presented at Netdev 0x14 and 0x17, and contributes three new bindings: `luanetfilter` for direct Netfilter hooks, `luaskb` for socket-buffer access and reply synthesis, and `luanftables`, a userspace `libnftables` wrapper. On a Wi-Fi 6 access point under a combined Ethernet and Wi-Fi HTTPS workload, the pipeline sustains 1.4 Gbps at parity with a plain Linux bridge in throughput and latency. The pipeline ships in production as Ring Zero Dome, the in-kernel engine of the NetExperience Secure Wireless Gateway on OpenWiFi access points.

Keywords

Lua, Lunatik, kernel scripting, Netfilter, nftables, L7 packet filtering, Secure Web Gateway, OpenWiFi

1 Introduction

Secure Web Gateways (SWGs) are web filters that protect outbound HTTPS traffic for enterprise users, providing policy-based access control and anti-malware services [4]. SWGs typically perform deep packet inspection (DPI) on HTTPS by intercepting TLS connections, terminating and reinitiating them through a CA installed on every client device, at the cost of per-session cryptographic work [10]. The commodity hardware of Wi-Fi access points (APs) typically cannot meet these demands: its CPU and memory are shared with the Wi-Fi datapath and the management plane. Cloud-hosted gateways centralize this work, but backhauling traffic to them adds latency, consumes uplink capacity, and routes traffic through the cloud operator's jurisdiction; running on the AP itself avoids the backhaul but must fit within its CPU and memory budget.

To deliver SWG enforcement within this hardware budget, we apply kernel scripting, a technique that embeds a scripting language inside the operating system kernel so that components written in C can be composed and customized through user scripts [31]. Our framework, Lunatik, runs on Linux and has scripted several networking subsystems.

At Netdev 0x14 we presented NFLua and XDPLua, monolithic Lua extensions to Netfilter (via xtables) and XDP (via eBPF helpers) [32]. At Netdev 0x17 we refactored Lunatik into a modular framework, presenting `luafib` for the routing table and `luanotifier` for kernel notifier chains [33]. NFLua and XDPLua were later redesigned as the `luanetfilter` and `luaxdp` bindings on top of this framework.

We apply this framework to cloud-managed APs running OpenWiFi [27], which forward client traffic at L2 as bridges, without performing IP routing, applying per-network policy pushed from a central controller. In multi-tenant deployments (e.g., multi-dwelling units, hospitality), VLAN sub-interfaces are provisioned at association time, so policy must follow interfaces that appear at runtime.

In this paper, we describe a kernel-userspace datapath pipeline for packet filtering on APs, structured around three new components. In the kernel, two new bindings extend the framework. The first, `luanetfilter`, replaces NFLua's xtables-based dispatch with direct Netfilter hook registration, supporting the bridge, inet, ipv4, ipv6, arp, and netdev families with configurable priorities and optional mark-based dispatch, so a hook fires only on packets carrying a configured mark. The second, `luaskb`, exposes the kernel's socket buffer (SKB) to Lua for inspection and modification; the hook uses this to generate synthetic replies (e.g., DNS NXDOMAIN, HTTP redirects) without crossing into userspace.

In userspace, `luanftables` wraps `libnftables` so a Lua script can translate configuration into nftables rules

without invoking the `nft` binary. The Netfilter hook and the nftables data plane cooperate through packet marks: the hook runs only until each flow is classified; the verdict is then cached in dynamic nftables sets, and subsequent packets bypass the hook.

Beyond the mark protocol, interface lifecycle events flow through two pre-existing bindings: `luanotifier` and `luarcu`. A Lua callback registered via `luanotifier` on the kernel netdevice chain publishes new interface state into a shared table provided by `luarcu`, which the Netfilter hook reads. `luarcu` wraps the Linux kernel's Read-Copy-Update (RCU) [18] synchronization mechanism.

Ring Zero Dome serves as the in-kernel engine of the NetExperience Secure Wireless Gateway [5], a content filtering, threat protection, and microsegmentation service deployed on OpenWiFi APs without TLS interception. Dome is built upon open-source components: `luanetfilter` and `luaskb` ship as part of Lunatik [16], and `luanftables` is a standalone project [22].

The remainder of the paper traces the evolution of Lunatik (Section 2), describes direct Netfilter hooks (Section 3), userspace ruleset generation (Section 4), the datapath pipeline (Section 5), synthetic reply generation (Section 6), and runtime interface tracking (Section 7), reports evaluation results (Section 8), and concludes (Section 9).

2 Evolution of the Lunatik Framework

At Netdev 0x14 [32] we presented NFLua, a Netfilter scripting environment coupled to iptables that invoked Lua functions through the `-m lua` match or `-j LUA` target. The architecture was monolithic: NFLua acted as the runtime entry point, embedding the in-kernel Lua interpreter (Lunatik) and a fixed set of supporting modules (`luadata`, `luajson`); scripts could not load additional bindings, and adding a new one required source-level changes to NFLua and a rebuild. Lua's role was confined to packet filtering within iptables.

At Netdev 0x17 [33] we presented the redesign of Lunatik into a modular framework with a core plus an open set of binding modules. The core contains the Lua interpreter modified to run in the kernel and a runtime abstraction that allows the framework to select synchronization and allocation primitives appropriate to each script's execution context. Bindings expose kernel resources to Lua as Lunatik objects, which are Lua userdata that wrap a C kernel structure with reference counting and synchronization [37]. For example, the `luadata` binding provides `data` objects, views onto regions of kernel memory that Lua scripts can read and write directly.

Two further bindings, `luanotifier` (kernel netdevice chain) and `luafib` (kernel routing table), demonstrated the architecture by composing into an adaptive routing service that swapped the default route on link-state changes. Figure 1 illustrates this modular architecture.

The XDPLua kernel patch was replaced by `luaxdp`, a loadable binding that exposes a BPF kfunc so an eBPF program at the XDP hook can call into a Lua script while keeping the fast-path in eBPF [36]. The binding ships in Dome's mobile-edge deployment on the OpenRAN@Brasil testbed [14, 8], inspecting traffic on disaggregated radio nodes.

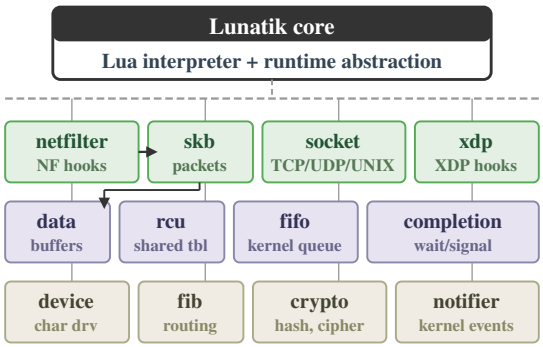

Figure 1: Lunatik: modular core with independent binding modules.

The framework has continued to grow modularly since Netdev 0x17. Beyond the original libraries (sockets, character devices, kernel threads, notifier chains, RCU, FIB), the catalog now spans networking, data handling, inter-runtime coordination, kernel events, device drivers, and general kernel APIs, including the `luanetfilter` and `luaskb` bindings of this paper.

The modular runtime has also evolved its memory model. An earlier NFLua deployment [35] kept its in-kernel Lua state within an explicit memory limit on the `kmalloc`-backed allocator [7] and delegated lookups against larger intelligence bases to a userspace agent and a cloud backend. The current Lunatik runtime allocates Lua memory through `kvmalloc`, falling back to `vmalloc` when physically contiguous pages are unavailable, so a Lua state can hold large data structures previously confined to userspace. Dome's domain intelligence base lives entirely in the Lua state, consulted by the hook without leaving the kernel.

3 Direct Netfilter Hooks

The `luanetfilter` binding lets a Lua script register itself directly as a Netfilter hook in the kernel. NFLua, its predecessor, was an xtables module (`xt_lua`) with a match and a target, and required a companion userspace plugin that shipped only for iptables, leaving ip6tables, arptables, and ebtables without Lua support. The `luanetfilter` binding removes this dispatch chain, registering Lua functions directly with Netfilter via `nf_register_net_hook`. Hooks can be placed at any Netfilter hook point across the supported families; the `BRIDGE` family in particular enables transparent filtering of bridged traffic on APs without requiring IP routing.

The `priority` parameter fixes the hook's evaluation order relative to other hooks and nftables chains on the same hook point. In the datapath pipeline of Section 5, this lets the hook be placed between an nftables chain that selects packets for inspection and another that enforces the verdict, so explicit rules and cached decisions short-circuit the hook.

The `mark` parameter and the callback's return value tie the

binding to the pipeline through the packet mark. At registration time, a `mark` parameter restricts delivery to packets bearing a specific mark value, so packets that do not match never reach the hook and incur no script invocation. At callback return time, the function returns an action and, optionally, a mark; when a mark is returned, it is set on the packet's SKB and is propagated to subsequent nftables chains. The two halves form a small protocol: nftables writes marks to direct packets toward the hook, and the hook writes marks to communicate verdicts back.

```
local netfilter = require("netfilter")
local nf        = require("linux.nf")
local eth       = require("linux.eth")
local pack      = require("pack")
local config    = require("config")
local ipv4      = require("ipv4")
local ipv6      = require("ipv6")
local tls       = require("tls")

local action      = nf.action
local unpacker    = pack.unpacker
local blocklist   = config.blocklist
local extract_sni = tls.extract_sni

-- EtherType offset in Ethernet frame
local ETH_TYPE = 12

local parsers = {
  [eth.IP]   = ipv4.parse,
  [eth.IPV6] = ipv6.parse,
}

local function classify(skb)
  local frame = skb:data("mac")
  local _, _, be16 = unpacker(frame)
  local parse = parsers[be16(ETH_TYPE)]
  local _, offset = parse(frame)
  local sni = extract_sni(frame, offset)

  if blocklist[sni] then
    return action.ACCEPT, 0xd2  -- deny
  end
  return action.ACCEPT, 0xd1    -- allow
end

netfilter.register{
  hook     = classify,
  pf       = nf.proto.BRIDGE,
  hooknum  = nf.br.PRE_ROUTING,
  priority = nf.br.pri.FILTER_OTHER,
  mark     = 0xd0,
}
```

Listing 1: A complete Netfilter hook in Lua: composing bindings to classify a flow.

Listing 1 is a complete hook, composing several independent bindings into one in-kernel classifier. Lua's `require` loads a module by name. In kernel space, Lunatik resolves C bindings by looking up the module's `luaopen_*` entry point in the kernel symbol table (the binding's kernel module must already be loaded) and Lua scripts by reading the corresponding `.lua` file through the kernel's VFS API.

The `require` block at the top of Listing 1 loads the modules this hook composes, with `luanetfilter` registering the Netfilter hook. The others are implemented in Lua: `linux.nf` and `linux.eth` supply kernel constants (Netfilter actions, Ethernet types); `pack` provides raw packet access via `luadata`; `config` carries the user blocklist; and `ipv4`, `ipv6`, and `tls` parse L3 and TLS headers. The callback receives the packet as an SKB exposed by `luaskb`, and `skb:data` returns the frame bytes that `luadata` reads through `unpacker`. An EtherType dispatch table selects the L3 parser, `ipv4` or `ipv6`; the TLS parser extracts the SNI; and a blocklist lookup produces the verdict.

The registration block names the callback (`hook`), the hook point (`pf`, `hooknum`, `priority`), and the mark filter. The callback returns `(action, mark)`, where `ACCEPT` continues through subsequent chains rather than allowing the packet directly: the final verdict is taken against the mark in the chains downstream. The nftables ruleset around the hook is built in userspace through `luanftables`, the subject of the next section.

4 Userspace Ruleset Generation

OpenWiFi [27] is the open AP software stack of the Telecom Infra Project [26]: an OpenWrt firmware paired with a microservice cloud, deployed in production by commercial controllers such as NetExperience [5].

APs communicate with the cloud through uCentral [28], a WebSocket initiated by the AP. The cloud pushes configurations as JSON validated against a published schema [29]. The schema defines a `third-party` extension point, keyed by vendor name, that lets services such as Dome reach the AP through uCentral without a custom transport.

The Dome agent uses `luanftables`, a Lua wrapper around `libnftables`, to write nftables rules directly from Lua rather than spawning `nft` subprocesses.

Each uCentral push restarts Dome through `ubus` [21], OpenWrt's IPC bus. The agent then performs two tasks: it converts the `third-party.dome` JSON blob into a Lua script that the in-kernel hook loads through `require` (Section 3), and it rewrites the dynamic per-profile section of the nftables ruleset through `luanftables` (Figure 2). The static template (chain layout, cache sets, port-marking rules), fixed across pushes, defines the pipeline of Section 5.

Policy is structured as named *profiles*, each matching a group of devices (by MAC address or by network interface) and specifying the filtering rules to apply. Both the in-kernel hook (Section 3) and the userspace agent are written in Lua.

Listing 2 shows two example profiles. The `firetv` profile applies to a Fire TV stick identified by MAC, default-blocked except for its vendor cloud and authorized streaming services. The `household` profile applies to clients of an SSID, blocking malicious domains, trackers, and advertising. Listing 3 shows the corresponding `luanftables` translation for the MAC case: a per-profile MAC set populated from `profile.devices`, plus a per-profile catch-all drop when `policy = "block"`.

Profiles may also carry L3/L4 firewall rules (the `rules` field), which the agent translates into the bridge `acl` chain

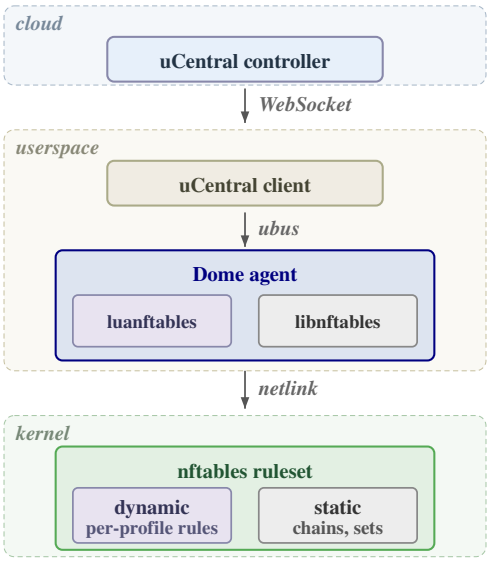

Figure 2: On each uCentral push, the Dome agent rewrites the dynamic per-profile section of the nftables ruleset using `luanftables`; the static template is fixed.

ahead of `dpi` marking (Section 5 details the chain structure). For a default-block profile such as `firetv`, these rules let through L3 services the device needs to operate (DHCP, NTP, mDNS); the L7 hook continues to filter against the allowlists, and any other traffic falls into the catch-all drop.

For the SSID case, the agent resolves each SSID name in `profile.ssids` to its underlying interface (e.g., `wlan0`) and identifies the profile's traffic by `iifname`. A `profile.ssids` entry without an explicit `vlan` field matches both `wlan0` and any of its VLAN sub-interfaces (e.g., `wlan0-v100` for VLAN 100) via a wildcard. Section 7 covers the runtime tracking of sub-interfaces created by hostapd at association time (via 802.1X or multiple pre-shared keys, MPSK).

5 The Datapath Pipeline

The datapath splits packet filtering between nftables chains and a Lua hook, cooperating through marks [19]: nftables handles set lookups, tuple matching, and accept/drop/reject; the Lua hook handles protocol parsing, string matching, hash table lookups with variable-length keys, and synthetic reply generation.

Every chain reads and writes the packet's Netfilter mark, threading a single byte of classification state through the pipeline. Mark values group by role in the verdict lifecycle: not selected for inspection, submitted to Lua, allow/deny verdicts back from the hook, reassembly pending, and explicit firewall pass. Table 1 lists the six values used.

5.1 Multi-Chain Architecture

The pipeline consists of six chains on the bridge prerouting hook, five of them attached directly to it in priority order; the sixth (`fallback`) is reached only through an explicit `jump` from another chain. All chains and sets live in a ded-

```
{
  "profiles": {
    "firetv": {
      "devices": ["f0:81:73:8a:1f:42"],
      "policy": "block",
      "allowlists": ["amazon-devices",
        "amazon-prime", "netflix"],
      "rules": [
        {"proto": "udp", "dport": "67-68",
          "action": "allow"},
        {"proto": "udp", "dport": "123",
          "action": "allow"},
        {"proto": "udp", "dport": "5353",
          "action": "allow"}
      ]
    },
    "household": {
      "ssids": [{"name": "Home"}],
      "blocklists": ["threats",
        "trackers", "ads"]
    }
  }
}
```

Listing 2: Example policy profiles in a uCentral third-party.dome blob.

```
local nft = require("nftables")
local fmt = string.format

local new_set = [[
  add set bridge dome %s
    { type ether_addr; }
]]
local add_mac = [[
  add element bridge dome %s { %s }
]]
local drop = [[
  add rule bridge dome fallback
    ether saddr @%s counter drop
]]

local ctx <close> = nft.new()
for name, p in pairs(profiles) do
  if p.devices then
    local set = "profile_" .. name
    ctx:cmd(fmt(new_set, set))

    for _, mac in ipairs(p.devices) do
      ctx:cmd(fmt(add_mac, set, mac))
    end

    if p.policy == "block" then
      ctx:cmd(fmt(drop, set))
    end
  end
end
```

Listing 3: Per-profile MAC set and catch-all drop, written through `luanftables`.

Table 1: Mark values used in the datapath pipeline.

Mark	Name	Meaning
`0x00`	NONE	Untouched, not selected for inspection
`0xd0`	INSPECT	Submitted to Lua for classification
`0xd1`	SKIP	Classified as allow; cache and forward
`0xd2`	DENY	Classified as block; cache and reject
`0xd3`	PENDING	Reassembly in progress; bypass cache
`0xd4`	PASS	Explicit firewall allow; bypass cache

icated nftables table (`bridge dome`), keeping the pipeline isolated from any other nftables consumer on the AP. Figure 3 shows the packet flow.

`cache`. Bypasses the Lua hook for already-classified sessions. Two dynamic nftables sets, `@skip` and `@deny`, cache the verdict keyed on the 4-tuple $\langle src\_ip, dst\_ip, src\_port, dst\_port \rangle$; a hit transfers it to the mark and accepts the packet so subsequent chains do not re-evaluate it.

`acl`. Holds the explicit L3/L4 firewall rules populated by the agent: an allow rule sets mark `0xd4` and accepts, bypassing inspection; a deny rule drops the packet directly.

`dpi`. Marks traffic on ports of interest (e.g., 53, 80, 443) with `0xd0` so the Lua hook receives it for L7 classification. A guard condition (`meta mark 0x00`) ensures that packets already marked by `cache` or `acl` are not re-marked.

Lua hook (bridge prerouting callback, registered via `luanetfilter`). Receives packets marked `0xd0`. The hook parses protocol headers from the SKB, classifies, and returns `0xd1` (allow) or `0xd2` (deny); blocking verdicts may also produce a synthetic reply (Section 6). When classification needs more data (e.g., a fragmented TLS ClientHello), the hook stashes partial data and returns `0xd3` (pending) so subsequent segments reach the hook for reassembly.

`verdict`. Routes on the mark: `0xd1` and `0xd2` accept so `learn` records the verdict; `0xd3` and `0xd4` also accept but bypass `learn`; `0xd0` jumps to `fallback`. After the jump returns (still `0xd0`, meaning the catch-all drop did not match), the chain re-marks the packet `0xd1` so `learn` caches it as allowed.

`fallback` (regular chain, jumped from `verdict`). Holds a catch-all drop per block-policy profile, reached only by packets the hook did not classify; allow-policy profiles emit no rule here, so their unclassified packets fall through.

`learn`. Writes verdicts to the cache sets and emits the deny side-effect. `0xd1` adds the flow to `@skip`; `0xd2` adds it to `@deny` and rejects with TCP RST (TCP) or drops (UDP). `0xd3` and `0xd4` match no rule here, keeping reassembly-in-progress flows and explicit acl allows out of the cache.

5.2 Inside the Lua Hook

The Lua hook extracts the Server Name Indication (SNI) from a TLS ClientHello, the QNAME from a DNS query, or the Host header from an HTTP request, and looks the resulting domain up against the policy. Listing 4 shows the TLS case as exemplar: the SNI is nested inside a variable-length chain of structures (handshake header, session ID, cipher suites, compression methods, extensions list), and the parser walks the extensions until the Server Name extension (type 0x0000) is found. Each offset depends on a length field read from the previous structure, creating a chain of length-prefixed reads.

DNS QNAMEs follow the same length-prefixed shape, walked label by label. Either lookup completes inside the hook: the domain database lives entirely in the kernel-resident Lua state (Section 2), without crossing into userspace, in contrast to earlier NFLua deployments where larger intelligence bases required a userspace agent for classification [35].

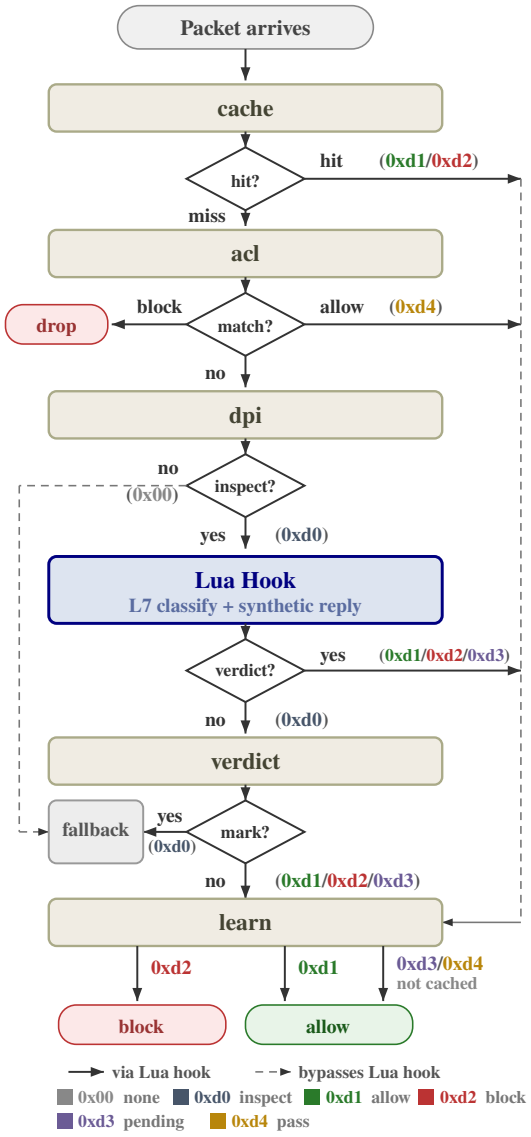

Figure 3: Packet flow through the datapath pipeline. Marks mediate transitions between nftables chains and the Lua hook. Solid arrows show the Lua-classified path; dashed arrows show paths that bypass the Lua hook (cache hit, explicit allow, or not selected for inspection).

```lua
local pack = require("pack")

local unpacker = pack.unpacker

local SERVER_NAME    = 0x0000
local SESSION_ID_OFF = 43
local MAX_EXTENSIONS = 17

local tls = {}

function tls.extract_sni(packet, offset)
  local str, u8, be16 =
    unpacker(packet, offset)

  -- Skip variable-length fields
  local cipher = SESSION_ID_OFF + 1
    + u8(SESSION_ID_OFF)
  local comp = cipher + 2 + be16(cipher)
  local ext  = comp + 3 + u8(comp)

  -- Walk extensions
  for _ = 1, MAX_EXTENSIONS do
    if be16(ext) == SERVER_NAME then
      local len = be16(ext + 4 + 3)
      return str(ext + 4 + 5, len)
    end
    ext = ext + 4 + be16(ext + 2)
  end
end

return tls
```

Listing 4: TLS SNI extraction from ClientHello extensions.

When the SNI is hidden by Encrypted Client Hello (ECH) or by DNS-over-HTTPS (DoH), Dome falls back to two controls: the TLS hook denies connections matching a blocklist of known DoH resolvers (e.g., `cloudflare-dns.com`, `dns.google`), and a DNS hook returns synthetic replies (Section 6) to DNS HTTPS RR (QTYPE 65) [24] queries to prevent ECH discovery.

5.3 Cache Dynamics

Verdict caching applies per flow: one classification decision is reused for every subsequent packet of that flow, illustrated below through the TLS lifecycle. DNS is the exception: each query is classified independently, so DNS traffic never enters the cache sets.

Cache miss and hit. The first data packet of a TLS connection (a ClientHello in a single segment) finds no entry in `@skip` or `@deny` when `cache` runs. `dpi` marks it `0xd0`, the Lua hook parses the SNI and returns `0xd1` (allow) or `0xd2` (block), and `learn` adds the 4-tuple to `@skip` or `@deny`. Subsequent packets hit that set in `cache`, are marked with the cached verdict, and bypass the Lua hook entirely; the `learn` chain refreshes the timeout on each hit and re-applies the reject or drop on `@deny` hits.

Fragmented ClientHello. When the ClientHello spans multiple TCP segments, the first segment does not contain the complete SNI extension. The Lua hook stashes the partial data in a per-connection LRU buffer and returns mark `0xd3` (pending). The `verdict` chain accepts the packet on `0xd3` before the fallback jump (so it is not subjected to per-profile catch-all drops); the `learn` chain has no rule matching `0xd3`, so the flow is not added to `@skip`. The next segment is re-marked `0xd0` in `dpi`, the Lua hook reassembles, extracts the SNI, and returns the verdict; the `learn` chain records the 4-tuple normally from there.

Timeout and re-evaluation. Set entries expire after 2 minutes of inactivity, capping how long a cached verdict persists for an idle flow. New connections always re-enter the Lua hook (a new TCP source port makes the 4-tuple a miss), so configuration changes (e.g., policy updates pushed from the cloud controller) take effect immediately for new flows and within the timeout window for any quiescent cached entries.

6 Synthetic Reply Generation

Netfilter already supports reply injection through the `REJECT` target, which generates fixed kernel templates: TCP RST segments and ICMP unreachable messages. These templates carry no application-layer content, so REJECT cannot express a DNS NXDOMAIN response or an HTTP 302 redirect to a custom URL.

The `luaskb` binding makes reply generation scriptable. The Lua hook clones the SKB, modifies its headers and payload, and transmits the clone to the sender, then drops the original. The blocked flow is deliberately not cached: each new query or request re-enters the hook and receives a fresh synthetic reply, so the client always sees the block page or the synthetic DNS answer rather than a dead connection.

The binding exposes the kernel socket buffer to Lua through five operations:

`skb:copy()`. Independent deep copy with its own memory, linearizing any fragmented data.

`skb:data([layer])`. Returns a `luadata` object at the network header (default `"net"`, L3) or at the Ethernet header with `"mac"` (L2).

`skb:resize(len)`. Adjusts the packet length when the payload changes size.

`skb:checksum()`. Recomputes the IP and transport-layer checksums.

`skb:forward()`. Queues the result for transmission on the ingress interface.

The hook-generated reply types share a header-swap routine for L2, L3, and L4 (handling both IPv4 and IPv6); replies are queued via `dev_queue_xmit` from the hook.

DNS NXDOMAIN. The hook copies the packet, swaps L2/L3/L4 addresses, and flips the DNS flags in place (setting `QR=1`, `RA=1`, `RCODE=3` while preserving the client's `RD` flag); the question section is preserved intact so the client receives a well-formed response. Listing 5 shows the bit manipulation (`dns_nxdomain`) and the fallback path that returns only NXDOMAIN; the production code also synthesizes an A or AAAA record when a block-page IP is configured. The dropped query never reaches the upstream resolver.

HTTP redirect or block response. The TCP payload is replaced by a precomputed response (302 redirect when a block-page URL is configured, otherwise 403 Forbidden), the IP/SKB lengths are updated, and the TCP sequence numbers

```lua
local pack = require("pack")

local unpacker = pack.unpacker
local packer   = pack.packer

local function dns_nxdomain(frame, offset)
  local _, _, be16 = unpacker(frame)
  local _, _, pbe16 = packer(frame)
  local flags = be16(offset + 2)
  flags = (flags & 0x0100) | 0x8083
  pbe16(offset + 2, flags)
end

local function dns_reply(skb, offset)
  local new_skb = skb:copy()
  local frame = new_skb:data("mac")
  dns_nxdomain(frame, offset)
  swap_headers(frame)
  new_skb:checksum()
  new_skb:forward()
end
```

Listing 5: Synthesizing a DNS NXDOMAIN reply.

are swapped with `FIN|PSH|ACK` so the client accepts the response and closes the connection.

TCP RST for blocked TLS. Dome does not intercept TLS, so the hook holds no session key to forge an encrypted application-layer response; the hook leaves the reply to nftables' `reject with tcp reset`, emitted from the `learn` chain on the DENY mark match.

7 Tracking Dynamic Interfaces

Section 4 noted that hostapd creates VLAN sub-interfaces at association time, after the agent has run. In larger Wi-Fi deployments, matching by MAC alone becomes impractical at scale: many devices per user, frequent device churn, and occupant turnover. Operators instead match by VLAN sub-interface. The sub-interfaces may appear minutes or hours later; the policy layer must track them as they appear, without reloading the hook. The `luanotifier` binding, introduced with the modular Lunatik at Netdev 0x17 [33], registers Lua callbacks on kernel notifier chains and dispatches their events from process context. For dynamic VLANs, we register on the netdevice chain, which fires on every interface lifecycle event (registration, unregistration, link up or down).

7.1 Dynamic VLAN Resolution

The notifier callback (Listing 6) fires when a new interface appears (`REGISTER`) and matches its name against the configured interfaces, trying an exact match first and then a wildcard (e.g., `wlan0-v100` matches `wlan0-v*`). On a hit it publishes the interface into a `luarcu` table that the Lua hook consults whenever its static interface cache misses.

Writer and reader run in different kernel contexts: the notifier in process context and the Netfilter hook in softirq. Lunatik runs each as its own runtime over a single RCU-protected table (Listing 7 is the read side), so the notifier's writes reach the hook's per-packet reads without locking.

On a cache miss, `resolve` consults the `luarcu` table, copies the resolved interface entry, and caches it for subsequent lookups. The copy is necessary because the wildcard interface entry is shared across all VLANs of one radio, so writing per-`ifindex` state into it would corrupt the source. The `box`/`unbox` helpers in both sides wrap and unwrap the ifname as a `luadata` object, since the `luarcu` table accepts only userdata values.

```lua
local notifier = require("notifier")
local linux    = require("linux")
local netdev   = require("linux.netdev")
local common   = require("common")

local box    = common.box
local lookup = common.lookup

local function attacher(vlans)
  local function tracker(event, ifname)
    if event == netdev.REGISTER
       and lookup(ifname) then
      local idx = linux.ifindex(ifname)
      if idx then
        vlans[tostring(idx)] = box(ifname)
      end
    end
  end
  notifier.netdevice(tracker)
end

return attacher
```

Listing 6: Notifier callback for interface tracking (process context).

```lua
local common = require("common")

local unbox  = common.unbox
local lookup = common.lookup

local cache = {}  -- ifindex -> entry

local function resolve(ifindex, vlans)
  local key = tostring(ifindex)
  local ifname = unbox(vlans[key])
  local entry, vlan_id = lookup(ifname)

  if entry then
    local new_entry = {
      profile = entry.profile,
      ifname  = entry.ifname,
      ssid    = entry.ssid,
      vlan    = vlan_id or entry.vlan,
    }
    cache[ifindex] = new_entry
    return new_entry
  end
end
```

Listing 7: Per-packet interface resolution, called by the Netfilter hook (softirq).

The tracker runs infrequently (once per interface creation) and writes to the RCU table; the hook consults the table on every packet whose `ifindex` is not already in the static cache, lock-free under RCU read-side critical sections. After the first successful lookup, the interface entry lives in the static cache, so the RCU path is taken at most once per interface.

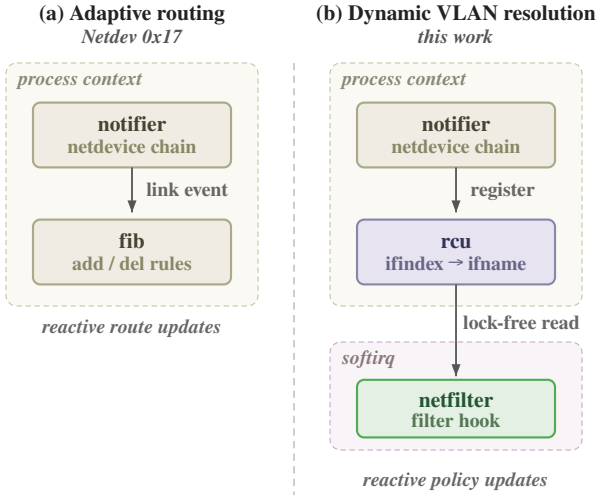

Figure 4: Use of `luanotifier` in two services: (a) adaptive routing via `luafib` [33]; (b) interface tracking via `luarcu`, whose RCU table is consulted by the Netfilter hook.

Figure 4 contrasts the present pipeline with the adaptive routing service of Netdev 0x17. Both services register a Lua callback on the netdevice notifier chain; the difference is the downstream consumer (the kernel FIB at Netdev 0x17, a Netfilter hook here). Both bindings (`luanotifier` and `luarcu`) are reused unchanged: policy that would otherwise require a dedicated kernel module reduces to the two Lua functions in Listings 6 and 7.

8 Evaluation

This section evaluates the datapath pipeline of Section 5 with the TLS SNI classifier (Listing 4) as the workload. An aggregated Ethernet and Wi-Fi HTTPS workload exercises the cache mechanism under production-realistic load; a LANforge cross-tool campaign corroborates the parity claim under HTTPS foreground plus non-inspected Ethernet background. Single-traffic-generator runs on each isolated path support the per-DUT cap argument.

8.1 Methodology

The device under test (DUT) is an Edgecore EAP101 [11] access point built on the Qualcomm IPQ6018 SoC (four ARM Cortex-A53 cores at 1.8 GHz) running Telecom Infra Project (TIP) [26] OpenWiFi [27]. The DUT exposes a 1 Gbps LAN port and a 2.5 Gbps WAN port.

The Ethernet client and the HTTPS server are hosted on a single LANforge [2] appliance (Figure 5): the `wrk2` [30]

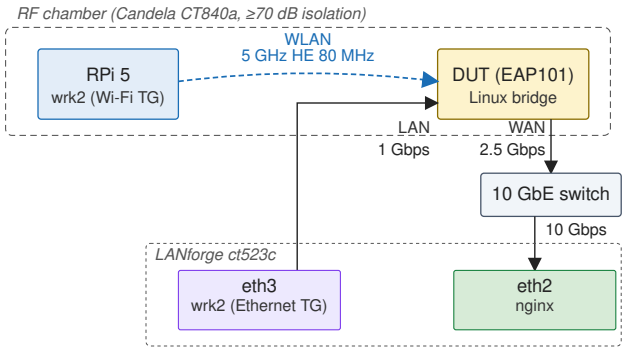

Figure 5: Experimental topology. Two `wrk2` traffic generators (Wi-Fi via the RPi 5 station, Ethernet via LANforge `eth3`) drive HTTPS requests through the DUT bridge to `nginx` on LANforge `eth2`.

client drives traffic to the DUT LAN port, and `nginx` [20] serves HTTPS responses through a 10 GbE switch to the DUT WAN port. The Wi-Fi station is a Raspberry Pi 5 (RPi 5) with an Intel BE200 client card associating with the DUT on 5 GHz; the station and the AP share an RF-shielded chamber (Candela Technologies CT840a [3]). Path-ceiling references from `iperf3` [13] are ∼941 Mbps on the Ethernet path and ∼894 Mbps on the Wi-Fi path; the additive per-link reference of 1.8 Gbps brackets the aggregate `wrk2` throughput, and the 2.5 Gbps DUT WAN port exceeds the aggregate throughput in every cell.

Three conditions are compared in the `wrk2` experiments. In *baseline*, Dome is not loaded and the DUT bridges layer-2 traffic without Dome's nftables tables. In *Dome*, the engine runs in its default configuration: the Lua hook classifies each new flow from its first data packets and commits the verdict to an nftables set; subsequent packets of the same flow match that set in the bridge prerouting chain and bypass the hook. In *Dome without cache*, the cache-match rules are removed from the same chain, so every packet the `dpi` chain dispatches for inspection enters the Lua hook; this condition mirrors per-packet scripting designs such as NFLua [32], where classification runs on every packet of every flow.

The traffic generators (TGs) drive `wrk2` in open-loop rate control to avoid coordinated omission, with four threads × 100 persistent TLS connections per TG (the Ethernet TG on LANforge and the Wi-Fi TG on the RPi 5) requesting fixed-size objects from `nginx` on port 443. Per-cell target rates are calibrated against the baseline ceiling and pinned at $0.9\times$ the highest target rate the TG can deliver while holding achieved/target ≥ 0.95. Each production cell comprises ten repetitions of 30 s after a 15 s warmup; medians are reported with bootstrap 95% confidence intervals on the median (10 000 resamples) [12]; condition comparisons use the same bootstrap on the per-cell difference. Latency is reported at the median and the 99th percentile: stall-type costs (garbage-collection pauses, hook serialization, queue buildup) surface in the tail rather than the median [9]. The calibrated target rates and per-cell raw data are in the dataset [34].

The aggregate campaign covers four HTTPS response sizes

Table 2: Aggregated Ethernet + Wi-Fi HTTPS workload through the DUT. Totals (req/s, Mbps) across both TGs; p50 and p99 are the larger of the two per-TG medians; sys% is the DUT-side peak (threaded NAPI accounts softirq under `%sys`, not `%sirq`). Medians over ten repetitions of 30 s per cell.

Size / condition	req/s	Mbps	p50	p99	sys%
10 KB					
baseline	12 758	1 161	176 ms	1.64 s	31.1
Dome	12 774	1 163	163 ms	1.61 s	48.1
Dome without cache	1 017	93	18.3 s	27.8 s	99.5
100 KB					
baseline	1 607	1 403	12 ms	239 ms	31.2
Dome	1 599	1 399	15 ms	245 ms	45.7
Dome without cache	845	742	6.3 s	21.6 s	94.8
300 KB					
baseline	521	1 367	145 ms	315 ms	31.7
Dome	521	1 368	146 ms	350 ms	45.7
Dome without cache	454	1 190	539 ms	19.0 s	79.3
1024 KB					
baseline	137	1 245	503 ms	716 ms	34.5
Dome	137	1 246	555 ms	786 ms	53.1
Dome without cache	135	1 230	710 ms	1.45 s	60.8

anchored to the per-resource size distributions in the HTTP Archive Web Almanac [25, 15]. 10 KB matches the median individual image and the median HTML document. 100 KB sits between p75 and p90 of individual images. 300 KB sits above p90 of individual images. 1024 KB sits well above the Almanac's p90 for individual web resources, in the regime of bulk transfers such as on-demand video streaming. Single-TG runs on each isolated path extend the sweep to 1 KB, where at the aggregate level both TGs hit their own client-side throughput caps before the DUT becomes the binding constraint, so the size is reported only via single-TG runs. This small-object regime sits below the Almanac's p25 for individual web resources, within the small-payload, short-lived connections that dominate at Internet scale [1], including Internet-of-Things (IoT) heartbeat traffic [17].

8.2 Cache mechanism under aggregated load

Table 2 reports the aggregated Ethernet and Wi-Fi HTTPS workload sustained through the DUT. Figure 6 plots the sustained throughput against response size. Across the four response sizes, *Dome* tracks the *baseline* within bootstrap rounding on throughput (95% CI widths $\leq$ 0.9% on *baseline* and *Dome* rows; wider on *Dome without cache* at small sizes), and *Dome* latency tracks the *baseline*: per-cell p50 and p99 differences are indistinguishable from run-to-run variability (bootstrap 95% CIs on the difference include zero). The pipeline absorbs the workload at aggregate rates reaching 1.4 Gbps at 100 KB with the cache active.

Hook saturation point. Without the cache, the Lua hook saturates the DUT under continuous open-loop load. The ceiling is per-DUT rather than per-TG. A single ingress alone reaches ~1 060 req/s at 10 KB, yet two ingresses loaded together sum to only 1 017 req/s (Table 2), not the doubling a per-TG limit would produce, so the DUT, not the TG, is the bottleneck. The single-ingress runs hold at ~1 050 req/s for 1 KB responses as well. At one-tenth the payload the request rate does not rise, so the cap is request-rate-bound rather than byte-bound. The mechanism is a single-state Lua hook, with the four Cortex-A53 cores serializing on each invocation. Per-CPU Lua runtimes, an extension of Lunatik's multi-runtime model, would lift the bound. The bound manifests only under continuous load; real users browse in bursts [6], so this regime is a worst case, not the load of ordinary browsing.

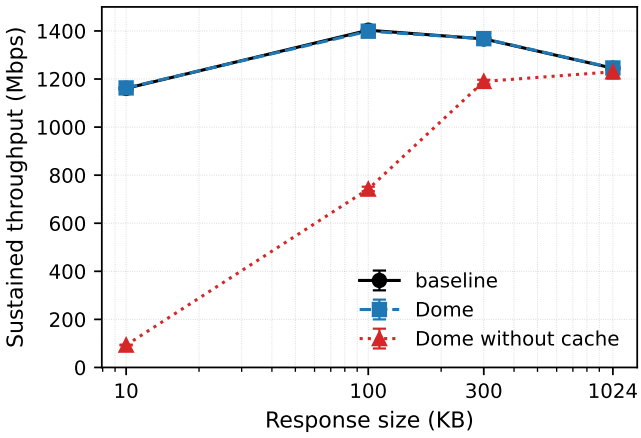

Figure 6: Sustained aggregate throughput against HTTPS response size (log-scaled axis). Error bars are bootstrap 95% CIs on the median. *Dome* tracks the *baseline* across all sizes; *Dome without cache* converges as response size grows.

Cache cost. With the cache active, the hook inspects each flow's data segments only until that flow is classified; thereafter its packets match the verdict cache and bypass the hook. The per-packet inspection cost is thus amortized to a per-flow cost paid once at handshake, producing a +14 to +19 pp sys% overhead between *Dome* and *baseline* across all response sizes (Table 2). The *baseline* sits at 31–35% sys% under the same aggregate workload, the bare cost of Linux bridge forwarding.

Bandwidth-bound convergence. The large-response regime is bandwidth-bound rather than call-rate-bound. By 1024 KB the *Dome*, *Dome without cache*, and *baseline* cells converge at ~1.25 Gbps of HTTPS throughput. *Dome without cache* reaches 87% of *baseline* at 300 KB and 99% at 1024 KB, as the per-packet inspection cost amortizes over response bytes at large sizes. This aggregate sits against the additive per-link `iperf3` reference of 1.8 Gbps; the residual gap to the additive reference is consumed by TLS framing, HTTP headers, and the request-response duty cycle, none of which `iperf3` incurs. Latency does not converge as fully: *Dome without cache* p99 stays at 2× the *baseline* at 1024 KB and tens of seconds at smaller sizes; the cache is what holds the tail at *baseline* levels.

Table 3: LANforge cross-tool campaign: 100 L4 HTTPS endpoints (10 KB) over Wi-Fi with a concurrent 1 Gbps wired TCP background. One repetition per condition.

Condition	URLs/s	L4 Mbps	BG Mbps	CV
baseline	6 180	506	940	2.9%
Dome	5 994	491	940	3.6%
Dome without cache	924	76	739	0.6%

Workload realism. Lua-per-packet classification matches baseline throughput in the large-response regime. Such transfers are common in practice: on-demand video streaming accounts for 54% of fixed broadband downstream traffic [23], delivered as multi-second HLS/DASH adaptive-bitrate segments. The cache extends parity to smaller responses where per-packet Lua would otherwise hit the per-DUT bound. These results characterize a single DUT driven by one Wi-Fi station and one Ethernet generator; many-station and multi-DUT scaling is outside the present scope.

LANforge cross-tool verification. An independent campaign with the LANforge L4 HTTPS endpoint framework on the same hardware corroborates the *Dome* parity claim under a different traffic-generation model: LANforge L4 schedules each URL download independently across 100 endpoints, exercising the bridge chain under a different connection-management model than `wrk2`'s threaded persistent pool. Endpoints download a fixed 10 KB object from `nginx` to the RPi 5 station over Wi-Fi. *Dome* sustains the foreground at 5 994 URLs/s (491 Mbps), tracking the *baseline* of 6 180 URLs/s (506 Mbps) within 3.0% (Table 3). Under *Dome without cache* the foreground falls to 924 URLs/s. Methodology and CSV-derived measurements are in the dataset [34].

Background-flow isolation. Alongside the LANforge L4 foreground, a concurrent LANforge L3 TCP background was offered at 1 Gbps through the DUT bridge over Ethernet. The background uses TCP ports outside Dome's L7 inspection set, so the bridge `dpi` chain does not mark these packets for the Lua hook and they traverse the prerouting pipeline without entering it. The background sustains 940 Mbps under *Dome*, matching the 940 Mbps *baseline*. Dome's L7 classification path leaves no measurable trace on the non-inspected Ethernet flow sharing the AP. The cache is what sustains this isolation; without it, foreground CPU saturation reduces the background to 739 Mbps.

9 Final Remarks

This paper presented a kernel-userspace datapath pipeline for packet filtering on APs, built from three new components, two kernel bindings (`luanetfilter`, `luaskb`) and a userspace library (`luanftables`). These compose with the existing `luanotifier` and `luarcu` bindings for interface lifecycle tracking. The verdict cache lives in nftables sets rather than in Lua state, so once a flow is classified, every subsequent packet of the same flow is matched in native kernel code without returning to Lua.

This enforcement runs without TLS interception: the Lua hook performs DPI on TLS handshake metadata (Section 5), avoiding the per-session cryptographic work and CA distribution that anchor conventional SWG appliances to non-commodity hardware.

Section 8 evaluated the pipeline under a combined Ethernet and Wi-Fi HTTPS workload at 1.4 Gbps aggregate throughput. The cache mechanism delivered parity with the *baseline* (Linux bridge without Dome's Lua hook or nftables ruleset) in throughput and latency, on both the median and the tail, at a $+14$ to $+19$ pp sys% overhead across all response sizes. At the saturated 1.4 Gbps aggregate, Dome's sys% sits in the 45–54% range, well below the four-core ceiling and leaving headroom for additional inspection rules. *Dome without cache* measurements established that the Lua hook is a per-DUT, single-state saturation rather than a per-TG one. Lua-per-packet inspection approached the *baseline* at large response sizes even without the cache; the cache extended parity to the small-object regime. On-demand video streaming, the dominant share of fixed broadband downstream traffic [23], falls within this regime, so the architecture covers a substantial fraction of real traffic even without the cache. An independent LANforge cross-tool campaign corroborated the throughput parity.

Both the cache and the classification state are kernel-resident: the per-flow cache lives in nftables sets, and the domain intelligence base lives in the `kvmalloc`-backed Lua state (Section 2), so the hook completes a verdict without a userspace round-trip, an architectural shift from earlier NFLua deployments [35].

The cache is parametric in the fields nftables can match on (source and destination addresses, ports, protocol numbers, marks, interfaces): a classifier whose decision can be expressed as a function of these fields can write verdicts to an nftables set and have the cache deliver them without further invocation. The same mechanism applies in production to signature-based classifiers for BitTorrent, OpenVPN, and WireGuard. The same pattern, a Lua hook gating a verdict cache, is also substrate-agnostic: it runs on XDP/eBPF in Dome's mobile-edge OpenRAN deployments via `luaxdp`, where line rate is the binding constraint.

Dome ships in production as the in-kernel engine of NetExperience's Secure Wireless Gateway [5] on OpenWiFi access points.

Acknowledgments

We thank the Google Summer of Code program for sponsoring Lua-in-kernel projects from 2010 to 2026, including `luanetfilter`; these were mentored primarily by LabLua, and the 2010 project by the NetBSD Foundation. We thank Savio Sena, Jacques Peron, Ashwani Kamal, and Carlos Carvalho for their ongoing contributions to Lunatik, and Jamal Hadi Salim and Iruatã Souza for reviewing preliminary drafts. We thank the Brazilian National Research and Education Network (RNP) for its continued support.

References

[1] Ahmad, S., and Wu, P. 2025. Measuring characteristics of TCP connections at Internet scale. The Cloudflare Blog, https://blog.cloudflare.com/measuring-network-connections-at-scale/.

[2] Candela Technologies. 2024. LANforge-GUI User Guide. https://www.candelatech.com/lfgui_ug.php.

[3] Candela Technologies. 2026. CT840a: large 2D-turntable LANforge chamber. https://www.candelatech.com/ct840a_product.php.

[4] Chandramouli, R. 2022. Guide to a secure enterprise network landscape. NIST Special Publication 800-215, National Institute of Standards and Technology.

[5] Chenier, M., and Rees, H. 2022. The NetExperience cloud management and controller platform: a disaggregated approach to WLAN management and control. NetExperience White Paper, https://www.netexperience.com/wp-content/uploads/2022/06/NetExp-White-Paper-FinalHR062222.pdf.

[6] Crovella, M. E., and Bestavros, A. 1997. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking* 5(6):835–846.

[7] CUJO AI. 2019. NFLua: a Netfilter extension module for Lua-based packet filtering. https://github.com/cujoai/nflua.

[8] Damasceno Barbosa, A. G.; Valle, R.; Bruno, G.; de Moura, M. S. A.; and Calabria, L. 2026. Testbed OpenRAN Brasil como plataforma de inovação aberta para desenvolvimento tecnológico da indústria. In *5º Congresso Latino-Americano de Casos de Open Innovation*.

[9] Dean, J., and Barroso, L. A. 2013. The tail at scale. *Communications of the ACM* 56(2):74–80.

[10] Durumeric, Z.; Ma, Z.; Springall, D.; Barnes, R.; Sullivan, N.; Bursztein, E.; Bailey, M.; Halderman, J. A.; and Paxson, V. 2017. The security impact of HTTPS interception. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[11] Edgecore Networks. 2026. EAP101: Wi-Fi 6 indoor access point. https://wifi.edge-core.com/access-point/wifi-6-access-point/eap101/.

[12] Efron, B., and Tibshirani, R. J. 1993. *An Introduction to the Bootstrap*. Chapman & Hall.

[13] ESnet, and Lawrence Berkeley National Laboratory. 2024. iperf3. https://software.es.net/iperf/.

[14] Farias, F. N. N.; Borges, L.; and Silva, M. 2025. Do conceito à prática: Softwarização e orquestração de redes Open RAN no testbed OpenRAN@Brasil. Simpósio Brasileiro de Telecomunicações e Processamento de Sinais (SBrT). https://sbrt2025.sbrt.org.br/wp-content/uploads/2025/10/Minicurso-OpenRAN-SBrT.pdf.

[15] Judis, S., and Portis, E. 2024. Media. In *The 2024 Web Almanac*. HTTP Archive. https://almanac.httparchive.org/en/2024/media.

[16] Lunatik Project. 2026. Lunatik: a framework for scripting the Linux kernel with Lua. https://github.com/luainkernel/lunatik.

[17] Mazhar, M. H., and Shafiq, Z. 2020. Characterizing smart home IoT traffic in the wild. In *IEEE/ACM International Conference on Internet-of-Things Design and Implementation (IoTDI)*.

[18] McKenney, P. E., and Walpole, J. 2007. What is RCU, fundamentally? https://lwn.net/Articles/262464/.

[19] Netfilter Project. 2024. The netfilter.org "nftables" project. https://netfilter.org/projects/nftables/.

[20] Nginx Project. 2024. nginx. https://nginx.org/.

[21] OpenWrt. 2024. ubus: OpenWrt micro bus architecture. https://openwrt.org/docs/techref/ubus.

[22] Ring Zero Networks. 2026. luanftables: Lua binding for libnftables — programmatic access to nftables from Lua. https://github.com/ring0networks/luanftables.

[23] Sandvine. 2024. The global internet phenomena report. https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2024/GIPR/GIPR%202024.pdf. Top Content Categories by Downstream Volume (Fixed).

[24] Schwartz, B.; Bishop, M.; and Nygren, E. 2023. RFC 9460: Service binding and parameter specification via the DNS (SVCB and HTTPS resource records). https://www.rfc-editor.org/rfc/rfc9460.

[25] Smart, D., and Indigo, J. 2024. Page weight. In *The 2024 Web Almanac*. HTTP Archive. https://almanac.httparchive.org/en/2024/page-weight.

[26] Telecom Infra Project. 2024a. Telecom infra project. https://telecominfraproject.com/.

[27] Telecom Infra Project. 2024b. TIP OpenWiFi. https://openwifi.tip.build/.

[28] Telecom Infra Project. 2026a. uCentral Protocol Specification. https://github.com/Telecominfraproject/wlan-cloud-ucentralgw/blob/master/PROTOCOL.md.

[29] Telecom Infra Project. 2026b. wlan-ucentral-schema: configuration schema and renderer for uCentral. https://github.com/Telecominfraproject/wlan-ucentral-schema.

[30] Tene, G. 2024. wrk2: a constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[31] Vieira Neto, L.; Ierusalimschy, R.; de Moura, A. L.; and Balmer, M. 2014. Scriptable operating systems with Lua. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS)*.

[32] Vieira Neto, L.; Nogueira, V.; de Moura, A. L.; and Ierusalimschy, R. 2020. Linux network scripting with Lua. In *Proceedings of Netdev 0x14*.

[33] Vieira Neto, L.; Moura, M.; de Moura, A. L.; and Ierusalimschy, R. 2023. Scripting the Linux routing table with Lua. In *Proceedings of Netdev 0x17*.

[34] Vieira Neto, L.; Tausif, M. S. Y.; Shaari, F.; de Moura, M. S. A.; and Alam, A. 2026. Scripting Netfilter with Lua: measurement data. https://github.com/luainkernel/netdev-0x1A.

[35] Vieira Neto, L. 2017. CUJO – Safe Browsing with Lua. Lua Workshop 2017, https://www.lua.org/wshop17/Lourival.pdf.

[36] Vieira Neto, L. 2024. Is eBPF driving you crazy? Let it run Lunatik instead! https://medium.com/p/4aca7d63e6fd.

[37] Vieira Neto, L. 2025. From the kernel to the moon: a journey into Lunatik bindings. https://medium.com/p/c2cac8816f9e.