

Network and Infrastructure Security: A Comparative Performance and Security Analysis of VPN vs. SSH Tunneling

Hubert YAKPE TROLINDO

hu881229@ucf.edu

University of Central Florida, Orlando, Florida, USA

ABSTRACT

Layer-3 VPNs and SSH tunnels are often deployed interchangeably to protect data in transit, without measured evidence of what each costs. We built an automated, single-host testbed of three Linux network namespaces with a custom loss/delay emulator and compared WireGuard, OpenVPN (UDP and TCP), SSH port forwarding and SSH layer-3 tunnelling against an unencrypted baseline over 3–5 randomized trials. On a clean link, SSH port forwarding reached 8.60 Gbit/s with AES-GCM, userspace WireGuard 1.58 Gbit/s and OpenVPN about 727 Mbit/s; changing OpenVPN’s cipher did not change its throughput, indicating per-packet processing rather than cryptography as the bottleneck. Under random loss with CUBIC, every tunnel tracked plain TCP within about 10% and no TCP-over-TCP collapse appeared; with BBR, however, TCP-in-TCP tunnels lost 28–43% of throughput. Security differs more than speed: WireGuard and OpenVPN with `tls-crypt` stayed silent to unauthenticated probes, whereas SSH disclosed its software version. We conclude with deployment guidelines.

Code & artifacts: <https://github.com/cybsecoff31/vpn-vs-ssh-tunnel-benchmark>

CCS Concepts: • Security and privacy → Network security; • Networks → Network performance evaluation.

Keywords: VPN, SSH tunneling, WireGuard, OpenVPN, TCP-over-TCP, benchmarking, attack surface

1 INTRODUCTION & PROBLEM STATEMENT

Remote work, multi-cloud deployments and fleets of edge gateways have made encryption of data in transit a default requirement rather than an option. Two architectural families dominate in practice: *layer-3 virtual private networks* (VPNs), which encapsulate whole IP packets in an encrypted envelope, and *Secure Shell (SSH) tunnelling*, which multiplexes application byte streams over one authenticated TCP connection [21, 22]. Administrators frequently choose between them by familiarity—an `ssh -L` command is one line—rather than by measured cost, even though the two differ in where packets are processed, which transport carries them, and how much attack surface they expose.

Problem statement. There is little controlled, reproducible evidence that compares modern VPNs and SSH tunnels on the *same* host, link and workload, and that reports performance and security side by side. This report asks:

- (1)**RQ1 (efficiency):** How much throughput, latency and CPU does each tunnel cost relative to an unencrypted baseline on an uncongested link?
- (2)**RQ2 (robustness):** How does throughput degrade as random packet loss grows, and does carrying TCP inside TCP cause the “meltdown” described in the literature [8, 19]?

- (3)**RQ3 (security):** How do the tunnels differ in handshake cost, exposure to unauthenticated probing, cryptographic agility and code size?

Contributions. We (i) release an automated, single-host namespace testbed with a userspace loss/delay emulator; (ii) measure six tunnel configurations plus two cipher variants over 3–5 randomized trials with 95% confidence intervals; (iii) revisit two causal claims from our milestone—user-space context-switching and SSH “TCP-over-TCP meltdown”—and qualify them with evidence; and (iv) derive a concrete security comparison and deployment guide.

2 RELATED WORK

VPN performance. Donenfeld introduced WireGuard as a small in-kernel tunnel built on the Noise framework and reported higher throughput than IPsec and OpenVPN [5, 14]. Mackey et al. confirmed WireGuard’s throughput advantage over OpenVPN on commodity hardware [11], and Pudelfko et al. showed that OpenVPN is limited by its single-threaded user-space data path while kernel WireGuard and IPsec scale differently with cores and packet sizes, and that small packets expose per-packet costs most clearly [15]. These studies compare VPNs with one another; none includes SSH tunnels or a loss sweep. OpenVPN has since added Data Channel Offload to move encryption into the kernel [16].

TCP-over-TCP. Titz argued that stacking TCP on TCP lets the outer connection’s retransmissions inflate the inner connection’s timers, causing collapse under loss [19]. Honda et al. measured this effect and found it depends strongly on loss rate and buffer sizes [8]; Kim et al. later showed that congestion-control choice and transfer buffers change the outcome [9]. Loss-based TCP throughput itself scales roughly with $1/\sqrt{p}$ [12], which is why CUBIC [7] and model-based BBR [4] respond very differently to random loss.

SSH throughput and security. Rapier and Bennett showed that SSH’s fixed channel windows throttle bulk transfers on high bandwidth-delay paths and proposed HPN-SSH [17]. On the security side, WireGuard’s handshake has been analysed computationally [6] and mechanically verified [10]; SSH has suffered plaintext-recovery [1] and prefix-truncation (Terrapin) attacks [2]; and OpenVPN has been hit by 64-bit block-cipher collisions (Sweet32) [3] and is fingerprintable by active probing [20]. Prior work studies these dimensions separately; we measure them together on one testbed.

3 METHODOLOGY

Figure 1 summarizes the pipeline: tunnel configurations and network conditions (inputs) are deployed on a three-namespace testbed, exercised by performance and security probes, and reduced to the metrics of Section 4.1 (outputs).

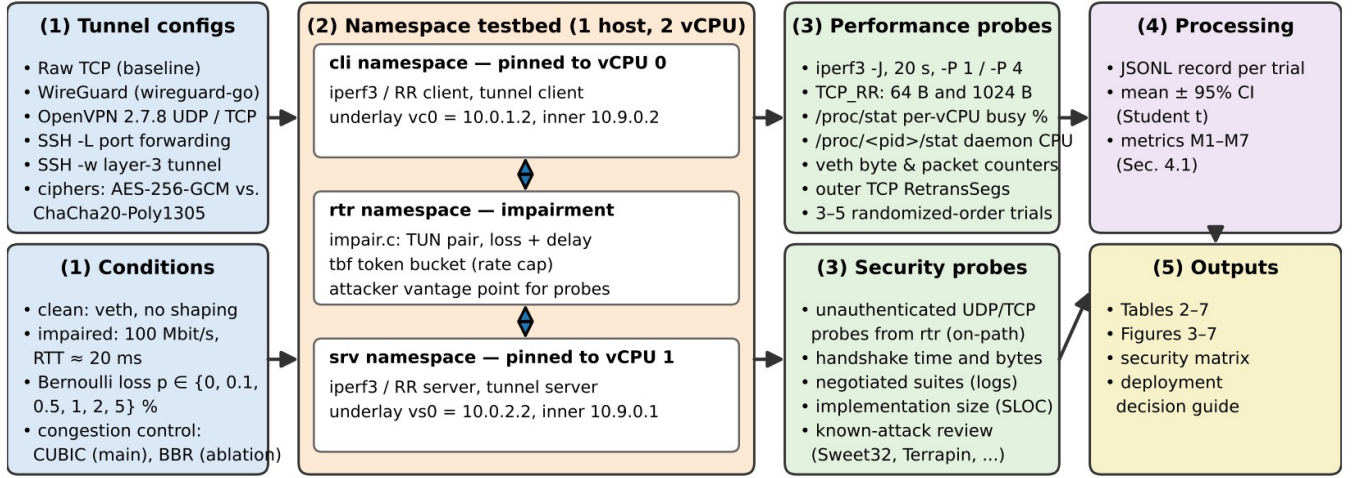


Figure 1. Experimental pipeline. Inputs (1) are deployed on a three-namespace testbed (2); performance and security probes (3) produce per-trial records that are aggregated (4) into the tables, figures and guidelines of Sections 4–5 (5).

3.1 Testbed

All experiments run on one Linux 6.18 virtual machine (2 vCPUs, Intel Xeon @ 2.8 GHz with AES-NI, 7 GiB RAM). Three network namespaces—client (*cli*), router (*rtr*) and server (*srv*)—are joined by veth pairs (MTU 1500), with routed underlay subnets 10.0.1.0/24 and 10.0.2.0/24 and tunnel addresses in 10.9.0.0/24. Client-side processes are pinned to vCPU 0 and server-side processes to vCPU 1 with *taskset*, so each endpoint behaves like a single-core gateway. All tools were built from source: OpenVPN 2.7.8 (OpenSSL 3.0.13), OpenSSH 10.6p1, wireguard-go (commit b3bc704), iperf3 3.22 [23] and iproute2.

Impairment. The kernel lacks the *netem* queueing discipline, so we wrote *impr*, a 70-line C forwarder that sits between two TUN devices in *rtr* and applies independent Bernoulli loss and a fixed one-way delay in each direction; policy routing steers all transit traffic through it. A *tbf* token bucket (64 KB burst, 512 KB queue) caps the bottleneck at 100 Mbit/s. A sanity check with plain TCP reproduced the expected $1/\sqrt{p}$ collapse (96, 5.8 and 2.8 Mbit/s at 0, 1 and 2% loss) [12].

3.2 Tunnel Configurations

Table 1 lists the configurations and Figure 2 their protocol stacks. Two SSH modes are included deliberately: *-L* *port forwarding* terminates the application’s TCP at the SSH endpoints and carries only bytes, whereas *-w* creates a *layer-3 TUN tunnel* that carries whole TCP/IP packets inside the SSH TCP stream—the genuine TCP-over-TCP case [19]. *iperf3* has no SOCKS support, so dynamic forwarding (*-D*) is represented by *-L*; both open identical direct-tcpip channels [22].

Table 1. Tunnel configurations under test.

ID	Layer / transport	Crypto (data / handshake)	Path
Raw	none / TCP	—	kernel
WG	L3 / UDP	ChaCha20-Poly1305 / Noise IK, X25519	user (wg-go)
OV-U	L3 / UDP	AES-256-GCM / TLS 1.3, RSA-4096	user
OV-U-C	L3 / UDP	ChaCha20-Poly1305 / same	user
OV-T	L3 / TCP	AES-256-GCM / same	user
SSH-L	L7 / TCP	chacha20-poly1305@openssh / Ed25519	user
SSH-L-A	L7 / TCP	aes256-gcm@openssh / Ed25519	user
SSH-W	L3 / TCP	chacha20-poly1305@openssh / Ed25519	user

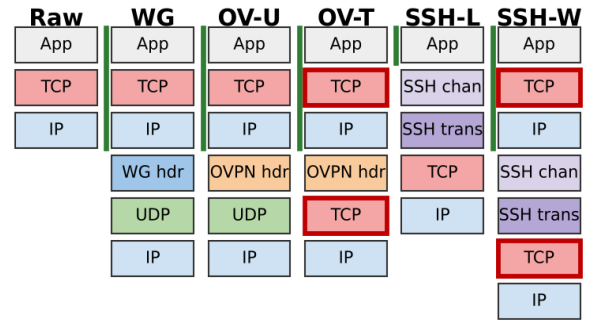


Figure 2. Protocol stacks as seen on the wire. Green bars mark encrypted content; red frames mark the two configurations in which TCP is carried inside TCP. SSH-L terminates the application’s TCP locally, so only one TCP crosses the network.

OpenVPN negotiates a TLS 1.3 control channel [18] with *tls-crypt*, *--dh none* (ECDHE X25519) and its default *mssfix*; the ChaCha20-Poly1305 variants use the AEAD of RFC 8439 [13]; WireGuard uses MTU 1420; SSH disables compression. All keys were generated fresh for the study.

3.3 Workloads and Procedure

- Bring-up.** Tear down any tunnel, record underlay counters, start the tunnel, and poll a TCP connect every 50 ms until the inner path answers (records setup time and bytes).

2. **Bulk transfer.** iperf3 upload for 20 s after a 2 s warm-up, with one stream (-P 1, 5 trials) and four streams (-P 4, 3 trials). CPU busy time per vCPU (/proc/stat), tunnel-daemon CPU (/proc/<pid>/stat), underlay bytes and TCP retransmissions are sampled before and after.
3. **Request/response.** A TCP_RR probe with TCP_NODELAY exchanges 3000 messages of 64 B and of 1024 B and records median and 99th-percentile round-trip time.
4. **Loss sweep.** On the impaired link (100 Mbit/s, 20 ms RTT) each tunnel is measured at $p \in \{0, 0.1, 0.5, 1, 2, 5\}\%$ with three trials and a fresh random seed per trial; the default congestion control is CUBIC, with a BBR ablation at 1% and 5%.
5. **Security probes.** From rtr (an on-path attacker without keys) we send random and protocol-shaped datagrams/segments to each listener and record whether anything is returned; we also log negotiated suites and count source lines (SLOC) of each implementation.

3.4 Threat Model for the Security Analysis

The adversary sits on the path between client and server (the rtr namespace). It can observe, drop, inject and replay packets and can open connections to any listening port, but it holds no keys and cannot compromise endpoints. Its goals are to (a) detect and identify the tunnel service, (b) elicit responses or computation before authentication, and (c) exploit known protocol weaknesses. Cryptanalysis of primitives and endpoint compromise are out of scope; for those we rely on the published analyses cited in Section 4.4.

3.5 Statistical Treatment

Configurations run in a randomized order within each trial to spread slow drift (thermal, noisy neighbours) across all tunnels; each loss trial uses a fresh random seed. Every trial is stored as one JSON record. For n trials with mean $\bar{x}$ and standard deviation s we report $\bar{x} \pm t_{0.975, n-1} \cdot s/\sqrt{n}$ (95% Student-t interval) and call two configurations different only when their intervals do not overlap.

4 EVALUATION & RESULTS

4.1 Metrics

Each metric is computed per trial and then averaged; notation: B_{rx} bytes received by the application in t_m measured seconds, B_{wire} bytes sent on the client's underlay interface in wall time t_w , U_0 , U_1 the busy percentage of vCPU 0 and 1.

1. **Sustained throughput (M1).** Application-level goodput reported by the iperf3 receiver after warm-up. It is the quantity users experience.

$$T = 8 \cdot B_{rx} / t_m \quad [\text{Mbit/s}] \quad (1)$$

2. **Latency overhead (M2).** Extra median request/response time over the raw path for a message of size s . Medians resist scheduling outliers; we also report the 99th percentile.

$$\Delta RTT(s) = RTT_{cfg}(s) - RTT_{raw}(s) \quad [\mu s] \quad (2)$$

3. **CPU efficiency (M3).** Throughput per percent of one vCPU, counting both endpoints (0–200%). It captures user and kernel work, so kernel-heavy and daemon-heavy designs are judged on the same scale.

$$E = T / (U_0 + U_1) \quad [\text{Mbit/s per vCPU-}\%] \quad (3)$$

4. **Goodput ratio (M4).** Share of bytes on the wire that is useful payload; it measures encapsulation and header overhead including Ethernet framing.

$$G = (B_{rx}/t_m) / (B_{wire}/t_w) \times 100 \quad [\%] \quad (4)$$

5. **Loss retention (M5).** Throughput at loss p relative to plain TCP at the same p , which isolates the tunnel's own effect from TCP's $1/\sqrt{p}$ law [12]; $R \approx 1$ means no extra penalty, $R \ll 1$ indicates collapse.

$$R(p) = T_{cfg}(p) / T_{raw}(p) \quad (5)$$

6. **Cycles per byte (M6).** CPU cycles spent per delivered payload byte at clock $f = 2.8$ GHz. Unlike E it is independent of link speed and lets us project the ceiling of slower edge processors.

$$C_B = ((U_0 + U_1)/100) \cdot f / (T/8) \quad [\text{cycles/B}] \quad (6)$$

7. **Setup cost (M7).** Time and underlay bytes from process start until the first inner TCP connection succeeds; it reflects handshake complexity and matters for short-lived or on-demand tunnels.

4.2 Efficiency on a Clean Link (RQ1)

Table 2 and Figure 3 report the clean-link results. The raw veth path reaches 28.07 Gbit/s because the kernel moves 64 KB segments without per-packet work. Three groups emerge.

Table 2. Clean-link results, one stream (mean $\pm$ 95% CI, $n = 5$). Raw row shows absolute RTT in parentheses.

Config	T (Mbit/s)	U_0+U_1 (%)	E	C_B	G (%)	ΔRTT_{64} (μs)	Setup (s)
Raw	28,065 $\pm$ 1,442	164	171.0	1.3	99.0	(60)	0.04
WG	1,581 $\pm$ 173	138	11.4	19.6	92.4	+113	0.09
OV-U	727 $\pm$ 96	171	4.3	52.6	91.6	+48	2.25
OV-U-C	742 $\pm$ 107	172	4.3	51.8	92.6	+46	2.00
OV-T	652 $\pm$ 102	175	3.7	60.2	90.2	+43	2.71
SSH-L	4,095 $\pm$ 121	190	21.5	10.4	99.9	+61	0.44
SSH-L-A	8,596 $\pm$ 297	187	45.9	4.9	100.0	+52	0.45
SSH-W	575 $\pm$ 43	185	3.1	72.2	90.1	+52	0.47

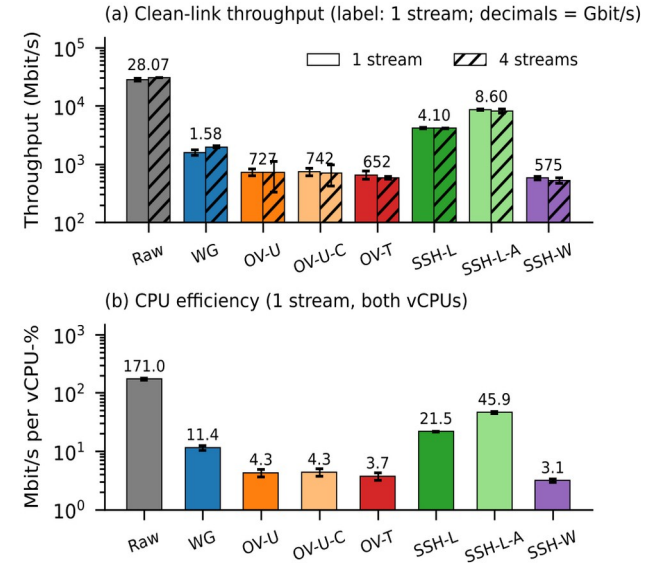


Figure 3. Clean-link throughput with one and four streams (a) and CPU efficiency (b), log scale; error bars are 95% CIs.

(i) **Byte-stream SSH forwarding is fastest.** SSH-L moved 4.10 Gbit/s with ChaCha20-Poly1305 and 8.60 Gbit/s with AES-256-GCM, with near-perfect goodput (99.9%). Because it relays bytes, the kernel can still coalesce data into large segments, so ssh and sshd perform a few large reads, encryptions and writes rather than one per packet.

(ii) **Packet tunnels are bounded by per-packet work.** WireGuard reached 1.58 Gbit/s (1.97 with four streams); OpenVPN-UDP 727,

OpenVPN-TCP 652 and SSH-W 575 Mbit/s, with both vCPUs above 85% busy. Swapping OpenVPN’s cipher from AES-256-GCM to ChaCha20-Poly1305 left throughput unchanged (727 vs. 742 Mbit/s, overlapping CIs), whereas in SSH-L moving from ChaCha20-Poly1305 to AES-256-GCM (AES-NI) doubled the rate. Cryptography therefore dominates only once per-packet overhead is removed.

(iii) Context switches explain the gap. A supplementary run counted system-wide context switches per megabyte delivered (3 trials, 10 s): Raw 10, SSH-L 43, WG 105, OV-U 1,350, OV-T 1,317, SSH-W 1,361. OpenVPN and SSH-W wake up roughly once per packet; wireguard-go batches packets across reads and writes and incurs about 13× fewer switches.

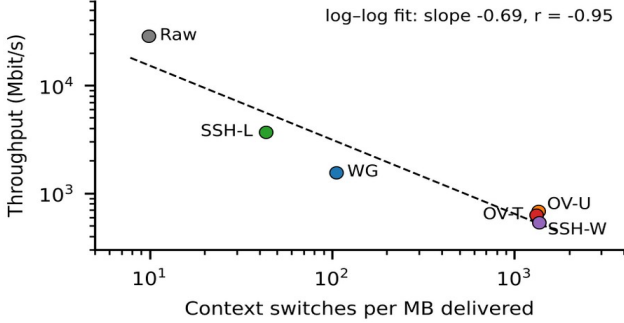


Figure 4. Throughput vs. context switches per delivered megabyte (mean of 3 runs, log-log). The fit has slope -0.69 and $r = -0.95$.

Across the six configurations throughput falls almost monotonically with switching rate ($r = -0.95$, Figure 4). Correlation over six points does not prove causation by itself, but combined with the cipher-swap control and the saturated CPUs it is the most consistent explanation of the gap.

(iv) Multi-core scaling. Only WireGuard benefited from four parallel streams (1.58 → 1.97 Gbit/s, +25%), because wireguard-go spreads encryption across goroutines. OpenVPN, ssh and sshd each process a tunnel in one thread, so extra streams only share the same core (OV-U 712, SSH-L 4.09 Gbit/s with four streams).

Edge-gateway projection. M6 converts these results into a hardware budget. Per payload byte, OpenVPN-UDP cost 53 cycles, SSH-W 72, WireGuard 20, SSH-L 10.4 and raw forwarding 1.3. On a single 1 GHz core and assuming similar cycles per byte (a rough approximation across instruction sets), the same software would saturate near 152 Mbit/s for OpenVPN and 409 Mbit/s for WireGuard—an order-of-magnitude gap that matters for low-power gateways exposed to traffic floods.

Overhead and latency. Measured goodput ratios match header arithmetic: WireGuard’s 32-byte header and tag plus UDP/IP outer headers predict 91.6% for 1368-byte payloads; we measured 92.4%. All tunnels added 43–113 μ s to a 64-byte round trip; WireGuard’s higher value reflects goroutine scheduling in wireguard-go. On the 20 ms path the extra delay was below 0.4 ms for every tunnel.

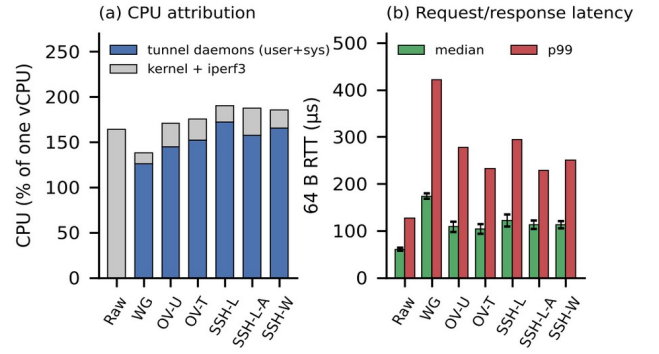


Figure 5. CPU attribution during one-stream transfers (a) and 64-byte request/response latency (b). Daemon time includes system calls made by ssh/sshd, openvpn or wireguard-go.

Where the CPU goes. Figure 5a shows that the tunnel daemons themselves consume 145–165% of one vCPU for OpenVPN and SSH-W, close to the two-core budget, while wireguard-go needs 126% for more than twice the throughput. Tail latency (Figure 5b) is 2–3× the median for every tunnel and highest for WireGuard (p99 421 μ s), again a scheduler effect of the userspace implementation.

4.3 Robustness to Packet Loss (RQ2)

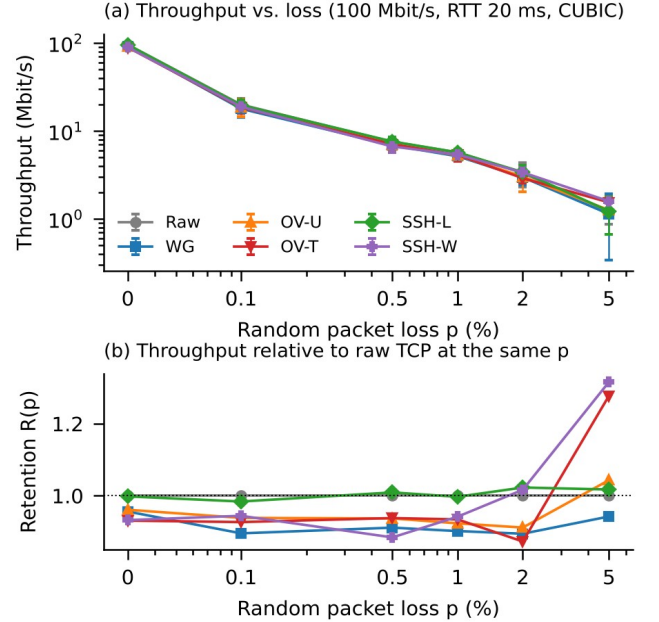


Figure 6. Loss sweep on a 100 Mbit/s, 20 ms RTT link with CUBIC ($n = 3$): absolute throughput (a) and retention $R(p)$ relative to raw TCP (b).

Figure 6 shows that loss, not the tunnel, dominates: from 0.1% loss upward every configuration falls along the same $1/\sqrt{p}$ curve as plain TCP [12], and retention stays between 0.87 and 1.02 up to 2%. At 0% the ~5–10% gap of the packet tunnels equals their header overhead.

Table 3. Raw TCP on the emulated link vs. the Mathis model $T \approx (MSS/RTT) \cdot \sqrt{(3/2)/p}$ [12] (MSS 1448 B, measured RTT).

p (%)	0.1	0.5	1	2	5
Model (Mbit/s)	21.4	9.5	6.8	4.8	3.0
Measured (Mbit/s)	19.9	7.5	5.7	3.3	1.2

The emulator reproduces the expected trend; measured values sit 7–30% below the model up to 2% loss and fall further at 5%, where retransmission timeouts—ignored by the simple model—dominate.

No TCP-over-TCP collapse with CUBIC. For the two genuine TCP-in-TCP tunnels (OV-T, SSH-W) the inner connection’s own retransmissions fell to zero at $p \geq 2\%$ (outer: 641 and 199 per run): the outer TCP repairs every loss, and the inner flow sees only delay. Their throughput at 5% loss (1.54 and 1.58 Mbit/s) was not below raw TCP (1.20). SSH-L, which is not TCP-over-TCP, behaved exactly like raw TCP.

Table 4. Mean retransmitted segments per 22 s run, inner / outer TCP (CUBIC, $n = 3$). For Raw, WG and OV-U the inner flow is the only TCP on the lossy path.

Config	$p = 0.5\%$	$p = 2\%$	$p = 5\%$	R(2%)
Raw	84 / —	121 / —	138 / —	1.00
WG	77 / —	117 / —	126 / —	0.89
OV-U	81 / —	114 / —	144 / —	0.91
OV-T	551 / 355	0 / 641	0 / 440	0.87
SSH-L	46 / 113	23 / 208	8 / 254	1.02
SSH-W	37 / 1,228	0 / 199	0 / 341	1.02

In SSH-L the “inner” legs run over loopback, so its few inner retransmissions are local; its outer count is the SSH connection itself.

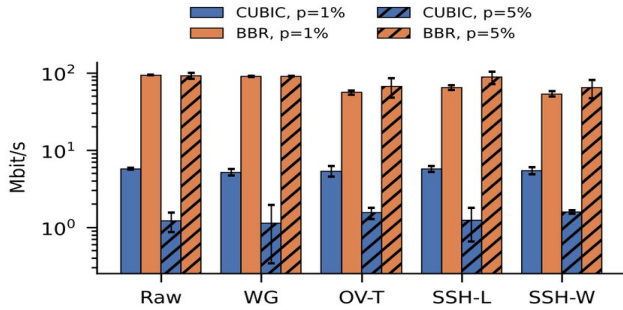


Figure 7. Congestion-control ablation at 1% and 5% loss ($n = 3$). With BBR the UDP tunnel keeps pace with raw TCP while TCP-in-TCP tunnels lose 28–43%.

BBR exposes a stacking penalty. BBR ignores random loss, so raw TCP and WireGuard sustained 93 and 90 Mbit/s at 1% loss (Figure 7). The TCP-based tunnels reached only 55 (OV-T), 53 (SSH-W) and 64 Mbit/s (SSH-L). Here the outer connection’s head-of-line blocking inflates the delay samples of the inner flow, a milder form of the interaction predicted by Titz and by Kim et al. [9, 19]. When the bottleneck is CUBIC’s loss response, this penalty is hidden.

4.4 Security Comparison (RQ3)

Table 5 combines our probes with the literature. From the attacker position, WireGuard answered none of 2 unauthenticated datagrams, including a correctly typed handshake-initiation message, because responses require a valid MAC keyed by the server’s public key [5]. OpenVPN with `tls-crypt` also stayed silent; *without* `tls-crypt` the same reset packet drew a 26-byte reply, exactly the behaviour exploited for fingerprinting [20]. `sshd` returned its banner, `SSH-2.0-OpenSSH_10.6`, to any TCP client, disclosing product and version. OpenSSH negotiated the hybrid post-quantum key exchange `mlkem768x25519-sha256` and applies strict-KEX counter-measures against Terrapin [2].

Table 5. Security comparison (measured items marked †).

Property	WireGuard	OpenVPN	SSH
Handshake	Noise IK, 1-RTT, X25519 [14]	TLS 1.3 + X.509 (RSA-4096) [18]	ML-KEM-768+X25519†, Ed25519
Crypto agility	none (fixed suite)	negotiated; legacy BF-CBC risk [3]	negotiated
Formal / attacks	proofs [6, 10]	Sweet32 [3], fingerprint [20]	Terrapin [2], CBC leak [1]
Unauth. probe†	silent	silent (<code>tls-crypt</code>); 26 B reply w/o	version banner
Access scope	routed subnet	routed subnet	-L: one port; -w: subnet
Setup† (s / KB)	0.09 / 1.9	2.25 / 11.6	0.44 / 9.4
SLOC†	8.0 k (wireguard-go)	86.8 k + OpenSSL	125.9 k

Handshake cost. Tunnel establishment differed by more than an order of magnitude: WireGuard needed 0.09 s and 1.9 KB (one round trip of fixed-size messages), SSH 0.44 s and 9.4 KB, and OpenVPN 2.00–2.71 s and up to 13.6 KB, dominated by the certificate chain and RSA-4096 signatures of the TLS exchange. Expensive pre-authentication work is also a denial-of-service lever, which `tls-crypt` and WireGuard’s cookie mechanism are designed to blunt [5].

Two further differences matter in practice. First, scope: SSH-L exposes one service, which enforces least privilege by construction, whereas both VPNs grant routed access to a subnet unless filtered. Second, attack surface: `wireguard-go` is an order of magnitude smaller than OpenVPN or OpenSSH, and OpenVPN additionally links a full TLS library; small code eases review but does not by itself prove security.

5 DISCUSSION & CHALLENGES

5.1 Key Insights

- 1. Packet granularity, not user space, sets the cost.** Our milestone attributed OpenVPN’s limit to user-space context switching. The data support a narrower claim: OpenVPN does switch context roughly once per packet, but SSH-L also runs in user space and is the fastest tunnel. What matters is how many system calls and wake-ups each byte costs. Batching (`wireguard-go`) or kernel offload such as OpenVPN DCO [16] attack exactly this.
- 2. “Meltdown” is conditional.** The milestone claimed an SSH collapse at 2% loss. We could not reproduce it: SSH-L is not TCP-over-TCP at all, and the genuine TCP-in-TCP tunnels matched raw TCP under CUBIC. A real penalty of 28–43% appears only with a loss-insensitive controller (BBR), consistent with reports that the effect depends on congestion control and buffers [8, 9].
- 3. Security, not speed, should drive most choices.** Above a few hundred Mbit/s every tunnel is fast enough for typical remote-access links, and latency costs are sub-millisecond. The decisive differences are silence to probes, access scope, agility and code size (Table 5).
- 4. The two risks in our problem statement need re-weighting.** *Performance bottlenecks* from TCP forwarding did not materialize for SSH port forwarding, which was the fastest tunnel; they did appear for the genuine TCP-in-TCP tunnels, but only under BBR. *Resource saturation*, by contrast, is real and large: per-packet tunnels spent 53–72 cycles per byte and saturated both cores at well under 1 Gbit/s, so a low-power gateway running OpenVPN or SSH -w is the configuration most exposed to CPU exhaustion and flooding.

5. **Speed and security do not trade off here.** WireGuard was both the most efficient packet tunnel and the most probe-resistant, while SSH-L combined top throughput with the narrowest access scope but the most visible fingerprint. The real tension is between network-wide access (VPNs) and per-service least privilege (SSH-L), which is an architectural choice rather than a performance one.

Table 6. Milestone claims revisited against the final measurements.

Milestone claim	Final evidence	Revised claim
WireGuard 942 Mbit/s at 14% CPU (kernel)	wireguard-go 1.58 Gbit/s at 138% of 200%	efficient via batching; kernel variant untested
OpenVPN 480 Mbit/s, one thread at 100%, context switching	727 Mbit/s; 1,350 vs 105 switches/MB; cipher-neutral	per-packet wake-ups limit OpenVPN (supported, narrowed)
SSH 310 Mbit/s; meltdown at 2% loss	SSH-L 4.10 Gbit/s; $R(2\%) = 1.02$	no meltdown under CUBIC; 28–43% loss for TCP-in-TCP under BBR

The preliminary milestone numbers came from single, undocumented runs; Table 6 replaces them with the repeated, configuration-documented measurements requested in the milestone review.

Table 7. Deployment guide derived from Sections 4.2–4.4.

Scenario	Recommendation	Reason
Site-to-site, edge gateway	WireGuard	best packet-tunnel E; silent; small code
Admin access to one service	SSH -L	least privilege; fastest single flow; hide port
UDP blocked, TCP/443 only	OpenVPN-TCP	parity under CUBIC; avoid BBR on lossy paths
Enterprise PKI, full network	OpenVPN-UDP + tls-crypt	certificate integration; probe-silent; DCO
Production L3 over SSH	avoid SSH -w	lowest E; root and PermitTunnel needed

5.2 Challenges

- **Restricted environment.** Package mirrors were unreachable, so OpenVPN, OpenSSH, iperf3, wireguard-tools, wireguard-go (with locally mirrored Go modules), libcap-ng and iproute2 were compiled from source.
- **No netem, no kernel WireGuard.** We wrote the TUN-based impair emulator and validated it against the $1/\sqrt{p}$ law; WireGuard ran as wireguard-go.
- **Default BBR.** The VM shipped with BBR, which masked loss effects entirely; we switched the namespaces to CUBIC for the main sweep and kept BBR as an ablation.
- **Measurement artefacts.** Back-to-back iperf3 runs through SSH-L at 5% loss failed, most likely because data still buffered in the SSH channel kept the server busy; fresh tunnels per trial fixed this. OpenVPN’s point-to-point TCP mode refuses new connections once a peer is attached, so its probe was run before the client connected.

5.3 Limitations and Improvements

- **Single host.** Both endpoints share one VM; veth segmentation offload favours byte-stream transports, so absolute raw and SSH-L rates exceed what a physical NIC would allow.

- **Userspace WireGuard.** Kernel WireGuard and OpenVPN-DCO avoid TUN copies and would likely narrow the gaps [5, 15, 16].
- **Simple impairment.** Loss is Bernoulli at one RTT; bursty (Gilbert–Elliott) loss, RTT sweeps and longer runs would strengthen RQ2.
- **Workload coverage.** We measured uploads with bulk TCP and small request/response messages only; UDP applications, downloads and mixed traffic were not tested, and loss sweeps used $n = 3$, giving wide intervals at 5% loss.
- **Probe overhead.** The Python RR probe adds a constant $\approx 50 \mu\text{s}$ to every round trip; ΔRTT cancels it, but absolute RTTs are pessimistic. Setup times have 50 ms polling granularity.
- **Security depth.** Probing and literature review are not cryptanalysis; a handshake-flood DoS experiment is a natural next step.

5.4 Hardening Recommendations

Our probes translate into concrete configuration advice:

- **WireGuard:** keep AllowedIPs to the minimum prefixes and filter forwarded traffic, because the tunnel otherwise grants subnet-wide access.
- **OpenVPN:** always enable tls-crypt (it removed the only probe response we observed), restrict data-ciphers to AEAD suites to exclude Sweet32-prone 64-bit ciphers [3], and prefer DCO where the kernel supports it [16].
- **SSH:** disable password authentication, restrict forwarding with PermitOpen or per-key permitopen=, keep OpenSSH at a version with strict key exchange against Terrapin [2], and hide the listener behind a firewall or VPN since the version banner cannot be suppressed.

6 CONCLUDING REMARKS

We compared WireGuard, OpenVPN and two SSH tunnelling modes on a reproducible namespace testbed, with repeated trials and confidence intervals. Three takeaways stand out. First, per-packet processing dominates cost: byte-stream SSH forwarding and batching WireGuard outperform per-packet OpenVPN and SSH -w regardless of cipher. Second, TCP-over-TCP meltdown did not appear under CUBIC, but stacking TCP costs 28–43% of throughput under BBR. Third, the protocols differ most in security posture: WireGuard and tls-crypt-protected OpenVPN are silent to unauthenticated probes, whereas SSH advertises its version but limits access to a single port. Beyond the measurements, the study contributes a reusable namespace testbed with a netem-free impairment emulator, a metric set that separates throughput, CPU, overhead, loss retention and setup cost, and a deployment guide grounded in data rather than familiarity. Future work will repeat the study on two physical hosts with kernel WireGuard and OpenVPN DCO, add bursty loss and handshake-flood tests, and measure energy on ARM edge gateways.

REFERENCES

- [1] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. 2009. Plaintext Recovery Attacks against SSH. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE, 16–26. <https://doi.org/10.1109/SP.2009.5>
- [2] Fabian Bäume, Marcus Brinkmann, and Jörg Schwenk. 2024. Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*. USENIX Association, Philadelphia, PA, USA. <https://www.usenix.org/conference/usenixsecurity24/presentation/b%C3%A4umer>
- [3] Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 456–467. <https://doi.org/10.1145/2976749.2978423>
- [4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2016), 20–53. <https://doi.org/10.1145/3012426.3022184>
- [5] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '17)*. Internet Society. <https://doi.org/10.14722/ndss.2017.23160>
- [6] Benjamin Dowling and Kenneth G. Paterson. 2018. A Cryptographic Analysis of the WireGuard Protocol. In *Applied Cryptography and Network Security (ACNS 2018)* (LNCS, Vol. 10892). Springer, 3–21. https://doi.org/10.1007/978-3-319-93387-0_1
- [7] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74. <https://doi.org/10.1145/1400097.1400105>
- [8] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. 2005. Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency. In *Proceedings of SPIE 6011, Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*. SPIE, 60110H. <https://doi.org/10.1117/12.630496>
- [9] Yoohwan Kim, Russell Harkanson, Ju-Yeon Jo, and Khanh Pham. 2019. Effects of TCP Transfer Buffers and Congestion Avoidance Algorithms on the End-to-End Throughput of TCP-over-TCP Tunnels. In *16th International Conference on Information Technology–New Generations (ITNG 2019)* (AISC, Vol. 800). Springer, 401–408. https://doi.org/10.1007/978-3-030-14070-0_55
- [10] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. 2019. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 231–246. <https://doi.org/10.1109/EuroSP.2019.00026>
- [11] Steven Mackey, Ivan Mihov, Alex Nosenko, Francisco Vega, and Yuan Cheng. 2020. A Performance Comparison of WireGuard and OpenVPN. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20)*. ACM, 162–164. <https://doi.org/10.1145/3374664.3379532>
- [12] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review* 27, 3 (1997), 67–82. <https://doi.org/10.1145/263932.264023>
- [13] Yoav Nir and Adam Langley. 2018. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. IETF. <https://doi.org/10.17487/RFC8439>
- [14] Trevor Perrin. 2018. *The Noise Protocol Framework*, Revision 34. <https://noiseprotocol.org/noise.html>
- [15] Maximilian Pudelko, Paul Emmerich, Sebastian Gallenmüller, and Georg Carle. 2020. Performance Analysis of VPN Gateways. In *2020 IFIP Networking Conference (Networking)*. IFIP, Paris, France. ISBN 978-3-903176-28-7
- [16] Antonio Quartulli. 2022. Pushing OpenVPN down the Stack: Data Channel Offload (DCO). In *Netdev 0x16, The Technical Conference on Linux Networking*. Lisbon, Portugal. <https://netdevconf.info/0x16/sessions/talk/pushing-openvpn-down-the-stack-data-channel-offload-dco.html>
- [17] Chris Rapier and Benjamin Bennett. 2008. High Speed Bulk Data Transfer Using the SSH Protocol. In *Proceedings of the 15th ACM Mardi Gras Conference (MG '08)*. ACM, Article 11. <https://doi.org/10.1145/1341811.1341824>
- [18] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF. <https://doi.org/10.17487/RFC8446>
- [19] Olaf Titz. 2001. Why TCP Over TCP Is A Bad Idea. Retrieved October 26, 2026 from <http://sites.inka.de/bigred/devel/tcp-tcp.html>
- [20] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. 2022. OpenVPN is Open to VPN Fingerprinting. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association, Boston, MA, USA. <https://www.usenix.org/conference/usenixsecurity22/presentation/xue-diwen>
- [21] Tatu Ylonen and Chris Lonvick. 2006. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. IETF. <https://doi.org/10.17487/RFC4253>
- [22] Tatu Ylonen and Chris Lonvick. 2006. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. IETF. <https://doi.org/10.17487/RFC4254>
- [23] ESnet. 2026. iperf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool (version 3.22). Retrieved October 26, 2026 from <https://github.com/esnet/iperf>