ISSN 2797-7889

A WireGuard Mesh Overlay Resisting Internet Provider Port Policing

Aviv Maghridlo[1*], Nardi[2], Marzuki Sinambela[3]
[1,2,3]Undergraduate Program in Applied Instrumentation Meteorology Climatology Geophysics, STMKG, Tangerang, Indonesia
Email: aviv1.qwerty@gmail.com, nardi@stmkg.ac.id, sinambela.m@gmail.com

Overlay networks built on WireGuard can make dispersed devices appear to share one local network, but two obstacles remain: endpoints behind address translation are not directly reachable, and WireGuard's fixed default port is an obvious target for provider traffic discrimination. This study aims to design and empirically evaluate a hybrid overlay that mitigates both. We built WireGuard Manager, a single-binary Windows application combining a hub-and-spoke baseline with an opportunistic direct mesh over an in-process WireGuard data plane and host-orchestrated hole punching, and evaluated it on three physical nodes across two cities and three providers using throughput, latency, and packet-loss measurements. Results show that carrying the tunnel over the default port collapsed throughput to roughly 1–4% of a control port over the identical link (from 7.6–22.1 Mbps to at most 0.33 Mbps), while a randomized stealth port restored it; direct and relayed paths were validated independently through the observed time-to-live, and a live trace captured the automatic failover between them. A secondary result is that a direct path is not always superior to a well-provisioned relay. We conclude that provider port discrimination is a decisive, reproducible factor for such overlays and that a stealth-port strategy is an effective, low-cost mitigation, within the limits of a three-node case study.
Keywords: WireGuard, overlay network, network address translation, hole punching, port policing, peer-to-peer, mesh network.

Corresponding Author:
Aviv Maghridlo
STMKG
Tangerang, Indonesia
aviv1.qwerty@gmail.com

1. Introduction

Modern computing increasingly requires devices in different locations to interact as though they shared a single local network. Remote collaboration, distributed services, and legacy applications that assume same-subnet peers all depend on this illusion of locality, which overlay networks and VPNs provide by building a virtual subnet over the public Internet. Among available protocols, WireGuard has become a favored foundation for its modern cryptography, small codebase, and near-native performance, and now underpins a family of full-mesh tools linking personal and organizational devices across the Internet.

Building such an overlay in practice runs into two obstacles WireGuard does not resolve, which jointly determine whether it is usable. First, most home and mobile endpoints sit behind Network Address Translation and are not directly reachable, so traffic must detour through a relay unless NAT traversal opens a direct path. Second, and less studied, Internet Service Providers can discriminate against traffic by port or protocol, and WireGuard's fixed default port is an easily matched throttling target. When either strikes, an otherwise correct tunnel becomes slow or unusable—especially for latency-sensitive uses such as real-time collaboration and games—making the mitigation of these effects under real provider conditions an urgent, practical problem.

This research designs and empirically evaluates a hybrid overlay addressing both obstacles on commodity Windows hosts. The system combines an always-available hub-and-spoke baseline with an opportunistic direct mesh, promotes and demotes direct paths automatically, and relocates its transport off the default

port to resist port-based discrimination. The evaluation is scoped to a small set of real, geographically separated nodes on different providers and to two questions: how strongly providers police the default port and whether a stealth port mitigates it, and how direct and relayed paths compare in throughput and latency.

Accordingly, this study has three objectives: (i) to design and implement WireGuard Manager, a single-binary Windows overlay embedding WireGuard in-process, with receive-liveness-based mesh promotion, host-orchestrated hole punching, and live transport management; (ii) to characterize, through controlled per-port measurements on real providers, the extent of default-port policing and the effectiveness of a stealth-port mitigation; and (iii) to evaluate the direct and relayed paths—validated through the observed time-to-live—in throughput and latency, including whether a direct path is always preferable. The remainder reviews related work, describes the system and method, presents results, and concludes.

2. Literature Review and Problem Statement

WireGuard implements a secure Layer-3 tunnel using fixed modern primitives—Noise, Curve25519, and ChaCha20-Poly1305—in a deliberately small codebase [1]. Comparative evaluations report higher throughput and lower latency than OpenVPN and IPsec at low CPU cost, with tunneled throughput near an unencrypted link, though the margin is context-dependent across environments and IP versions [2], [3], [4]. Its minimal design underpins full-mesh overlays such as ZeroTier, Tailscale, and Netbird that let end-user devices behave as one virtual network [2], [5], which are mature but reveal little about how their NAT-traversal and relay decisions are made.

Direct connectivity is obstructed by NAT; the standard remedy is UDP hole punching [6], coordinated by frameworks standardized as ICE [7] with STUN [8] and TURN [9], in which a rendezvous host observes each peer's public endpoint so peers open reciprocal mappings. This is probabilistic—a large-scale campaign reports about 70% success, with a minority of peers seeing worse paths after a direct upgrade [10]—and symmetric NAT defeats it outright. Providers also discriminate against traffic by port and protocol, including throttling [11], [12], [13]; where content rather than port is inspected, VPN protocols can be fingerprinted [14] and obfuscation becomes necessary [15]. Because such degradation most harms latency-sensitive traffic, whose quality falls as delay rises [16], [17], resisting it is central to a usable overlay.

Two gaps motivate this work. First, although mesh overlays and hole punching are individually well studied, there is little openly described empirical evidence of how a WireGuard overlay behaves under adverse, real provider conditions—particularly port-based policing of the default port—on commodity Windows hosts. Second, the assumption that a promoted direct path is always superior to a relayed one is untested for this class of system. The problem is therefore to design a hybrid overlay that resists port-based discrimination and traverses NAT transparently, and to measure both the magnitude of default-port policing and the true throughput-and-latency ordering of direct versus relayed paths.

3. Method

This section presents the proposed system (subsections A–D) and the evaluation testbed, metrics, and procedure (E–F).

Proposed System: Architecture and Data Planes

WireGuard Manager is a single-binary Windows application acting as both host and client. Its interface uses Wails v2 (a Go backend with a WebView2/React frontend), while the data plane embeds WireGuard in-process—linking the wireguard-go library and creating the interface through wintun.dll rather than an

 ISSN 2797-7889

external wireguard.exe—and runs elevated because adapter creation needs driver privileges. Keeping the tunnel, control channel, and path-management logic in one address space simplifies coordinating the additive features below.

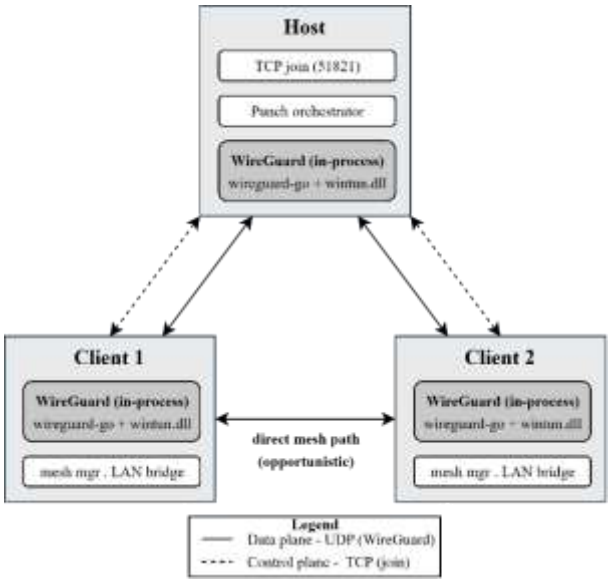

Fig. 1. System architecture. The host runs a TCP join/subscribe server and a hole-punch orchestrator alongside its in-process WireGuard data plane; each client embeds the same stack plus mesh manager and LAN bridge. Control (TCP) and data (UDP) planes are separate, so control messages survive UDP disruption.

The architecture separates two planes important to the evaluation. The control plane is a TCP channel for enrollment, peer notifications, and hole-punch coordination; the data plane is the UDP WireGuard tunnel carrying user traffic. Their independence lets transport-level changes—such as switching the UDP port—be signaled reliably over the still-connected control plane.

Hybrid Topology: Hub-and-Spoke Baseline and Opportunistic Mesh

The overlay uses a hub-and-spoke path as its always-available baseline, with all inter-client traffic forwarded through the host and identified by TTL 127. When two clients can communicate directly, the mesh manager installs a temporary /32 route that bypasses the host, creating a peer-to-peer path identified by TTL 128. This direct route remains active only while the connection is proven live; if liveness is lost, the route is removed and traffic automatically returns to the hub-and-spoke path, ensuring that the mesh improves connectivity without disrupting the baseline.

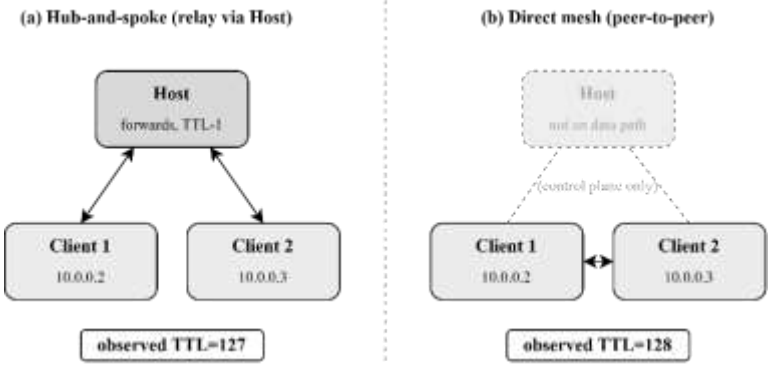

Fig. 2. Data path in the two modes. (a) Hub-and-spoke: the host forwards inter-client traffic and decrements the TTL, giving observed TTL 127. (b) Direct mesh: peers exchange traffic directly with the host off the data path, giving TTL 128.

 ISSN 2797-7889

NAT Hole Punching and Path Management

Establishing the direct paths of Section III-B requires traversing NAT. The host acts like a STUN server combined with a coordinator, without the full ICE/STUN/TURN stack: from the WireGuard control interface it observes each client's public endpoint and distributes it. To open reciprocal mappings it issues a punch_now command to both peers of a candidate pair back-to-back, so the endpoints emit packets almost simultaneously—the synchronization that makes hole punching succeed. Each client first tries LAN-local candidates before the public endpoint.

Path state is governed by receive-liveness. A direct peer is promoted—its /32 installed, path at TTL 128—only after a handshake completes and received traffic is fresh; if it stalls, the peer is demoted, the /32 withdrawn, and traffic falls back to the host at TTL 127. To recover temporarily blocked paths, the orchestrator re-issues synchronized punches about every ten seconds. Symmetric NAT, which assigns a distinct mapping per destination, cannot be traversed and stays on the hub-and-spoke path—a fundamental limitation, not a defect—with transparent fallback.

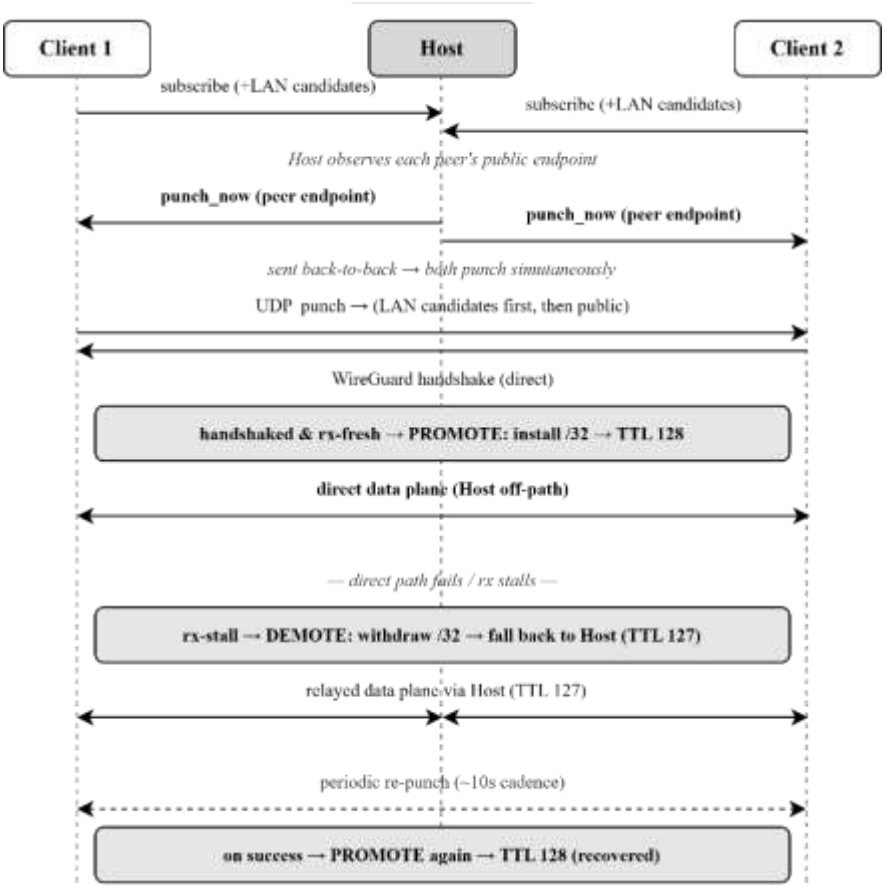

Fig. 3. Hole-punching and promotion/demotion sequence. The host observes both peers' endpoints and sends punch_now back-to-back; on a live direct handshake the path is promoted to TTL 128, and on receive-liveness stall it is demoted to the TTL-127 relay, with periodic re-punching restoring it.

Anti-Policing Mechanisms and Application Bridging

Because the default WireGuard UDP port is a static signature that operators can target, the host uses a randomized stealth port instead of port 51820 and can change it dynamically at runtime, distributing the new endpoint to clients through the control plane so they reconnect automatically. The resulting bootstrapping issue, in which clients cannot know the randomized UDP port in advance, is resolved through an auto-join mechanism: because the observed discrimination targets UDP ports while TCP remains

unaffected, the host operates its join and subscribe service on the fixed TCP port 51821, allowing clients to learn the current UDP port from the join response without additional configuration. The system also provides an optional VPN-wide XOR obfuscation layer for content-based discrimination, although it was disabled during the experiments to isolate the port variable and is reserved for future evaluation. An MTU of 1400 bytes was selected because the standard WireGuard MTU caused packet black-holing on some paths, whereas 1400 bytes remained stable, and all experimental parameters were kept fixed to prevent configuration drift. Finally, because WireGuard does not transport broadcast or multicast traffic, a bridging component captures local discovery packets, transmits them through the tunnel as unicast, and reinjects them at the remote endpoint, enabling applications that rely on same-subnet discovery, including Minecraft and Warcraft III, to operate across the Internet.

Table 1. Controlled Experimental Parameters

Parameter	Setting
WireGuard UDP port (baseline for throughput/RTT tests)	Stealth port 29041
WireGuard UDP port (port-policing test only)	Varied: 5201 (control) vs. 51820 (default)
Traffic obfuscation	Off (disabled to isolate the port variable)
Tunnel MTU	1400 bytes
Overlay subnet	10.0.0.0/24
Throughput tool	iperf3, TCP, 30 s per run, forward and reverse (-R)
Latency tool	ICMP ping, n = 100 packets per direction
Path label	Observed TTL (128 = direct, 127 = relayed)

Testbed and Scenarios

The system was evaluated on three physical Windows machines—one host and two clients—in two cities across three ISP configurations (Table 2). The host uses a datacenter connection with a public endpoint; the clients sit behind consumer ISPs typical of real deployments. In total the evaluation draws on repeated 30-second iperf3 runs per condition and 100-packet ping samples per direction across the three nodes.

Table 2. Test Node Hardware and Deployment

Node	Role	Device	ISP	Access Type	City	Scenario
PC A	Host	Datacenter server	PT Layanan Server Indonesia	Datacenter (public endpoint)	Jakarta Selatan	Both
PC B (A15)	Client	Asus TUF FA506NU	Lintas Jaringan Nusantara	Fixed broadband	Tangerang	Both
PC C (X450JB)	Client	Asus X450JB	Lintas Jaringan Nusantara	Fixed broadband (same LAN as PC B)	Tangerang	Same-LAN
PC C' (A456UQ)	Client	Asus A456UQ	IndiHome	Fixed broadband (fiber)	Semarang	Cross-network

Two scenarios were used, and their distinction is methodologically important. In the Same-LAN scenario, the two clients share one physical LAN—the only configuration allowing a meaningful no-VPN baseline, since the machines reach each other over identical infrastructure—so direct and relayed overlay paths can be compared against raw LAN throughput. In the Cross-network scenario, the clients are on different ISPs; here a no-VPN baseline is impossible in principle, because two hosts behind independent NATs cannot address each other without the overlay—precisely the problem the system solves. Only direct and relayed paths are compared, and the absent baseline is a property of the scenario, not an omission.

Metrics, Procedure, and Limitations

The evaluation measured throughput using 30-second bidirectional TCP iperf3 tests [18], while latency and packet loss were recorded through 100 ICMP ping packets per direction [19]. Direct and relayed paths were distinguished by the observed TTL, since routers decrement TTL at each forwarding hop [20]: TTL 128 indicated a direct path and TTL 127 indicated relay through the host. Each test followed a fixed sequence of ping, forward iperf3, and reverse iperf3, while core functions were validated using black-box pass/fail testing. In the port-policing experiment, all parameters were held constant except the UDP port, comparing control port 5201 with WireGuard's default port 51820. Because the study involved only three machines and a limited number of ISPs, the results are indicative rather than generalizable, and occasional RTT spikes may have resulted from ICMP de-prioritization by intermediate routers.

4. Results And Discussion

Functional Validation

Table 3 summarizes the black-box functional test. Every core feature behaved as specified in both scenarios: clients enrolled and connected, revocation removed access, the mesh promoted direct paths (TTL 128) and demoted them (TTL 127) on failure, stealth-port and auto-join let clients connect without knowing the UDP port, the port changed live, and sessions resumed on restart. Representative legacy applications assuming same-subnet peers also ran across the overlay, confirming the bridging layer works.

Table 3. Functional (Black-Box) Test Results

Feature	Expected result	Observed result / Status
Create host / server	Overlay interface created, host reachable	As expected (Pass)
Client join / enroll	Client provisioned, appears in overlay	As expected (Pass)
Connect	Tunnel established, overlay IP reachable	As expected (Pass)
Revoke client	Access removed, peer unreachable	As expected (Pass)
Mesh promotion (direct)	Direct path installed, TTL 128	As expected (Pass)
Fallback (symmetric NAT)	Relayed path via host, TTL 127	As expected (Pass)
Stealth-port (auto)	Non-51820 UDP port selected automatically	As expected (Pass)
Auto-join	Client joins over fixed TCP without knowing UDP port	As expected (Pass)
Live port change	Peers reconnect on new UDP port without manual steps	As expected (Pass)
Obfuscation toggle	Layer enables/disables without breaking tunnel	As expected (Pass)
Resume on startup	Sessions restored after application restart	As expected (Pass)
LAN gaming	Cross-Internet LAN session playable	As expected (Pass)

Characterization of ISP Port Policing

The central finding concerns the WireGuard UDP port. Holding every other Table 1 parameter constant and varying only the port, carrying the tunnel over the default port (51820) collapsed throughput to a small fraction of that over a control port (5201) on the same link. Table 4 reports per-client results on both ISPs, and Fig. 4 visualizes the contrast.

Table 4. iperf3 Throughput by UDP Port

Scenario / ISP	Client	UDP:5201 (control)	UDP:51820 (default)	Retained
Same-LAN / Lintas Jaringan	A15	17.5 Mbps	0.210 Mbps	≈ 1.2%

 ISSN 2797-7889

Scenario / ISP	Client	UDP:5201 (control)	UDP:51820 (default)	Retained
Same-LAN / Lintas Jaringan	X450JB	7.58 Mbps	0.334 Mbps	≈ 4.4%
Cross-network / IndiHome	A15	20.4 Mbps	0.292 Mbps	≈ 1.4%
Cross-network / IndiHome	A456UQ	22.1 Mbps	0.244 Mbps	≈ 1.1%

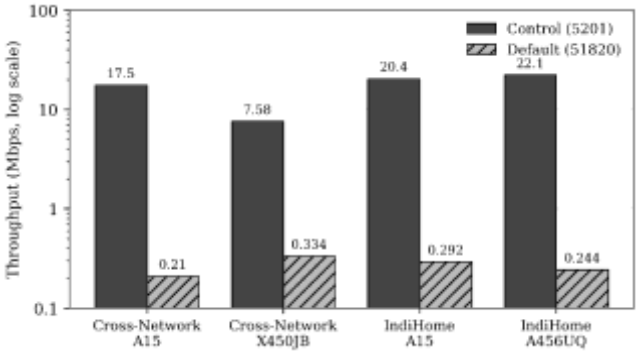

Fig. 4. Throughput on the control port (5201) versus WireGuard's default port (51820), per client on two ISPs (log scale). Traffic on port 51820 is throttled to roughly 1–4% of the throughput obtained on port 5201 over the identical link.

Two properties identify the mechanism as port-based, not content-based. First, the effect appears only on the default port: identical UDP on port 5201 flows normally, so the payload is not inspected. Second, the TCP control plane was unaffected throughout, consistent with discrimination keyed on the UDP port. This matches documented traffic-differentiation and throttling practices across many providers [11], [12], [13]; suppressing a single well-known UDP port mirrors the port- and protocol-keyed throttling those studies describe. We scope the claim to these two ISPs rather than asserting universality; within that scope it is clean and reproducible, and it validates defaulting to a stealth port—relocating off 51820 restored throughput at no user cost, since auto-join conveys the new port over the unaffected TCP channel. Against content-based fingerprinting [14], a port change alone would be insufficient, motivating the optional obfuscation layer left to future work.

Throughput: Direct vs. Relay vs. Baseline

With the tunnel on a stealth port, we compared the direct-mesh path, the relayed hub-and-spoke path, and (where possible) a no-VPN LAN baseline. Table 5 and Fig. 5 report the results; the two scenarios tell markedly different stories.

Table 5. iperf3 Throughput by Path

Scenario	Condition	Forward (Mbps)	Reverse -R (Mbps)
Same-LAN	Direct (TTL 128)	36.0	38.6
Same-LAN	Relay (TTL 127)	9.01	7.06
Same-LAN	LAN baseline (no VPN)	37.4	42.3
Cross-network	Direct (TTL 128)	1.22	0.559
Cross-network	Relay (TTL 127)	18.4	11.5

 ISSN 2797-7889

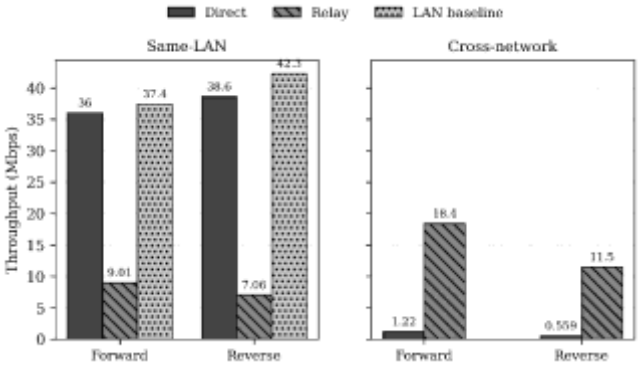

Fig. 5. Throughput by path in each scenario (forward and reverse). In the same-LAN scenario the direct path recovers nearly the full no-VPN baseline while the relay is the bottleneck; in the cross-network scenario the ordering reverses, with the relay substantially outperforming the direct path.

In the same-LAN scenario, the direct path (36.0 Mbps forward) recovers essentially the full no-VPN baseline (37.4 Mbps), while the relay collapses to 9.01 Mbps: forcing LAN peers through a remote datacenter host is a severe detour the mesh eliminates, restoring near-native throughput. Here TTL 128 is the higher-performing path.

The cross-network scenario inverts this: the direct path sustains only about 1 Mbps while the relay reaches 18.4 Mbps. We report this openly because it is informative: a direct path (TTL 128) does not guarantee higher throughput. Two residential endpoints behind independent NATs may reach each other only over a poor public path, whereas relaying through a well-provisioned datacenter host—despite the extra hop—yields far more. This is consistent with the large-scale NAT-traversal measurement, which likewise found a minority of peers with worse paths after a direct upgrade [10]. The practical implication, which we flag as future work, is that path promotion should follow measured link quality rather than the mere existence of a direct route; unconditional promotion can degrade throughput in exactly this case.

Latency, Packet Loss, and TTL Validation

Table 6 reports round-trip time and loss for the client-to-client pair in each scenario, measured with n = 100 packets per direction, together with the observed TTL.

Table 6. RTT, Packet Loss, and Observed TTL (N= 100 per direction)

Scenario	Condition	Pair	RTT min/avg/max (ms)	Loss	TTL
Same-LAN	Direct	A15 ↔ X450JB	6 / 21 / 82	0%	128
Same-LAN	Relay	A15 ↔ X450JB	11 / 24 / 58	0%	127
Cross-network	Direct	A15 ↔ A456UQ	37 / 74 / 138	0%	128
Cross-network	Relay	A15 ↔ A456UQ	15 / 26 / 124	1%	127

The TTL column is the key validation. Across all 100-packet runs the direct path showed TTL 128 and the relay TTL 127, exactly as the single host-forwarding hop predicts (Section III-F). Because TTL is a property of returned packets, not an application-set value, it is data-path evidence independent of the logs. The latency also reinforces Section IV-C: in the cross-network case the direct path shows a higher average RTT (74 ms) than the relay (26 ms)—the latency counterpart of its lower throughput—confirming the direct route is genuinely worse, not merely bandwidth-limited.

NAT Connectivity and Live Failover

Fig. 6 shows one continuous ping trace as a direct session is deliberately disrupted and recovers, making receive-liveness (Section III-C) directly observable: the path begins direct (TTL 128, low RTT), a burst of

 ISSN 2797-7889

timeouts marks the loss, the system demotes to the relay (TTL 127, higher and more variable RTT), and re-punching restores the direct path (TTL 128) near the end—entirely automatically.

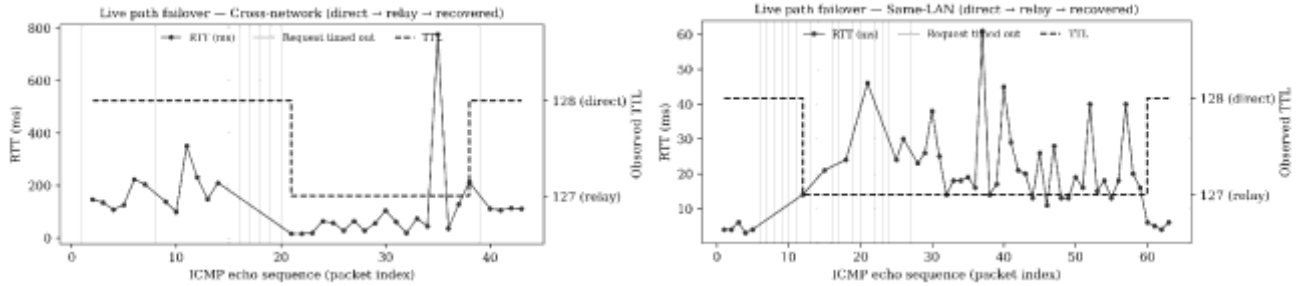

Fig. 6. Live path failover in one ping trace (same-LAN, 63 packets). RTT (left axis) and observed TTL (right axis) show the automatic sequence direct (128) → timeouts → relay (127) → recovered direct (128); the cross-network trace (43 packets) shows the same pattern.

Table 7 summarizes connectivity. Endpoints capable of direct traversal settled on the direct path (TTL 128), while one whose NAT and ISP conditions blocked hole punching fell back transparently to hub-and-spoke (TTL 127). With three nodes this is a case study, but the outcome is consistent with the roughly 70% hole-punching success reported at scale, the rest relayed [10].

Table 7. NAT Connectivity Summary (N= 3, case study)

Pair	Access / NAT condition	Path achieved	Observed TTL
A15 ↔ X450JB (same LAN)	Shared LAN, direct traversable	Direct mesh	128
A15 ↔ A456UQ (cross-network)	Independent ISPs, traversable	Direct mesh (promoted)	128
Client ↔ Host	Always available baseline	Hub-and-spoke	127 relayed / 128 to host
Non-traversable (symmetric/CGNAT)	Per-destination NAT mapping	Hub-and-spoke fallback	127

Comparison with Prior Work and Preliminary Conclusions

Taken together, the findings align with and extend prior work. WireGuard's context-dependent throughput echoes evaluations reporting that its advantage varies with environment and configuration [2], [3], [4]. The port-policing result is a concrete instantiation, on named Indonesian ISPs, of the port- and protocol-based differentiation documented at scale [11], [12], [13], and complements content-based fingerprinting [14] by isolating a purely port-keyed mechanism. The finding that a direct path can be slower than a relay is consistent with the NAT-traversal campaign that reported worse post-upgrade paths for some peers [10]. The emphasis on latency is justified by evidence that added delay degrades interactive and gaming experience [16], [17].

Several preliminary conclusions follow. First, provider port policing is not marginal but a decisive, reproducible determinant of overlay usability, capable of reducing throughput by about two orders of magnitude on the default port. Second, a randomized stealth port, bootstrapped over an unaffected TCP control channel, is an effective, low-cost mitigation. Third, the existence of a direct path is not equivalent to its optimality, so path selection should weigh measured link quality. Within a three-node case study, the hybrid overlay thus meets the three objectives of Section I: implemented as specified, its port-policing mitigation demonstrated, and the direct-versus-relay comparison quantified and TTL-validated.

Threats to Validity

Several limitations bound these results. First, three nodes across few ISPs make the figures indicative, not generalizable. Second, TTL-based labeling, though independent of the logs, assumes standard per-hop decrement and cannot distinguish paths of equal hop count. Third, the direct-slower-than-relay result was seen on one cross-network pairing; its prevalence needs a broader sample. Finally, the port-policing characterization covers two ISPs and does not establish prevalence, and the obfuscation layer—relevant to fingerprinting rather than port-based discrimination—was disabled and remains to be evaluated.

5. Conclusion

This study designed a WireGuard-based hybrid overlay that resists provider port discrimination and traverses NAT transparently, and measured its behavior on real providers. All three objectives of Section I were met. First, WireGuard Manager was implemented as a single-binary Windows overlay embedding WireGuard in-process, with receive-liveness-based mesh promotion, host-orchestrated hole punching, and live transport management, and passed every functional test in both scenarios. Second, controlled per-port measurements characterized default-port policing: on two ISPs, port 51820 collapsed throughput to roughly 1–4% of a control port over the identical link, while a randomized stealth port—bootstrapped over an unaffected TCP channel—restored it, concretely instantiating the differentiation documented at scale [11], [12], [13]. Third, direct and relayed paths were evaluated and TTL-validated (128 versus 127), showing a direct path is not always superior: in the cross-network case it was substantially slower than a well-provisioned relay in both throughput and latency, consistent with prior NAT-traversal measurement [10].

The principal implication is that provider port policing is a decisive, reproducible factor for WireGuard overlays, that a stealth-port strategy is an effective, low-cost mitigation, and that path promotion should follow measured link quality rather than the mere existence of a direct route. These conclusions hold within a three-node case study on a small provider set. Future work will extend the port-policing study to more diverse ISPs, evaluate the obfuscation layer against fingerprinting-based discrimination, broaden the NAT-connectivity matrix with repeated trials, quantify in-game quality of service on direct versus relayed paths, and compare against tools such as ZeroTier, Tailscale, and Radmin VPN.

6. References

[1] J. A. Donenfeld, "WireGuard: Next generation kernel network tunnel," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2017, doi: 10.14722/ndss.2017.23160.

[2] V. Kjorveziroski, C. Bernad, K. Gilly, and S. Filiposka, "Full-mesh VPN performance evaluation for a secure edge-cloud continuum," *Softw., Pract. Exper.*, vol. 54, no. 8, pp. 1543–1564, 2024, doi: 10.1002/spe.3329.

[3] J. Anyam, R. R. Singh, H. Larijani, and A. Philip, "Empirical performance analysis of WireGuard vs. OpenVPN in cloud and virtualised environments under simulated network conditions," *Computers*, vol. 14, no. 8, art. no. 326, 2025, doi: 10.3390/computers14080326.

[4] S. T. Oktavia, D. F. Priambodo, N. Trianto, and R. Purwoko, "Comparative quality of service analysis of VPN protocols on IPv6," *J. Nasional Pendidikan Teknik Informatika (JANAPATI)*, vol. 12, no. 3, pp. 461–471, 2024, doi: 10.23887/janapati.v12i3.69264.

[5] A. F. Gentile, D. Macrì, E. Greco, and P. Fazio, "Overlay and virtual private networks security performances analysis with open source infrastructure deployment," *Future Internet*, vol. 16, no. 8, art. no. 283, 2024, doi: 10.3390/fi16080283.

[6] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators,"

in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Anaheim, CA, USA, 2005, pp. 179–192.
[7] A. Keränen, C. Holmberg, and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A protocol for Network Address Translator (NAT) traversal," IETF, RFC 8445, 2018, doi: 10.17487/RFC8445.
[8] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session Traversal Utilities for NAT (STUN)," IETF, RFC 8489, 2020, doi: 10.17487/RFC8489.
[9] T. Reddy, A. Johnston, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay extensions to Session Traversal Utilities for NAT (STUN)," IETF, RFC 8656, 2020, doi: 10.17487/RFC8656.
[10] D. Trautwein, C. Ihle, M. Schubotz, and B. Gipp, "Challenging tribal knowledge: A large-scale measurement campaign on decentralized NAT traversal," arXiv, 2025, doi: 10.48550/arXiv.2510.27500.
[11] F. Li, A. M. Niaki, D. Choffnes, P. Gill, and A. Mislove, "A large-scale analysis of deployed traffic differentiation practices," in *Proc. ACM SIGCOMM*, Beijing, China, 2019, pp. 130–144, doi: 10.1145/3341302.3342092.
[12] A. Molavi Kakhki *et al.*, "Identifying traffic differentiation in mobile networks," in *Proc. ACM Internet Meas. Conf. (IMC)*, Tokyo, Japan, 2015, pp. 239–251, doi: 10.1145/2815675.2815691.
[13] D. Xue *et al.*, "Throttling Twitter: An emerging censorship technique in Russia," in *Proc. 21st ACM Internet Meas. Conf. (IMC)*, 2021, pp. 435–443, doi: 10.1145/3487552.3487858.
[14] D. Xue, R. Ramesh, A. Jain, M. Kallitsis, J. A. Halderman, J. R. Crandall, and R. Ensafi, "OpenVPN is open to VPN fingerprinting," in *Proc. 31st USENIX Secur. Symp.*, Boston, MA, USA, 2022, pp. 483–500.
[15] M. C. Tschantz, S. Afroz, Anonymous, and V. Paxson, "SoK: Towards grounding censorship circumvention in empiricism," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, San Jose, CA, USA, 2016, pp. 914–933, doi: 10.1109/SP.2016.59.
[16] M. Claypool and K. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, pp. 40–45, 2006, doi: 10.1145/1167838.1167860.
[17] D. Halbhuber, P. Schauhuber, V. Schwind, and N. Henze, "The effects of latency and in-game perspective on player performance and game experience," *Proc. ACM Hum.-Comput. Interact.*, vol. 7, no. CHI PLAY, art. no. 424, 2023, doi: 10.1145/3611070.
[18] ESnet, "iPerf3: A TCP, UDP, and SCTP network bandwidth measurement tool," Lawrence Berkeley National Laboratory. [Online]. Available: https://software.es.net/iperf/
[19] J. Postel, "Internet Control Message Protocol," IETF, RFC 792, 1981, doi: 10.17487/RFC0792.
[20] J. Postel, "Internet Protocol," IETF, RFC 791, 1981, doi: 10.17487/RFC0791.