Hop: A Modern Transport and Remote Access Protocol

Paul Flammarion
Stanford University

George Hosono*
Georgia Tech

Wilson Nguyen
Stanford University

Laura Bauman
Stanford University

Daniel Rebelsky
Stanford University

Gerry Wan
Stanford University

David Adrian
Independent

Zakir Durumeric
Stanford University

Abstract

Since SSH's standardization nearly 20 years ago, real-world requirements for a remote access protocol and our understanding of how to build secure cryptographic network protocols have both evolved significantly. In this work, we introduce Hop, a transport and remote access protocol designed to support today's needs. Building on modern cryptographic advances, Hop reduces SSH protocol complexity and overhead while simultaneously addressing many of SSH's shortcomings through a cryptographically-mediated delegation scheme, native host identification based on lessons from TLS and ACME, client authentication for modern enterprise environments, and support for client roaming and intermittent connectivity. We present concrete design requirements for a modern remote access protocol, describe our proposed protocol, and evaluate its performance. We hope that our work encourages discussion of what a modern remote access protocol should look like in the future.

1 Introduction

SSH (Secure Shell or Secure Socket Shell) is the de facto network protocol for server remote access and one of the most widely used cryptographic protocols after Transport Layer Security (TLS) [16,62,120]. First introduced in 1995, SSH-1 replaced rsh, rlogin, and rexec, and enabled secure remote shell access [123]. The Internet Engineering Task Force (IETF) later standardized SSH-2 in 2006, both addressing several security issues and formally documenting the protocol. Since then, SSH has been iteratively updated and is now standardized across 20 RFCs. Today, SSH usage remains widespread [44,54,76], used for both remote access and tunneling cleartext protocols like rsync [116].

Despite SSH's longevity, the protocol has several recognized shortcomings. At its cryptographic foundation, recent work has uncovered downgrade attacks [13] and surfaced privacy leaks [2, 89, 92]. A long string of work has also demonstrated shortcomings in SSH's user and host authentication mechanisms [11,39,58,117,119] resulting in the original creator of SSH, Tatu Ylönen, calling for fundamentally re-thinking SSH's host authentication [124]. Beyond the protocol itself, there have been repeated challenges with common implementations. The de facto OpenSSH implementation is heavyweight and has deviated from the SSH-2 protocol to meet user needs; the IETF revived the SSH working group in 2024 to reconcile these differences [65]. Alternatives like Dropbear have attempted to shrink the protocol's software footprint, but have been riddled with remotely exploitable vulnerabilities [4,6,41,63,68,84,111].

Several prior works have proposed iterative improvements to SSH that maintain its fundamental structure. Winstein et al. introduced Mosh [121], a remote terminal protocol that uses SSH for authentication, but transitions connections to UDP to support roaming users and intermittent connectivity. Kogan et al. proposed Guardian Agent to better protect user credentials [74]. Separately, Michel et al. proposed rebuilding SSH on QUIC and HTTP/3 [86]. None of these patches have seen widespread adoption. Yet, in other domains, protocols like Wireguard [43] have shown that our community is willing to adopt altogether new approaches.

The significant changes to SSH that would be needed to address modern security requirements suggest that it may be time to consider a clean-slate approach to server remote access. In this work, we systematize today's requirements and we introduce Hop, a simple and secure remote access protocol. Beyond introducing a lightweight cryptographic foundation, Hop addresses several long-standing problems in SSH by (1) adding secure credential delegation, (2) eliminating "trust-on-first-use" server authentication through a native Automatic Certificate Management Environment (ACME) inspired protocol [1], and (3) enabling roaming and unreliable tunneling through a UDP-based transport. We provide a reference implementation in Go[1], which we evaluate across a geographically distributed testbed. We demonstrate comparable throughput to OpenSSH, but with faster connection

*Work conducted primarily at Stanford University.
[1]Source code available: https://github.com/hop-proto/hop-go

establishment and improved terminal responsiveness.

Our work makes the following contributions:

1. We survey prior cryptanalysis of the SSH protocol, reported weaknesses, proposed improvements, and modern enterprise needs, from which we systemize the design requirements for a modern remote access protocol;
2. We introduce Hop, a three-layer network protocol ensemble that provides a lightweight cryptographic foundation, reliable and unreliable network transport, and a more secure SSH remote access alternative;
3. We evaluate Hop under real-world conditions and show that it can provide comparable throughput, lowered latency, and improved security and privacy guarantees.

We hope that our work prompts conversation on the future of server remote access protocols.

2 Protocol Requirements

Here, we overview shortcomings in SSH and modern needs, which we use to develop concrete protocol requirements.

2.1 Simple Cryptographic Protocol

In 2023, Bäumer et al. demonstrated that SSH is vulnerable to cipher attacks, showing that an attacker can downgrade the public key algorithms for user authentication and disable eavesdropping protections because SSH does not protect the full handshake transcript [13]. Historically, protocol complexity has repeatedly led to protocol logic bugs [2,3,10,20,23,57,90,104]. In contrast, WireGuard [43] stands out in the Virtual Private Network (VPN) domain as a simple versioned protocol composed of modern cryptographic primitives. Despite WireGuard's simple design, the protocol improved VPN performance [82] over historic options like OpenVPN and IPsec [71].

Requirement 1. Hop's handshake must provide downgrade resistance, handshake integrity, session uniqueness, and replay resistance. The handshake should use a fixed, versioned set of cryptographic primitives with no in-band negotiation, such that any unsupported version or parameter results in explicit handshake failure. These properties must hold against an active network adversary capable of observing, modifying, replaying, and reordering packets.

2.2 Trustworthy Host Identification

Today, SSH users primarily rely on "trust-on-first-use" ("tofu") to authenticate servers [119]. Unfortunately, Gutmann showed that most users will blindly accept SSH server keys [58]. Several works have explored whether alternative key representations help users more effectively verify identity keys [11, 39, 117]. Wendlandt et al. proposed Perspectives, a system in which multiple vantage points are used to verify server host keys [119]. However, none of these methods have been adopted by the core protocol.

In 2019, the creator of SSH, Tatu Ylönen, agreed with Gutmann, noting that "users do not understand the warnings about changed host keys and even for experts, verifying the keys is too cumbersome to do reliably." Critically, he argued that "the host authentication in SSH is not reliably serving its function of preventing man-in-the-middle attacks. Thus, the mechanism needs to be augmented or replaced by a mechanism that provides better security and smoother operations in large environments." and called for a new approach [124].

Requirement 2. Hop must enable clients to reliably verify servers' identities to prevent man-in-the-middle attacks during any connection, including the first. Server authentication must fail if identity verification cannot be completed successfully. This requirement must hold against an active network adversary capable of intercepting, modifying, or injecting messages, even if the attacker compromises the handshake initiator's long-term authentication key (KCI).

2.3 Extensible Client Verification

SSH is primarily used in situations where a relationship and out-of-band communication channel exist between client and server operators. Despite this, there is no in-protocol mechanism built-in for providing or verifying key material distributed out-of-band, short of distributing `known_hosts` and `authorized_keys` files, and non-standard authentication remains a challenge for many enterprises [106]. Recently, in 2025, Kayali et al. introduced SSH-Passkeys as a passwordless approach to client authentication [70]. Similarly, Cloudflare has implemented OAuth-based SSH authentication using OPKSSH [61]. Other authentication mechanisms include external, hardware-backed FIDO2 authentication [126].

Requirement 3. Hop must natively support client authentication mechanisms that leverage existing out-of-band trust relationships. Authentication must fail under long-term identity compromise or unauthorized reuse of delegated authority. These guarantees must hold even when authentication is performed via intermediate hosts or delegated actions.

2.4 Privacy and Confidentiality

The SSH protocol acts as an oracle that allows querying whether a given public key is accepted by a server, without first proving ownership of the private key [89,92,105]. While OpenSSH does not recognize this as a vulnerability [88], operators typically do not expect such information to be exposed, and it can unintentionally reveal the server ownership unless all users have single-use keys. Passively observable SNI values have also been used for censorship [127].

Requirement 4. Hop servers must not act as oracles for active adversaries and must hide private information, including SNI, from passive network observers. Additionally, Hop's

handshake must ensure client identity hiding with forward secrecy against unknown or unauthenticated servers.

2.5 Secure Credential Delegation

Key forwarding attacks are a known shortcoming of SSH, which "allows remote machines (delegates) to authenticate as the user, without knowing which remote machine is asking, what command it will run, and what server it will run it on." [74]. To support delegation, SSH uses `ssh-agent` to "forward" keys from the client to a delegate server, allowing the delegate to access the target server using the user's identity. This forwarding model enables any process with root access on the delegate server to authenticate as the user without user confirmation [74]. This weakness has enabled lateral movement in repeated attacks where adversaries pivot across systems using stolen or forwarded credentials [95]. Despite bundling `ssh-agent`, many users fail to configure it correctly, and instead copy their private key material to remote hosts, which is an even larger security weakness.

Requirement 5. Hop must support secure client credential delegation without exposing long-lived credentials. Delegated credentials must be bound to a specific user, action, target, and validity period, and must not be reusable beyond their intended scope. These guarantees must hold even if delegate hosts are compromised.

2.6 Secure Transport for Unreliable Traffic

QUIC [67] popularized the use of UDP for HTTP, eliminating the need for a three-way handshake to establish a transport connection. UDP further supports roaming, intermittent connectivity, fast session resumption, and efficient transmission of small amounts of data at a time. This allows securely tunneling UDP-based protocols, avoids potential slowdowns associated with tunneling TCP over TCP [34], and enables native support for UDP-based applications. A prominent example is Mosh: a UDP-based alternative to SSH that "supports intermittent connectivity, allows roaming, and speculatively and safely echoes user keystrokes for better interactive response over high-latency paths." [121]. Mosh relies on SSH for connection setup and authentication before transitioning to a secondary UDP channel for remote access.

Requirement 6. Hop must support secure communication over unreliable networks while preventing an unauthenticated adversary from causing the server to emit responses larger than the triggering request. Intermittent connectivity and roaming must preserve authentication and must not enable session hijacking or spoofing.

2.7 Constrained Environment Support

A large body of work has shown that (D)TLS and QUIC remain unnecessarily heavyweight for embedded devices where SSH is often deployed [8, 76], despite numerous proposed modifications [12, 31, 60, 94, 99, 115, 118]. These characteristics make them poor alternatives to a dedicated lightweight protocol, regardless of their recent updates, formal proofs, and suggestions as potential alternatives to SSH [21, 22, 35, 36, 40, 85, 86]. Beyond the protocols themselves, server authentication in both (D)TLS and QUIC is typically achieved through a supporting PKI in which Certificate Authorities (CAs) issue ASN.1-based X.509 certificates; the flexibility and inherent complexity of ASN.1 and X.509 continue to pose challenges for the WebPKI ecosystem [29, 32, 46, 72, 77, 108, 114].

Several papers have proposed modifications to QUIC and (D)TLS to better suit embedded devices. Raza discusses how (D)TLS needs to be altered to fit within 6LoWPAN packets (e.g., when transporting CoAP requests) [103]. Gallenmüller identified that cryptographic operations are the main cost factor of (D)TLS and suggested using more efficient ciphers like ChaCha20 [53]. Others propose altering (D)TLS to support more efficient handshakes without X.509 and public-key exchanges [28, 60, 80, 102, 107]. Others propose modifications to X.509 certificates for IoT environments [52, 83, 110]. Tange introduces ratchetTLS, a symmetric-ratchet-based extension to TLS 1.3 that allows IoT devices to more efficiently use 0-RTT TLS sessions [113].

Requirement 7. Hop must operate in computationally constrained environments without requiring complex parsing or validation logic. Protocol credentials must follow a fixed, deterministic validation path with a bounded size. These properties must hold under adversaries attempting to exploit implementation complexity.

2.8 Post-Quantum Security

Current secure-channel protocols are in the process of transitioning to post-quantum key establishment [5, 7, 9, 18, 64, 101]. NIST has standardized ML-KEM as a replacement for classical Diffie-Hellman key exchange [97], and has explicitly stated that Internet protocols may retain classical authentication (peer identity establishment) mechanisms during the transition to post-quantum cryptographic standards [47].

Requirement 8. Hop must ensure post-quantum forward secrecy against adversaries capable of recording encrypted traffic. Compromise of long-term authentication keys must not retroactively compromise confidentiality.

3 Threat Model

We describe the threat model that Hop protects against:

Network Adversary. An adversary with full control over the network may observe all traffic [112], interfere with packet delivery, and actively manipulate message contents or timing [42]. Such an adversary can attempt scanning or

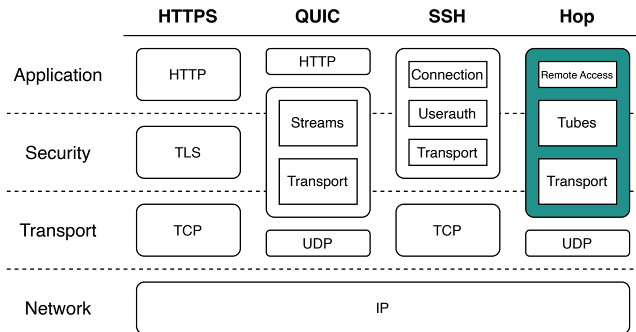


Figure 1: **Layered Architecture**—Hop is composed of three sub-protocols: Hop Transport, Hop Tubes, and Hop Remote Access. Similar to QUIC, Hop is responsible for transport-layer functionality through application-layer functionality.

probing to discover running services [44, 45, 54, 92], launch replay or downgrade attacks [13], or try to shape the handshake transcript in ways that induce inconsistent or malicious states [13]. We also assume the attacker may abuse UDP-based network protocols [75].

Endpoint Adversary. An attacker may compromise semi-trusted delegate hosts [74], steal long-term private keys, obtain previously issued intent-grant credentials, or exploit implementation vulnerabilities such as unchecked parsing logic or memory-safety errors [14, 55]. We also assume the adversary may record encrypted traffic today and attempt decryption in the future using large-scale quantum resources [18], known as ("harvest now, decrypt later").

4 Hop Protocol Overview

In this section, we introduce Hop, a new protocol that provides secure, modern, and privacy-preserving remote access and meets the requirements set forth in the previous section.

Hop is composed of three sub-protocols: (1) a lightweight cryptographic transport layer (*Hop Transport*), (2) a reliable and best-effort connection multiplexing layer (*Hop Tubes*), and (3) an application-layer protocol (*Hop Remote Access*) that provides a remote shell (Figure 1). While the top-most application layer depends on functionality provided by the lower layers, the lower Hop Transport and Hop Tubes layers can be used in other contexts (e.g., IoT–cloud communication). In this section, we briefly describe each layer and the motivation behind the proposed division of responsibilities.

Hop Transport. Hop Transport is a lightweight transport-layer cryptographic foundation that foregoes per-handshake cryptographic agility and instead uses a single duplex construction [37] for all symmetric cryptographic operations. Hop Transport operates over UDP, delegating reliability and ordering responsibilities to the upper Tubes layer. This allows efficiently tunneling UDP protocols (e.g., Mosh [121]), supporting both reliable and best-effort streams over a single tunnel, providing seamless client roaming, and supporting other use cases that do not require reliability.

Hop Tubes. Hop Tubes extend the cryptographic transport layer to provide multiplexed reliable and best-effort communication channels, which we term "tubes." (We choose not to call these "channels" due to a naming collision with Go Channels.) Hop supports multiple simultaneous tubes over a single transport connection, each of which can have their own reliability and congestion control configuration. For example, the Hop Remote Access application-layer protocol uses a reliable channel for control messages, but a best-effort channel for UDP port forwarding.

Hop Remote Access. Building upon the lower transport and connection layers, we introduce a new secure remote access protocol that improves SSH by providing secure identity forwarding, IP roaming, improved client authentication, and server identity (e.g., automatic server certificate issuance and short-lived client certificates).

We implement Hop in approximately 18K Lines of Code (LoC) in Go (Transport: 6k LoC, Tubes: 2k LoC, Remote Access: 5k LoC, Tests: 5k LoC) as reported by `cloc` [38]. In the following three sections, we detail each layer, their security guarantees, and their interactions.

5 Hop Transport Protocol

Hop Transport provides a lightweight cryptographically protected channel between two mutually authenticated endpoints. Contrasting complex protocols like TLS, SSH, and QUIC, which allow for cryptographic agility, Hop Transport defines a static set of lightweight cryptographic primitives for authentication and key exchange derived from the Noise Protocol Framework [100]. Noise provides several formally verified handshake patterns [24, 73, 81] that minimize implementation and reduce network round-trips (Req. 1). Since remote access users typically have a preexisting relationship or established communication channel with server operators, Hop uses protocol versioning to adapt to cryptographic developments rather than in-handshake negotiation.

In addition, we introduce a lightweight certificate format and an ACME-like [1] extension that enables automated certificate issuance (Req. 2). This allows consistent but flexible client and server identity verification without incurring the complexity and overhead associated with ASN.1 and X.509.

5.1 Connection Establishment

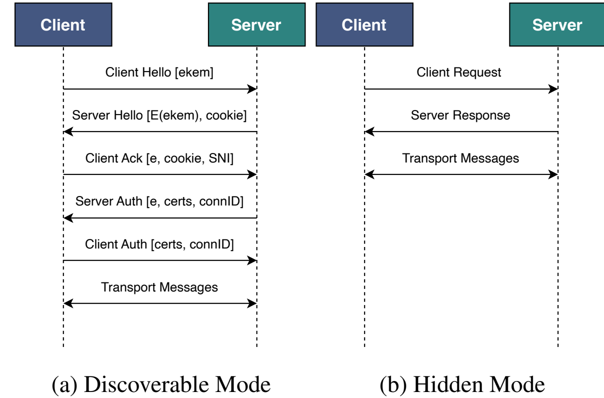
Hop Transport supports two modes of operation: *discoverable mode*, which uses the Noise XX handshake, and *hidden mode*, which uses the Noise IK handshake [100]. While the standard Noise XX handshake pattern requires 1.5 round-trips for key exchange and authentication, we extend the

handshake by one round-trip to prevent denial-of-service attacks and to enable post-quantum forward secrecy. This extra round-trip is not required in Hop hidden mode as a client ensures the same security properties in a single round-trip, but must know the server static key a priori through out-of-band communication. Beyond the handshake efficiency (two messages), Hop hidden mode—built on UDP—also enables Hop servers to remain completely transparent to scanners and unauthenticated clients, in part fulfilling our privacy Req. 4. Hop servers will immediately fail the handshake if the version number does not match the one advertised in the first client message (§5.4).

We mitigate the "harvest now, decrypt later" attack posed by future quantum computers [109] by ensuring that ephemeral key exchanges use NIST standardized PQ-safe Key Encapsulation Mechanisms (KEMs) [97]. While the Noise Framework was originally designed to be instantiated with Diffie–Hellman functions [100], we specifically build on Angel et al.'s proposed PQNoise as a secure replacement of Diffie–Hellman key exchanges with KEMs [7]. However, unlike PQNoise, Hop's authentication remains based on static Diffie–Hellman keys. This allows each primitive to serve a distinct security goal, without conflating assumptions. This approach reduces the transmission of large post-quantum KEM keys and secrets, while ensuring long-term post-quantum forward secrecy (Req. 8).

While Noise is used for Hop's handshake pattern, it does not dictate the cryptographic primitives used for key derivation, hashing, encryption, and authentication. To minimize code size and complexity, Hop Transport relies on duplex objects based on a single permutation from which all of symmetric cryptography can be derived. Specifically, we use the Cyclist [37] duplex construction instantiated with Keccak (the permutation used for SHA-3 [48]). Cyclist is a modern, lightweight duplex construction that can be instantiated using any sponge permutation. Hop could similarly use ASCON in place of Keccak, now that it has won the NIST Lightweight Cryptography contest [96]. In the case of an update of the cryptographic primitives, we will deprecate the current version and move to a new major one, thus preventing cryptography downgrade attacks (Req. 1).

We extend Noise with Cyclist, similar to the Disco [122] and Strobe [59] frameworks. Since duplex objects can be used to implement pseudorandom functions (PRFs), hash functions, authenticated encryption, message authentication codes (MACs), key derivation functions, and support ratchet operations, Hop Transport replaces all symmetric cryptographic algorithms and objects with a single duplex object that is continuously updated. In each handshake message, the transport parameters, KEM keys and their shared secret keys, and Diffie-Hellman results are sequentially "absorbed" into the running duplex state, and MACs are "squeezed" out and appended to each sent message to provide message integrity and transcript consistency (Req. 1). After the handshake is completed, Hop Transport uses a standard AEAD record construction using the same primitives.



(a) Discoverable Mode (b) Hidden Mode

Figure 2: **Connection Establishment**—Hop supports two modified handshakes that use Diffie–Hellman static keys for authentication and post-quantum ML-KEM for forward secrecy. Hop Discoverable Mode extends Noise XX, and Hop Hidden Mode adapts PQNoise IK.

Both handshake participants maintain their own duplex state and verify that their views of the ongoing handshake are identical by validating received MACs on each message. Duplex objects also allow for incrementality [15], enabling output MACs and derived keys to be computed without buffering handshake data or maintaining multiple hash chains, making them efficient at authenticating the full transcript of operations at each step of the protocol. As a result, implementations of Hop Transport benefit from a significant reduction in code size and working memory, which facilitates security audits, a factor important for many embedded devices with limited resources (Req. 7).

5.2 Discoverable Mode Handshake

The Hop Discoverable Mode handshake extends the Noise XX pattern (Figure 2a). In a standard XX construction, the client initiates the connection by sending a clear-text ephemeral public key, to which the server replies with its own clear-text ephemeral public key, followed by its static public key encrypted using the Diffie-Hellman result between both ephemerals (*ee*). In the last handshake message, the client sends its static key encrypted using key material influenced by the Diffie-Hellman result between the client ephemeral and server static (*es*) and all prior transport parameters. Finally, both parties perform a Diffie-Hellman key exchange between the client static key and the server ephemeral key (*se*) to complete the handshake.

To provide post-quantum forward secrecy and prevent Hop servers from becoming UDP DoS amplifiers, Hop introduces an additional round trip before the standard Noise XX sequence. This modification is inspired by PQNoise [7]

and integrates ML-KEM for the initial key exchange.

$$\rightarrow\ ekem$$
$$\leftarrow\ Encaps(ekem), cookie$$
$$\rightarrow e,\ ekem, cookie$$
$$\leftarrow e, ee, s, es$$
$$\rightarrow s, se$$

Fields highlighted in blue in the Hop handshake are additions beyond the original Noise XX pattern. *ekem* denotes the client ephemeral ML-KEM encapsulation key. *Encaps()* is the ML-KEM ciphertext encapsulating the post-quantum shared secret using that key. *e* and *s* represent the Diffie–Hellman ephemeral and static public keys, respectively, and *ee*, *es*, and *se* denote the corresponding derived Diffie–Hellman shared secrets.

A client initiates a handshake by sending a *Client Hello* message to the server, which communicates the client's ML-KEM ephemeral encapsulation key for the session. As with all handshake messages, the client's duplex object computes a MAC and attaches it to the client hello. The client hello elicits a *Server Hello* with a ML-KEM encapsulated shared secret and a handshake cookie: an authenticated-encryption block that encrypts the server ephemeral shared secret key with associated data consisting of the client ML-KEM ephemeral encryption key, IP address, and port.

The client sends this cookie back to the server in the next message, demonstrating that it is maintaining handshake state prior to being authenticated. The cookie decryption key can be rotated by the server on a regular basis (e.g., every two minutes), and the corresponding encryption key never needs to be revealed beyond the server process, nor is it directly used as part of the handshake. This symmetric construction does not compromise the long-term post-quantum security of the handshake and enables a stateless handshake from the server's perspective before client authentication. The size of the server hello message is equivalent to the client hello to account for Req. 6 and DDoS amplification.

After the client receives the cookie, the handshake information is encrypted with the post-quantum shared secret from the ephemeral KEM keys. Hop's handshake then returns to the traditional Noise structure using Diffie–Hellman keys. The client replies with a *Client Acknowledgement* to send its Diffie–Hellman ephemeral public key, echo the cookie back to the server, and indicate the hostname or Server Name Identification (SNI) it is trying to connect to. This design follows our specification of Req. 4, hiding the SNI to eavesdroppers. Since the duplex state is fully encoded in the cookie, the server can use it to reconstruct the full handshake transcript and decrypt the hostname.

The server replies with a *Server Authentication* message that contains (1) a unique connection ID, (2) an encrypted server certificate chain, and (3) a certificate authentication tag. The connection ID identifies the Hop Transport connection for post-handshake transport messages independent of the IP/port 4-tuple, allowing connections to roam across IP addresses (Req. 6) in a manner similar to QUIC [78]. Certificate chains in Hop Transport are always linear and are at most length three (leaf, intermediate, root). We further detail Hop Transport identity and authentication in Section 5.6. The certificate authentication tag is derived from the duplex state and allows the remote party to validate the authenticity of the certificate prior to parsing it and performing key exchange. When the client receives the server authentication, it validates the server and sends a symmetric *Client Authentication* message, which completes the handshake.

After verifying the final MACs, both parties are authenticated, have an identical view of the handshake transcript, and have shared secrets from ML-KEM ephemeral and derivation of Diffie–Hellman key pairs (ephemeral and static, preventing KCI Req. 2). Hop Transport's cryptography allows subsequent transport messages to benefit from sender and receiver authentication, as well as strong message forward secrecy (Req. 8). Further details can be found in Appendix A.1.

5.3 Hidden Mode Handshake

To hide from network scanners and minimize abuse, Hidden Mode uses a single round-trip handshake that requires the first handshake message to authenticate the client (Figure 2b). Hop Hidden Mode handshake's design is based on the PQNoise IK pattern [7], but uses Diffie–Hellman static keys in Hop certificates for authentication. This change in the PQNoise IK pattern enables Hop to conserve lightweight certificates and results in the following flow:

$$\leftarrow skem$$
$$\ldots$$
$$\rightarrow Encaps(skem), ekem,\ s$$
$$\leftarrow Encaps(ekem),\ s, ss$$

Here, *skem* is the static ML-KEM encapsulation key and *ss* is the Diffie–Hellman calculation of the certificates keys during the key exchange. All other notations are the same as described in the Hop Discoverable Mode. Fields highlighted in green indicate modifications to the use of Diffie–Hellman keys compared with the original PQNoise IK pattern [7].

In the Hidden Mode, the server ML-KEM static encapsulation key is distributed out-of-band prior to handshake. It can also be generated and transmitted during a previous session using a Discoverable Mode handshake. Unlike the Discoverable Mode, there is no need to modify the IK pattern to prevent DoS attacks (Req. 6), as the client must authenticate in the first message. The resulting security guarantees of the Hop Transport connection remain identical to those provided by the Hop Discoverable Mode handshake.

The client begins by sending a *Client Request* message that contains: (i) the ML-KEM client ephemeral encapsulation key, (ii) a (KEM) ciphertext of the shared secret keys encapsulated by the ML-KEM server static key (iii) an encrypted Hop Certs chain authenticating the client, (iv) an authentication tag, and (v) an encrypted timestamp. Like in Discoverable Mode, the tag allows the remote party to validate the authenticity of the certificate prior to parsing. The timestamp is necessary to prevent replay attacks (Req. 6). Note that the server does not respond to messages from clients that do not know the server key, allowing it to remain "hidden". The server responds with a *Server Response* message that contains (i) a ML-KEM ciphertext of a second shared secret encapsulated by the ML-KEM client ephemeral key, (ii) an encrypted Hop Certs chain authenticating the server, (iii) an authentication tag, and (iv) a connection ID.

After authentication, both client and server derive the final transport keys in a manner identical to the discoverable handshake. To retrieve the key pair used to encrypt the client request, a server with a multi-host configuration will loop through the configured hosts to match the static key used by the client to encrypt the client request. The number of hidden hosts should therefore be kept small to ensure performance. Additional information about the Hop Hidden Mode handshake and its messages is available in Appendix A.2.

5.4 Handshake Failure

By design, there are no branches in either handshake except for failures due to a rejection, timeout, or error. On a failed handshake or handshake packet loss, both sides will discard local state and remain silent except for a certificate validation failure. In this case, both parties still complete the handshake, but the rejecting party will immediately close the newly established connection by transmitting a *Connection Close* message (encrypted to prevent UDP spoofing, Req. 6).

5.5 Key Derivation

After a successful Hop Transport handshake, the client and server have a post-quantum safe shared secret key from the ephemeral KEM and Diffie-Hellman results from client and server static keys. They both have identical duplex objects keyed using each of these key pairs. Hop Transport then uses the duplex object to derive two encryption keys: one for client-to-server transport messages and one for server-to-client transport messages.

5.6 Identity and Authentication

Hop uses lightweight certificates to authenticate servers and clients. While the Noise framework does not specify how to verify a remote party's static public key, Hop explicitly integrates PKI-based authentication into the handshake. Hop

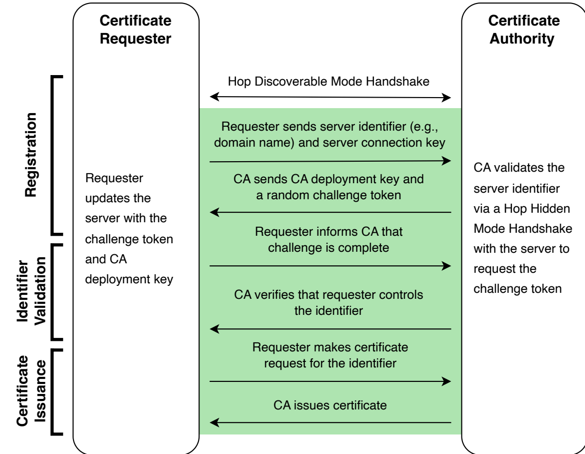


Figure 3: **Certificate Issuance Protocol for Servers—** ACME enables Hop Transport to support automatic certificate issuance. For Hop clients, Identifier Validation may take some other form, such as an OAuth authorization request.

uses certificates that (1) minimize parsing and validation complexity, (2) provide a fixed and deterministic validation path, and (3) fit within a single handshake message without fragmentation, consistent with Req. 7.

Existing certificate formats, including X.509, do not satisfy these constraints. X.509 is complex [55], has a long history of configuration errors [46], incorrectly issued certificates [77], and its processing overheads preclude constrained devices [98]. While it is possible to fit X.509 certificates into small enough packets for Hop [56], it is not possible to simplify the parsing logic to meet Req. 7. X.509 certificates can include arbitrary extensions [27], which means that any implementation must parse and interpret extensions that it has never seen before. Rather than adopt a subset of X.509, we introduce a lightweight certificate structure with the minimal fields needed to bind an identity to a public key (Appendix Table 1).

Hop Certificates only support a single cryptographic algorithm or scheme for each field, simplifying parsing and validation. To further simplify certificate validation, we enforce that each certificate has at most one parent, which ensures linear paths. We restrict the path length to be exactly three: leaf, intermediate, and root. Hop Certs are sized such that all Hop handshake authentication messages will never exceed a 1400 byte maximum transmission unit (MTU), eliminating the need to reassemble certificates across L4 packet boundaries. Hop certificates are intentionally simple and scoped for use within Hop, following recent guidance for maintaining separate PKIs for different application domains [30].

5.6.1 Automatic Server Certificate Issuance

Hop supports clients and servers acquiring certificates signed by a Certificate Authority (CA) through a protocol extension

based on the ACME protocol [1]. The ACME flow runs over Hop Transport, ensuring that certificate issuance inherits the security properties of Hop. It requires only that the certificate application obtain the CA root certificate out-of-band (e.g., during ACME client or Hop installation).

The Hop issuance protocol consists of three phases: (1) Registration, (2) Identifier Validation, and (3) Certificate Issuance (Figure 3). During Registration, the requester connects to a CA using a Discoverable Mode handshake and sends the identifier it wants a certificate for (e.g., domain name) as well as any information needed to complete Identifier Validation. For example, the requester would send the domain name and an ephemeral KEM key that the CA will use to validate that the requester controls the domain name.

Identity validation can take various forms. For a Hop server, identity validation means confirming that the requester can operate a Hop server on the requested domain name or IP address. For a client, identity validation may include performing an OAuth authorization request or other forms of verification. To validate that a Certificate Requester owns a domain name, the CA sends the Requester a 256-bit challenge string and a Hop client KEM key. The Requester then creates a Hop server on the requested domain name and creates a special authorization grant for the CA's client KEM key that allows the CA to retrieve the challenge. The CA then connects to the Requester's Hop server using a Hop hidden mode handshake and verifies that the requester can correctly echo back the challenge string. Finally, the Requester sends the CA the name and public key they want signed in their certificate. The CA ensures that the requested name matches the validated name and sends over the signed certificate. This mechanism allows server identity verification (Req. 2).

5.6.2 Client Authentication

Unlike TLS and QUIC, client authentication is required for all Hop handshakes. Enterprises increasingly rely on client-side certificates for remote-access authentication, while IoT infrastructures use them for device authentication [33]. Hop is designed to explicitly support these issuance profiles. The Hop Cert format, with its root/intermediate/leaf hierarchy and symmetric authentication messages, makes the process straightforward to implement and practical to deploy (Req. 3). Hop also supports an `authorized_keys` file for static authentication of client keys that have been distributed out-of-band. Hop supports three types of client certificates:

1. **CA-Signed Client Certificates:** Enterprises can issue short-lived, CA-signed certificates to users as part of their authentication infrastructure, for example, via Hop ACME or custom distribution mechanisms.
2. **Self-Issued Client Certificates:** Clients may generate their own certificates and distribute their public static key out-of-band, similar to SSH public keys today.
3. **Ephemeral Client Certificates:** Hop's delegation scheme (§7.3) issues short-lived certificates to securely enable delegation.

5.7 Session Establishment

Once a Hop Transport connection is established, the server authenticates the client through certificate-based verification before allowing access to network services. If the client is authorized, the Hop session is considered fully established, and the server begins processing service requests.

6 Hop Tubes Protocol

The Hop Tubes layer sits above Hop Transport, providing an interface to build secure applications that run over the encrypted and authenticated transports. Applications can use a single transport connection to multiplex several logical channels, called "tubes," which support either reliable or best effort communication directly aligning with the Req. 6. Each tube has a "tube type" (e.g., remote command execution, network proxy, port-forwarding control), which is an identifier that determines how the application layer should interpret received data. The tube abstraction is inspired by a combination of stream multiplexing in QUIC and channel multiplexing in SSH. Since latency is critical for interactive applications such as remote shells, Hop does not implement any variant of Nagle's algorithm [93].

Hop Tubes encapsulates a single *tube frame* (Appendix Figure 11) within the Encrypted Data section of the transport message (Appendix Figure 10). Each tube is identified by a unique Tube ID in the frame header. Like QUIC, Hop Tubes allocate odd-numbered Tube IDs for client-initiated tubes and even-numbered IDs for server-initiated tubes to avoid collisions during the tube creation. Unlike QUIC, tube creation is explicit, much like opening a new channel in the SSH Connection Protocol [125]. This allows each party to restrict actions from the remote party (e.g., allowing packet forwarding but not remote command execution) by denying tube creation attempts for unsupported tube types.

6.1 Tube Creation

Tube creation involves a single round-trip handshake that begins when the initiator sends a tube frame with the `REQ` bit set to one and the Frame Number set to zero (this packet is called a *Tube Initiation Request*), which eliminates the need for sequence number synchronization. The Tube Initiation Request contains a non-negotiable Tube ID and tube type chosen by the initiator. The responder simply replies with a *Tube Initiation Response* (`RESP`) indicating acceptance.

Since the entire frame (including the Frame Number) is authenticated and encrypted within a Hop Transport message, sequence number spoofing attacks [17] are not possi-

ble. Tube creation messages do not use explicit acknowledgments, as the response is implicitly an acknowledgment of the request. If the initiator does not receive a Tube Initiation Response after some period of time, it will assume that the tube was not created and will retransmit the request. The initiator ignores all application data arriving before a Tube Initiation Response and resends requests until it properly receives a corresponding response.

6.2 Tube Closing

Tubes can be closed without tearing down the entire Hop connection. To close a tube, Hop Tubes uses the four-way FIN-ACK/FIN-ACK handshake similar to TCP. If tube frames are received by an endpoint after a `FIN` has been sent, the endpoint will continue sending acknowledgments with the `FIN` flag set to signal closing the tube. Once both endpoints have received acknowledgments for each `FIN`, the tube is closed. While Tube IDs can be reused by the same Hop connection, it is the responsibility of each end of the connection to manage its Tube IDs (odd or even) and ensure that it does not reuse a Tube ID until it is confident that the Tube ID is no longer in use. To prevent multiple tubes from being created with the same Tube ID, the side that initiated the tube reserves the Tube ID for a duration of four times the measured round-trip time (RTT) after the tube has closed.

6.3 Reliable Transport

Reliable Hop Tubes use acknowledgment-based loss detection and a congestion control algorithm similar to NewReno. We describe these mechanisms below.

6.3.1 Loss Recovery

Reliable Hop Tubes rely on duplicated acknowledgments to detect packet loss and a complementary Retransmission Timeout (RTO) strategy. When the sender receives a duplicate acknowledgment, it retransmits the missing frame with the same data, unlike segment coalesce in TCP [49]. Upon receiving additional acknowledgments for the same lost frame, the sender increments the missing frame number index and proactively retransmits subsequent frames. This strategy anticipates burst losses and reduces timeouts. Hop must also account for packet reordering. In TCP, best practices for loss detection rely on a reordering threshold of three duplicate acknowledgments [25, 26]. However, packet reordering is expected to be more common in Hop than in TCP, since network elements that may observe and reorder TCP packets cannot do so for Hop, due to encrypted Frame Numbers. To mitigate spurious loss detection, we empirically set the reordering threshold in Hop to ten. This threshold is maintained by the receiver, which issues a duplicate acknowledgment only when the threshold is reached. The receiver resets the reordering counter each time it transmits a duplicate acknowledgment.

The second loss recovery mechanism in Hop using an estimated RTT of the network is the Retransmission Timeout (RTO). Inspired by QUIC [66], Hop smooths its RTT calculation using an exponentially weighted moving average, assigning the new RTT sample a weight of 1/8 relative to the previous estimate. To maintain an accurate RTT estimate and avoid TCP retransmission ambiguity [69], Hop stores a timestamp with each frame and updates it upon retransmission. This timestamp is only maintained on the sender side and is not sent alongside the frame payload.

To account for network variability, the RTO is set to 9/8 of the smoothed RTT. When the RTO expires, the sender retransmits the oldest frame with an `RTR` flag. This flag notifies the receiver of the congestion, requiring the receiver to prioritize sending the latest acknowledged frame number. This mechanism serves two purposes: (i) to update the sender with the most recently acknowledged frames in case acknowledgments were lost, and (ii) to unblock the receiver as frames are processed in order. If multiple consecutive RTOs occur, the RTO timer is backed off exponentially by duplication. If the RTO exceeds ten seconds, the sender removes the corresponding frames from its retransmission list in case the receiver has already interrupted the connection.

6.3.2 Congestion Control

Hop implements sender-side congestion using a loss-based scheme similar to TCP NewReno [51] and its congestion control phases: Slow Start, Additive Increase/Multiplicative Decrease (AIMD), and Fast Recovery. In Hop, the window determines the number of outstanding frames in flight, bounded with a minimum of ten frames and a maximum of one thousand.

Congestion control begins in Slow Start with the congestion window set to ten frames. The window doubles every RTT until the first loss event (§6.3.1). On the first loss, the slow start threshold is set to half of the current congestion window, and Hop transitions out of Slow Start. On reception of a new acknowledgment, Hop updates RTT and RTO estimates, removes acknowledged frames, and adjusts the congestion window. In AIMD, the window increases by one frame per RTT. In Slow Start and Fast Recovery, the window doubles every RTT. When the window grows beyond the slow start threshold, Hop transitions to AIMD.

When a loss occurs, Hop adjusts the window size conservatively to 25% of its current size to avoid underutilization of the network and indexes the slow start threshold on the newly calculated value. The congestion window is never reduced below ten frames. For each loss, the missing frame is immediately retransmitted to avoid further timeouts (§6.3.1). On timeout, the sender retransmits the oldest outstanding frame. A timeout also forces entry into Fast Recovery: the

congestion window is reduced by 25% on each retransmission. Consecutive timeouts trigger multiple retransmissions to compensate for burst losses.

Unlike TCP, we define Hop Tubes' congestion window as a quantity of frames rather than bytes, which simplifies flow-control logic at the cost of reduced precision. Hop intentionally overestimates the window size to better tolerate bursty losses. Too aggressive reductions of the window size on each loss event would require Hop to stop sending frames and resynchronize with the receiver. Hop does not implement pacing when transmitting frames, which results in short-term bursts and transient congestion. We leave both pacing and a pluggable congestion-control interface as future work.

7 Hop Remote Access Protocol

Building on the foundation provided by Hop Transport and Hop Tubes, Hop Remote Access offers a significantly smaller and simpler implementation (total of 18K LoC in Go vs. 130K LoC in C for OpenSSH). In addition to the improved security provided by lower layers such as server identity verification, Hop introduces secure identity forwarding through a novel authorization grant mechanism and adds support for roaming and intermittent connectivity.

7.1 Remote Access Services

Hop Remote Access initially supports four network services: (1) opening a remote login shell, (2) executing a specific command on a remote host, (3) local port forwarding, and (4) remote port forwarding. A Hop Session can contain several of these network services (e.g., a user can have a remote shell and also use the same Hop Session to perform port forwarding). A Hop Session's duration is tied to the lifetime of either the remote login shell or the command being executed. If a Hop Session is started in headless mode (no remote login or command executed), then the session persists until the client is manually terminated. Beyond forwarding TCP and Unix domain socket connections over reliable tubes, Hop also supports UDP port forwarding using best-effort tubes.

7.2 Secure Delegation

SSH does not provide first-class support for delegating credentials. Rather, it uses workarounds like `ssh-agent` forwarding [87], which prior work has demonstrated to be insecure [74] as it gives any process with root privileges on the remote third-party machine carte blanche to perform any action as the user. For example, if a user forwards their key to perform a `git pull`, an attacker could also open a shell on a remote machine using that user's forwarded key. In contrast, a secure protocol should guarantee that the user's local credentials will only be used to perform the requested action on the desired server at the desired time.

```
Would you like to
  allow bob.cloud.com
  to run the command 'sudo reboot'
  as alice
  on private.server.com
  from Wed Apr  9 16:03:28 EDT 2025
  until Wed Apr  9 16:04:28 EDT 2025?
     Yes
   > No
```

Figure 4: **Secure Delegation**—Intent dialogue presented to the Principal during step ③ of the authorization grant protocol illustrated in Figure 5. If the intent is accepted by the Principal and the Target, this intent will give the permission to the Delegate (`bob.cloud.com`) to execute a specific command as the user `alice` on the Target (`private.server.com`) for a one-minute duration.

Kogan et al. have denoted that a delegation protocol exists between three parties: a principal, a delegate, and a target. According to their prior work, Secure Delegation Principle [74], a secure delegation protocol should guarantee that a delegate may only act under a principal's authority after the principal can verify and enforce the delegate's intent. The intent consists of the *who* (the identity of the delegate requesting authorization), the *what* (a single action the delegate can perform), the *when* (a window of time when the delegate is authorized to perform the action) and the *to whom* (the identity of the target on whom the delegate can perform the action). Hop provides users explicit control over delegation when granting permissions. Figure 4 shows an example of a dialog prompted to the user when a delegate server requests a one-minute duration authorization to execute a specific command on a target server.

7.3 Authorization Grant Protocol

Hop natively provides a secure identity forwarding with a novel delegation protocol that enables explicit delegation through authorization grants. An authorization grant represents a short-lived, single-use, pre-approved token (called an "intent") that allows a semi-trusted machine (a Delegate) to act with limited, specified authority on behalf of a trusted machine (the Principal). Authorization grants provide the Target with a cryptographic guarantee that the Principal gave the Delegate permission to perform certain actions for a certain period of time. The authorization grant protocol flow follows the design principles of Req. 5 and is illustrated in Figure 5. There are three parties involved in the protocol:

1. **Principal**: A Hop client on a trusted user machine, holding long-term credentials and managing the delegation requests.
2. **Delegate**: A Hop client on a semi-trusted machine that requests delegated access to the Target.

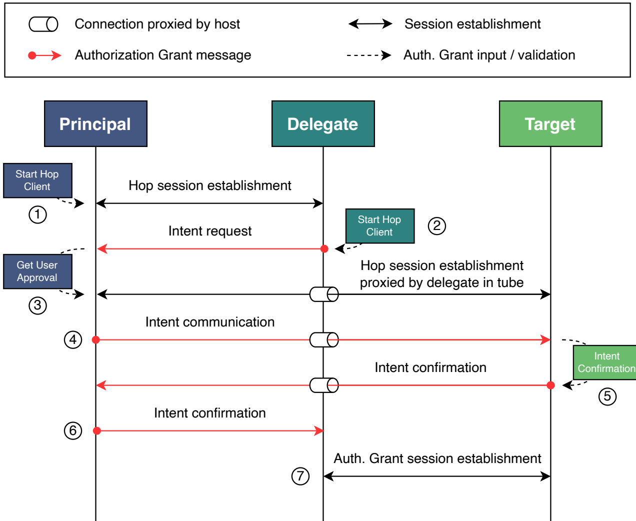


Figure 5: **Authorization Grant Protocol**—This protocol enables secure identity forwarding, allowing a Principal to grant a semi-trusted Delegate permission to act on a Target. The process consists of: ① The Principal starts a Hop client and establishes a Hop Session. ② The Delegate requests authorization (*Intent Request*) for an action on the Target. ③ The Principal verifies the request and asks for user confirmation via a dialogue (see Figure 4). If approved, the Principal establishes a session through a handshake with the Target, proxied by the Delegate. ④ The Principal sends an *Intent Communication* message to the Target. ⑤ If the Target approves the *Intent Communication* (e.g., `alice` is allowed to reboot the Target), it allocates an authorization grant and sends an *Intent Confirmation* message. ⑥ The Principal relays approval to the Delegate. ⑦ The Delegate establishes a Hop Session with the Target to execute the action, after which the temporary authorization is revoked. Each intent, and its corresponding action, is authorized only once, ensuring controlled delegation. If either the Principal or the Target denies the intent (③ or ⑤), an *Intent Denied* message is forwarded to the Delegate to terminate the authorization grant process and no further steps are executed.

3. **Target**: A Hop server that the Delegate wants to access.

The authorization grant protocol begins with ① Hop Session establishment between the Principal and the Delegate, authenticated by the Principal identity. As the Delegate does not have credentials to authenticate to the Target, ② it sends an *Intent Request* to the Principal over the already established Hop Session. This message includes a description of the requested action that the Delegate wants to execute (e.g., initiating a `git pull` or opening a shell) along with an Ephemeral Client Certificate (§5.6).

Upon receiving the request, the authorization grant flow stops and the Principal requests explicit approval from the user (Figure 4). If the user does not consent, the Principal sends an *Intent Denied* message to the Delegate and does not proceed further in the authorization grant protocol. If the user consents, ③ the Principal initiates a proxied handshake with the Target through its existing connection with the Delegate. Once completed, ④ the Principal sends an *Intent Communication* message to the Target, forwarding the original *Intent Request* from the Delegate along with metadata indicating the Principal approval.

The Target evaluates the received *Intent Communication* and, if the Delegate is authorized to perform the action on the Target (e.g., `git pull` or `rsync secret.txt`), it creates an authorization grant for the Delegate and adds the action information from the *Intent Communication*. The authorization grant is stored in an in-memory mapping maintained by the Target. This intent is scoped to a specific action and session, and it cannot be reused. ⑤ The target then confirms and issues an *Intent Confirmation* message, which ⑥ is subsequently forwarded to the Delegate by the Principal. Using its own identity, ⑦ the Delegate can now establish a new Hop session with the Target.

The Delegate may send additional *Intent Requests* to the Principal if there are multiple actions it needs to perform. If at least one of the Delegate's *Intent Requests* is confirmed, the Delegate can establish a Hop Session with the Target. After the Delegate completes Hop Session establishment, the Target removes the authorization grant from its in-memory mapping so that it cannot be reused. It keeps track of the approved actions from the authorization grant for the Hop Session and allows each one to be performed at most once.

The authorization grants are bound to a specific set of actions, Target, and Delegate, and are immediately invalidated after use or timeout. This makes this delegation non-transitive, time-bound, and non-repayable by a semi-trusted Delegate, and thus, limits lateral movements in a given infrastructure. This system can also be extended to multi-hop delegation: if a Delegate connects to a Target that itself runs a Hop client, that client can become a delegate in a further chain, issuing its own intent requests back to the original principal. This enables deep delegation chains without exposing any credentials or keys to semi-trusted environments.

8 Evaluation

Having presented Hop's design, we now evaluate both its security properties and performance.

8.1 Security Requirements

We start by considering how our proposed protocol addresses the requirements set forth in Section 2:

Requirement 1. Hop leverages the shared operational relationship between client and server (§5) (cf. web servers must support uncoordinated web browsers) and implements a simple, versioned handshake protocol (§5.1) with a fixed

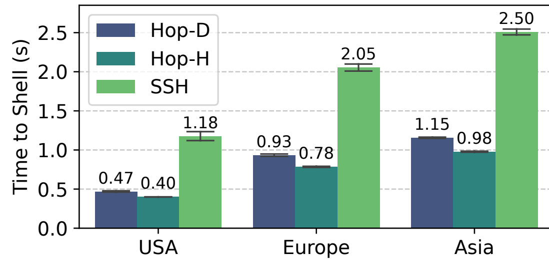


Figure 6: **Time to Shell**—Hop and SSH session establishment from the US. Hop initiates a session in 5 and 6 round-trips for Hop-H and Hop-D compared to 12 round-trips for SSH. In each case, Hop outperforms SSH due to its inherent protocol design and demonstrates greater stability. RTT: USA = 69 ms, Europe = 144 ms, Asia = 186 ms.

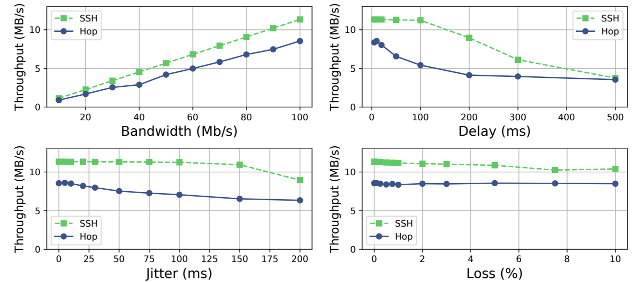


Figure 7: **Performance Under Simulated Conditions**—Hop and SSH in simulation during the transfer of 1 GB of random data. Before isolated variation bandwidth = 100 Mb/s, delay = 10 ms, jitter = 0 ms, and loss = 0%. Hop shows consistent reliability comparable to SSH.

set of cryptographic primitives and no in-band negotiation, ensuring downgrade resistance. All handshake messages are authenticated using the Noise Framework (§5.2 and §5.3) and bound to a duplex construction providing full transcript integrity, session uniqueness, and replay resistance against an active adversary.

Requirement 2. Hop does not rely on any weak authentication mechanisms and uses certificates for both client and server authentication (§5.6.1), preventing man-in-the-middle attacks during the first and following connections. Hop prevents initiator KCI by either combining ephemeral and static Diffie-Hellman keys or using an out-of-band shared KEM key. Hop is also designed to enable a semi-trusted third-party to operate an ACME [1] certificate issuer that validates server identity and issues credentials that chain to a root of trust distributed in the client (e.g., a cloud provider could operate an ACME server and deploy a root certificate, optionally constrained to the cloud network or select hostnames).

Requirement 3. Hop can be used with a "key vending machine" in Hop, backed by web service login or backed by single-sign-on and multi-factor authentication in enterprise settings for client identification. The key vending machine issues a scoped and time-limited client certificates that form a certificate-based chain of trust, which prevents credential theft reuse (§5.6.2). Authorization Grants bind delegated actions and key usage to a specific remote context and require user approval from the principal client (§7.3). In Hop, client credentials are never exposed directly in server memory, nor as a signing oracle via Remote Procedure Call.

Requirement 4. Hop does not disclose client and server identities to network observers by encrypting with forward secrecy all handshake identifiers, including certificates and SNI (§5.2). In addition, in Hop hidden mode, servers only respond to clients possessing the correct static public KEM key, rendering them invisible to active scanners and unauthenticated probes (§5.3). Hop clients never reveal their identities to unknown or unauthenticated servers.

Requirement 5. Inspired by Guardian Agent [74], Hop's Authorization Grant Protocol provides a cryptographically fine-grained control over specific actions on a remote server for a short period of time, preventing additional lateral movement even if delegate hosts are compromised (§7.3).

Requirement 6. Hop prevents DoS amplification by either generating a stateless cryptographically protected cookie during initial contact (§5.2) or requiring an authenticated client request (§5.3). Encrypted Connection IDs preserve authentication during roaming and prevent session hijacking or UDP spoofing, including during handshake failure (§5.4). Hop natively supports reliable and unreliable traffic facilitated by Hop Tubes (§6).

Requirement 7. Hop is designed for constrained environments by using lightweight certificates with a fixed, deterministic validation path, and bounded size (§5.1 and §5.6). This minimizes parsing complexity and reduces implementation risk on resource-limited devices.

Requirement 8. Hop implements post-quantum key-establishment mechanisms. During the handshake, endpoints first perform an ephemeral post-quantum ML-KEM key exchange [97], ensuring confidentiality of session keys against "harvest now, decrypt later" adversaries. Authentication is provided separately using classical Diffie-Hellman–based mechanisms (§5.1).

8.2 Performance

We evaluate Hop's performance under simulation and real-world end-to-end tests to match important use cases for a remote access protocol, considering session establishment latency, throughput, and terminal responsiveness across multiple scenarios. We emphasize that our goal is not to *beat* SSH's performance: OpenSSH is a highly optimized implementation written in C, and the Linux kernel implementation

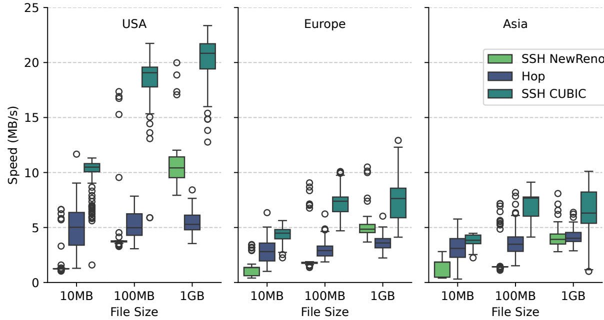


Figure 8: **File Transfer Speed**—Hop and SSH using TCP NewReno and CUBIC when uploading different files from the US to cloud-hosted servers. Hop shows comparable throughput to SSH NewReno across the experiments, especially for small files, due to its optimistic and larger window size in its congestion control. Hop remains consistent over the same network configuration regardless of the transfer size. Hop accurately evaluates its network capabilities, leading to fewer outliers over the transfers.

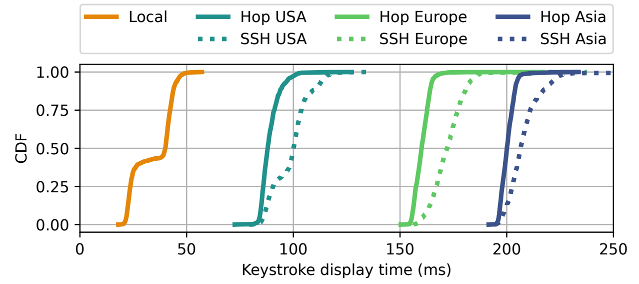


Figure 9: **Keystroke latency**—CDF of keystroke display times in the Vim editor for Hop and SSH sessions. Hop consistently shows lower and more tightly clustered latencies, suggesting a more responsive experience.

of TCP is some of the most highly scrutinized network code. Rather, we seek to show that Hop is a fully functional implementation, capable of operating in real-world environments with similar characteristics.

Experimental Setup. We designed our experiment to have a broad coverage to capture network congestion effects. For real-world experiments, we set up three cloud-based Hop servers in the US, Europe, and Asia. We used a client located in the US to connect via WiFi to those servers using both Hop and SSH to compare their performance. We used ChaCha20-Poly1305 cryptography for SSH evaluations to ensure no cryptographic hardware acceleration is involved.

Session Establishment ("Time to Shell"). We first evaluated the time required to establish a non-interactive shell and execute an initial command using SSH and Hop. For each host, we conducted 300 session establishments, cycling between the protocols. By design, Hop uses fewer round-trips to establish a session: 5 round-trips for Hop Hidden Mode (Hop-H), 6 round-trips for Hop Discoverable Mode (Hop-D), and 12 round-trips for SSH. In Figure 6, Hop shows lower variability in connection time. The "time to shell" metric is strongly correlated with the RTT and the number of round-trips required to complete the handshake, and serves as a quantitative measure of connection latency.

File Transfer Speed. We also evaluated Hop throughput in a controlled simulation environment to isolate the effects of network variation. Using Mininet [79], we connected two hosts to a router with a bottleneck link initially configured at 100 Mb/s bandwidth, 10 ms delay, no jitter, and no loss. As shown in Figure 7, Hop utilized the network similarly to SSH over TCP NewReno, though it exhibited about 40% lower throughput under bandwidth variation across experiments.

To complement the simulation, we conducted real-world experiments by sending different files (10 MB, 100 MB, 1 GB) using the `rsync` tool [116] from a US workstation to globally distributed servers. Each file transfer was repeated at least 50 times over three days to account for variability. As shown in Figure 8, Hop achieved stable and predictable performance across file sizes, in contrast to SSH with TCP NewReno and CUBIC, which were more sensitive to bandwidth and latency differences. Consistent with our simulation results, the throughput gap between Hop and SSH narrowed as network bandwidth decreased. For 100 MB files, TCP NewReno demonstrated highly consistent performance, with tightly clustered quartiles. TCP NewReno sometimes experienced significantly low congestion, resulting in similar throughput as CUBIC. A contrario, TCP CUBIC occasionally overestimates its congestion window, producing lower throughput. Nonetheless, SSH with TCP CUBIC consistently delivered the highest throughput across all regions and file sizes, especially in the US (the shortest distance).

We attribute Hop's behavior to its current implementation, which extends TCP NewReno. Hop benefits from a larger initial congestion window that accelerates transfer startup. However, its reliance on UDP without pacing leads to more frequent burst losses, resulting in slower linear throughput growth compared to TCP-based congestion control. These characteristics affect real-world performance but are tunable and independent of the Hop security protocol itself.

Keystroke Latency. We define keystroke latency as the time between pressing a key and seeing the character appear on the screen. We argue that it is a critical factor for user-perceived responsiveness in remote access protocols. To isolate terminal performance, we conducted measurements over already-established sessions with a 150 ms keystroke interval. We followed the methodology of SSH3 evaluations [86], and measured latency inside the Vim text editor [91] using the Typometer tool [50] to record the keystroke latency.

Figure 9 shows that Hop has a consistently tighter latency distribution, with values closer to the actual RTT than SSH. While part of this difference comes from Hop's use of UDP, which avoids mechanisms such as Nagle's algorithm and head-of-line blocking, it also reflects Hop's lightweight framing and fast processing pipeline. Hop design choices lead to more stable and responsive keystroke rendering without implementing speculative keystroke prediction [121].

9 Conclusion

In this work, we introduced Hop, a reimagined remote access protocol. Hop avoids the complexity of SSH by being cryptographically opinionated, using a key exchange handshake based on the Noise framework, and deploying lightweight certificates for mutual authentication. Hop's simpler protocol allows for smaller implementations that are practically verifiable and easier to deploy on resource-constrained platforms. Improving upon the functionality of SSH, Hop provides native support for secure identity forwarding through the Authorization Grant Protocol. Ultimately, Hop maintains the core functionality of SSH while aligning better with modern use cases and using modern cryptographic best practices. Our evaluation of Hop's Go implementation demonstrates its practicality. We hope our work sparks conversation about what a modern remote access protocol should look like going forward.

Acknowledgments

We thank Keith Winstein and members of the Stanford Empirical Security Research Group. This work was supported in part by a Sloan Research Fellowship and the National Science Foundation under Grant Number #2319080. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

Ethical Considerations

We discuss the ethical considerations of our work below:

Stakeholders. The stakeholders in this research are: end-users of remote access protocols; operators of the DigitalOcean cloud infrastructure, which hosted virtual machines for our real-world experiments; other tenants sharing the same infrastructure; the broader research community; and the authors themselves.

Ethical Principles. We use the the Menlo Report principles to structure our ethical considerations:

- **Respect for Persons:** No personal or user data was collected, and no human subjects were involved.
- **Beneficence:** We minimized risks by conducting controlled experiments and conducting real-world experiences using infrastructure that we fully controlled. The benefits are the assessment of the functioning of a new network protocol under real-world conditions.
- **Justice:** The research outcomes do not exclude or disadvantage any specific group.
- **Respect for Law and Public Interest:** All tests were conducted within our own accounts on DigitalOcean, complying with its terms of service and within allocated resource limits.

Potential Harms and Mitigation. Potential harms include: (1) excessive load on shared infrastructure; (2) unintended interference with other tenants; and (3) insecure deployment if Hop were prematurely adopted. To mitigate (1) and (2), we limited traffic volumes to remain within typical usage patterns for rented virtual machines and continuously monitored for abnormal behavior (such as overconsumption of the resources). We did not attempt to probe, measure, or interact with other tenants. To mitigate (3), Hop was not publicly deployed and was only used on controlled servers under our own accounts, containing no sensitive or private data.

Decision to Publish. We judge that the benefits of sharing this research outweigh the minimal risks. The protocol and evaluation results advance the state of knowledge in secure remote access protocols, which we argue can improve the security and privacy of users globally long term.

Open Science

We make all artifacts including available online, including:

- **Source code:** Implementation of Hop client and server.
- **Simulation scripts:** Mininet scripts for controlled experiments reproducing Figure 7.
- **Measurement scripts:** Scripts for session establishment latency, file transfer throughput, and keystroke latency experiments.
- **Datasets and results:** Plotting scripts used to generate all evaluation figures.
- **Documentation:** A README with instructions for running the software and reproducing experiments.

Artifacts are available at: `https://doi.org/10.5281/zenodo.17953396`. We note that cloud experiments were conducted on DigitalOcean infrastructure. Since commercial cloud instances cannot be redistributed directly, we provide scripts and configuration files to enable reproduction.

References

[1] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-Lopez, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and B. Warren. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *ACM Conference on Computer and Communications Security*, 2019.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM Conference on Computer and Communications Security*, 2015.

[3] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[4] C. Alberca, S. Pastrana, G. Suarez-Tangil, and P. Palmieri. Security analysis and exploitation of arduino devices in the internet of things. In *ACM International Conference on Computing Frontiers*, 2016.

[5] N. Alnahawi, J. Müller, J. Oupický, and A. Wiesmaier. A Comprehensive Survey on Post-Quantum TLS. *IACR Communications in Cryptology*, 1(2), 2024.

[6] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security evaluation of home-based IoT deployments. In *IEEE Symposium on Security and Privacy*, 2019.

[7] Y. Angel, B. Dowling, A. Hülsing, P. Schwabe, and F. Weber. Post quantum noise. In *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[8] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the Mirai botnet. In *USENIX Security Symposium*, 2017.

[9] B. Auerbach, Y. Dodis, D. Jost, S. Katsumata, and R. Schmidt. How to compare bandwidth constrained two-party secure messaging protocols: a quest for a more efficient and secure post-quantum protocol. In *USENIX Security Symposium*, 2025.

[10] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, et al. DROWN: Breaking TLS using SSLv2. In *USENIX Security Symposium*, 2016.

[11] M. Azimpourkivi, U. Topkara, and B. Carbunar. Human distinguishable visual key fingerprints. In *USENIX Security Symposium*, 2020.

[12] U. Banerjee, C. Juvekar, S. H. Fuller, and A. P. Chandrakasan. eedtls: Energy-efficient datagram transport layer security for the internet of things. In *IEEE Global Communications Conference*, 2017.

[13] F. Bäumer, M. Brinkmann, and J. Schwenk. Terrapin attack: Breaking SSH channel integrity by sequence number manipulation. In *USENIX Security Symposium*, 2024.

[14] F. Bäumer, M. Maehren, M. Brinkmann, and J. Schwenk. Finding ssh strict key exchange violations by state learning. In *ACM Conference on Computer and Communications Security*, 2025.

[15] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *14th Annual International Cryptology Conference on Advances in Cryptology*, 1994.

[16] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in ssh: provably fixing the ssh binary packet protocol. In *ACM Conference on Computer and Communications Security*, 2002.

[17] S. M. Bellovin. Security Problems in the TCP/IP Protocol Suite. In *Computer Communication Review*, 1989.

[18] B. Benčina, B. Dowling, V. Maram, and K. Xagawa. Post-quantum cryptographic analysis of SSH. Cryptology ePrint Archive, Paper 2025/684, 2025.

[19] D. J. Bernstein. SYN cookies. `cr.yp.to/syncookies.html`.

[20] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. *Communications of the ACM*, 2017.

[21] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy*, 2017.

[22] K. Bhargavan, V. Cheval, and C. Wood. A symbolic analysis of privacy for TLS 1.3 with encrypted client hello. In *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[23] K. Bhargavan and G. Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium*, 2016.

[24] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. In *Foundations and Trends in Privacy and Security*, 2016.

[25] E. Blanton, M. Allman, L. Wang, I. Järvinen, M. Kojo, and Y. Nishida. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675, Aug. 2012.

[26] E. Blanton, D. V. Paxson, and M. Allman. TCP Congestion Control. RFC 5681, Sept. 2009.

[27] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[28] E. Boo, S. Raza, J. Hoglund, and J. Ko. FDTLS: supporting DTLS-based combined storage and communication security for IoT devices. In *IEEE Intl. Conf. on Mobile Ad Hoc and Sensor Systems*, 2019.

[29] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *IEEE Symposium on Security and Privacy*, 2014.

[30] CA/Browser Forum Code Signing Certificate Working Group. Baseline requirements for the issuance and management of publicly-trusted code signing certificates, v3.10.0.

[31] A. Capossele, V. Cervo, G. De Cicco, and C. Petrioli. Security as a CoAP resource: an optimized dtls implementation for the iot. In *IEEE International Conference on Communications (ICC)*, 2015.

[32] Y. Chen and Z. Su. Guided differential testing of certificate validation in ssl/tls implementations. In *10th Joint Meeting on Foundations of Software Engineering*, 2015.

[33] Cloudflare. What is mTLS? `https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/`.

[34] I. Coonjah, P. C. Catherine, and K. M. S. Soyjaudah. Experimental performance comparison between TCP vs UDP tunnel using OpenVPN. In *International Conference on Computing, Communication and Security (ICCCS)*, 2015.

[35] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[36] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy*, 2016.

[37] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology*, 2020.

[38] A. Danial. cloc software v2.04. `github.com/AlDanial/cloc`.

[39] S. Dechand, D. Schürmann, K. Busse, Y. Acar, S. Fahl, and M. Smith. An empirical study of textual Key-Fingerprint representations. In *USENIX Security Symposium*, 2016.

[40] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *IEEE Symposium on Security and Privacy*, 2021.

[41] J. D. D. H. Diego, J. Saldana, J. Fernández-Navajas, and J. Ruiz-Mas. Iotsafe, decoupling security from applications for a safer IOT. *IEEE Access*, 2019.

[42] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[43] J. A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Network and Distributed System Security Symposium*, 2017.

[44] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *ACM Conference on Computer and Communications Security*, 2015.

[45] Z. Durumeric, M. Bailey, and J. A. Halderman. An internet-wide view of internet-wide scanning. In *USENIX Security Symposium*, 2014.

[46] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the https certificate ecosystem. In *ACM Internet Measurement Conference*, 2013.

[47] Dustin Moody, Ray Perlner, Andrew Regenscheid, Angela Robinson, David Cooper. NIST IR 8547 - Transition to Post-Quantum Cryptography Standards. Technical report, NIST, 2024. https://doi.org/10.6028/NIST.IR.8547.ipd.

[48] M. J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015.

[49] W. Eddy. Transmission Control Protocol (TCP). RFC 9293, 2022.

[50] P. Fatin. typometer v1.0.1. github.com/pavelfatin/typometer.

[51] S. Floyd, T. Henderson, and A. Gurtov. The NewReno modification to TCP's fast recovery algorithm. Technical report, RFC Editor, 2004.

[52] F. Forsby, M. Furuhed, P. Papadimitratos, and S. Raza. Lightweight x.509 digital certificates for the internet of things. In *Interoperability, Safety and Security in IoT*, 2018.

[53] S. Gallenmüller, D. Schöffmann, D. Scholz, F. Geyer, and G. Carle. DTLS performance: How expensive is security? *arXiv:1904.11423*.

[54] O. Gasser, R. Holz, and G. Carle. A deeper understanding of SSH: results from internet-wide scans. In *IEEE Network Operations and Management Symposium*, 2014.

[55] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *ACM Conference on Computer and Communications Security*, 2012.

[56] A. Ghedini and V. Vasiliev. TLS Certificate Compression. RFC 8879.

[57] Y. Gluck, N. Harris, and A. Prado. BREACH: reviving the CRIME attack. *Unpublished manuscript*, 2013.

[58] P. Gutmann. Do users verify SSH keys. *Login*, 36(4):35–36, 2011.

[59] M. Hamburg. The STROBE protocol framework. *Cryptology ePrint Archive*, 2017.

[60] A. Haroon, S. Akram, M. A. Shah, and A. Wahid. E-lithe: A lightweight secure dtls for iot. In *IEEE Vehicular Technology Conference (VTC-Fall)*, 2017.

[61] E. Heilman. Open-sourcing openpubkey ssh (opkssh): integrating single sign-on with ssh. https://blog.cloudflare.com/open-sourcing-openpubkey-ssh-opkssh-integrating-single-sign-on-with-ssh/, Mar. 2025. Cloudflare Blog.

[62] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, 2012.

[63] T. Horák, M. Šimon, L. Huraj, and R. Budjač. Vulnerability of smart iot-based automation and control devices to cyber attacks. In *Applied Informatics and Cybernetics in Intelligent Systems*, 2020.

[64] A. Hülsing, K.-C. Ning, P. Schwabe, F. J. Weber, and P. R. Zimmermann. Post-quantum wireguard. In *IEEE Symposium on Security and Privacy*, 2021.

[65] IETF SSHM. Secure Shell Maintenance (sshm): Charter for working group, 2024. datatracker.ietf.org/wg/sshm/about/.

[66] J. Iyengar and I. Swett. QUIC Loss Detection and Congestion Control. IETF RFC 9002, 2021.

[67] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. IETF RFC 9000, 2021.

[68] P.-M. Junges, J. François, and O. Festor. Software-based analysis of the security by design in embedded devices. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.

[69] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *SIGCOMM CCR*, 1987.

[70] M. Kayali, J. Schmitt, and F. Roesner. Ssh-passkeys: Leveraging web authentication for passwordless ssh, 2025.

[71] S. Kent and K. Seo. Rfc 4301: Security architecture for the internet protocol, 2005.

[72] K. Kleine and D. E. Simos. Coveringcerts: Combinatorial methods for x. 509 certificate testing. In *IEEE International conference on software testing, verification and validation (ICST)*, 2017.

[73] N. Kobeissi, G. Nicolas, and K. Bhargavan. Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols. In *IEEE European Symposium on Security and Privacy*, 2019.

[74] D. Kogan, H. Stern, A. Tolbert, D. Mazières, and K. Winstein. The Case For Secure Delegation. In *ACM Workshop on Hot Topics in Networks*, 2017.

[75] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from hell? reducing the impact of amplification ddos attacks. In *USENIX Security Symposium*, 2014.

[76] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric. All things considered: an analysis of iot devices on home networks. In *USENIX Security Symposium*, 2019.

[77] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey. Tracking certificate misissuance in the wild. In *IEEE Symposium on Security and Privacy*, 2018.

[78] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, 2017.

[79] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[80] P. Li, J. Su, and X. Wang. itls: Lightweight transport-layer security protocol for iot with minimal latency and perfect forward secrecy. *IEEE Internet of Things Journal*, 2020.

[81] B. Lipp, B. Blanchet, and K. Bhargavan. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. In *IEEE European Symposium on Security and Privacy*, 2019.

[82] S. Mackey, I. Mihov, A. Nosenko, F. Vega, and Y. Cheng. A performance comparison of wireguard and openvpn. In *ACM Conference on data and application security and privacy*, 2020.

[83] M. Malik, M. Dutta, J. Granjal, et al. L-ECQV: lightweight ECQV implicit certificates for authentication in the internet of things. *IEEE Access*, 2023.

[84] E. McMahon, R. Williams, M. El, S. Samtani, M. Patton, and H. Chen. Assessing medical device vulnerabilities on the internet of things. In *IEEE International Conference on Intelligence and Security Informatics*, 2017.

[85] K. L. McMillan and L. D. Zuck. Formal specification and testing of quic, 2019.

[86] F. Michel and O. Bonaventure. Towards SSH3: how HTTP/3 improves secure shells. *arXiv preprint arXiv:2312.08396*, 2023.

[87] D. Miller. Openssh protocol vendor extensions, July 2008.

[88] D. Miller. About OpenSSH "user enumeration" / CVE-2018-15473. Message to the oss-security mailing list, Aug 2018. Message ID: <alpine.BSO.2.21.1808241046220.67512@haru.mindrot.org>.

[89] MITRE Corporation. CVE-2016-20012, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-20012.

[90] B. Möller, T. Duong, and K. Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *Security Advisory*, 21:34–58, 2014.

[91] B. Moolenaar. Vim, the editor, 2025. https://www.vim.org.

[92] C. Munteanu, G. Smaragdakis, A. Feldmann, and T. Fiebig. Catch-22: Uncovering compromised hosts using SSH public keys. In *USENIX Security Symposium*, 2025.

[93] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896.

[94] K. T. Nguyen, M. Laurent, and N. Oualha. Survey on secure communication protocols for the internet of things. *Ad Hoc Networks*, 2015.

[95] A. Niakanlahiji, J. Wei, M. R. Alam, Q. Wang, and B.-T. Chu. ShadowMove: A stealthy lateral movement strategy. In *USENIX Security Symposium*, 2020.

[96] NIST. Lightweight Cryptography. https://csrc.nist.gov/Projects/Lightweight-Cryptography, 2017.

[97] NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical report, NIST, 2025.

[98] NXP. A light-weight TLS and X.509 profile. https://www.nxp.com/docs/en/white-paper/LWTLSWP.pdf.

[99] A. Oak and R. Daruwala. Assessment of message queue telemetry and transport (mqtt) protocol with symmetric encryption. In *Secure Cyber Computing and Communication*, 2018.

[100] T. Perrin. The noise protocol framework, 2018.

[101] M. Raavi, S. Wuthier, P. Chandramouli, X. Zhou, and S.-Y. Chang. QUIC Protocol with Post-quantum Authentication. In *Information Security: 25th International Conference*, 2022.

[102] S. Raza, L. Seitz, D. Sitenkov, and G. Selander. S3k: Scalable security with symmetric keys—DTLS key establishment for the internet of things. *IEEE Tran. on Automation Science & Engineering*, 2016.

[103] S. Raza, D. Trabalza, and T. Voigt. 6LoWPAN compressed DTLS for CoAP. In *IEEE Distributed Computing in Sensor Systems*, 2012.

[104] J. Rizzo and T. Duong. The crime attack. In *ekoparty security conference*, 2012.

[105] L. Roy, S. Lyakhov, Y. Jang, and M. Rosulek. Practical privacy-preserving authentication for SSH. In *USENIX Security Symposium*, 2022.

[106] K. Ruth, V. A. Rivera, G. Akiwate, A. Fass, P. G. Kelley, K. Thomas, and Z. Durumeric. "perfect is the enemy of good": The ciso's role in enterprise security as a business enabler. In *CHI Conference on Human Factors in Computing Systems*, 2025.

[107] H. Sardeshmukh and D. Ambawade. A DTLS based lightweight authentication scheme using symmetric keys for Internet of Things. In *International Conference on Wireless Communications, Signal Processing and Networking*, 2017.

[108] Q. Scheitle, O. Gasser, T. Nolte, J. Amann, L. Brent, G. Carle, R. Holz, T. C. Schmidt, and M. Wählisch. The rise of certificate transparency and its implications on the internet ecosystem. In *ACM Internet Measurement Conference*, 2018.

[109] S. Schmieg, S. Kölbl, and G. Endignoux. Google's threat model for post-quantum cryptography, 2024. https://bughunters.google.com/blog/5108747984306176/google-s-threat-model-for-post-quantum-cryptography.

[110] S. Sciancalepore, A. Capossele, G. Piro, G. Boggia, and G. Bianchi. Key management protocol with implicit certificates for iot systems. In *IoT challenges in Mobile & Industrial Systems*, 2015.

[111] M. Shobana and S. Rathi. IoT malware: An analysis of IoT device hijacking. *International Journal of Scientific Research in Computer Science, Computer Engineering, and Information Technology*, 2018.

[112] D. X. Song, D. Wagner, and X. Tia. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, 2001.

[113] K. Tange, D. Howard, T. Shanahan, S. Pepe, X. Fafoutis, and N. Dragoni. rtls: Lightweight tls session resumption for constrained iot devices. In *Information and Communications Security: 22nd International Conference*, 2020.

[114] C. Tian, C. Chen, Z. Duan, and L. Zhao. Differential testing of certificate validation in SSL/TLS implementations: an RFC-guided approach. *ACM Tran. on Software Engineering & Methodology*, 2019.

[115] R. T. Tiburski, L. A. Amaral, E. de Matos, D. F. de Azevedo, and F. Hessel. Evaluating the use of tls and dtls protocols in iot middleware systems applied to e-health. In *14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2017.

[116] A. Tridgell, P. Mackerras, et al. The rsync algorithm, 1996.

[117] D. Turner, S. F. Shahandashti, and H. Petrie. The effect of length on key fingerprint verification security and usability. *arXiv preprint arXiv:2306.04574*, 2023.

[118] M. Vučinić, B. Tourancheau, T. Watteyne, F. Rousseau, A. Duda, R. Guizzetti, and L. Damon. Dtls performance in duty-cycled networks. In *Symposium on Personal, Indoor, and Mobile Radio Communications*, 2015.

[119] D. Wendlandt and A. Perrig. Perspectives: Improving ssh-style host authentication with Multi-Path probing. In *USENIX Annual Technical Conference (ATC)*, 2008.

[120] S. C. Williams. Analysis of the ssh key exchange protocol. In *Cryptography and Coding: 13th IMA International Conference*, 2011.

[121] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *USENIX Annual Technical Conference*, 2012.

[122] D. Wong. Noise extension: Disco, 2018.

[123] T. Ylonen. SSH–secure login connections over the internet. In *USENIX Security Symposium*, 1996.

[124] T. Ylonen. Ssh key management challenges and requirements. In *IFIP Conf. on New Technologies, Mobility and Security*, 2019.

[125] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. IETF RFC 4254, 2006.

[126] Yubico. Securing SSH with FIDO2. https://developers.yubico.com/SSH/Securing_SSH_with_FIDO2.html.

[127] A. Zohaib, Q. Zao, J. Sippe, A. Alaraj, A. Houmansadr, Z. Durumeric, and E. Wustrow. Exposing and circumventing SNI-based QUIC censorship of the great firewall of china. In *USENIX Security Symposium*, 2025.

A Appendix

A.1 Discoverable Mode

While Noise XX enables strong handshake security properties, it does not prove the validity of the public key and is not post-quantum resistant. Over a UDP-based protocol, sending a certificate chain does not prevent denial of service or amplification attacks from illegitimate clients. Consider a malicious client that sends spoofed Client Hello messages, which

prompts the server to respond with a larger payload that includes certificates and maintain “half-open” cryptographic state for each connection attempt. To prevent both types of attacks and implement post-quantum forward secrecy, we modify Noise XX to include an additional round-trip that enables the server to forget all handshake state until the client has acknowledged the server’s ML-KEM 512 ephemeral key and proven that it is maintaining state for the connection, and embeds KEM ephemeral key exchange for post-quantum forward secrecy. The XX pattern in the PQNoise extension is not ideal as it extends the handshake negotiation by one more message. We detail the modified handshake in Figure 2. Note that reserved bytes in Hop messages are used for alignment and protocol versioning flexibility.

Client Hello. The client initiates a handshake by sending a Client Hello, which specifies the protocol version and communicates an unencrypted KEM ephemeral encapsulation key. The client’s duplex object squeezes out a MAC for the Client Hello, keyed on the message header and ephemeral key. This MAC provides integrity only for the Client Hello message, as it is not keyed using any private data.

Type (1 byte)	Version (1 byte)	Reserved (2 bytes)
Client ML-KEM Ephemeral $ekem_c$ (800 bytes)		
MAC (16 bytes)		

The client keeps state for its own view of the handshake transcript using a duplex object that it maintains throughout the handshake duration. The client absorbs the header and ephemeral encapsulation key into the duplex object, and squeezes out a MAC keyed using the absorbed duplex material. This MAC only provides integrity for the Client Hello, as it is not keyed using any private data.

When the server receives a Client Hello, it instantiates a new duplex object that represents the server’s view of the handshake thus far. Mirroring the client, the server absorbs the message header and the client ephemeral encapsulation key into its own duplex object and verifies that the MAC attached to the Client Hello matches. If this verification is successful, the server can confirm that it is speaking the same protocol and its view of the handshake (up to the Client Hello) is identical to that of the client. The server then replies with a Server Hello. Otherwise, it does not respond. We note that discoverable servers respond to Client Hellos from unauthenticated clients, provided that both are speaking the same Hop protocol and the computed MAC matches.

Server Hello.

- Acknowledge the start of the handshake
- Transmit the (KEM) ciphertext, which encapsulates a shared secret key by using the client encapsulation key
- Prevent denial-of-service and amplification attacks from stateless clients

The server produces a shared secret key and an associated ciphertext as an output of applying the ML-KEM `Encaps` algorithm with the client encapsulation key randomness.

Green indicates encrypted fields.

Type (1 byte)	Reserved (3 bytes)
(KEM) Ephemeral Ciphertext ct (768 bytes)	
Cookie := $\texttt{AEAD}(K_r, shared_secret_key, \texttt{SHA3}(ekem_c, IP_c, Port_c))$ (64 bytes)	
MAC (16 bytes)	

To mitigate denial-of-service attacks, the server sends a cookie, which encrypts the newly generated shared secret key along with associated client metadata extracted from the Client Hello. The cookie is constructed as follows:

$$\texttt{cookie} := \texttt{AEAD}(K_r, shared_secret_key, \texttt{SHA3}(ekem_c, IP_c, Port_c))$$

where K_r is a key known only to the server that is rotated every 2 minutes (or up to the AEAD encryption limit). This cookie eliminates the need for the server to store half-open state about incomplete handshakes until it can verify that the client is also maintaining connection state, similar to TCP SYN Cookies [19]. It also reduces the impact of UDP amplification attacks, since the Server Hello is not significantly larger than the Client Hello. After sending the Server Hello, the server discards all handshake state associated with the connection.

Client Acknowledgment.

- Prove to the server that the client maintains the state
- Indicate the hostname the client is trying to connect to

Type (1 byte)	Reserved (3 bytes)
Client Ephemeral e_c^{pub} (32 bytes)	
Client ML-KEM Ephemeral $ekem_c$ (800 bytes)	
Cookie (64 bytes)	
Hostname := ID Type || ID Label (256 bytes)	
MAC (16 bytes)	

After verification of the Server Hello MAC, the client sends a Client Acknowledgment. The client retransmits the client encapsulation key alongside a newly generated Curve25519 ephemeral client public key (which will be used for later Diffie-Hellman (DH) calculations), echoes the cookie back to the server with the desired encrypted SNI. The plaintext SNI consists of a 1 byte type indicator for the server ID (e.g., domain name, IoT serial number) concatenated with a 255 byte server ID. The client continues with its duplex object and absorbs the Client Acknowledgment header (message type and reserved bytes), client ephemeral public key, ephemeral encapsulation key, the decapsulated shared secret, and the cookie received from the Server Hello. A MAC is then squeezed from the client’s duplex object and appended to the Client Acknowledgment.

At this point, the server has no state associated with the ongoing handshake. After receiving the Client Acknowledgment, the server decrypts the cookie using the SHA-3 hash of the provided client ephemeral encapsulation key, IP address, and port as the associated data. Successful decryption ensures that the cookie echoed back was originally encrypted by the server, and indicates that the client had previously sent a Client Hello with the same client ephemeral encapsulation

key. The server retrieves its shared secret key associated with the session from the decrypted cookie. Now, the server has all the data required to re-simulate the duplex. The server instantiates a new duplex object, absorbs all prior handshake data according to protocol (including the fields in the Client Acknowledgment message), and squeezes out a MAC to verify the Client Acknowledgment MAC. Successful verification proves that the owner of the encapsulation key sent both the Client Hello and the Client Acknowledgment messages.

Server Authentication.

- Prove the server's identity
- Transmit the server static (encrypted)
- Transmit the connection identifier

Type (1 byte)	Reserved (1 byte)	Certificates Length (2 bytes)	
Connection ID (4 bytes)			
Server Ephemeral e_s^{pub} (32 bytes)			
Leaf Certificate (* bytes)		Intermediate Certificate (* bytes)	
Server Certificate Authentication Tag (16 bytes)			
MAC (16 bytes)			

The server authentication message authenticates the server to the client and verifies that the identifier in the Client Acknowledgment controls the server's static key. Hop uses PKI-based authentication, but avoids the complexity of X.509 by using its own minimal certificate format (§5.6). Certificates are encrypted using the server's duplex state, and an authentication tag is squeezed and appended to the message payload for client verification. The server also chooses a random 4 byte string as the Connection ID, which is used to identify the Hop connection for post-handshake transport messages. Like QUIC [78], the Connection ID allows Hop to identify connections independent of the IP/port 5-tuple, allowing connections to roam. The server generates and sends its ephemeral key to perform DH exchanges.

Using the state of the duplex, which should be identical to that of the client's, the server decrypts the SNI. It searches for the leaf certificate and intermediate certificate to authenticate the server ID. Our PKI only uses a single intermediate certificate to ensure that the Server Authentication message fits into a single packet. At this stage, both parties begin keeping session state, but are not yet authenticated to each other. Continuing from the duplex prior, the server absorbs the message header (message type, reserved bytes, and size of the encrypted certificates), and the Session ID, which is a random 4 byte opaque string chosen by the server.

The server performs a DH calculation between the client ephemeral and the server ephemeral key pair ($\mathtt{DH}(e_s^{priv}, e_c^{pub})$), and absorbs the server's ephemeral public key and the DH result. The server's static public key is delivered as part of the certificate. It encrypts the leaf and intermediate certificates using the current duplex state and squeezes out the certificate authentication tag. As with the ephemeral key, the server now performs a DH calculation between the client ephemeral and the server static key pair ($\mathtt{DH}(s_s^{priv}, e_c^{pub})$), and absorbs the DH result. The MAC is squeezed out and appended to the Server Authentication.

Upon receiving the Server Authentication message, the client absorbs the message header, Session ID, and the ephemeral server public key. The client calculates and absorbs the DH result between the client ephemeral and the server ephemeral and then decrypts the certificates using the duplex material, and checks that its computed certificate authentication tag matches. The client then verifies the certificate to prove that the sender of the Server Authentication message owns the server static private key (s_s^{priv}) (the server is who it says it is) and that it is the actual owner of the Server ID. Finally, the client calculates and absorbs the DH result between the client ephemeral and the server static (obtained from the certificate), and verifies that the handshake transcript is identical via the squeezed MAC.

Client Authentication.

- Prove the client's identity
- Transmit the client static (encrypted)

Type (1 byte)	Reserved (1 byte)	Certificates Length (2 bytes)	
Session ID (4 bytes)			
Leaf Certificate (* bytes)		Intermediate Certificate (* bytes)	
Client Certificate Authentication Tag (16 bytes)			
MAC (16 bytes)			

The Client Authentication step is symmetric to that of Server Authentication. At this point, both parties are authenticated, have an identical view of the handshake transcript, and have a shared secret derived from KEM ephemeral and static keys.

The client obtains a signed certificate from their identity provider, and absorbs the Client Authentication message header and the Session ID. Using the current duplex state, it encrypts the leaf and intermediate certificates and squeezes out the client certificate authentication tag. The client then absorbs the DH between the client static and the server ephemeral ($\mathtt{DH}(s_c^{priv}, e_s^{pub})$), squeezes out the Client Authentication MAC, and sends the Client Authentication message to the server.

When the server receives the Client Authentication Message, it verifies that it has an in-progress handshake and absorbs the message header. The server confirms that the Session ID matches that of the existing handshake state, and absorbs the Session ID. Using the duplex object, it decrypts the client certificate and verifies using the duplex state that the client certificate authentication tag is correct and validates the certificate. Finally, the server absorbs the DH between the client static and server ephemeral, and verifies that the squeezed MAC matches. This confirms that the sender of the Client Authentication message owns the client static private key (s_c^{priv}) (the client is who it says it is). It also confirms that both parties have identical views of the handshake. The Hop discoverable handshake completes after the client authentication message is verified by the server.

A.2 Hidden Mode

Client Request.

- Indicate the start of the Hop Hidden handshake
- Transmit the client KEM ephemeral, static ciphertext
- Prove the client's identity

Type (1 byte)	Version (1 byte)	Certificates Length (2 bytes)
Client ML-KEM Ephemeral $ekem_c$ (800 bytes)		
(KEM) Static Ciphertext ct (768 bytes)		
Leaf Certificate (* bytes)		Intermediate Certs. (* bytes)
Client Static Authentication Tag (16 bytes)		
Timestamp (8 bytes)		
MAC (16 bytes)		

The client initiates a new duplex object that it maintains throughout the handshake duration. It first absorbs the header, including the type, the version, the certificate length, and the newly generated ML-KEM 512 ephemeral encapsulation key. It generates a shared secret and encapsulates it with the previously received server's static KEM key. Upon the absorption of the post-quantum safe shared secret key, the client can encrypt the Hop client certificates with the duplex object. The client squeezes the client certificate authentication tag and includes in the client request an encrypted timestamp to prevent replay attacks, both absorbed. The client now squeezes out the MAC for the integrity and appends it to the message and sends it to the server. In Hop Hidden, the server authenticates the client on the first request (removing the UDP message size constraint) and does not respond to invalid clients remaining "hidden". This cryptographic construction does not allow the SNI to be in the Client Request.

Server Response.

- Complete negotiation of transport keys
- Transmit the Ephemeral Ciphertext
- Prove the server's identity
- Transmit the connection identifier

Type (1 byte)	Reserved (1 byte)	Certificates Length (2 bytes)
Connection ID (4 bytes)		
(KEM) Ephemeral Ciphertext ct (768 bytes)		
Leaf Certificate (* bytes)		Intermediate Certs. (* bytes)
Certificate Authentication Tag (16 bytes)		
MAC (16 bytes)		

If the client is authenticated and authorized to access the server, the server issues a Server Response. It contains the connection ID, the ephemeral ciphertext from the client's ephemeral KEM key, and the server certificates. As proof of identity, the server performs the DH calculation between the client's public static key and its private static key. It absorbs all the aforementioned information and squeezes the server certificate authentication tag and the MAC. The server sends the Server Response to the client, who decrypts and verifies the certificates before deriving the final transport keys used for post-handshake messages (§5.5).

A.3 Key Derivation and Message Structures

```
1 duplex.ratchet()
2 duplex.absorb("client_to_server_key")
3 client_to_server_key = duplex.squeeze()
4 duplex.ratchet()
5 duplex.absorb("server_to_client_key")
6 server_to_client_key = duplex.squeeze()
```

Subsequent transport data (Figure 10) is encrypted using these shared session keys, which rotate every 2^{64} messages.

Type (1 byte)	Reserved (3 bytes)
Session ID (4 bytes)	
Counter (8 bytes)	
Encrypted Data (* bytes)	
MAC (16 bytes)	

Figure 10: **Hop Transport Message**—The payload is in the Encrypted Data segment (green) of the transport message.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Tube ID	REQ	RESP	REL	ACC	FIN	RTR	Res.	Data Length
Acknowledgment Number								
Frame Number								
Data (* bytes)								

Figure 11: **Hop Tube Frame**—Within the Encrypted Data segment of the transport message (see Figure 10).

Type (1 byte)	Target Port Number (2 bytes)	
Start Time (8 bytes)		Expiration Time (8 bytes)
Target Username (32 bytes)		
Target SNI (<=256 bytes)		
Delegate Client Certificate (<=660 bytes)		
Associated Data (* bytes)		

Figure 12: **Intent**—The Intent is authenticated and encrypted. The `Data` in *Intent Confirmation* messages is empty, and *Intent Denied* messages can optionally indicate the reason the *Intent Request* or *Intent Communication* was denied.

Field	Size (B)	Description
Protocol Version	1	Version number indicator
Cert. Type	1	Leaf (0x1), Int. (0x2), Root (0x3)
Reserved	2	0x0
IssuedAt	8	Timestamp (not valid before)
ExpiresAt	8	Timestamp (not valid after)
Public Static Key	32	Leaf: x25519 (for key exchange) Int./Root: ed25519 (for signature)
Parent Fingerprint	32	SHA3(parent cert, no signature); Root: 0x0
ID Chunk Size	2	Number of bytes (4–512 bytes, aligned)
ID Chunk	4–512	Array of ID Blocks
ID Block	4–256	Identifier block
Size	1	Total ID block length (label + 3 bytes)
Type	1	DNSName, IPv4Addr, IPv6Addr, Raw
Label Size	1	Number of ID label bytes
Label	1–253	UTF-8 label
Parent Signature	64	ed25519(parent (or self for root) static key, SHA3(cert. bytes excluding signature))

Table 1: **Certificate Fields**

USENIX Security '26 Artifact Appendix: Hop: A Modern Transport and Remote Access Protocol

Paul Flammarion
Stanford University

George Hosono*
Georgia Tech

Wilson Nguyen
Stanford University

Laura Bauman
Stanford University

Daniel Rebelsky
Stanford University

Gerry Wan
Stanford University

David Adrian
Independent

Zakir Durumeric
Stanford University

A Artifact Appendix

A.1 Abstract

We provide artifacts supporting our claims on the functionality and performance of our Hop reference implementation. Our artifact includes the full client and server source code, scripts for controlled simulation experiments, and measurement scripts used to evaluate session establishment latency, file transfer throughput, and keystroke latency. The artifact also includes the datasets and plotting scripts to reproduce all performance figures presented in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our software requires the user to install external dependencies. Our remote communication protocol can be used in a local network as well as on the public Internet. The use of hidden mode should be preferred as a means of connection for Internet-exposed Hop servers. Do not use Hop to grant access to anyone you do not trust. The code does not execute any destructive steps, and there are no ethical concerns or risks for the evaluators.

A.2.2 How to access

The artifact is hosted at Zenodo and can be accessed at: `https://doi.org/10.5281/zenodo.17953396`

A.2.3 Hardware dependencies

To ensure correct building and execution of our experiments, we recommend using at least two machines (physical or virtual), each with a minimum of 2 vCPUs, 4 GB of RAM, and 8 GB of storage.

*Work conducted primarily at Stanford University.

A.2.4 Software dependencies

To run all the experiments, you will need a Linux-based machine. We suggest using Ubuntu 24.04 LTS with the following dependencies:

- `Golang v1.23`, for running and building Hop. Installation available at `https://go.dev`. Example installation: `apt install golang`
- `Docker v28`, for building test environments. Installation available at `https://docs.docker.com`
- `Python3`, for evaluation only. Installation available at `https://www.python.org`. Example installation: `apt install python3`
- `OpenSSH v9.7p1`, for evaluation baseline. Installation available at `https://www.openssh.org`. Example installation: `apt install openssh-client openssh-server`
- `Rsync v3.3`, for file transfer. Installation available at `https://github.com/RsyncProject/rsync`. Example installation: `apt install rsync`
- `GNU Make v3.81`, for script automation. Installation available at `https://www.gnu.org/software/make`. Example installation: `apt install make`
- `Mininet v2.3.0`, for the evaluation in simulation. Installation available at `https://mininet.org`. Example installation: `apt install mininet`
- `Java 11`, to run Typometer software only used for keystroke timing evaluation. Installation available at `https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html`. Example installation: `apt install openjdk-11-jre`

To ensure proper execution of Hop, we recommend using `Go 1.23` for the project, as an issue with Hop's use of `creack/pty` may prevent a shell from opening when running on newer versions of Go.

A.2.5 Benchmarks

Our artifact includes the dataset of our SSH evaluation, serving as a benchmark for Hop evaluation under the same environment. However, since the evaluator setup will likely be different (e.g., different machines, different Internet infrastructure, different times, etc.), the artifact also provides a guide to collect this data from the evaluator's setup.

A.3 Set-up

Please be sure that the machines used for evaluation have installed all software dependencies and granted permission to listen on the port or socket specified in Hop configuration (e.g., port `77` or `7777`).

When generating credentials, make sure to specify the IP address of the remote machine and to report it correctly in the client configuration file. If DNS resolution is not available for these IP addresses, use the IP address itself as both the DNS name and the server name.

Since our experiments use SSH measurements as a baseline, you may set up a working SSH communication channel between the clients and a remote SSH server for the time to shell (E1) and the file transfer in real-world (E3), and a local SSH server for file transfer in simulation (E2) and the keystroke latency (E4) experiments.

A.3.1 Installation

1. Download the Hop artifact from `https://doi.org/10.5281/zenodo.17953396` and unarchive the file if required.
2. Go to Hop's directory `cd hop-go/`
3. Fetch dependencies `go get ./...; go get -t ./...`
4. Create the default credentials to locally connect to the Docker container configuration `make cred-gen`
5. Build and start the Docker container `make serve-dev`

A.3.2 Basic Test

Be sure you have a Hop server up-and-running in the previously created Docker container.

1. Connect to the Docker container `go run hop.computer/hop/cmd/hop -C containers/client_config.toml user@127.0.0.1:7777`
2. Success is having a shell as `user` in the Docker container `example.com`.

While this strictly describes a basic Hop shell access, more configurations can be found in the artifact `README` files.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Time to shell:* Hop starts a non-interactive shell and exits with fewer network round-trips than SSH due to its handshake efficiency. This is proven by the experiment (E1) described in Section 8.2, whose results are reported in Figure 6.

(C2): *File Transfer Speed:* Hop claims having a functional congestion control in both simulation and real-world experiments. This is proven by the experiment (E2) and (E3) described in Section 8.2, whose results are reported in Figure 7 and Figure 8.

(C3): *Keystroke latency:* Hop shows a faster keystroke response time through an interactive shell than SSH. This is proven with the experiment (E4), which is described in Section 8.2 and shown in Figure 9.

A.4.2 Experiments

Each experiment requires a running Hop instance, and setting up a fully functional environment takes approximately 15 human-minutes per machine. As this is a one-time setup shared across all experiments, we do not include these initial 15 minutes in the reported time for each experiment below.

- Generate client and server credentials as specified in `hop-go/CONFIGURATION.md` or reuse the ones created for the Docker basic test. For (E2), create experiment-specific credentials with `hop-go/artifact/simulation/config/cred-gen-simulation.sh` script or manually.
- Start a Hop Hidden Docker container with `make serve-dev-hidden` or configure it manually to evaluate Hop Hidden mode.
- Generate file transfer files with random data by running `dd if=/dev/urandom of=100MB_file bs=1M count=100`. Reproduce for `10MB` and `1GB` files.

Most of the paths within the measurement scripts are relative to the root of the repository. You can adapt the paths to your environment. Each experiment's dataset can be found in the `hop-go/artifact` folder.

(E1): *[Time to shell] [15 human-minutes + 20 computer-minutes]:* Measures the time to perform a full handshake and a session setup with a remote machine for each protocol.
How to: Run the Python script to automatically measure connections and collect the session connection data.
Preparation: Have at least one server to which you can connect with the desired protocol (Hop Discoverable, Hop Hidden, and/or SSH). Update the config-

uration at the top of the `tts_measure.py` file in the `artifact/time-to-shell` folder.

Execution: Run the `tts_measure.py` script, which will output a `CSV` file. By default, the script runs 10 experiments. In the paper, we ran 100 measures for each of the 3 protocols on each machine. This value can also be updated at the top of the file.

Results: Visualize the results with any desired software or configure and run the `tts_plot.py` script.

(E2): *[File transfer in simulation] [15 human-minutes + 15 computer-minutes]:* This evaluates file transfer performance under controlled network conditions using Mininet. It isolates the effects of bandwidth, latency, jitter, and loss on end-to-end transfer time for Hop and SSH.

How to: Run the provided Mininet-based experiment script, which automatically sets up the network topology, performs file transfers, and records results.

Preparation: Ensure Mininet is installed and can be executed with elevated privileges, as well as Golang or Hops builds. This might require editing the sudoers file. Be sure that you have all the credentials for both Hop and SSH correctly located according to their respective configurations, including the `.hop/authorized_keys` file on Hop's server side. Experiment parameters such as queue size, bandwidth, delay, and file sizes can be adjusted at the top of the simulation script.

Execution: Execute the simulation script to initialize the Mininet topology, run the file transfer experiments for each configured network condition, and store the results in a `CSV` file.

Results: The output can be visualized using the `simulation_plot.py` script.

(E3): *[File transfer in real-world] [15 human-minutes + 2 computer-hour]:* Measures end-to-end file transfer time to a remote host in a real-world deployment. Transfers are performed for multiple file sizes (10 MB, 100 MB, and 1 GB).

How to: Run the measurement script, which performs repeated file transfers using `rsync` over SSH and Hop and records the elapsed time.

Preparation: Generate the files that you want to transfer during the experiment. Ensure that the target host is reachable and running the required SSH and/or Hop services. Configure host addresses, users, data files, and Hop configuration file paths at the top of `transfer_measure.py`.

Execution: Execute `transfer_measure.py` and monitor the file transfers to ensure their completion. On a separate execution, the TCP congestion control algorithm can be changed (e.g., Cubic to NewReno). You can change the congestion control of the machine to NewReno with `sysctl net.ipv4.tcp_congestion_control=reno`

Results: Results can be visualized using the `transfer_plot.py` script.

(E4): *[Keystroke latency] [30 human-minutes + 1 computer-hour]:* Evaluates interactive latency by measuring keystroke-to-echo delay over SSH and Hop sessions.

How to: Establish an interactive terminal session with the target host using either SSH or Hop, then use the Typometer third-party software to record keystroke latency for the active terminal window.

Preparation: Ensure that an interactive SSH or Hop session can be established with the target host. The Typometer software and its license are provided in the `keystroke-latency` directory. Follow the included Typometer 'README' to install and launch Typometer. Ensure that you are using Java 11 to start Typometer. Keep the default Typometer settings.

Execution: Start Typometer and select the active terminal window. Typometer will write a keystroke and automatically measure the time to its display. Repeat the measurement for each protocol and host under identical conditions. Export the recorded latency data as a `CSV` file.

Results: Results can be visualized using the `keystrokes_plot.py` script.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.