

HORNET: High-speed Onion Routing at the Network Layer

Chen Chen
chenche1@cmu.edu
CMU/ETH Zürich

Daniele Enrico Asoni
dasoni@student.ethz.ch
ETH Zürich

David Barrera
david.barrera@inf.ethz.ch
ETH Zürich

George Danezis
g.danezis@ucl.ac.uk
UCL

Adrian Perrig
adrian.perrig@inf.ethz.ch
ETH Zürich

ABSTRACT

We present HORNET, a system that enables high-speed end-to-end anonymous channels by leveraging next generation network architectures. HORNET is designed as a low-latency onion routing system that operates at the network layer thus enabling a wide range of applications. Our system uses only symmetric cryptography for data forwarding yet requires no per-flow state on intermediate nodes. This design enables HORNET nodes to process anonymous traffic at over 93 Gb/s. HORNET can also scale as required, adding minimal processing overhead per additional anonymous channel. We discuss design and implementation details, as well as a performance and security evaluation.

1. INTRODUCTION

Recent revelations about global-scale pervasive surveillance programs have demonstrated that the privacy of Internet users worldwide is at risk. These revelations suggest massive amounts of private data, including web browsing activities, location information, and personal communications are being harvested in bulk by domestic and foreign intelligence agencies. The surveillance-prone design of the Internet accompanied by the decreasing cost of data storage have enabled mass-surveillance, through indiscriminate data collection and storage [12, 8].

To protect against these and other surveillance threats, several anonymity protocols, tools, and architectures have been proposed. Among the most secure schemes for anonymous communications are mix networks [28, 36, 20, 21], which are useful for cases where high-latency asynchronous messaging can be tolerated. Onion routing networks (most notably Tor [23]), offer a balance between security and performance, enabling low-latency anonymous communication suitable for typical Internet activities (e.g., web browsing, instant messaging, etc.). Tor is the system of choice for over 2 million daily users [13], but its design as an overlay network suffers from performance and scalability issues: as more clients use Tor, more relays must be added to the network. Additionally, Tor's design requires per-connection state to be maintained by intermediate nodes, limiting the total number of concurrent anonymous connections that can take place simultaneously.

The scalability and performance limitations of anonymous networks have been partially addressed by building protocols into the network layer rather than implementing them as overlays. Among these high-performing schemes are LAP [29] and Dovetail [42], which offer network-level low-latency anonymous communication on next-generation network architectures. The high performance of both schemes, however, results in significantly degraded security guarantees; endpoints can be de-anonymized if the adversary

has global data collection abilities, and payload protection relies on upper layer protocols which increases complexity.

In this paper, we present HORNET (High-speed Onion Routing at the Network layer), a highly-scalable anonymity system that leverages next-generation Internet architecture design. HORNET offers payload protection by default, and can defend against some global observation attacks. HORNET is designed to be highly efficient: instead of keeping state at each relay, connection state (including, e.g., onion layer decryption keys) is carried within packet headers, allowing intermediate nodes to quickly forward traffic for large numbers of clients.

While this paper proposes and evaluates a concrete anonymity system, a secondary goal herein is to broadly re-think the design of low-latency anonymity systems by envisioning networks where anonymous communication is offered as an in-network service to all users. For example, what performance trade-offs exist between keeping anonymous connection state at relays and carrying state in packets? If routers perform anonymity-specific tasks, how can we ensure that these operations do not impact the processing of regular network traffic, including in adversarial circumstances? And if the network architecture should provide some support for anonymous communication, what should that support be? Throughout the paper we consider these issues in the design of our own system, and provide intuition for the requirements of other network-level anonymity systems.

Specifically, our contributions are the following:

- We design and implement HORNET, an anonymity system that uses source-selected paths and shared keys between endpoints and routers to support onion routing. Unlike other onion routing implementations, HORNET routers do not keep per-flow state or perform computationally expensive operations for data forwarding, allowing the system to scale as new clients are added.
- We analyze the security of our system, showing that it can defend against passive attacks, and certain types of active attacks. Our system provides stronger security guarantees than existing network-level anonymity systems.
- We evaluate the performance of our system, showing that anonymous data processing speed is comparable to that of LAP and Dovetail (up to 93.5 Gb/s on a 120 Gb/s software router). Each HORNET node can process traffic for a practically unlimited number of sources.

2. PROBLEM DEFINITION

We aim to design a network level anonymity system to frustrate adversaries with mass surveillance capabilities. Specifically, an adversary observing traffic traversing our system should be unable to link (at large scale) pairs of hosts communicating over the network.

This property is known as end-to-end unlinkability [39].

We define *sender anonymity* as a communication scenario where unlinkability is guaranteed for the source, but the destination’s location is public (e.g., web sites for The Guardian or Der Spiegel). We define *sender-receiver anonymity* as a scenario where the unlinkability guarantee is extended to the destination (e.g., a hidden service that wishes to conceal its location). Sender-receiver anonymity therefore offers protection for both ends, implying sender anonymity. Depending on users’ needs, HORNET can support either sender anonymity or sender-receiver anonymity.

Since our scheme operates at the network layer, network location is the only identity feature we aim to conceal. Exposure of network location or user identity at upper layers (e.g., through TCP sessions, login credentials, or browser cookies) is out of scope for this work.

2.1 Network Model

We consider that provisioning anonymous communication between end users is a principal task of the network infrastructure. The network’s anonymity-related infrastructures, primarily routers, assist end users in establishing temporary *anonymous sessions* for anonymous data transmission.

We assume that the network layer is operated by a set of nodes (e.g., ASes). Each node cooperates with sources to establish anonymous sessions to the intended destinations, and processes anonymous traffic within the created sessions. We require that routing state allows each node to determine only the next hop. In particular, the destination is only revealed to the last node and no others. This property can be satisfied by IP Segment Routing [10], Future Internet Architectures (FIAs) like NIRA [46] and SCION [48], or Pathlets [26].

We assume the underlying network architecture provides a mechanism for a source to obtain a path to a destination. This path is the combination of routing state of all nodes between the source and the intended destination. Additionally, we assume that the same mechanism allows the source to fetch the public keys and certificates¹ of on-path nodes. The source retrieves the above information anonymously using one of two methods: 1) using unprotected queries to retrieve the necessary information to reach a public path lookup server and then creating an anonymous session to the server; or 2) using any form of Private Information Retrieval (PIR), as was recently proposed for Tor [35]. In Section 7.1, we briefly sketch how to obtain this information anonymously in selected FIAs. While a general solution represents an important avenue for future work, it remains outside of our present scope.

We assume that end hosts and on-path nodes have public keys accessible and verifiable by all entities. End hosts can retrieve the public keys of other end hosts through an out-of-band channel (e.g., websites) and verify them following a scheme like HIP [37], in which the end hosts can publish hashes of their public keys as their service names. Public keys of on-path nodes are managed through a public-key infrastructure (PKI). For example, the source node can leverage Resource Public Key Infrastructure (RPKI) [16] to verify the public keys of on-path nodes.

2.2 Threat Model

We consider an adversary attempting to conduct mass surveillance. Specifically, the adversary collects and maintains a list of “selectors” (e.g., targets’ network locations, or higher-level protocol identifiers), which help the adversary trawl intercepted traffic and extract parts of it for more expensive targeted analysis [8]. An

¹Depending on the underlying PKI scheme, the source might need to fetch a chain of certificates leading to a trusted anchor to verify each node’s public key.

anonymity system should provide no way for such an adversary to leverage bulk access to communications to select traffic that belongs to the targets. Thus an adversary will have to collect and analyze all traffic and cannot reliably select traffic specific to targets unless it has access to the physical links next to the targets.

We consider an adversary that is able to compromise a fraction of nodes on the path between source and destination. For sender anonymity, the adversary can also compromise the destination. For sender-receiver anonymity, the adversary can compromise at most one of the two end hosts.

By compromising a node, the adversary learns all keys and settings, observes all traffic that traverses the compromised node, and is able to control how the nodes behave including redirecting traffic, fabricating, replaying, and modifying packets. However, like other low-latency schemes, we do not solve confirmation attacks based on the analysis of flow dynamics [43, 31, 38] and active packet tagging [40]. Resisting such attacks using dynamic link padding [45] is no more difficult than in onion routing, although equally expensive.

2.3 Desired Properties

HORNET is designed to achieve the following anonymity and security properties:

1. **Path information integrity and secrecy.** An adversary should not be able to modify a packet header to change path without detection. The adversary should not learn forwarding information of uncompromised nodes, node’s positions, or the total number of hops on a path.
2. **No cross-link identification.** An adversary that can eavesdrop on multiple links in the network cannot correlate two or more packets on those links by observing the bit patterns in the packet headers or payloads.
3. **Session unlinkability.** An adversary cannot link packets from different sessions, even between the same set of sources and destinations.
4. **Payload secrecy and end-to-end integrity.** Without compromising end hosts, an adversary cannot learn any information from the data payload except for its length and timing among sequences of packets.

3. HORNET OVERVIEW

The basic design objectives for HORNET are *scalability* and *efficiency*. To enable Internet-scale anonymous communication, HORNET intermediate nodes must avoid keeping per-session state (e.g., cryptographic keys and routing information). Instead, session state is offloaded to end hosts, who then embed this state into packets such that each intermediate node can extract its own state as part of the packet forwarding process.

Offloading the per-session state presents two challenges. First, nodes need to prevent their offloaded state from leaking information (e.g., the session’s cryptographic keys). To address this, each HORNET node maintains a local secret to encrypt the exported per-session state. We call this encrypted state a *Forwarding Segment* (FS). The FS allows its creating node to dynamically retrieve the embedded information (i.e., next hop, shared key, session expiration time), while hiding this information from unauthorized third parties.

The second challenge in offloading the per-session state is to combine this state (i.e., the FSes) in a packet in such a way that each node is able to retrieve its own FS, but no information is leaked about the network location of the end hosts, the path length, or a specific node’s position on the path. Learning any of this information could assist in de-anonymization attacks (see Section 5.5). To

address this challenge, the source constructs an *anonymous header* (AHDR) by combining multiple FSes, and prepends this header to each packet in the session. An AHDR grants each node on the path access to the FS it created, without divulging any information about the path except for a node’s previous and next nodes (see Section 4.4.1).

For efficient packet processing, each HORNET node performs one Diffie-Hellman (DH) key exchange operation once per session during setup. For all data packets within the session, HORNET nodes use only symmetric cryptography to retrieve their state, process the AHDR and onion-decrypt (or encrypt) the payload. To reduce setup delay, HORNET uses only two setup packets within a single round trip between the source and the destination. Therefore, session setup only incurs $O(n)$ propagation delay in comparison to $O(n^2)$ by the iterative method used in Tor (where n is the number of anonymity nodes traversed on the path). While for Tor the default value of n is 3, for HORNET n might be as large as 14 (and 4.1 in the average case [7]), which emphasizes the need to optimize setup propagation delay in our protocol.

3.1 Sender Anonymity

Anonymous sessions between a source and destination require the source to establish state between itself and every node on the path. The state will be carried in subsequent data packets, enabling intermediate nodes to retrieve their corresponding state and forward the packet to the next hop. We now describe how the state is collected without compromising the sender’s anonymity, and how this state is used to forward data packets.

Setup Phase. To establish an anonymous session between a source S and a public destination D , S uses a single round of Sphinx [21], a provably secure onion routing protocol (an overview of Sphinx is given in Section 4.3.1). This round consists of two Sphinx packets (one for the forward path and one for the backward path) each of which will anonymously establish shared symmetric keys between S and every node on that path. For HORNET, we extend the Sphinx protocol to additionally anonymously collect the forwarding segments (FSes) for each node. Our modified Sphinx protocol protects the secrecy and integrity of these FSes, and does not reveal topology information to any node on the path. We note that using Sphinx alone for data forwarding would result in low throughput due to prohibitively expensive per-hop asymmetric cryptographic operations. Therefore, we use Sphinx only for session setup packets, which are amortized over the subsequent data transmission packets. We explain the details of the setup phase in Section 4.3.

Data Transmission Phase. Having collected the FSes, the source is now able to construct a forward AHDR and a backward AHDR for the forward and backward paths, respectively. AHDRs carry the FSes which contain all state necessary for nodes to process and forward packets to the next hop. When sending a data packet, the source onion-encrypts the data payload using the session’s shared symmetric keys, and prepends the AHDR. Each node then retrieves its FS from the AHDR, onion-decrypts the packet and forwards it to the next hop, until it reaches the destination. The destination uses the backward AHDR (received in the first data packet²) to send data back to S , with the only difference being that the payload is encrypted (rather than decrypted) at each hop. We present the details of the data transmission phase in Section 4.4.

3.2 Sender-Receiver Anonymity

Sender-receiver anonymity, where neither S nor D know each other’s location (e.g., a hidden service), presents a new challenge:

²If the first packet is lost the source can simply resends the backward AHDR using a new data packet (see Section 4.4).

since S does not know D ’s location (and vice versa), S cannot retrieve a path to D , precluding the establishment of state between S and nodes on the path to D as described in Section 3.1.

A common approach to this problem (as used by Tor, LAP, and Dovetail) is for the destination to advertise a path (or similar) back to itself through a known, public *rendezvous point* (RP). Sources establish anonymous sessions to the RP, who in turn forwards traffic to the destination while keeping S and D ’s location hidden from each other. This solution would also work for HORNET. However, it requires the RP to maintain per-session state between sources and destinations, which increases complexity, bounds the number of receivers, and introduces a state exhaustion denial of service attack vector.

Nested a-headers. Our proposal for sender-receiver anonymity requires no state to be kept at the RP by nesting the AHDRs from the source to the rendezvous and from the rendezvous to the destination. Briefly, to establish a HORNET session between S and D keeping both parties hidden from each other, D selects a public rendezvous point R and completes a HORNET session setup between D and R . D publishes $AHDR_{R \rightarrow D}$ to a public directory. Note that this AHDR leaks no information about D ’s location and can only be used to send data to D through R within a specific time window.

When S wants to send traffic to D , S retrieves (from a public directory) $AHDR_{R \rightarrow D}$. S then establishes a HORNET session between S and R and constructs a nested AHDR with $AHDR_{R \rightarrow D}$ inside $AHDR_{S \rightarrow R}$. S includes $AHDR_{R \rightarrow S}$ in the data payload to D , allowing D to create a return path to S .

One of the advantages of our scheme is that any node on the network can serve as a rendezvous point. In fact, multiple points can be selected and advertised, allowing the source to pick the RP closest to it. Moreover, once a HORNET session has been established, S and D can negotiate a better (closer) RP (e.g., using private set intersection [25]). A disadvantage of the nested AHDR technique is that it doubles the size of the header.

For space reasons, the formal protocol details and evaluation sections focus on sender anonymity only. Details of sender-receiver anonymity can be found in the full paper [5].

3.3 Packet Structure

HORNET uses two types of packets: *setup packets* and *data packets* (see Figure 1). Both types of packets begin with a common header which describes the packet type, the length of the longest path that the session supports, and a type-specific field. For session setup packets, the type-specific field contains a value *EXP* which indicates the intended expiration time of the session. For data packets, the specific value is a random nonce generated by the sender used by intermediate nodes to process the data packet.

Session setup packets include a nested Sphinx packet and an FS payload. Data packets carry an AHDR and an onion-encrypted data payload. We explain each field in detail in Section 4.

4. FORMAL PROTOCOL DESCRIPTION

We now describe the details of our protocol, focusing on sender anonymity. We first list the information required by the source to start an anonymous communication (Section 4.2), and then present the specification of the setup phase (Section 4.3) and of the data transmission phase (Section 4.4).

4.1 Notation

Let k be the security parameter used in the protocol. For evaluation purposes we consider $k = 128$. \mathcal{G} is a prime order cyclic group of order q ($q \sim 2^{2k}$), which satisfies the Decisional Diffie-Hellman Assumption. \mathcal{G}^* is the set of non-identity elements in \mathcal{G} and g is a

HORNET Session Setup Packet

type	hops	EXP
Sphinx Header		
Sphinx Payload		
FS Payload		

HORNET Data Packet

type	hops	nonce
AHDR		
Data Payload		

Figure 1: HORNET packet formats.

generator of \mathcal{G} .

Let r be the maximum length of a path, i.e., the maximum number of nodes on a path, including the destination. We denote the length of an FS as l_{FS} and the size of an AHDR block, containing an FS and a MAC of size k , as $c = l_{FS} + k$.

HORNET uses the following cryptographic primitives:

- MAC : $\{0, 1\}^k \times \{0, 1\}^* \rightarrow \{0, 1\}^k$: Message Authentication Code (MAC) function.
- PRG0, PRG1 : $\{0, 1\}^k \rightarrow \{0, 1\}^{r(c+k)}$; PRG2 : $\{0, 1\}^k \rightarrow \{0, 1\}^{rc}$: Three cryptographic pseudo-random generators.
- PRP : $\{0, 1\}^k \times \{0, 1\}^a \rightarrow \{0, 1\}^a$: A pseudo-random permutation, implementable as a block cipher. The value of a will be clear from the context.
- ENC : $\{0, 1\}^k \times \{0, 1\}^k \times \{0, 1\}^{mk} \rightarrow \{0, 1\}^{mk}$: Encryption function, with the second parameter being the Initialization Vector (IV) (e.g., Stream cipher in counter mode). m is a variable integer parameter.
- DEC : $\{0, 1\}^k \times \{0, 1\}^k \times \{0, 1\}^{mk} \rightarrow \{0, 1\}^{mk}$: Decryption function to reverse ENC.
- h_{op} : $\mathcal{G}^* \rightarrow \{0, 1\}^k$: a family of hash functions used to key op, with $op \in \{MAC, PRG0, PRG1, PRP, ENC, DEC\}$.

We denote by $RAND(l)$ a function that generates a new uniformly random string of length l .

Furthermore, we define the notation for bit strings. 0^l stands for a string of zeros of length l . $|x|$ is the length of the bit string x . $x_{[a\dots b]}$ represents a substring of x from bit a to bit b , with sub-index a starting from 0; $x_{[a\dots end]}$ indicates the substring of x from bit a till the end. ε is the empty string. $x \parallel y$ is the concatenation of string x and string y .

In the following protocol description, we consider a source S communicating with a destination D using forward path p^f traversing $n_0^f, n_1^f, \dots, n_{l^f-1}^f$ and backward path p^b traversing $n_0^b, n_1^b, \dots, n_{l^b-1}^b$, with $l^f, l^b \leq r$, where n_0^f and $n_{l^b-1}^b$ are the nodes closest to the source. Without loss of generality, we let the last node on the forward path $n_{l^f-1}^f = D$ and refer to the destination by these two notations interchangeably. In general we use $dir \in \{f, b\}$ as superscripts to distinguish between notation referring to the forward and backward path, respectively. Finally, to avoid redundancy, we use $\{sym_i^{dir}\}$ to denote $\{sym_i^{dir} | 0 \leq i \leq l^{dir} - 1\}$, where sym can be any symbol.

4.2 Initialization

Suppose that a source S wishes to establish an anonymous session with a public destination D . First, S anonymously obtains (from the underlying network) paths in both directions: a forward path $p^f = \{R_0^f, R_1^f, \dots, R_{l^f-1}^f\}$ and a backward path $p^b = \{R_0^b, R_1^b, \dots, R_{l^b-1}^b\}$ over the plaintext of the payload that the destination sent.

R_i^{dir} denotes the routing information needed by the node n_i^{dir} to forward a packet. S also anonymously retrieves and verifies a set of public keys $g^{x_{n_i^{dir}}}$ for the node n_i^{dir} on path p^{dir} (see Section 2.1). Note that g^{x^D} is also included in the above set (as $n_{l^f-1}^f = D$). Finally, S generates a random DH key pair for the session: x_S and g^{x_S} . The per-session public key g^{x_S} is used by the source to create shared symmetric keys with nodes on the paths later in the setup phase. S locally stores $\{(x_S, g^{x_S}), \{g^{x_{n_i^{dir}}}\}, p^{dir}\}$, and uses these values for the setup phase.

4.3 Setup Phase

As discussed in Section 3, in the setup phase, HORNET uses two Sphinx packets, which we denote by $P\blacklozenge$ and $P\blacktriangle$, to traverse all nodes on both forward and backward paths and establish per-session state with every intermediate node, without revealing S 's network location. For S to collect the generated per-session state from each node, both Sphinx packets contain an empty FS payload into which each intermediate node can insert its FS, but is not able to learn anything about, or modify, previously inserted FSes.

4.3.1 Sphinx Overview

Sphinx [21] is a provably-secure onion routing protocol. Each Sphinx packet allows a source node to establish a set of symmetric keys, one for each node on the path through which packets are routed. These keys enable each node to check the header's integrity, onion-decrypt the data payload, and retrieve the information to route the packet. Processing Sphinx packets involves expensive asymmetric cryptographic operations, thus Sphinx alone is not suitable to support high speed anonymous communication.

Sphinx Packets. A Sphinx packet is composed of a Sphinx header SHDR and a Sphinx payload SP. The SHDR contains a group element y_i^{dir} that is re-randomized at each hop. Each y_i^{dir} is used as S 's ephemeral public key in a DH key exchange with node n_i^{dir} . From this DH exchange node n_i^{dir} derives a shared symmetric key $s_{n_i^{dir}}$, which it uses to process the rest of the SHDR and mutate the y_i^{dir} . The rest of the SHDR is an onion-encrypted data structure, with each layer containing the routing information to decide the next node to forward the packet and a per-hop MAC to protect the header's integrity. The Sphinx payload SP allows end hosts to send confidential content to each other. Each intermediate node processes SP by using a pseudo-random permutation.

Sphinx Core Functions. We abstract the Sphinx protocol into the following six functions:

- GEN_SPHX_HDR. The source nodes uses this function to generate two Sphinx headers, $SHDR^f$ and $SHDR^b$, for the forward and backward path, respectively. It also outputs a series of DH public-private key pairs $\{(x_i^{dir}, y_i^{dir})\}$ (ephemeral keys of the source), and the symmetric keys $\{s_{n_i^{dir}}\}$, each established with the corresponding node's public key $g^{x_{n_i^{dir}}}$.
- GEN_SPHX_PL_SEND. The function allows the source to generate an onion-encrypted payload SP^f encapsulating confidential data to send to the destination.
- UNWRAP_SPHX_PL_SEND. The function removes the last encryption layer added by GEN_SPHX_PL_SEND, and allows the destination to decrypt the SP^f .
- GEN_SPHX_PL_RECV. The function enables the destination to cryptographically wrap a data payload into SP^b before sending it to the source.
- UNWRAP_SPHX_PL_RECV. The function allows the source to

- **PROC_SPHX_PKT.** Intermediate nodes use this function to process a Sphinx packet, and establish symmetric keys shared with the source. The function takes as inputs the packet (SHDR, SP), and the node's DH public key $g^{x_{n_i^{dir}}}$. The function outputs the processed Sphinx packet (SHDR', SP') and the established symmetric key $s_{n_i^{dir}}$.

4.3.2 Forwarding Segment

We extend Sphinx to allow each node to create an FS and add it to the FS payload. An FS contains a node's per-session state, which consists of a secret key s shared with the source, a routing segment R , and the session's expiration time EXP. To protect these contents, the FS is encrypted with a PRP keyed by a secret value SV known only by the node that creates the FS. A node seals and unseals its state using two opposite functions: FS_CREATE and FS_OPEN. We introduce only the form of FS_CREATE as follows:

$$FS = \text{PRP}(h_{\text{PRP}}(SV); \{s \parallel R \parallel \text{EXP}\}) \quad (1)$$

4.3.3 FS Payload

At the end of each HORNET setup packet is a data structure we call FS payload (see Figure 1). The FS payload is an onion-encrypted construction that allows intermediate nodes to add their FSes as onion-layers.

Processing the FS payload leaks no information about the path's length or about an intermediate node's position on the path. All FS payloads are padded to a fixed length, which is kept constant by dropping the right number of trailing bits of the FS payload before an FS is added to the front. Moreover, new FSes are always added to the beginning of the FS payload, eliminating the need for intermediate nodes to know their positions in order to process FS payloads.

An FS payload also provides both secrecy and integrity for the FSes it contains. Each node re-encrypts the FS payload after inserting a new FS and computes a MAC over the resulting structure. Only the source, with symmetric keys shared with each node on a path, can retrieve all the FSes from the FS payload and verify their integrity.

Functions. There are three core functions for the FS payload: INIT_FS_PAYLOAD, ADD_FS, and RETRIEVE_FSES.

INIT_FS_PAYLOAD. A node initializes a FS payload by using a pseudo-random generator keyed with a symmetric key s to generate rd random bits:

$$P_{FS} = \text{PRG1}(h_{\text{PRG1}}(s))_{[0..rd-1]} \quad (2)$$

where d is the size of a basic block of the FS payload (cf. Line 2 in Algorithm 2).

ADD_FS. Each intermediate node uses ADD_FS to insert its FS and other meta-data MD into the payload, as shown in Algorithm 1. First, the trailing d bits of the current FS payload, which are padding bits containing no information about previously added FSes, are dropped, and then the FS and MD are prepended to the shortened FS payload. The result is encrypted using a stream cipher (Line 3) and MACed (Line 5). Note that no node-position information is required in ADD_FS, and verifying that the length of the FS payload remains unchanged is straight-forward.

RETRIEVE_FSES. The source uses this function to recover all FSes $\{FS_i\}$ and MDs $\{MD_i\}$ inserted into an FS payload P_{FS} . RETRIEVE_FSES starts by recomputing the discarded trailing bits (Line 4) and obtaining a complete payload P_{full} . Afterwards, the source retrieves the FSes and MDs from P_{full} in the reverse order in which they were added by ADD_FS (see lines 7 and 10).

4.3.4 Setup Phase Protocol Description

Algorithm 1 Adding FS into FS payload.

```

1: procedure ADD_FS
   Input:  $s, FS, MD, P_{in}$ 
   Output:  $P_{out}$ 
2:    $d \leftarrow |FS| + |MD| + k$ 
3:    $P_{tmp} \leftarrow \left\{ FS \parallel MD \parallel P_{in}[d..(r-1)d] \right\}$ 
      $\oplus \text{PRG0}(h_{\text{PRG0}}(s))_{[k..end]}$ 
4:    $\alpha \leftarrow \text{MAC}(h_{\text{MAC}}(s); P_{tmp})$ 
5:    $P_{out} \leftarrow \alpha \parallel P_{tmp}$ 
6: end procedure

```

Algorithm 2 Retrieve FSes from FS payload

```

1: procedure RETRIEVE_FSES
   Input:  $P_{FS}, s, \{s_i\}$ 
   Output:  $\{FS_i\}, MD_i$ 
2:    $d \leftarrow l_{FS} + |MD| + k$ 
3:    $P_{init} \leftarrow \text{INIT\_FS\_PAYLOAD}(s)$ 
4:    $\psi \leftarrow P_{init}[(r-l)d..rd-1]$ 
      $\oplus \text{PRG0}(h_{\text{PRG0}}(s_0))_{[(r-l+1)d..end]} \parallel 0^d$ 
      $\oplus \text{PRG0}(h_{\text{PRG0}}(s_1))_{[(r-l+2)d..end]} \parallel 0^{2d}$ 
      $\dots$ 
      $\oplus \text{PRG0}(h_{\text{PRG0}}(s_{l-2}))_{[(r-1)d..end]} \parallel 0^{(l-1)d}$ 
5:    $P_{full} = P_{FS} \parallel \psi$ 
6:   for  $i \leftarrow (l-1), \dots, 0$  do
7:     check  $P_{full}[0..k-1] = \text{MAC}(h_{\text{MAC}}(s_i); P_{full}[k..rd-1])$ 
8:      $P_{full} \leftarrow P_{full} \oplus (\text{PRG0}(h_{\text{PRG0}}(s_i)) \parallel 0^{(i+1)d})$ 
9:      $FS_i \leftarrow P_{full}[k..k+l_{FS}-1]$ 
10:     $MD_i \leftarrow P_{full}[k+l_{FS}..d-1]$ 
11:
12:     $P_{full} \leftarrow P_{full}[d..end]$ 
13:   end for
14: end procedure

```

Source Processing. With the input

$$I = \left\{ (x_S, g^{x_S}), \left\{ g^{x_{n_i^{dir}}} \right\}, p^{dir} \right\}$$

the source node S bootstraps a session setup in 5 steps:

1. S selects the intended expiration time EXP for the session and specifies it in the common header CHDR.
2. S generates the send and the reply Sphinx headers by:

$$\{\text{SHDR}^f, \text{SHDR}^b\} = \text{GEN_SPHX_HDR}(I, \text{CHDR}) \quad (3)$$
 The common header CHDR (see Figure 1) is passed to the function to extend the per-hop integrity protection of Sphinx over it. GEN_SPHX_HDR also produces a series of keys: symmetric keys shared with each node on both paths $\{s_{n_i^{dir}}\}$, and the DH key pairs $\{(x_i^{dir}, y_i^{dir})\}$.
3. In order to enable the destination D to reply, S places the reply Sphinx header SHDR^b into the Sphinx payload:

$$\text{SP} = \text{GEN_SPHX_PL_SEND}(\{s_{n_i^f}\}, \text{SHDR}^b) \quad (4)$$
4. S creates an initial FS payload $P_{FS} = \text{INIT_FS_PAYLOAD}(x_S)$.
5. S composes $\mathbf{P}\bullet = \{\text{CHDR} \parallel \text{SHDR}^f \parallel \text{SP} \parallel P_{FS}\}$ and sends it to the first node on the forward path n_0^f .

Intermediate Node Processing. An intermediate node n_i^f receiving a packet $\mathbf{P}\bullet = \{\text{CHDR} \parallel \text{SHDR}^f \parallel \text{SP} \parallel P_{FS}\}$ processes it as follows:

1. n_i^f first processes SHDR^f and SP in $\mathbf{P}\bullet$ according to the Sphinx

protocol (using PROC_SPHX_PKT). As a result n_i^f obtains the processed header and payload (SHDR $^{f'}$, SP') as well as the routing information R_i^f , S 's DH public key y_i^f , and the established symmetric key $s_{n_i^f}$ shared with S . During this processing the integrity of the CHDR is verified.

- n_i^f obtains EXP from CHDR and checks that EXP is not expired. n_i^f also verifies that R_i^f is valid.
- To provide forward secrecy, the shared key $s_{n_i^f}$ is not used for the data transmission phase, since $s_{n_i^f}$ depends on n_i^f 's long-term DH key. Instead, n_i^f generates an ephemeral DH key pair $(x'_{n_i^f}, y'_{n_i^f})$ and derives s_i^f by

$$s_i^f = (y_i^f)^{x'_{n_i^f}} \quad (5)$$

This is the symmetric key that is included in n_i^f 's FS and used during the data transmission phase.

- n_i^f generates its FS FS_i^f by using its local symmetric key SV_i to encrypt s_i^f , R_i^f , and EXP:

$$FS_i^f = \text{FS_CREATE}(SV_i, \{s_i^f \parallel R_i^f \parallel \text{EXP}\}) \quad (6)$$

- n_i^f adds its FS_i^f and MD = $y'_{n_i^f}$ into the FS payload P_{FS} .

$$P_{FS} = \text{ADD_FS}(s_{n_i^f}, FS_i^f, y'_{n_i^f}, P_{FS}) \quad (7)$$

Adding $y'_{n_i^f}$ as the meta-data into the FS payload allows S to later retrieve $y'_{n_i^f}$ and derive the symmetric key s_i^f shared with n_i^f for the session. The MAC computed using $s_{n_i^f}$ shared between S and n_i^f (Line 5 in Algorithm 1) allows S to authenticate the DH public key $y'_{n_i^f}$.

- Finally node n_i^f assembles the processed packet $\mathbf{P}\bullet = \{\text{CHDR} \parallel \text{SHDR}^{f'} \parallel \text{SP}' \parallel P_{FS}\}$ and routes it to the next node according to the routing information R_i^f .

Destination Processing. As the last node on the forward path, D processes $\mathbf{P}\bullet$ in the same way as the previous nodes: it processes the Sphinx packet in $\mathbf{P}\bullet$ and derives a symmetric key s_D shared with S ; it generates a new DH key pair (x'_D, y'_D) and derives a second shared key s'_D ; it encrypts per-session state including s'_D into FS_D and inserts FS_D into the FS payload.

After these operations, however, D moves on to create the second setup $\mathbf{P}\bullet$ as follows:

- D retrieves the Sphinx reply header using the symmetric key s_D :

$$\text{SHDR}^b = \text{UNWRAP_SPHX_PL_SEND}(s_D, \text{SP}) \quad (8)$$

- D places the FS payload P_{FS} of $\mathbf{P}\bullet$ into the Sphinx payload SP^b of $\mathbf{P}\bullet$ (this will allow S to get the FSes $\{FS_i^f\}$):

$$\text{SP}^b = \text{GEN_SPHX_PL_RECV}(s_D, P_{FS}) \quad (9)$$

Note that since D has no knowledge about the keys $\{s_i^f\}$ except for s_D , D learns nothing about the other FSes in the FS payload.

- D creates a new FS payload $P_{FS}^b = \text{INIT_FS_PAYLOAD}(s_D)$ to collect the FSes along the backward path.
- D composes $\mathbf{P}\bullet = \{\text{CHDR} \parallel \text{SHDR}^b \parallel \text{SP}^b \parallel P_{FS}^b\}$ and sends it to the first node on the backward path, n_0^b .

The nodes on the backward path process $\mathbf{P}\bullet$ in the exact same way nodes on the forward path processed $\mathbf{P}\bullet$. Finally $\mathbf{P}\bullet$ reaches the source S with FSes $\{FS_i^b\}$ added to the FS payload.

Post-setup Processing. Once S receives $\mathbf{P}\bullet$ it extracts all FSes, i.e., $\{FS_i^f\}$ and $\{FS_i^b\}$, as follows:

- S recovers the FS payload for the forward path P_{FS}^f from SP^b :

$$P_{FS}^f = \text{UNWRAP_SPHX_PL_RECV}(\{s_i^b\}, \text{SP}^b) \quad (10)$$

- S retrieves the FSes for the nodes on the forward path $\{FS_i^f\}$:

$$\{FS_i^f\} = \text{RETRIEVE_FSSES}(\{s_i^f\}, P_{FS}^f) \quad (11)$$

- S directly extracts from P_{FS}^b the FSes for the nodes on the backward path $\{FS_i^b\}$:

$$\{FS_i^b\} = \text{RETRIEVE_FSSES}(\{s_i^b\}, P_{FS}^b) \quad (12)$$

With the FSes for all nodes on both paths, $\{FS_i^f\}$ and $\{FS_i^b\}$, S is ready to start the data transmission phase.

4.4 Data Transmission Phase

Each HORNET data packet contains an anonymous header AHDR and an onion-encrypted payload O as shown in Figure 1. Figure 2 demonstrates the details of an AHDR. The AHDR allows each intermediate node along the path to retrieve its per-session state in the form of an FS and process the onion-encrypted data payload. All processing of data packets in HORNET only involves symmetric-key cryptography, therefore supporting fast packet processing.

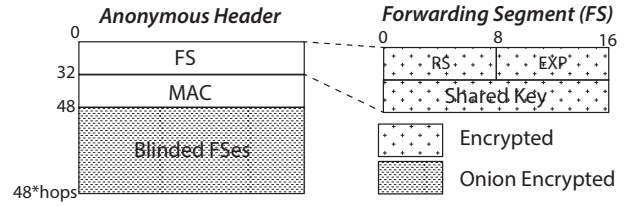


Figure 2: Format of a HORNET anonymous header with details of a forwarding segment (FS).

At the beginning of the data transmission phase, S creates two AHDRs, one for the forward path (AHDR f) and one for the backward path (AHDR b), by using FSes collected during the setup phase. AHDR f enables S to send data payloads to D . To enable D to transmit data payloads back, S sends AHDR b as payload in the first data packet. If this packet is lost, the source would notice from the fact that no reply is seen from the destination. If this happens the source simply resends the backward AHDR using a new data packet.

4.4.1 Anonymous Header

Like an FS payload, an AHDR is an onion-encrypted data structure that contains FSes. It also offers similar guarantees, i.e., secrecy and integrity, for the individual FSes it contains, for their number and for their order. Its functionalities, on the other hand, are the inverse: while the FS payload allows the source to collect the FSes added by intermediate nodes, the AHDR enables the source to re-distribute the FSes back to the nodes for each transmitted data packet.

Functions. The life cycle of AHDRs involves two functions: GET_FS and CREATE_AHDR. GET_FS allows each intermediate node to retrieve its FS from the input AHDR and compute the AHDR for the next hop (see Algorithm 3). CREATE_AHDR enables S to create two AHDRs, AHDR f and AHDR b (see Algorithm 4).

4.4.2 Onion Payload

HORNET data payloads are protected by onion encryption. To send a data payload to the destination, the source adds a sequence of encryption layers on top of the data payload, one for each node on the forward path (including the destination). As the packet is forwarded, each node removes one layer of encryption, until the destination removes the last layer and obtains the original plaintext.

Algorithm 3 Obtain FS from AHDR

```
1: procedure GET_FS
   Input:  $SV, \text{AHDR}$ 
   Output:  $FS, \text{AHDR}$ 
2:  $\{FS \parallel \gamma \parallel \beta\} \leftarrow \text{AHDR}$ 
3:  $s \leftarrow \text{FS\_OPEN}(FS, SV)_{[0..k]}$ 
4: check  $\gamma = \text{MAC}(h_{\text{MAC}}(s); \beta \parallel 0^c)$ 
5:  $\beta' \leftarrow \{\beta \parallel 0^c\} \oplus \text{PRG2}(h_{\text{PRG2}}(s))_{[0..rc]}$ 
6:  $\text{AHDR} \leftarrow \beta'$ 
7: end procedure
```

Algorithm 4 Anonymous header construction

```
1: procedure CREATE_AHDR
   Input:  $\{s_i\}, \{FS_i\}$ 
   Output:  $(FS_0, \gamma_0, \beta_0)$ 
2:  $\phi_0 \leftarrow \varepsilon$ 
3: for  $i \leftarrow 1, \dots, l-1$  do
4:    $\phi_i \leftarrow (\phi_{i-1} \parallel 0^c) \oplus \left\{ \text{PRG2}(h_{\text{PRG2}}(s_{i-1}))_{[(r-i)c..rc]} \right\}$ 
5: end for
6:  $\beta_l \leftarrow \{\text{RAND}(c(r-l)) \parallel \phi_{l-1}\}$ 
7: for  $i \leftarrow (l-1), \dots, 0$  do
8:    $\beta_i \leftarrow \left\{ FS_{i+1} \parallel \gamma_{i+1} \parallel \beta_{i+1} \parallel 0^{c(r-1)-1} \right\} \oplus \text{PRG2}(h_{\text{PRG2}}(s_i))$ 
9:    $\gamma_i \leftarrow \text{MAC}(h_{\text{MAC}}(s_i); FS_i \parallel \beta_i)$ 
10: end for
11: end procedure
```

To send a data payload back to the source, the destination adds only one layer of encryption with its symmetric key shared with the source. As the packet is forwarded, each node on the backward path re-encrypts the payload until it reaches the source. With all the symmetric keys shared with nodes on the backward path, the source is capable of removing all encryption layers, thus obtaining the original data payload sent by the destination.

Functions. Processing onion payloads requires two functions: ADD_LAYER and REMOVE_LAYER .

ADD_LAYER . The function's full form is:

$$\{O', IV'\} = \text{ADD_LAYER}(s, IV, O) \quad (13)$$

Given a symmetric key s , an initial vector IV , and an input onion payload O , ADD_LAYER performs two tasks. First, ADD_LAYER encrypts O with s and IV :

$$O' = \text{ENC}(h_{\text{ENC}}(s); IV; O) \quad (14)$$

Then, ADD_LAYER mutates the IV for next node:

$$IV' = \text{PRP}(h_{\text{PRP}}; IV) \quad (15)$$

REMOVE_LAYER . The function is the inverse of ADD_LAYER . Without going into details, we only list its full form below:

$$\{O', IV'\} = \text{REMOVE_LAYER}(s, IV, O) \quad (16)$$

4.4.3 Initializing Data Transmission

To start the data transmission session, S generates AHDR^f and AHDR^b as follows:

$$\text{AHDR}^f = \text{CREATE_AHDR}(\{s_i^f\}, \{FS_i^f\}) \quad (17)$$

$$\text{AHDR}^b = \text{CREATE_AHDR}(\{s_i^b\}, \{FS_i^b\}) \quad (18)$$

S then sends AHDR^b to D as payload of the first data packet (which uses AHDR^f), as specified in the following section.

4.4.4 Data Transmission Protocol Description

Source Processing. With the forward AHDR (AHDR^f), the source

S can send a data payload P with the following steps:

1. S ensures that the session is not expired by checking that the current time $t_{\text{curr}} < \text{EXP}$.
2. S creates an initial vector IV . With $\{s_i^f\}$, s_D , and IV , S onion encrypts the data payload P .
$$\{O_{lf}, IV_{lf}\} = \text{ADD_LAYER}(s_D, IV, P) \quad (19)$$
$$\{O_i, IV_i\} = \text{ADD_LAYER}(s_D, IV_{i+1}, O_{i+1}) \quad i \leftarrow (l^f - 1)..0 \quad (20)$$
3. S places IV_0 in the common header CHDR .
4. S sends out the resulting data packet $\{\text{CHDR}, \text{AHDR}^f, O_0\}$.

Processing by Intermediate Nodes. Each intermediate node n_i^f on the forward path processes a received data packet $\{\text{CHDR}, \text{AHDR}^f, O\}$ with its local secret key SV_i^f as follows:

1. n_i^f retrieves its FS FS_i^f from AHDR^f :
$$\{FS_i^f, \text{AHDR}^{f'}\} = \text{GET_FS}(SV_i^f, \text{AHDR}^f) \quad (21)$$
2. n_i^f extracts its per-session state, i.e., the symmetric key s_i^f shared with S , the routing information R_i^f , and the session's expiration time EXP , by decrypting FS with SV_i^f (cf. with Equation 1):
$$\{s_i^f, R_i^f, \text{EXP}\} = \text{PRP}^{-1}(h_{\text{PRP}}(SV_i^f); FS_i^f) \quad (22)$$
3. n_i^f checks the session is not expired by verifying that the current time $t_{\text{curr}} < \text{EXP}$.
4. n_i^f obtains IV from CHDR and removes one layer of encryption from the data payload:
$$\{O', IV'\} = \text{REMOVE_LAYER}(s_i^f, IV, O) \quad (23)$$
5. n_i^f updates the IV field in CHDR with IV' .
6. n_i^f sends the resulting packet $\{\text{CHDR}', \text{AHDR}^{f'}, O'\}$ to the next node according to R_i^f .

The above procedures show that the intermediate node processing requires only symmetric-cryptography operations.

Destination Processing. D processes incoming data packets as the intermediate nodes. Additionally, for the first data packet D retrieves AHDR^b from the payload, and stores the $\{s_D, R_0^b, \text{AHDR}^b\}$ locally so that D can retrieve AHDR^b when it wishes to send packets back to S .

Processing for the Backward Path. Sending and processing a HORNET packet along the backward path is the same as that for the forward path, with the exception of processing involving the data payload. Because D does not possess the symmetric keys that each node on the backward path shares with S , D cannot onion-encrypt its payload. Accordingly, intermediate nodes use ADD_LAYER instead of REMOVE_LAYER to process the data payload, and the source node recovers the data by REMOVE_LAYER .

4.5 Nested Anonymous Header Construction

As discussed in Section 3.2, the main difference of the protocols between sender anonymity and sender-receiver anonymity is that the latter requires nested AHDRs. We present in detail the process of composing an AHDR with a nested AHDR in Algorithm 5.

Constructing a new AHDR based on a nested AHDR A has essentially the same procedures as constructing a normal AHDR from ASes, except for the initialization process and the size of the resulted AHDR. For the AHDR initialization in Line 10 in Algorithm 5, the nested AHDR A is appended to the random bits generated. Thus, when the last node n_i^{dir} (RP) decrypts the AHDR, A is revealed to the node. For the size of the resulted AHDR, instead of r for a normal AHDR, the length of the generated AHDR with a nested AHDR is $2r$, doubling the bandwidth cost incurred by the protocol headers.

Algorithm 5 Creating an AHDR with a nested AHDR.

```
1: procedure CREATE_PADDING_STRING_NESTED
   Input:  $\{s_i\}, r$ 
   Output:  $\phi_{l-1}$ 
2:    $\phi_0 \leftarrow \varepsilon$ 
3:   for  $0 < i < l$  do
4:      $\phi_i \leftarrow (\phi_{i-1} \parallel 0^c) \oplus$ 
5:        $\left\{ \text{PRG0}(h_{\text{PRG0}}(s_{i-1}))_{[(2r-i)c..2rc]} \right\}$ 
6:   end for
7: end procedure
8: procedure CREATE_ANONYMOUS_HEADER_NESTED
   Input:  $\{s_i\}, \{FS_i\}, A$ 
   Output:  $(FS_0, \gamma_0, \beta_0)$ 
9:    $\phi_{l-1} \leftarrow \text{CREATE_PADDING_STRING_NESTED}(\{s_i\})$ 
10:   $\beta_{l-1} \leftarrow \left\{ \{A \parallel \text{RAND}(c(r-l))\} \right.$ 
11:     $\left. \oplus \text{PRG0}(h_{\text{PRG0}}(s_{l-1}))_{[0..c(2r-l)-1]} \right\} \parallel \phi_{l-1}$ 
12:   $\gamma_{l-1} \leftarrow \text{MAC}(h_{\text{MAC}}(s_{l-1}); FS_{l-1} \parallel \beta_{l-1})$ 
13:  for  $i \leftarrow (l-2), \dots, 0$  do
14:     $\beta_i \leftarrow \left\{ FS_{i+1} \parallel \gamma_{i+1} \parallel \beta_{i+1} \right.$ 
15:       $\left. \oplus \text{PRG0}(h_{\text{PRG0}}(s_i))_{[0..c(2r-l)-1]} \right\}$ 
16:     $\gamma_i \leftarrow \text{MAC}(h_{\text{MAC}}(s_i); FS_i \parallel \beta_i)$ 
17:  end for
18: end procedure
```

5. SECURITY ANALYSIS

In this section, we first presents formal proofs showing that HORNET satisfies the correctness, security, and integrity properties defined by Camenisch and Lysyanskaya [17]. Then, we describes how HORNET defends against well-known de-anonymization attacks, where an adversary actively or passively attempts to reveal the sender's (and/or the receiver's) network location. We also present defenses against denial of service attacks.

5.1 Formal Proof of Security for HORNET Data Transmission Phase

We prove HORNET's data transmission phase realizes ideal onion routing functionalities in the Universal Composability (UC) framework [18]. Conceptually, with an ideal onion routing protocol, adversaries have no access to the routing information or the message within packets except for opaque identifiers that vary across links.

As demonstrated by Camenisch and Lysyanskaya [17], to prove that a protocol conforms to an ideal onion routing model, it is sufficient to show that the protocol provides four properties: *correctness*, *integrity*, *wrap-resistance*, and *security*.³

5.1.1 Correctness

Proving the correctness property requires that HORNET protocol functions correctly in the absence of adversaries. A scrutiny of protocol description in Section 4 should suffice.

5.1.2 Integrity

To prove the integrity property, we need to prove that an adversary cannot forge a message that can traverse more than N uncompromised nodes, where Q is a fixed upper bound for HORNET. Equivalently, we demonstrate that the adversary, with significantly less than 2^k computation, can only produce a requisite message with a negligible probability. In our proof, we choose $Q = r + 1$.

Suppose that an adversary can constructs an HORNET AHDR $(FS_0, \gamma_0, \beta_0)$ that can succeed in traversing $r + 1$ honest nodes $n_0,$

³See definitions by Camenisch and Lysyanskaya [17].

$n_2, \dots, n_r,$ without knowing secrets $SV_0, \dots, SV_r.$ According to Algorithm 4, $FS_r, \beta_r,$ and γ_r satisfy:

$$\gamma_r = \text{MAC}(h_{\text{MAC}}(\text{PRP}^{-1}(h_{\text{PRP}}(SV_r); FS_r)_{[0..c]}); \beta_r) \quad (24)$$

For convenience, for $i \leq j \leq r - 1,$ we introduce the following notations:

$$\phi(SV, FS) = \text{PRP}^{-1}(h_{\text{PRP}}(SV); FS) \quad (25)$$

$$\rho(SV, FS) = \text{PRG}(h_{\text{PRG}}(\phi(SV, FS))) \quad (26)$$

$$\rho_i = \rho(SV_i, FS_i^*) \quad (27)$$

$$\rho_i^{FS} = \{\rho_i\}_{[c(r-1-i)..c(r-1-i)+l_{FS}-1]} \quad (28)$$

$$\rho_i^\gamma = \{\rho_i\}_{[c(r-1-i)+l_{FS}..c(r-1-i)-1]} \quad (29)$$

$$\rho_i^\beta = \{\rho_i\}_{[0..c(i+1)-1]} \parallel 0^{c(r-1-i)} \quad (30)$$

$$\rho_{i,j}^c = \{\rho_i\}_{[j..(j+1)c-1]} \quad (31)$$

where FS_i^* are defined recursively as follows:

$$FS_0^* = FS_0 \quad (32)$$

$$FS_i^* = FS_i \oplus \bigoplus_{j=0}^{i-1} \{\rho_j\}_{[c(j+i-1)..c(j+i-1)+l_{FS}-1]} \quad (33)$$

We observe that FS_i^* is a function of $\{FS_j \mid \forall 0 \leq j \leq i\}$ and $\{SV_j \mid \forall 0 \leq j \leq i-1\}.$ Accordingly, $\rho_i^{FS}, \rho_i^\gamma,$ and ρ_i^β are all functions of $\{FS_j \mid \forall 0 \leq j \leq i\}$ and $\{SV_j \mid \forall 0 \leq j \leq i-1\}.$

With a detailed inspection into Algorithm 4, we can express $FS_r, \beta_r,$ and $\gamma_r:$

$$FS_r = \bigoplus_{i=0}^{r-1} \rho_i^{FS} \quad (34)$$

$$\gamma_r = \bigoplus_{i=0}^{r-1} \rho_i^\gamma \quad (35)$$

$$\beta_r = \bigoplus_{i=0}^{r-1} \rho_i^\beta \quad (36)$$

(37)

With Equation 34, 35, 36 and 24, we can prove the following lemma:

LEMMA 1. *With less than 2^k work, an adversary can only distinguish $\text{MAC}(h_{\text{MAC}}(\phi(SV_r, FS_r)_{[0..c]}); \beta_r)$ from a random oracle with negligible probability.*

Proof. (Sketch) We will show that an adversary could not find two sets of

$$(SV_0, \dots, SV_r, FS_0 \dots, FS_{r-1}) \neq (SV'_0, \dots, SV'_r, FS'_0 \dots, FS'_{r-1})$$

that leads to the same value of $\text{MAC}(h_{\text{MAC}}(\phi(SV_r, FS_r)_{[0..c]}); \beta_r)$ with significant less than 2^k work.

Assume that the adversary, with much less than 2^k work, finds two sets,

$$(SV_0, \dots, SV_r, FS_0 \dots, FS_r) \neq (SV'_0, \dots, SV'_r, FS'_0 \dots, FS'_r)$$

that results in the same value of

$$\text{MAC}(h_{\text{MAC}}(\phi(SV_r, FS_r)_{[0..c]}); \beta_r)$$

We will show the assumption leads to an contradiction.

Because MAC is a random oracle, the only way for an attacker to distinguish the target function from a random oracle with much less than 2^k work is to ensure

$$\phi(SV_r, FS_r)_{[0..c]} = \phi(SV'_r, FS'_r)_{[0..c]}$$

and $\beta_r = \beta'_r.$ Because PRP is a pseudo-random permutation and h_{PRP} is collision resistant, we have $SV_r = SV'_r.$

Note that the last c bits of β_r and β'_r are $\rho_{r-1, r-1}^c$ and $\rho_{r-1, r-1}'$ respectively. Therefore, we have $\rho_{r-1, r-1}^c = \rho_{r-1, r-1}'.$ According to Equation 31, because PRG is a pseudo-random generator, we have $SV_{r-1} = SV'_{r-1}$ and $FS_{r-1}^* = FS'_{r-1}.$ Hence, $\rho_{r-1, j}^c = \rho_{r-1, j}', \forall 0 \leq j \leq r-1.$

A careful calculation shows that the c bits before the last c bits in β_r and β'_r are $\rho_{r-2,r-2}^c \oplus \rho_{r-1,r-2}^c$ and $\rho_{r-2,r-2}^c \oplus \rho_{r-1,r-2}^c$. Similarly, we have $SV_{r-2} = SV_{r-2}'$ and $FS_{r-2}^* = FS_{r-2}'$.

Continuing the logic in the way above, we finally have $SV_i = SV_i'$ and $FS_i^* = FS_i'$, $\forall 0 \leq i \leq r-1$. However, given Equation 33, $SV_i = SV_i'$, and $FS_0^* = FS_0'$, we have $FS_i = FS_i'$, $\forall 0 \leq i \leq r-1$. That says,

$$(SV_0, \dots, SV_r, FS_0, \dots, FS_{r-1}) = (SV_0', \dots, SV_r', FS_0', \dots, FS_{r-1}')$$

. Therefore, we obtain a contradiction. \square

We can substitute Equation 34, 35, and 36 into Equation 24, and rewrite the equation into:

$$\rho_0^\gamma = MAC(h_{MAC}(\phi(SV_r, FS_r)_{[0..c]}); \beta_r) \oplus \bigoplus_{i=1}^{r-1} \rho_i^\gamma \quad (38)$$

Because MAC is not used in ρ_i^γ , the right side of Equation 38 is a random oracle with respect to SV_i and FS_i , $\forall 0 \leq i \leq r-1$.

We can further simplify the notation by denoting ρ_0^γ as $f_0(SV_0, FS_0)$ and the right side of Equation 38 as

$$f_1(FS_0, \dots, FS_{r-1}, SV_0, \dots, SV_{r-1})$$

. Both f_0 and f_1 are random oracle with range $\{0, 1\}^k$. As a result, by creating a AHDR traversing $r+1$ honest nodes, the adversary equivalently finds a solution to

$$f_0(SV_0, FS_0) = f_1(FS_0, \dots, FS_{r-1}, SV_0, \dots, SV_{r-1})$$

which obviously can only be solved with negligible probability with significantly less than 2^k work. That says, with much less than 2^k work, the adversary can only generate a packet that traverse $r+1$ hops with negligible probability.

5.1.3 Wrap-resistance

To prove the wrap-resistance property, we show that given a data packet (FS, γ, β, P) , an adversary, with significant less than 2^k work, cannot generate a message $(FS', \gamma', \beta', P)$ so that processing $(FS', \gamma', \beta', P)$ on an uncompromised node yields data packet (FS, γ, β, P) .

To succeed, it is necessary that $\beta \oplus \{\beta'_{c..cr-1} || 0^c\} = \rho(SV', FS')$

$$\beta \oplus \{\beta'_{c..cr-1} || 0^c\} = \rho(SV', FS') \quad (39)$$

Consider the last c bits of the left side of Equation 39, we have:

$$\beta_{[c(r-1)..cr-1]} = \rho(SV', FS')_{[c(r-1)..cr-1]} \quad (40)$$

Because PRG , PRP , h_{PRG} , and h_{PRP} are all random oracles, an adversary could generate FS' and SV' that satisfy Equation 40 only with negligible probability if the adversary performs much less than 2^k work.

5.1.4 Security

In order to demonstrate the security property, we need to prove that an adversary with controls over all nodes on a path except one node N , cannot distinguish among data packets entering N . The adversary is able to select paths for the packets traversing N and payloads of the packets. The adversary can also observe packets entering and leaving node N except for packets whose headers match the challenge packets.

We construct the following game G . The adversary picks two paths $(n_0, n_1, \dots, n_{\nu-1})$ $0 < \nu \leq r$ and $(n'_0, n'_1, \dots, n'_{\nu'-1})$ $0 \leq \nu' \leq r$, where $n_i = n'_i$ $\forall 0 \leq i \leq j$ and $n_j = n'_j = N$. Note that the nodes after N in both paths are not necessarily the same set of nodes, and the lengths of the paths can also be different. The adversary chooses the public/private key pairs and $SV_i(SV'_i)$ for all nodes except N and can arbitrarily select payload M .

The challenger picks randomly a bit b and proceeds in one of the following two ways:

$b = 0$: The challenger creates an AHDR $(FS_0, \gamma_0, \beta_0)$ through the HORNET setup phase using the path $(n_0, n_1, \dots, n_{\nu-1})$ and uses it to construct a data packet with onion encrypted payload M^e

from M . The challenger outputs $(FS_0, \gamma_0, \beta_0, M^e)$, which could be sent to n_0 .

$b = 1$: The challenger creates an AHDR $(FS_0, \gamma_0, \beta_0)$ using the alternative path $(n'_0, n'_1, \dots, n'_{\nu'-1})$ instead and outputs $(FS_0, \gamma_0, \beta_0, M^e)$, which could be sent to n'_0 .

Given the output $(FS_0, \gamma_0, \beta_0)$, the adversary's goal is to determine b . The adversary can also input any messages $(FS', \gamma', \beta', M^{e'})$ to the honest node N and observes the output messages as long as $(FS', \gamma', \beta') \neq (FS_j, \gamma_j, \beta_j)$.⁴

We define the adversary's advantage as the difference between $\frac{1}{2}$ and the probability that the adversary succeeds. We will show that the adversary's advantage is negligible. That says, the adversary has no better chance to determine b than random guessing.

Proof. (Sketch) We adopt the method of hybrid games. First, we construct a modified game G_1 with exactly the same definition, except that we require $j = 0$. An adversary who can win G can thus immediately win G_1 . On the other hand, because the adversary controls nodes (n_0, \dots, n_{j-1}) ((n'_0, \dots, n'_{j-1})) and can thus emulate their processing, the adversary can also win game G if he/she can win game G_1 . Therefore, the adversary can win game G if and only if the adversary can win game G_1 .

We create a second game G_2 , which is the same as G_1 except that FS_0 , β_0 , and γ_0 are all randomly generated from their corresponding domains. If the adversary can distinguish G_2 from G_1 , we have:

1. The adversary can distinguish

$$FS_0 = PRP(h_{PRP}(SV_0); R_0 || s_0)$$

from randomness. Then it must be that the adversary is able to tell the output of a pseudo-random permutation with a random key $(h_{PRP}(SV_0))$ from random bits. The probability of success for the adversary is negligible.

2. The adversary can distinguish

$$\beta_0 = PRG(h_{PRG}(SV_0)) \oplus \{FS_1 || \gamma_1 || \beta_1\}$$

from randomness. Then it must be the adversary is able to distinguish the output of a secure pseudo-random number generator with a random key $(h_{PRG}(SV_0))$ from randomness. The probability that the adversary succeeds is negligible.

3. The adversary can distinguish

$$\gamma_0 = MAC(h_{MAC}(SV_0); \beta_0)$$

from randomness. Then it must be the adversary is able to distinguish the output of MAC with a random key $h_{MAC}(SV_0)$ from randomness. Under our random oracle assumption for MAC , the probability of success is negligible.

Therefore, the adversary cannot distinguish G_2 from G_1 .

At last, because in G_2 , $(FS_0, \gamma_0, \beta_0)$ are all random, the adversary's advantage is 0. Moreover, in our chain of game $G \rightarrow G_1 \rightarrow G_2$, the adversary can only distinguish a game from its previous game with negligible probability. As a result, the adversary's advantage in game G is negligible. \square

5.2 Passive De-anonymization

Session linkage. Each session is established independently from every other session, based on fresh, randomly generated keys. Sessions are in particular not related to any long term secret or identifier of the host that creates them. Thus, two sessions from the same host are unlinkable, i.e., they are cryptographically indistinguishable from sessions of two different hosts.

Forward/backward flow correlation. The forward and backward headers are derived from distinct cryptographic keys and therefore

⁴We follow the definition of security property in [17] and only care about header uniqueness.

cannot be linked. Only the destination is able to correlate forward and backward traffic, and could exploit this to discover the round-trip time (RTT) between the source and itself, which is common to all low-latency anonymity systems. Sources, willing to thwart such RTT-based attacks from malicious destinations, could introduce a random response delay for additional protection.

Packet correlation. HORNET obfuscates packets at each hop to prevent an adversary observing two points on a path from linking packets between those two points using packets’ bit-patterns. Besides the use of onion encryption, we also enforce this obfuscation by padding header and payload to a fixed length, thwarting packet-size-based correlation.⁵ While this does not prevent the adversary from discovering that the same flow is passing his observation points using traffic analysis, it makes this process non-trivial, and allows upper layer protocols to take additional measures to hide traffic patterns. The hop-by-hop encryption of the payload also hides the contents of the communication in transit, protecting against information leaked by upper layer protocols that can be used to correlate packets.

Flow-dynamics-based end-to-end correlation. In general it is difficult even for high latency mix networks to resist such powerful adversaries [34]. Low-latency anonymity systems are particularly prone to these types of attacks [43, 30]. HORNET cannot protect against them, but as mentioned above, the use of packet obfuscation makes these attacks more expensive and allows for potential additional measures to be taken (e.g., padding), either by upper layer protocols or by extensions of HORNET. Mass surveillance based on end-to-end confirmation attacks requires an adversary to monitor a large fraction of the nodes of the network and to store and process all intercepted traffic, so it falls outside our attacker model.

5.3 Active De-anonymization

Session state modification. The state of each node is included in an encrypted FS. During the session setup, the FSEs are inserted into the FS payload, which allows the source to check the integrity of these FSEs during the setup phase. When the FSEs are carried in the AHDR, they are integrity-protected as well thanks to per-hop MACs computed by the source. This second case needs clarification, however, since it involves a particular construction. Each MAC protecting an FS is computed using a key contained in that FS. This construction is secure because every FS is encrypted using a PRP keyed with a secret value known only to the node that created the FS: if the FS is modified, the authentication key that the node obtains after decryption is a new pseudo-random key that the adversary cannot control. Thus, the probability of the adversary being able to forge a valid MAC is still negligible.

Path modification. Both HORNET data structures that hold paths, i.e., the FS payloads in the setup phase and the AHDRs, use chained per-hop MACs to protect path integrity and thwart attacks like inserting new nodes, changing the order of nodes, and splicing two paths. The source can check such chained per-hop MACs to detect the modifications in the FS payload before using the modified FS payload to construct AHDRs, and similarly intermediate nodes can detect modifications to AHDRs and drop the altered packets.

Replay attacks. Replaying packets can facilitate some types of confirmation attacks. For example, an adversary can replay packets with a pre-selected pattern, and have a colluding node identify those packets downstream. HORNET offers replay protection through session expiration. Replayed packets whose sessions have expired are immediately dropped. Replay of packets whose ses-

sions are not yet expired is possible, but the risk of misbehaving nodes being detected⁶ might deter an adversary from using replays to conduct mass surveillance.

Payload tagging or tampering. HORNET does not use per-hop MACs on the payload of data packets for efficiency and because the destination would not be able to compute such MACs. The lack of integrity protection allows an adversary to tag payloads. Admittedly, by using tagging in conjunction with replay attacks, the adversary is able to improve the effectiveness of confirmation attacks. However, the end-to-end MACs protect the integrity of the data, making such attacks (at a large scale) detectable.

5.4 Denial-of-Service (DoS) Resilience

Computational DoS. The use of asymmetric cryptography in the setup phase makes HORNET vulnerable to computational DoS attacks, where adversaries can attempt to deplete a victim node’s computation capability by initiating a large number of sessions through this node. To mitigate this attack, HORNET nodes should rate-limit the number of session setup packets that are processed per neighbor.

State-based DoS. HORNET is not vulnerable to attacks where adversaries maintain a large number of active sessions through a victim node. One of HORNET’s key features is that all state is carried within packets, thus no per-session memory is required on nodes or rendezvous points.

5.5 Topology-based Analysis

Unlike onion routing protocols that use global re-routing through overlay networks (e.g., Tor [23] and I2P [47]), HORNET uses short paths created by the underlying network architecture to reduce latency, and is therefore bound by the network’s physical interconnection and ISP relationships. This is an unavoidable constraint for onion routing protocols built into the network layer [29, 42]. Thus, knowledge of the network topology enables an adversary to reduce the number of possible sources (and destinations) of a flow by only looking at the previous (and next) hop of that flow. For example, in Figure 3(a), assume that AS0 is controlled by a passive adversary. The topology indicates that any packet received from AS1 must have originated from a source located at one of {AS1, AS2, AS3, AS4, AS5}.

We evaluate the information leakage due to the above topology constraints in the scenario where a single AS is compromised. We derive AS-level paths from iPlane trace-route data [7]⁷, and use AS-level topology data from CAIDA [33]. For each AS on each path we assume that the AS is compromised and receives packets from a victim end host through that path. We compute the end host’s anonymity set size that the adversary learns according to the topology. Similar to Hsiao et al. [29], we use the number of IPv4 addresses to estimate the size of the anonymity set of end hosts. Figure 3(b) plots the CDF of the anonymity set size for different distances (in number of AS hops) between the adversary and the victim end host. For adversarial ASes that are 4 hops away, the anonymity set size is larger than 2³¹ in 90% of the cases. Note that the maximum anonymity set size is 2³² in our analysis, because we consider only IPv4 addresses.

Implications of path knowledge. Knowledge about the path, including the total length of the path and an adversarial node’s position on the path, significantly downgrades the anonymity of end hosts. Considering again Figure 3(a), if the adversary controlling AS0 sees a packet incoming from AS1 and knows that it is 4 hops

⁵An alternative for a more optimized bandwidth usage would be to allow two or three different payload sizes, at a cost of decreased anonymity.

⁶Volunteers and organizations might monitor the network, and honest ASes might control their own nodes as part of an IDS.

⁷Traceroute data was generated on October 12, 2014

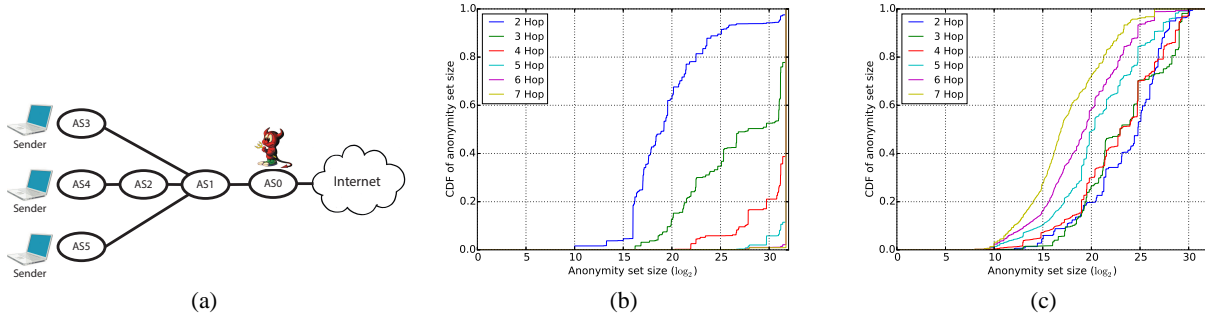


Figure 3: a) An example AS-level topology with an adversarial AS (AS0); b) CDF of anonymity-set size when a position-agnostic AS on path is adversarial. “Hops” indicates the number of ASes between the adversarial AS and the victim end host. c) CDF of anonymity-set size when an adversarial AS knows its own position on the path.

away from the source host, he learns that the source host is in AS4. Compared with the previous case, we see that the anonymity set size is strongly reduced.

We quantify additional information leakage in the same setting as the previous evaluation. Figure 3(c) represents the CDFs of the anonymity set sizes of end hosts according to the distance to the compromised AS. The anonymity set sizes are below 2^{28} in 90% of the cases when the adversarial ASes are 4 hops away, with an average size of 2^{23} . This average size decreases to 2^{17} for the cases where the adversarial ASes are 7 hops away from the target hosts.

Previous path-based anonymity systems designed for the network layer either fail to hide knowledge about the path [42] or only partially obscure the information [29]. In comparison, HORNET protects both the path length and the position of each node on the path, which significantly increases the anonymity-set size of the end hosts, as our analysis in this section showed.

6. EVALUATION

We implemented the HORNET router logic in an Intel software router using the Data Plane Development Kit (DPDK) [4]. To our knowledge, no other anonymity protocols have been implemented in a router SDK. We also implemented the HORNET client in Python. Furthermore, we assembled a custom crypto library based on the Intel AESNI crypto library [6], the curve25519-donna library [3], and the PolarSSL libraries [9]. For comparison, we implemented the data forwarding logic from LAP, Dovetail, and L3 Tor⁸ and Sphinx using DPDK and our crypto library. We use IP forwarding in DPDK as our performance baseline.

Our testbed contains an Intel software router connected to a Spirent TestCenter packet generator and analyzer [11]. The software router runs DPDK 1.7.1 and is equipped with an Intel Xeon E5-2680 processor (2.70 GHz, 2 sockets, 16 logical cores/socket), 64 GB DRAM, and 3 Intel 82599ES 40 Gb/s network cards (each with 4 10 Gb/s ports). We configured DPDK to use 2 receiving queues for each port with 1 adjacent logical core per queue.

6.1 Data Forwarding Performance

Forwarding latency. We measure the CPU cycles consumed to forward a data packet in all schemes. Figure 5(a) shows the average latency (with error bars) to process and forward a single data packet in all schemes when payload sizes vary. We observe that data forwarding in Sphinx is 3 orders of magnitude slower than that

⁸For fair comparison, we only implement payload encryption/decryption. We do not implement SSL/TLS or L4 transport control logic.

Scheme	Header Length	Sample Length (Bytes)
LAP	$12 + 2s \cdot r$	236
Dovetail	$12 + s \cdot r$	124
Sphinx	$32 + (2r + 2)s$	296
Tor	$3 + 11 \cdot r$	80
HORNET	$8 + 3r \cdot s$	344

Table 1: Comparison between the length of different packet header formats in bytes. s is the length of symmetric elements and r is the maximum AS path length. For the sample length, we select $s = 16$ Bytes and $r = 7$. Analysis of iPlane paths shows that more than 99% of all paths have less than 7 AS hops.

of HORNET, L3 Tor, LAP, and Dovetail because Sphinx requires per-packet asymmetric crypto operations. We further observe that HORNET, even with onion encryption/decryption over the entire payload and extensive header manipulation, is only 5% slower than LAP and Dovetail for small payload (64 bytes). For large payloads (1200 bytes⁹), HORNET is 71% slower (about 400 nanoseconds slower per packet when using a single core) than LAP and Dovetail. However, the additional processing overhead enables stronger security guarantees.

Header overhead. As a result of carrying anonymous session state (specifically cryptographic keys) within packet headers, HORNET headers are larger than Sphinx, L3 Tor, LAP, and Dovetail headers (see Table 1). While larger headers reduce net throughput (i.e., goodput), this tradeoff appears acceptable: compared to L3 Tor, no state is required at relay nodes, enabling scalability; compared to Sphinx, data processing speed is higher; compared to LAP and Dovetail, HORNET provides stronger security properties.

Throughput. To compare the throughput of data forwarding, we measure the throughput of all schemes on a single CPU core on a 10 Gb/s link with different payload sizes. The result is shown in Figure 6.1. We observe that all schemes except Sphinx can saturate the 10Gb/s link when the payload size is larger than 512 Bytes. When payload size is smaller than 256 Bytes, HORNET’s throughput is 20% - 30% smaller than LAP and Dovetail, whose throughput, in turn are 25% smaller than IP forwarding. Note, Tor’s small throughput at small payload sizes is mainly a reflection of its small per-hop header size, as later proved in Figure 5(b). In comparison, Sphinx can only achieve 0.05Gb at its maximum when the packet

⁹Because LAP, Dovetail, and HORNET all have large packet headers of 300+ bytes, we limit the largest payload in our experiments to be 1200 bytes.

size is 1500 Bytes because of its expensive data forwarding.

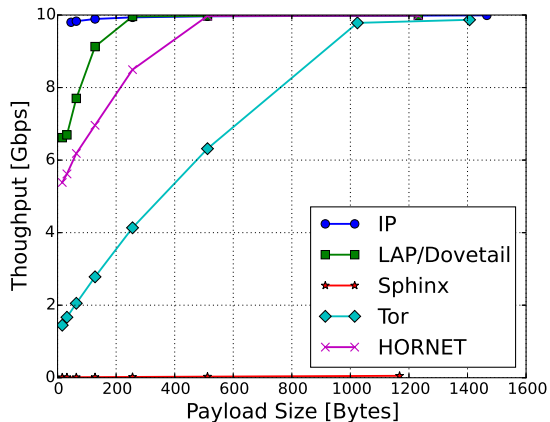


Figure 4: Data forwarding throughput on 10 Gb/s link

Goodput. We further compare all the schemes by goodput, which excludes the header overhead from total throughput. Goodput is a comprehensive metric to evaluate both the packet processing speed and protocol overhead. For example, a scheme where headers take up a large proportion of packets yields only low goodput. On the other hand, a scheme with low processing speed also results in poor goodput.

Figure 5(b) and Figure 5(c) demonstrate the goodput of all schemes on a 10Gb/s link when varying the number of hops, with 40-byte and 1024-byte payloads, respectively. We observe that Sphinx’s goodput is the lowest in both cases because of its large packet headers and slow asymmetric-crypto-based packet processing.

When the payload size is small, the goodput of all protocols remains stable. This is due to the fact that no scheme can saturate the link, and accordingly the goodput differences between the three schemes mainly reflect the different processing latencies among them. Consequently, L3 Tor’s and HORNET’s goodput is 32% less than that of LAP and Dovetail. On the other hand, when the payload size is large, all schemes except Sphinx can saturate the 10Gb/s link. HORNET can reach 93% of LAP and Dovetail’s goodput while providing stronger security guarantees.

6.2 Max Throughput on a Single Router

To investigate how our implementation scales with respect to the number of CPU cores, we use all 12 ports on the software router, generating HORNET data packets at 10 Gb/s on each port. Each packet contains a 7 AS-hop header and a payload of 512 bytes, and is distributed uniformly among the working ports. We monitor the aggregate throughput on the software router.

The maximal aggregate throughput of HORNET forwarding in our software router is 93.5 Gb/s, which is comparable to today’s switching capacity of a commercial edge router [2]. When the number of cores is less than 5, our HORNET implementation can achieve full line rate (i.e., 10 Gb/s per port). As the number of cores increases to 5 and above, each additional port, with the two logical cores binded to the port, adds an extra 6.8Gb/s.

6.3 Session-Setup Performance

We evaluate the latency introduced by processing setup packets on each border router. Similar to measuring the latency of data forwarding, we also instrument the code to measure CPU cycles consumed to process packets in the session-setup phase. Table 2 lists

the average per-node latency for processing the two setup packets in HORNET’s session-setup phase. Due to a Diffie-Hellman key exchange, processing the two setup packets in the session-setup phase increases processing latency (by about $240\mu s$) compared to data packet processing. However, HORNET must only incur in this latency once per session.

Packet	Latency (K cycles)	Latency (μs)
P1	661.95 ± 30.35	245.17 ± 11.24
P2	655.85 ± 34.03	242.91 ± 12.60

Table 2: Per-node latency to process session-setup packets with standard errors.

6.4 Network Evaluation

Distribution of AS-level path length. The bandwidth overhead of a HORNET packet depends on the number of ASes traversed by the packet. Figure 6 demonstrates the CDF of AS-level path lengths of the paths extracted from our data source. We observe that 99% of the paths have a path length smaller than 7, and the mean AS-level path length is 4.2. Thus, to achieve 128 bits of security, 48 bytes per AS hop are required, leading to an average overhead of 201.6 bytes.

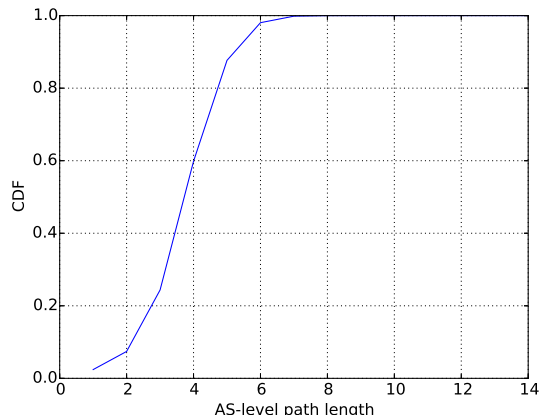


Figure 6: CDF of AS-level path length.

Non-scalability of a stateful design. We evaluate the memory capacity needed to maintain state required by a stateful design to support Internet-scale anonymous communication. We consider the design of Tor, one of the most popular onion routing systems today [23], and assume that each Tor node (*onion router* or OR) would correspond to an autonomous system (AS), as proposed by Liu et al. [32]. Analyzing the CAIDA Internet Traces [1], we found that a 10 GbE backbone link handles about 1M new flows every minute under normal operating conditions. Since the largest inter-AS links today have up to ten times that capacity (100 Gbps)¹⁰, this means that at the core of the network there are edge routers of ASes that handle about 10M new flows per minute.

If we assume that half of these flows would use a Tor circuit, because of the default lifetime of circuits of 10 minutes¹¹ we obtain that ORs on such edge routers would need to store state for

¹⁰E.g., see www.seattleix.net/participants.htm.

¹¹We actually measured the number of flows taking this lifetime into account, in particular we expired flows only if no packets were

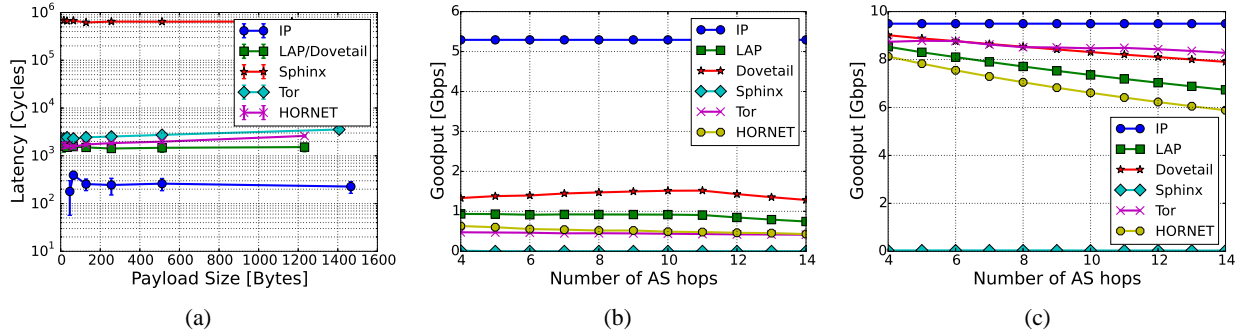


Figure 5: a) Per-node data forwarding latency on a 10 Gbps link; b) Data forwarding goodput on a 10 Gbps link for small packets (40 Bytes payload); c) Data forwarding goodput large packets (1024 Bytes payload). For a), lower is better; for others, higher is better.

approximately 50M circuits at all times. Since Tor stores at least 376 bytes per circuit, this translates to almost 20 GB of memory. This might still be acceptable for high-end devices, but there are a number of additional factors that make keeping state unfeasible, even for ASes handling less traffic:

- The growing number of user on the Internet and the increasing number of devices per user result in an increasing number of traffic flows;
- The state for each circuit would actually be larger, as for active circuits the ORs need to store the packets being transmitted until they are acknowledged by the next hop;
- A DDoS attack could force an OR to store much more state by opening a large number of new circuits through that OR.

7. DISCUSSION

7.1 Retrieving Paths Anonymously in FIAs

HORNET assumes that the source host can obtain a forward path and a backward path to an intended destination host anonymously in FIAs. We briefly discuss how a source host using HORNET can retrieve such two paths in NIRA, SCION and Pathlet.

NIRA and SCION hosts rely on path servers to retrieve paths. In both architectures, each destination node registers its path to/from the network “core” on a centralized path server. A source only needs to anonymously fetch the information registered by the destination to form forward and backward paths, which could be done in two ways. As a first method, the source can obtain the paths to/from the path servers from resolver configuration or local services similar to DHCP and establish an anonymous HORNET session to the servers. Once the HORNET session is created, the source can proceed to anonymously request a path to the destination. Alternatively, the source can leverage a PIR scheme to retrieve the path anonymously from the path server.

In Pathlet routing, the situation is different because the routing information is disseminated through a path vector protocol. The source can keep a local database of pathlets and simply query the database locally for the pathlets to a certain destination.

7.2 Composability with other protocols

HORNET operates at the network layer. As such, upper-layer anonymity protocols may be used in conjunction with HORNET to provide stronger anonymity guarantees. For example, to help mitigate concerns of topology-based attacks (as seen in Section 5.5), seen on them for over 10 minutes. Also note that in our setting it would not be possible to have multiple streams per circuit, unless the destinations those streams are in the same AS.

a single hop proxy or virtual private network (VPN) could be used to increase the size of the anonymity sets of end hosts. Similar solutions could also protect against upper-layer de-anonymization attacks, in particular fingerprinting attacks on the transport protocol [44].

At lower layers, HORNET is also compatible with link-layer protection such as link-level encryption. The role of link-level encryption in HORNET is comparable to SSL/TLS in Tor. Link encryption prevents an adversary eavesdropping on a link from being able to distinguish individual sessions from each other, therefore making confirmation attacks much harder for this type of adversary.

7.3 Targeted confirmation attacks

When an adversary controls more than one node on a path, it can launch confirmation attacks by leveraging flow-dynamics analysis, timing, and packet tagging, all of which can be further assisted by replay attacks. HORNET, like other low-latency onion routing schemes [23], cannot prevent such confirmation attacks targeting individual users. However, HORNET raises the bar of deploying such attacks for secretive mass surveillance: the adversary must be capable of controlling a significant percentage of ISPs often residing in multiple geopolitical boundaries, not to mention keeping such massive activity confidential.

8. RELATED WORK

Anonymity system using overlays. The study of anonymous communication began with Chaum’s proposal for mix relays [19]. A number of message-based mix systems have been proposed and deployed since [28, 36, 20, 21]. These systems can withstand an active adversary and a large fraction of compromised relays, but rely on expensive asymmetric primitives, and message batching and mixing. Thus, they suffer from large computational overhead and high latency.

Onion routing systems [41, 23] were proposed to efficiently support interactive web traffic. Onion routing systems are vulnerable against end-to-end correlation attacks [31], and may fail to provide unlinkability when two routers on the path are compromised [27, 30]. While these systems are generally intended as overlays, some proposals [14, 15] are closer to the network layer in that they use UDP (instead of TCP [23]) between pairs of onion routers. HORNET distinguishes itself from all of these systems in two ways: it does not store per-connection state on the nodes, and it requires only a single round trip to establish an anonymous path.

Anonymity systems in FIAs. Hsiao et al. [29] explored the design space of efficient anonymous systems with a relaxed adversary

model. In their scheme, LAP, the adversary can compromise only a single node, and the first hop must always be honest. Sankey and Wright proposed Dovetail [42] (based on Pathlets [26] and SCION [48]) which has the same attacker model as LAP, except it allows the first hop to be compromised. Compared to HORNET and to onion routing, LAP and Dovetail do not offer protection to the payload, and they allow nodes to learn their position on the path.

The research community has also explored applying onion routing to FIAs. Liu et al. [32] proposed Tor instead of IP as an FIA that regards anonymity as the principal requirement for the network architecture. However, details on how to scale Tor’s current design (requiring per-circuit state) to Internet scale were not addressed.

DiBenedetto et al. [22] proposed ANDaNA, to enable onion routing in Named Data Networking (NDN). NDN focuses on content delivery and thus inherently different from the FIAs we considered.

9. CONCLUSION

In this paper, we address the question of “what minimal mechanism can we use to frustrate pervasive surveillance [24]?” and study the design of a high-speed anonymity system supported by the network architecture. We propose HORNET, a scalable and high-speed onion routing scheme for future Internet architectures. HORNET nodes can process anonymous traffic at over 93 Gb/s and require no per-flow state, paving the path for Internet-scale anonymity. Our experiments show that small trade-offs in packet header size greatly benefit security, while retaining high performance.

10. REFERENCES

- [1] CAIDA UCSD Anonymized Internet Traces 2014. http://www.caida.org/data/passive/passive_2014_dataset.xml. Retrieved on 2015.04.30.
- [2] Cisco ASR-1000. <http://www.cisco.com/c/en/us/products/routers/>. Retrieved on 2015.04.28.
- [3] curve25519-donna. <https://code.google.com/p/curve25519-donna/>. Retrieved on 2014.12.13.
- [4] DPDK: Data Plane Development Kit. <http://dpdk.org/>. Retrieved on 2014.12.23.
- [5] HORNET: High-speed Onion Routing at the Network Layer. [anonymizedurl](http://anonymizedurl.com). Technical report.
- [6] Intel AESNI Sample Library. <https://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library>. Retrieved on 2014.12.13.
- [7] iPlane dataset. <http://iplane.cs.washington.edu/data/data.html>. Retrieved on 2014.12.23.
- [8] NSA targets the privacy-conscious. <http://daserste.ndr.de/panorama/aktuell/NSA-targets-the-privacy-conscious,nsa230.html>. Retrieved on 2015.05.13.
- [9] PolarSSL. <https://polarssl.org/>. Retrieved on 2014.12.13.
- [10] Segment Routing Architecture (IETF draft). <https://datatracker.ietf.org/doc/draft-ietf-spring-segment-routing/>. Retrieved on 2015.05.13.
- [11] Spirent TestCenter. http://www.spirent.com/Ethernet_Testing/Software/TestCenter. Retrieved on 2014.12.23.
- [12] The crux of the NSA story in one phrase: ‘collect it all’. <http://www.theguardian.com/commentisfree/2013/jul/15/crux-nsa-collect-it-all>. Retrieved on 2014.12.13.
- [13] Tor metrics. <https://metrics.torproject.org>. Retrieved on 2015.05.13.
- [14] Philippe Boucher, Adam Shostack, and Ian Goldberg. Freedom systems 2.0 architecture, 2000. White paper, Zero Knowledge Systems, Inc.
- [15] Zach Brown. Cebolla: Pragmatic IP anonymity. In *Ottawa Linux Symposium*, 2002.
- [16] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. IETF RFC 6810.
- [17] Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In *CRYPTO*, 2005.
- [18] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE FOCS*, 2001.
- [19] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981.
- [20] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE S&P*, 2003.
- [21] George Danezis and Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE S&P*, 2009.
- [22] Steven DiBenedetto, Paolo Gasti, Gene Tsudik, and Ersin Uzun. ANDaNA : Anonymous Named Data Networking Application. In *NDSS*, 2011.
- [23] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security*, 2004.
- [24] S. Farrell and H. Tschofenig. Pervasive Monitoring Is an Attack. IETF RFC 7258.
- [25] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *EUROCRYPT*, 2004.
- [26] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. *ACM SIGCOMM*, 2009.
- [27] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding routing information. In *ACM Information Hiding (IH) Conference*, 1996.
- [28] Ceki Gülcü and Gene Tsudik. Mixing email with babel. In *NDSS*, 1996.
- [29] Hsu Chun Hsiao, Tiffany Hyun Jin Kim, Adrian Perrig, Akira Yamada, Samuel C. Nelson, Marco Gruteser, and Wei Meng. LAP: Lightweight anonymity and privacy. In *IEEE S&P*, 2012.
- [30] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In *ACM CCS*, 2013.
- [31] Brian N. Levine, Michael K. Reiter, Chenxi Wang, and Matthew K. Wright. Timing attacks in low-latency mix-based systems. In *FC*, 2004.
- [32] Vincent Liu, Seungyeop Han, Arvind Krishnamurthy, and Thomas Anderson. Tor instead of IP. In *ACM HotNets*, 2011.
- [33] P. Mahadevan, D. Krioukov, M. Fomenkov, B. Huffaker, X. Dimitropoulos, K. Claffy, and A. Vahdat. The Internet AS-Level Topology: Three Data Sources and One Definitive Metric. In *ACM SIGCOMM*, 2006.
- [34] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *PETS*, 2004.
- [35] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security*, 2011.
- [36] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster Protocol v. 2. IETF Draft, 2003.
- [37] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. IETF RFC 4423.
- [38] Steven J. Murdoch and Piotr Zieliński. Sampled traffic analysis by Internet-exchange-level adversaries. In *PETS*, 2007.
- [39] Andreas Pfizmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In *PETS*, 2001.
- [40] Ryan Pries, Wei Yu, Xinwen Fu, and Wei Zhao. A new replay attack against anonymous communication networks. In *IEEE ICC*, 2008.
- [41] Michael G. Reed, Paul F. Syverson, and M. Goldschlag David. Anonymous connections and onion routing. *IEEE JSAC*, 1998.
- [42] Jody Sankey and Matthew Wright. Dovetail: Stronger anonymity in next-generation internet routing. In *PETS*, 2014.
- [43] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. In *ESORICS*, 2003.
- [44] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. In *USENIX Security*, 2000.
- [45] Wei Wang, Mehul Motani, and Vikram Srinivasan. Dependent link padding algorithms for low latency anonymity systems. In *ACM CCS*, 2008.

- [46] Xiaowei Yang, David Clark, and Arthur W Berger. Nira: a new inter-domain routing architecture. *IEEE/ACM Transactions on Networking*, 2007.
- [47] Bassam Zantout and Ramzi Haraty. I2P data communication system. In *ICN*, 2011.
- [48] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *IEEE S&P*, 2011.