MinimaLT: Minimal-latency Networking Through Better Security

W. Michael Petullo, mike@flyn.org
Xu Zhang, xzhang54@uic.edu
Jon A. Solworth, solworth@rites.uic.edu
University of Illinois at Chicago, USA

Daniel J. Bernstein, djb@cr.yp.to

Tanja Lange, tanja@hyperelliptic.org
TU Eindhoven, Netherlands

ABSTRACT

Minimal Latency Tunneling (MinimaLT) is a new network protocol that provides ubiquitous encryption for maximal confidentiality, including protecting packet headers. MinimaLT provides server and user authentication, extensive Denial-of-Service protections, and IP mobility while approaching perfect forward secrecy. We describe the protocol, demonstrate its performance relative to TLS and unencrypted TCP/IP, and analyze its protections, including its resilience against DoS attacks [56]. By exploiting the properties of its cryptographic protections, MinimaLT is able to eliminate three-way handshakes and thus create connections faster than unencrypted TCP/IP.

1 Introduction

Our goal is to protect *all* networking against eavesdropping, modification, and, to the extent possible, Denial of Service (DoS). To achieve this goal, networking must protect privacy, perform well, provide strong (i.e., cryptographic) authentication, and be easy to configure. These needs are not met by existing protocols.

Hardware and software improvements have eliminated historical cryptographic performance bottlenecks. Now, strong symmetric encryption can be performed on a single CPU core at Gb/s rates [45], even on resource-constrained mobile devices [13]. Public-key cryptography, once so agonizingly slow that systems would try to simulate it with symmetric key cryptography [43], is now performed at tens of thousands of operations per second on commodity CPUs. But one performance parameter is a fundamental limitation—network latency [33]. Latency is critical for users [58]. For example, Google found that a 500ms latency increase resulted in a 25% dropoff in page searches, and studies have shown that user experience degrades when interfaces incur latencies as small as 100ms [17]. This has prompted several efforts to reduce TCP and TLS latency [18, 46, 44, 53, 59].

We describe here MinimaLT, a secure network protocol which delivers protected data on the first packet of a typical client-server connection. MinimaLT provides substantial protections and is extraordinarily simple to configure and use. In particular, it provides cryptographic authentication of servers and users; encryption of communication; simplicity of protocol, implementation, and configuration; clean IP-address mobility; and DoS protections.

MinimaLT's design intentionally crosses network layers. It does so for two reasons: First, security problems often occur in the seams between layers. For example, Transport Layer Security (TLS) is unable to protect against attacks on TCP/IP headers due to its layering; RST and sequence number attacks interrupt TLS connections in such a way that is difficult to correct or even detect [23, 4, 62]. Second, multilayer design enables MinimaLT to improve performance.

Particularly challenging has been to provide Perfect Forward Secrecy (PFS) at low latency. PFS means that even an attacker who captures network traffic and later obtains all long-term private keys cannot decrypt past packets or identify the parties involved in communication. Traditionally, using Diffie-Hellman key exchange (DH) to achieve PFS requires a round trip before sending any sensitive data. MinimaLT eliminates this roundtrip, instead receiving the server's ephemeral key during a directory service lookup (§4.3.1). To establish connection liveness—necessary only if the connection is inactive and the server is running out of memory—we invert the normal mandatory start-of-connection handshake and replace it with an only-when-needed server-originated handshake (§4.3.4). Eliminating roundtrips makes MinimaLT *faster than unencrypted TCP/IP at establishing connections.*

A second challenge is to make connections portable across IP addresses to better support mobile computing. MinimaLT allows you to start a connection from home, travel to work, and continue to use that connection. This avoids application recovery overhead and lost work for operations which would otherwise be interrupted by a move. MinimaLT IP mobility does not require intermediary hosts or redirects, allowing it to integrate cleanly into protocol processing (§4.3.3). To provide better privacy, MinimaLT blinds third parties to IP-address portability to prevent linking a connection across different IP addresses.

A third challenge is DoS. A single host cannot thwart an attacker with overwhelming resources [36], but MinimaLT protects against attackers with fewer resources. In particular, MinimaLT dynamically increases the ratio of client (i.e., attacker) to server resources needed for a successful attack. MinimaLT deploys a variety of defenses to maintain proportional resource usage between a client and server (§5.3).

A fourth challenge is authentication and authorization. Experience indicates that network-based password authentication is fraught with security problems [34, 57, 47, 16], and that cryptographic authentication is needed. Our authentication framework supports both identified and non-identified (pseudonym) users (§4.1.1). We designed MinimaLT to integrate into systems with strong authorization controls.

To meet these challenges, we have done a clean-slate design, starting from User Datagram Protocol (UDP), and concurrently considering multiple network layers. We found an unexpected synergy between speed and security. The reason that the Internet uses higher-latency protocols is that, historically, low-latency protocols such as T/TCP have allowed such severe attacks [18] as to make them undeployable. It turns out that providing strong authentication elsewhere in the protocol stops all such attacks without adding latency.

In short, MinimaLT provides the features of TCP/IP (reliability, flow control, and congestion control), and adds in encryption, authentication, clean IP mobility, and DoS protections, all while preserving PFS and reducing latency costs. We describe MinimaLT and its implementation here. In §2 we describe related work, §3 describes our threat model, §4 describes the design, and §5 provides an evaluation of the security and performance of MinimaLT.

2 Related work

TLS provides cryptographic network protections above the transport layer and is normally implemented as a user-space library [19]. TLS is widely deployed as the primary network security layer in web browsers, yet many Internet applications avoid the use of TLS or use weak TLS options [61]. Even well-meaning developers routinely misuse complex TLS Application Programming Interfaces (APIs), resulting in security holes [31, 26]. Optionally, TLS can provide user-level authentication using client-side certificates but authorization is left to application logic. ForceHTTPS [38] and HTTPS Everywhere [25] attempt to make TLS/HTTP more robust; MinimaLT forgoes backwards compatibility to provide a simpler, less mistake-prone platform.

There have been many attempts to reduce the latency incurred by TLS and TCP. Recently, False Start (no longer used), Snap Start, and certificate pre-fetching have accelerated establishing a TLS session [46, 44, 59]. TCP Fast Open (TFO) [53] clients may request a TFO cookie that allows them to forgo the three-way handshake on future connections. Since any client may request a TFO cookie, a client may spoof its sending Internet Protocol (IP) address to mount a DoS attack against a server; under this condition, the server must again require a three-way handshake. To benefit from TFO, a server application must be idempotent, a requirement that MinimaLT does not impose. Datagram Transport Layer Security (DTLS) [54] provides TLS protections on top of UDP, which is useful when reliability is not needed. However, DTLS shares TLS' initial handshake latency.

Like MinimaLT, tcpcrypt [15] investigated ubiquitous encryption, but it maintains backwards compatibility with TCP/IP. Tcpcrypt provides hooks that applications may use to provide authentication services and determine whether a channel is encrypted. MinimaLT's approach is different; it is clean-slate and eases host assurance by moving authentication and encryption services to the system layer.

Internet Protocol Security (IPsec) provides very broad confidentiality and integrity protections because it is generally implemented in the Operating System (OS) kernel. For example, IPsec can be configured such that *all* communication between node A and node B is protected. This universality simplifies assurance. Many key management protocols have been proposed for IPsec; we were particularly inspired by Just Fast Keying due to its simplicity, focus on PFS, and DoS resilience [1]. IPsec's major shortcoming is that its protections stop at the host; it focuses on network isolation and host authentication/authorization. For example, IPsec does not authenticate or restrict users across the network.

Labeled IPsec [39] combines IPsec and Security-Enhanced Linux (SELinux) [49] to provide more comprehensive network protections. Using a domain-wide authorization policy, the system (1) associates SELinux labels with IPsec security associations, (2) limits a process' security associations (connections) using a kernel authorization policy, and (3) employs a modified `inetd` that executes worker processes in a security domain corresponding to the label associated with an incoming request. In this manner, labeled IPsec can solve many of the authentication deficiencies in plain IPsec. However, labeled IPsec depends on a verified Trusted Computing Base (TCB) and enforcement policy; it builds upon the Linux kernel, SELinux, and IPsec, each of which are very complex. Furthermore, IPsec security association granularity limits the granularity of controls in labeled IPsec. In contrast, MinimaLT is designed from scratch, significantly simplifying policy specification, implementation, and use.

Stream Control Transport Protocol (SCTP) is a transport-layer protocol that provides reliable delivery and congestion control [60]. SCTP differs from TCP in that it can bundle messages from multiple applications (i.e., *chunks*) into a single packet. MinimaLT borrows this technique. Structured Stream Transport (SST) [29] allows applications to associate lightweight network streams with an existing stream, reducing the number of three-way handshakes incurred by applications and providing semantics useful for applications that use both data and control connections. MinimaLT eliminates the handshake on even the first connection, and MinimaLT's tunnels do not require a programmer's explicit use of a lightweight stream API.

Table 1 compares several network protocols with MinimaLT. MinimaLT is unique in that it provides encryption and authentication with PFS while allowing a client to include data in the first packet sent to a server (often forgoing pre-transmission round trips entirely). MinimaLT is also notable in that it includes robust DoS protections directly in the protocol.

3 Threat model

We are concerned with the confidentiality and integrity of network traffic in the presence of an attacker that can observe or modify arbitrary packets—including a Man-in-the-Middle (MitM). Confidentiality and integrity attacks, mounted by both known and anonymous users should be thwarted. An attacker who gains complete control over clients and servers, through physical access or otherwise, can decrypt very recent and future packets but should still be unable to decrypt older packets.

DoS attacks from *known* users are expected to be addressed through de-authorizing abusive users or non-technical means. An *anonymous* attacker might try to affect availability, through transmission-, computation-, and memory-based DoS. An attacker with enough resources (or control over the network) can always affect availability, so we attempt to drive up his costs by making his attack at least as expensive as the cost to defend against it. Here we want to address equally non-MitM and MitM attackers. That is, the ability to spoof the source IP address of a packet *and* capture a reply should not allow much easier attacks.

4 Design

4.1 Overview

We begin with a high-level introduction of the MinimaLT protocol design. MinimaLT identifies hosts using public-key cryptography; multiplexes multiple authenticated user-to-service connections within a single encrypted host-to-host tunnel; lowers latency by replacing setup handshakes with the publication of ephemeral keys in a directory service; and builds on carefully designed cryptographic abstractions.

4.1.1 Public key MinimaLT is decidedly public-key-based. Both servers and users are identified by their public keys; such keys serve as a Universally Unique ID (UUID) [64, 55, 42]. Principals prove their identity by providing ciphertext which depends on both their and the server's

	TCP/IP	TCP Fast Open	SST	IPsec	Labeled IPsec	TLS	False Start	Snap Start	Tcpcrypt	MinimaLT
Encrypt			✓	✓	✓	✓	✓	✓	✓	✓
PFS				✓	✓	✓	✓	✓	✓	✓
User authentication					✓	✓	✓	✓	✓	✓
Robust DoS protections										✓
Round trips before client sends data*	2	2	2	$\geq$4	$\geq$4	4	3	2	3	1
... if server is already known	1	1	1	$\geq$3	$\geq$3	3	2	1	2	0
... in abbreviated case†	1	0	0	1	1	2	2	1	1	0

*Includes one round trip for DNS/directory service lookup of unknown server
†Assumes protocol-specific information cached from previous connection to same server

Table 1: Comparison of MinimaLT with other network protocols

keys. A principal may be known—i.e., the underlying OS is aware of a real-world identity associated with the principal's public key—or he may be a **stranger**—a user whose real world identity is unknown. We consider a stranger who produces a new identity for each request **anonymous**. Whether strangers or anonymous users are allowed is left to the underlying system's authorization policy.

4.1.2 Network tunnel A MinimaLT **tunnel** is a point-to-point entity that encapsulates the set of connections between two hosts. MinimaLT creates a tunnel on demand in response to the first packet received from a host or a local application's outgoing connection request. Tunnels provide server authentication, encryption, congestion control, and reliability; unlike with TLS/TCP, these services are not repeated for each individual connection.

Tunnels make it more difficult for an attacker to use traffic analysis to infer information about communicating parties. Of course, traffic analysis countermeasures have limits [48]; for obvious cost reasons we did not include extreme protections against traffic analysis, such as using white noise to maintain a constant transmission rate.

As with IPsec, a MinimaLT tunnel provides cryptographic properties that ensure confidentiality and integrity. However, MinimaLT has been designed to provide tighter guarantees than IPsec, which (as generally used in practice) provides host-based protections only. In MinimaLT, connections provide user authentication as described below.

4.1.3 Connections A MinimaLT tunnel contains a set of **connections**, that is, a single tunnel between two hosts encapsulates an arbitrary number of connections. Each connection is user-authenticated and provides two-way communication between a client application and service. In addition to multiplexing any number of standard application-to-service connections, each MinimaLT tunnel has a single **control connection**, along which administrative requests flow (§4.2.3).

4.1.4 Directory service Central to our protocol is the MinimaLT **directory service**. The directory service resolves queries for (server) hostname information. It provides the server's directory certificate, signed by the server's long-term key. This returned certificate contains the server's IP address, UDP port, long-term key, zero padding (the minimum payload size of the initial packet), and a server ephemeral key.

An additional certificate vouches for the server's long-term public key and ties it with the server's hostname. Servers register with a MinimaLT directory service to provide this information, and they update their ephemeral key at a rate depending on their security requirements. In §4.4 we describe how directory services integrate with DNS to span the Internet.

4.1.5 Cryptographic abstractions TLS builds on separate cryptographic primitives for public-key cryptography, secret-key encryption, etc. Unfortunately, composing these low-level primitives turns out to be complicated and error-prone. For example, the BEAST attack [22] and the very recent Lucky 13 attack [2] recovered TLS-encrypted cookies by exploiting the fragility of the "authenticate-pad-encrypt" mechanism used by TLS to combine secret-key encryption with secret-key authentication. TLS implementations have worked around these particular attacks by (1) sending extra packets to hide the "IV" used by BEAST and (2) modifying implementations to hide the timing leaks used by Lucky 13; however, further attacks would be unsurprising.

The modern trend is for cryptographers to take responsibility for providing secure higher-level primitives. For example, cryptographers have defined robust high-performance "AEAD" primitives that handle authentication and encryption all at once using a shared secret key [50], taking care of many important details such as padding and key derivation. This simplifies protocol design, eliminating the error-prone step of having each protocol combine separate mechanisms for authentication and encryption. TLS 1.2 (not yet widely deployed) supports AEAD primitives.

MinimaLT is built on top of an even higher-level primitive, *public-key* authenticated encryption (see, e.g., [11, 12, 40, 30]), protecting both confidentiality and integrity of messages sent from one public key to another. Our implementation of MinimaLT uses the high-performance high-security NaCl library [12], because it provides exactly this primitive. NaCl's `box` encrypts and authenticates a plaintext using the sender's private key, the receiver's public key, and a number-used-once (nonce); `box_open` verifies and decrypts the ciphertext using the sender's public key, the receiver's private key, and the same nonce. See Section 5.4 for underlying cryptographic details.

4.2 Packet format

The MinimaLT packet format can be broken into three conceptual layers: (1) **delivery**, routing and other information necessary to deliver a packet to its destination host; (2) **tunnel**, server authentication, reliability, and encryption; and (3) **connection**, user authentication and application-to-service multiplexing.

The packet format is simple and is given in Figure 1 and Table 2. The cleartext portion of the packet contains the Ethernet, IP[1], and UDP headers; the Tunnel ID (TID), a

[1]The MinimaLT protocol details are orthogonal to the

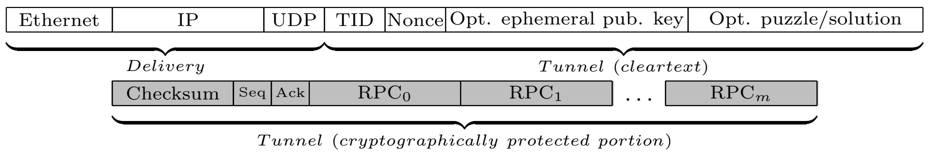


Figure 1: Packet format with cleartext in white and cryptographically protected portion in gray

		Size (bytes)	
	Field	**First**	**Successive**
Deliv.	Ethernet Header	14	14
	IP	20	20
	UDP	8	8
Crypto.	Tunnel ID	8	8
	Nonce	8	8
	Ephemeral public key	32	n/a
	Puzzle/solution	148	n/a
	Checksum	16	16
Rel.	Sequence Num.	4	4
	Acknowledgment	4	4
Con.	Connection ID	4	4
	RPC	variable	variable
	Total (except RPC)	282	86

Table 2: Tunnel's first/successive packets

nonce; and two optional fields used at tunnel initiation—an ephemeral public key and a puzzle. A client provides the public key only on the first packet for a tunnel, and a server requires puzzles opportunistically to prevent memory and computation attacks.

The packet's cryptographically protected portion contains ciphertext and the ciphertext's cryptographic checksum. The ciphertext contains a sequence number, acknowledgment number, and a series of Remote Procedure Calls (RPCs). In contrast to byte-oriented protocols, we use RPCs at this layer because they result in a clean design and implementation; they are also general and fast (§5). RPCs have a long history dating to the mid-1970s [63, 14].

4.2.1 Delivery layer The usual Ethernet, IP, and UDP headers allow the delivery of packets across existing network infrastructure; they play no role in any packet processing within MinimaLT. The UDP header allows packets to traverse NATed networks [24], and it enables user-space implementations of MinimaLT. Aside from the length field, the UDP fields are otherwise uninteresting for MinimaLT.

4.2.2 Tunnel layer (cryptography and reliability) The tunnel establishment packet (the first packet sent between two hosts) contains a TID, a nonce, and the sending host's (ephemeral) public DH key. The TID is pseudorandomly generated by the initiator. The public key is ephemeral to avoid identifying the client host to a third party[2].

The recipient cryptographically combines the client's ephemeral public key with its own ephemeral private key to generate the symmetric key (§4.3). The recipient then uses this symmetric key and the nonce to verify and decrypt the encrypted portion of the packet.

After tunnel establishment, the tunnel is identified by its TID. Successive packets embed the TID, which the recipient uses to look up the symmetric key and decrypt the payload. Thus MinimaLT reduces packet overhead by using TIDs rather than resending the public key. The TID is 64 bits—1/4 the size of a public key—with one bit indicating the presence of a public key in the packet, one bit indicating the presence of a puzzle/solution, and 62 bits identifying a tunnel.

structure of IPv4/v6 addresses.

[2]Of course, a client can sometimes be identified by IP; here onion routing can help [20].

The nonce ensures that each payload transmitted between two hosts is uniquely encrypted. The nonce is a monotonically increasing value; once used, it is never repeated for that (unordered) pair of keys. The nonce is odd in one direction (from the side with the smaller key) and even in the other direction, so there is no risk of the two sides generating the same nonce. Clients enforce key uniqueness by randomly generating a new ephemeral public key for each new tunnel; this is a low-cost operation. For a host which operates as both client and server, its client ephemeral key is in addition to (and different from) its server ephemeral key.

The tunnel layer also contains an optional field that might contain a puzzle request or solution (§4.3.1). The purpose of the puzzle here is to protect against spoofed tunnel requests. Such an attack might cause a server to perform relatively expensive DH computations. A puzzle both demonstrates connectivity and expends initiator resources, and thus limits attack rates. (In addition to the puzzle header fields, MinimaLT provides puzzle RPCs to defend against Sybil attacks [21, 41] after a tunnel has been established. We describe the details of both in §4.3.4 and evaluate them in §5.3.)

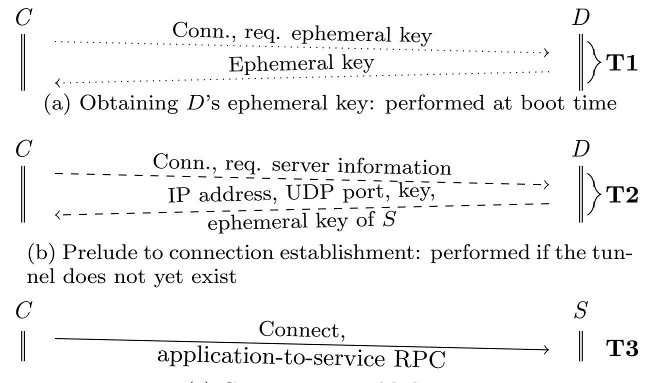
Beyond the cleartext fields mentioned so far, the tunnel layer contains a strong 128-bit cryptographic checksum (a keyed message-authentication code) of the ciphertext. An attacker does not have the shared symmetric key, so any attempt to fabricate ciphertext will be detected.

The final component of the tunnel header consists of reliability information in the form of sequence and acknowledge fields. MinimaLT's reliability and connection headers are part of the ciphertext, so they are protected against tampering and eavesdropping. The total size of the delivery, reliability, and connection headers is equal to that of the headers in TCP/IP. We discuss packet overhead further in §5.1.

4.2.3 Connection layer The connection layer supports an arbitrary number of connections, where each connection hosts a series of RPCs. An RPC is of the form $f_c(a_0, a_1, \ldots)$, where f is the name of the remote function, c is the connection that the RPC is sent to, and $a_0, a_1, \ldots$ are its arguments. On the wire this is encoded as $c, f, a_0, a_1, \ldots$ A single packet can contain multiple RPCs; this amortizes the overhead due to MinimaLT's delivery and tunnel fields across multiple connections.

One connection is distinguished: connection 0 is the control connection, which hosts all management operations. These include authenticating users (§4.3.6); creating, closing, accepting, and refusing connections; providing certificates (§4.3.1); rekeying (§4.3.2); IP address changes (§4.3.3); and puzzles (§4.3.4). Here we reference:

RPC	**Description**
$\mathsf{create}_0(c,\ s)$	create an anonymous connection c of type s


C
D
Conn., req. ephemeral key
Ephemeral key
T1
C
D
Conn., req. server information
IP address, UDP port, key,
ephemeral key of S
T2
C
S
Connect,
application-to-service RPC
T3

(a) Obtaining D's ephemeral key: performed at boot time

(b) Prelude to connection establishment: performed if the tunnel does not yet exist

(c) Connection establishment

Figure 2: MinimaLT protocol trace

$\mathsf{createAuth}_0(c, s, U, x)$	create an authenticated connection for the user with long-term public key U, who generates authenticator x
$\mathsf{close}_0(c)$	close connection c
$\mathsf{ack}_0(c)$	creation of c successful
$\mathsf{refuse}_0(c)$	connection c refused
$\mathsf{requestCert}_0(H)$	get host H's certificate
$\mathsf{provideCert}_0(X)$	provide the certificate X
$\mathsf{ok}_0()$	last request was OK
$\mathsf{nextTid}_0(t, C')$	advertise future TID to prepare for a rekey or IP address change
$\mathsf{puzzle}_0(p, h, w)$	pose a puzzle
$\mathsf{puzzleSoln}_0(p, h, w)$	provide a puzzle solution
$\mathsf{windowSize}_0(c, n)$	adjust connection receive window

The control connection RPCs maintain the tunnel and its other connections. All data on connections other than the control connection are sent unchanged to their corresponding applications.

In general, each service provided by a host supports a set of service-specific RPCs on standard connections. Our illustrations use the following sample RPC:

$\mathsf{serviceRequest}_c(...)$	a request for some type of service on connection c

The tunnel, and the RPCs within it, are totally sequenced. Thus RPCs are executed in order (as opposed to separately implemented connections—as in TLS—where ordering between connections is not fixed). This enables a clean separation of the control connection from other connections. We have found that this simplifies both the protocol and its implementation.

4.3 Protocol

The purpose of the protocol is to allow protected communication between a client and server. We discuss here symmetric key establishment, rekeying, IP-address mobility, puzzles, the absence of a three-way handshake, user authenticators, and congestion control.

4.3.1 Establishing the symmetric key MinimaLT approaches PFS, encrypting sensitive data using only ephemeral keys; it protects both client-side data and identity. In this section, we will show the negotiation of keys and the transmission of RPCs in the absence of puzzles. This is the normal case, when servers are not under heavy load.

At least four entities cooperate to establish a symmetric key: a client C which wants to communicate with server S, a directory service D with which C communicates, and an ephemeral key upload service E with which S communicates. Here we discuss the intra-organizational case, where a single organization maintains C, D, S, and E; for such a deployment D and E can be the same server. In §4.4 we show how MinimaLT scales to the Internet using DNS, while providing security at least as strong as DNSSEC with no additional latency. Eventually, a pure MinimaLT solution could span the entire Internet.

The client can compute the shared secret after a maximum of two round trips, and can include application data in the first packet sent to the server. (The common case is that the client can immediately compute the shared secret with zero round trips.) Each trip uses a different tunnel:

T1 C establishes a tunnel, anonymously, to D in order to obtain D's ephemeral public key;

T2 C establishes a tunnel to D using ephemeral keys to lookup S's contact information; and

T3 C establishes a tunnel to S using ephemeral keys.

Figure 2 depicts this process. C establishes tunnel T1 once, at boot time. This is the only tunnel that does not use a server ephemeral key, so C does not yet provide D with a user authenticator. Next, C establishes tunnel T2 to collect the information necessary for the first connection between C and S. It uses this information to establish tunnel T3. The tunnel establishment packet for tunnel T3 may include application-to-service RPCs. Successive connections to S skip T1 and T2, and tunnel T2 remains open to look up other servers. We use the following to describe the details:

t	A tunnel ID (described in §4.2)
n	A nonce (described in §4.2)
s	A sequence number
a	An acknowledgment number
0 or c	The connection ID
z	A puzzle
z'	A puzzle solution
C, D, E, S	The client, directory, upload, and server long-term public/private key
C', D', E', S'	An ephemeral client, directory, upload, and server public/private key
$C \to S$	A message from the client to the server, using keys C and S
$H(m)$	The cryptographic hash of message m
$\boxed{m}\,^{k}_{n}$	Encrypt and authenticate message m using symmetric key k and nonce n
$\boxed{m}\,^{S\to P}_{n}$	Encrypt and authenticate message m using a symmetric key derived from private key S and public key P; n is a nonce

We show each packet on a single line such as

$$t, n, C', \boxed{s, a, \ldots}\,^{C'\to S'}_{n}$$

which indicates a tunnel establishment packet (due to the presence of the unencrypted C', as described in §4.2.2) from C to S using keys C' and S' to box (encrypt and authenticate) the message '$s, a, \ldots$'. Each packet has a new nonce but for conciseness we simply write n rather than n_1, n_2, etc. The same comment applies for sequence numbers (s) and acknowledgements (a).

Communication of C with D At boot time, C establishes a tunnel with D (Figure 2a, tunnel T1); C's configuration contains D's IP address, UDP port, and long-term public key D. First, C generates a single-use public key C' and uses it to create a bootstrap tunnel with the directory service.

$$t, n, C', \boxed{s, a, \mathsf{requestCert}_0(D)} \quad \underset{n}{C' \to D}$$

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Dcert})} \quad \underset{n}{D \to C'}$$

D responds with a certificate containing its own ephemeral key, D', signed by D. C uses this to establish a PFS tunnel to request S's directory certificate. Tunnel T2 uses a fresh C' and is established by:

$$t, n, C', \boxed{s, a, \mathsf{requestCert}_0(S)} \quad \underset{n}{C' \to D'}$$

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Scert})} \quad \underset{n}{D' \to C'}$$

Communication of C with S After receiving S's Scert, C is ready to negotiate tunnel T3. C encrypts packets to the server using S' (from Scert) and a fresh C'. Because C places its ephemeral public key in the first packet, both C and S can immediately generate a shared symmetric key using DH without compromising PFS. Thus C can include application-to-service data in the first packet. That is,

$$t, n, C', \boxed{\begin{array}{l} s,\ a,\ \mathsf{nextTid}_0(t, C'), \\ \mathsf{createAuth}_0(1,\ \text{serviceName},\ U,\ x), \\ \mathsf{serviceRequest}_1(\ldots) \end{array}} \quad \underset{n}{C' \to S'}$$

We describe the purpose of $\mathsf{nextTid}_0$ in §4.3.2. Upon receiving $\mathsf{createAuth}_0$, S verifies the authenticator x (§4.3.6) and decides if the client user U is authorized to connect. If so, S creates the new connection (with ID 1). The server ensures no two tunnels share the same C'. The service-specific $\mathsf{serviceRequest}_1$ can then be processed immediately on the new connection.

Tunnels are independent of the IP address of C; this means that C can resume a tunnel after moving to a new location (typically prompting the use of the next ephemeral key as described below), avoiding tunnel-establishment latency and application-level recovery due to a failed connection. This reduced latency is useful for mobile applications, in which IP-address assignments may be short-lived, and thus overhead may prevent any useful work from being done before an address is lost.

Registering an ephemeral key Before a client may connect, S must register its own IP address, UDP port, public key, and current ephemeral public key with an upload service E. (E is the same as D in the local case, and we describe how E supports Internet-spanning lookups in §4.4.) This is done using the $\mathsf{provideCert}_0$ RPC:

$$t, n, \boxed{s, a, \mathsf{provideCert}_0(\mathsf{Scert})} \quad \underset{n}{S' \to E'}$$

$$t, n, \boxed{s, a, \mathsf{ok}_0()} \quad \underset{n}{E' \to S'}$$

4.3.2 Rekey PFS requires periodic rekeying, so that old encryption keys can be forgotten and are thus denied to an attacker who later compromises clients and servers. Prior to a rekey, a host creates the next TID t and sends it to the opposite host using $\mathsf{nextTid}_0$. (We describe the C' argument to $\mathsf{nextTid}_0$ below.) Although either side may invoke $\mathsf{nextTid}_0$, rekeying is initiated only by the client.

We reduce the rekeying workload for the client and server as follows. To rekey, the client sends a tunnel initiation packet using the next TID. The client generates a one-time valid key pair used for this initiation and places the public part of this key pair in this packet so that it is indistinguishable from a true tunnel initiation packet. However, instead of computing a new shared secret using DH, the client simply uses the hash of the previous symmetric key. (In fact, the client makes no use of the private part of the key pair, so rather than generating a valid key pair it can perform a cheaper operation to generate a random public key as a random point on the elliptic curve used in NaCl, without ever knowing the corresponding private key.) The client repeats this one-time public key inside the boxed part of the message (i.e., as the C' argument to another $\mathsf{nextTid}_0$).

When the server receives a packet whose TID matches a known next TID, the server hashes the existing key for that tunnel to produce the new key, and then verifies and decrypts the packet. The server also verifies that the public key sent in clear matches the public key inside the boxed part of the message. (Without this verification, an active attacker could modify the public key sent in the clear, observe that the server still accepts this packet, and confidently conclude that this is a rekey rather than a new tunnel.) If both verifications succeed then the server updates the tunnel's TID and handles the packet; otherwise it behaves as for failed tunnel initialization. Thus the rekey process inflicts neither superfluous round trips nor server public-key operations.

Typically, clients invoke $\mathsf{nextTid}_0$ immediately after creating a new tunnel, and after assuming a new TID/key. Servers invoke $\mathsf{nextTid}_0$ if their PFS interval expires. Clients then assume a new TID/key when their rekey interval expires or immediately after receiving a $\mathsf{nextTid}_0$ from the server (the latter implies that the server's key has expired).

A client-side administrator sets his host's rekey interval as a matter of policy. The server's policy is slightly more sensitive, because the server must maintain its ephemeral key pairs as long as they are advertised through the directory service. An attacker who seizes a server could combine the ephemeral keys with captured packets to regenerate any symmetric key within the ephemeral key window. Thus even if the client causes a rekey, the server's ephemeral key window dominates on the server side. This asymmetry reflects reality, because each side is responsible for their own physical security. The client knows server policies and can restrict communication to acceptable servers.

4.3.3 IP-address mobility Because MinimaLT identifies tunnels by their TID, a tunnel's IP and UDP port can change without affecting communication; indeed, one purpose of $\mathsf{nextTid}_0$ is to support IP-address mobility. After changing its IP address or UDP port, a host simply assumes the next TID as with a rekey. The other host will recognize the new TID and will transition the tunnel to the new key, IP address, and UDP port. Thus a computer can be suspended at home and then brought up at work; an application which was in the middle of something could continue without any recovery actions.

MinimaLT reduces an attacker's ability to link tunnels across IP address changes because its TID changes when its IP address changes. What remains is temporal analysis, where an attacker notices that communication on one IP address stops at the same time that communication on another starts. However, the attacker cannot differentiate for sure between IP mobility and an unrelated tunnel establishment. Blinding information below the network layer—for example, the Ethernet MAC—is left to other techniques.

4.3.4 Puzzles MinimaLT uses puzzles selectively, so their costs are only incurred when the server is under load. There are two ways MinimaLT can pose puzzles: as a part of the tunnel header (to avoid abusive DH computations) or by using the puzzle RPCs after establishing a tunnel. In the former case, a MinimaLT server under load that receives a tunnel establishment packet from a stranger for an authorized service does not create a tunnel. Instead, it responds with a puzzle:

$$z' = \boxed{C', S'}\ {}^{k}_{n'}$$

where k is a secret known only to the server. It then calculates z'' by zeroing z''s rightmost w bits (i.e., the client will take $O(2^{w-1})$ operations to solve the puzzle), where the server dynamically selects w based on its policy. The server sends the puzzle $z = [z'', H(z'), w, n']$ to the client:

$$t, n, z$$

The server forgets about the client's request. The client must solve the puzzle z and provide the solution z' along with n' in a new tunnel establishment packet using the same C' and S'. The client brute forces the rightmost w bits of z'' to find z' with a matching hash and responds to the server with:

$$t, n, [z', n'], \boxed{s, a, \ldots}\ {}^{C' \to S'}_{n}$$

To confirm a solution, the server decrypts z' using k and n', confirms C' and S' and ensures that n' is within an acceptable window. Although the server has forgotten z' these protections ensure that the puzzle cannot be reused for other tunnel establishment attempts.

Once a tunnel is established, hosts can use the puzzle RPCs to perform small-w proof-of-life/liveness challenges on idle tunnels that might be suitable for garbage collection. Stranger-authorized servers can also use the puzzle RPCs to slow Sybil attacks, whereby an attacker tries to generate many identities to cause public-key authenticator validations on the server. We evaluate MinimaLT's puzzles in §5.3.

4.3.5 No transport-layer three-way handshake As described above, MinimaLT establishes an ephemeral symmetric key with a minimal number of round trips; MinimaLT also subsumes the need for a transport-layer three-way handshake when establishing each application-to-service connection. TCP's three-way handshake establishes a random Initial Sequence Number (ISN). This is necessary for two reasons: (1) the ISN serves as a weak authenticator (and liveness check) because a non-MitM attacker must predict it to produce counterfeit packets, and (2) the ISN reduces the likelihood that a late packet will be delivered to the wrong application.

MinimaLT encrypts the sequence number, provides cryptographic authentication, and checks liveness using puzzles, addressing (1). MinimaLT uses TIDs, connection IDs, and nonces to detect late packets, addressing (2). Thus MinimaLT can include application data in a connection's first packet, as discussed above. Extra round trips are necessary only if the tunnel does not exist; and then only when the client does not have S's directory certificate or is presented with a puzzle. If the server provides a puzzle, it means that the server is under heavy load so that additional latency is unavoidable.

4.3.6 User authenticators Every user serviced by MinimaLT is identified by his public key. The $\mathsf{createAuth}_0$ authenticator is the server's long-term public key encrypted and authenticated using the server's long-term public key, the user's long-term private key U, and a fresh nonce n:

$$x = \boxed{S}\ {}^{U \to S}_{n}$$

Because authenticators are transmitted inside boxes (as ciphertext), they are protected from eavesdropping, and because the authenticator is tied to the server's public key, the server cannot use it to masquerade as the user to a third-party MinimaLT host. Of course, any server could choose to ignore the authenticator or perform operations the client did not request, but that is unavoidable. If third-party auditability is desired then users can choose to interact only with servers that take requests in the form of certificates.

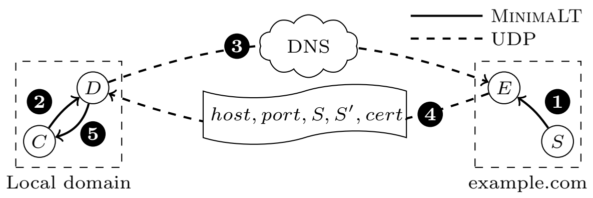


Figure 3: An external directory service query

4.3.7 Congestion/Flow control MinimaLT's tunnel headers contain the fields necessary to implement congestion control, namely sequence number and acknowledgment fields. We presently use a variation of TCP's standard algorithms [27]. As with TCP [32], efficient congestion control is an area of open research [28], and we could substitute an emerging algorithm with better performance. MinimaLT does have one considerable effect on congestion control: controls are aggregated for all connections in a tunnel, rather than on individual connections. Since a single packet can contain data for several connections, the server no longer needs to allocate separate storage for tracking the reliability of each connection. This also means that MinimaLT need not repeat the discovery of the appropriate transmission rate for each new connection, and a host has more information (i.e., multiple connections) from which to derive an appropriate rate. The disadvantage is that a lost packet can affect all connections in aggregate.

MinimaLT hosts adjust per-connection flow control using the $\mathsf{windowSize}_0$ RPC. MinimaLT subjects individual connections to flow control, so $\mathsf{windowSize}_0$ takes as parameters both a connection ID and size. MinimaLT currently implements TCP-style flow control.

4.4 A directory service that spans the Internet

Within an organization, an administrator maintains clients, servers, and a directory service. However, clients will often want to connect to services outside of their organization, so it becomes necessary to obtain external servers' directory service records. MinimaLT integrates disparate directory services using DNS in a way that does not add latency to the current requirement of performing a DNS lookup. MinimaLT directory services support their organization as described above, but also can make DNS queries about external hosts and can service DNS queries about local hosts.

The following specific mechanism is designed for easy deployability on the Internet today while guaranteeing at least as much security as is currently obtained from the X.509 PKI used in TLS. In particular, C checks an X.509 certificate chain leading to the long-term public key for S, the same way that web browsers today check such chains. We transmit this chain through DNS, obtaining three benefits compared to transmitting the chain later in the protocol:

- The chain automatically takes advantage of DNS caching.
- Even in non-cached cases the latency of transmitting the chain is usually overlapped with existing latency

for DNS queries.

- Any security added to DNS automatically creates an extra obstacle for the attacker, forcing the attacker to break *both* DNS security and X.509 security.

For comparison, if a client obtains merely an IP address from DNS and then requests an X.509 certificate chain from that IP address (the normal use of TLS today), then the attacker wins by breaking only X.509 security. If a client instead obtains the S public key from DNS as a replacement for X.509 certificate chains then the attacker wins by breaking only DNS security.

We depict an external lookup in Figure 3. As described in §4.3.1, servers such as S publish their ephemeral keys to their local upload service E (1). To connect to an external server, e.g., example.com's S, the client C requests S's information from C's directory service D (2), and, if cached, D immediately replies. Otherwise, D makes a DNS request for example.com's S (3). The DNS reply from example.com's DNS/upload service E is extended to contain a full MinimaLT server record, split into one long-term DNS record containing S's long-term key and a chain certifying the identity of this key, and one shorter-term DNS record containing an IP address, a UDP port, and an ephemeral server key, all signed by S. Once D receives this DNS reply (4), it can respond to C's request as earlier described (5).

The integration of MinimaLT's directory services with DNS affects DNS configurations in two ways. First, the shorter-term DNS record's time to live must be set to less than or equal to the rekey interval of the host it describes. We expect this to have a light impact, because most Internet traffic is to organizations that already select short times to live (e.g., 300 seconds for `www.yahoo.com`, 300 seconds for `www.google.com`, and 60 seconds for `www.amazon.com`). Second, DNS replies will grow due to the presence of additional fields. The largest impact is the identity certificate, which as mentioned above is encoded today as an X.509 certificate.

We have carefully separated the MinimaLT protocol per se, which describes how C–S, C–D, and S–E interact, from the use of DNS for the D–E interaction, and the use of X.509 for the certificate on S. We do not claim that DNS and X.509 are satisfactory from a performance perspective or from a security perspective, but improved systems and replacement systems will integrate trivially with MinimaLT.

5 Evaluation

Here we evaluate MinimaLT's packet overhead (§5.1); the performance of creating new tunnels, creating connections on existing tunnels, and transmitting data (§5.2); DoS defenses (§5.3); cryptography (§5.4); key isolation (§5.5); and prospects for further performance improvements (§5.6).

5.1 Packet header overhead

MinimaLT's network bandwidth overhead is modest. The overhead is due to the cryptography, and includes the nonce, TID and Checksum (the public key/puzzle fields are rarely present and are thus insignificant overall). MinimaLT requires 32 bytes more for its headers than TCP/IP; this represents 6% of the minimum Internet MTU of 576 bytes, and 2% of 1518-byte Ethernet packets.

5.2 Performance evaluation

We experimentally evaluate MinimaLT's performance in three areas: (1) the serial rate at which MinimaLT establishes tunnels/connections, primarily to study the effect of latency on the protocol; (2) the rate at which MinimaLT establishes tunnels/connections when servicing many clients in parallel; and (3) the throughput achieved by MinimaLT. All of our performance tests were run on two identical computers with a 4.3 GHz AMD FX-4170 quad-core processor, 16GB of memory, and a Gb/s Ethernet adapter. We benchmarked in 64-bit mode and on only one core to simplify cross-platform comparisons.

Serial tunnel/connection establishment latency In each of our serial connection benchmarks, a client sequentially connects to a server, sends a 28-byte application-layer request, and receives a 58-byte response. We measure the number of such operations completed in 30 seconds, where each measurement avoids a DNS/directory service lookup. We performed this experiment under a variety of network latencies using Linux's netem interface.

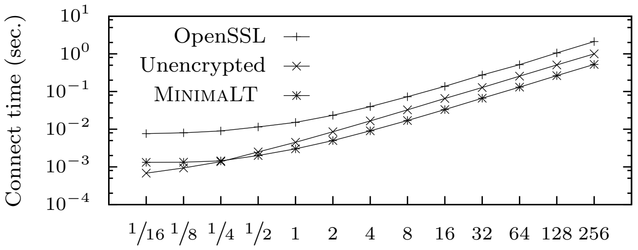
We compare against OpenSSL 1.0.0j using its s_server and s_time utilities, running on version 3.3.4 of the Linux kernel. We first configured OpenSSL to use 2,048-bit RSA as recommended by NIST [6] (although 2,048-bit RSA provides 112-bit security, less than that of MinimaLT, which provides 128-bit security), along with 128-bit AES, ephemeral DH, and client-side certificates. In order to ensure disk performance did not skew our results, we modified s_server to provide responses from memory instead of from the filesystem. We also wrote an unencrypted benchmark which behaves similarly, but makes direct use of the POSIX socket API, avoiding the use of cryptography.

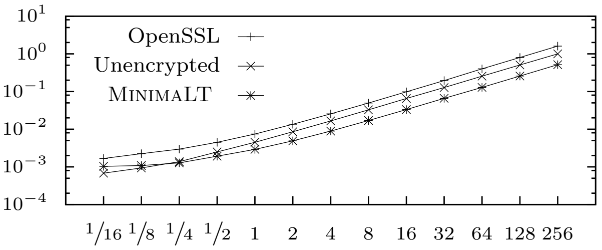
We benchmarked MinimaLT on Ethos, an experimental OS we have written to investigate robust security interfaces, because MinimaLT serves as Ethos' native network protocol (we have also ported MinimaLT to Linux). To produce results analogous to OpenSSL, simulating both (1) many abbreviated connection requests to one server and (2) many full connection requests to many servers, we tested both (1) the vanilla MinimaLT stack and (2) a MinimaLT stack we modified to artificially avoid tunnel reuse.

Figure 4a displays, in log scale, the rate of MinimaLT and OpenSSL when creating full, client-user-authenticated connections. For each connection, MinimaLT creates a new tunnel and authenticates the client user, and OpenSSL performs a full handshake; each requires public-key operations. At native LAN latencies plus $1/16$ ms (LAN+$1/16$ ms), MinimaLT took 1.32ms to complete a full connection, request, and response, and OpenSSL took 7.63ms. MinimaLT continued to outperform OpenSSL as network latency increased. At LAN+256ms, MinimaLT took 526.31ms, while OpenSSL took 2.13s.

Figure 4b displays abbreviated connection speed. In this case, MinimaLT reuses an already established tunnel and OpenSSL takes advantage of its session ID to execute an abbreviated connection. Here both systems avoid computing a shared secret using DH, except in the case of the first connection. At LAN+$1/16$ ms, MinimaLT took 1.03ms to complete a connection, request, and response over an existing tunnel. Under the same conditions, OpenSSL took 1.67ms to complete an abbreviated connection, request, and response. At LAN+256ms, MinimaLT took 517.24ms, while OpenSSL took 1.60s.

In all measurements, MinimaLT connections incur less latency than OpenSSL. More surprisingly, MinimaLT creates connections faster than raw TCP/IP, beginning before

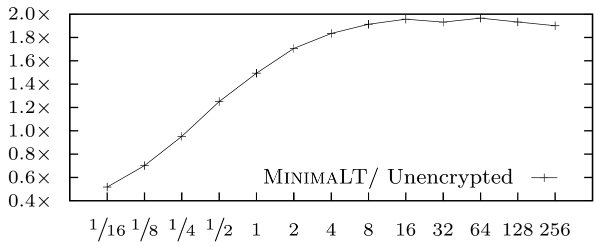


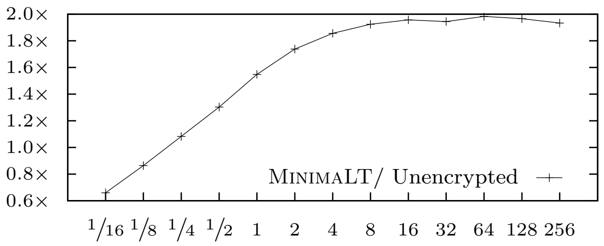


One-way additional simulated latency applied to network (ms); this is in addition to our native LAN latency of 1/13 ms

(a) Time spent creating a connection; full connections

(b) Time spent creating a connection; abbr. connections





One-way additional simulated latency applied to network (ms); this is in addition to our native LAN latency of 1/13 ms

(c) MinimaLT improvement over TCP/IP; full connections

(d) MinimaLT improvement over TCP/IP; abbr. connections

Figure 4: Serial tunnel/connection establishment latency

Tunnels per run	User Auth.	Connections per second	DH per conn.
One		18,453	0
One	✓	8,576	1
Many		7,827	1
Many	✓	4,967	2

Table 3: Connection establishment with many clients

System	Bytes per second
Line speed	125,000,000
Unencrypted	117,817,528
MinimaLT	113,945,258
OpenSSL	111,448,656

Table 4: Data throughput (ignoring protocol overhead)

LAN+1/4 ms latency in the case of abbreviated connections and beginning before LAN+1/2 ms latency otherwise. We experience latencies of at least this magnitude on any packets which leave our laboratory room (i.e., are processed by our router). Figures 4c and 4d show the ratio between MinimaLT and raw TCP/IP performance. We attribute our results to MinimaLT's efficient tunnel/connection establishment (especially at high latencies) and to the speed of the NaCl library (especially at low latencies).

Tunnel establishment throughput with many clients We created a second connection benchmark to estimate the CPU load on a MinimaLT server servicing many clients. To do this, we ran two client OS instances, each forking several processes that connected to the server repeatedly as new clients; here each virtual machine instance was running on a single computer. Because this experiment concerns CPU use and not latency, these clients do not write any application-layer data, they only connect. Using xenoprof and xentop, we determined that the server was crypto-bound (i.e., 63% of CPU use was in cryptography-related functions—primarily public key—and the server CPU load was nearly 100%). We measured the number of full connections per second achieved under this load, and varied our configuration from accepting fully-anonymous users (no authenticators), to verifying a new user authenticator for each connection request. MinimaLT established 4,967–7,827 tunnels per second, as shown in Table 3 (rows 3–4).

Given the minimal tunnel request size of 1,024 bytes, our hosts can (on a single core) service 61.15Mb/s of tunnel requests from anonymous users and at least 38.80Mb/s of tunnel requests from authenticated users. We note that this is the worst case for authenticated users. In general, we would cache the result of the DH computations necessary to validate user authenticators, as authenticators use long-term keys. Thus in practice we expect the authenticated user case to approach that of anonymous users.

Connection establishment throughput with many clients We repeated the previous experiment, but this time repeatedly used a single tunnel between each client and the server. Table 3 (rows 1–2) shows that our rates ranged from 8,576–18,453 per second, depending on the presence of user authenticators. The connection rate over a single tunnel is important for applications which require many connections and when using many applications to communicate with the same server. Our comments about cached DH results in the preceding experiment applies here as well; we would expect in practice the rate of the authenticated case will approach the anonymous case.

A theoretical throughput limit We used SUPERCOP [10] to measure the time it takes our hardware to compute a shared secret using DH, approximately 293,000 cycles or 14,000 operations per second. MinimaLT's tunnel establishment rate approaches 56% of this upper bound, with the remaining time including symmetric key cryptography, scheduling, and the network stack.

Single-connection data throughput Table 4 describes our throughput results, observed when running programs that continuously transmitted data on a single connection for thirty seconds. MinimaLT approaches the throughput achieved by unencrypted networking and runs at 91% of line speed (Gb/s). Indeed, MinimaLT's cryptography runs at line speed; header size differences are the primary reason the unencrypted benchmark outperforms MinimaLT.

5.3 Denial of service

DoS protections in MinimaLT are intended to maintain availability against much more severe attacks than are handled by current Internet protocols. Of course, an extremely powerful attacker will be able to overwhelm a MinimaLT server, but DoS protections are useful even in such extreme situations as a way to consume the attacker's resources and limit the number of DoS victims. Of particular concern are DoS attacks which consume memory or computational resources; the protocol cannot directly defend against network exhaustion attacks, although it can avoid contributing to such attacks by preventing amplification.

We introduced anonymous and stranger-authorized services in §4.1.1. Anonymous services (i.e., permit create_0) must perform a DH computation to compute a shared secret and maintain tunnel data structures that consume just under 5KB each (this is configurable, most memory use is due to incoming and outgoing packet buffers). Stranger-enabled services (i.e, require $\mathsf{createAuth}_0$, but permit strangers) must additionally perform a public-key decryption to validate each new user authenticator encountered.

5.3.1 Before establishing a tunnel In the case of anonymous services, a single attacker could employ a large number of ephemeral public keys to create many tunnels, with each tunnel requiring a DH computation and consuming memory on the server. Furthermore, the attacker's host might avoid creating a tunnel data structure or performing any cryptographic operations, thus making the attack affect the server disproportionately.

As discussed in §4.3.1, MinimaLT addresses these attacks using puzzles present in its tunnel headers. Servicing tunnel requests in excess of the limits discussed in §5.2 would cause a server to require puzzles. Our server can generate and verify 386,935 puzzles per second on our test hardware. Since a puzzle interrogation and padded solution packet are 206 and 1,024 bytes, respectively, a single CPU core can handle puzzles at 394% of Gb/s line speed.

Amplification attacks against third parties At tunnel establishment, MinimaLT may respond to packets from clients which spoof another host's IP address. This is always the case with the directory service, which initially must react to a request from an unknown party before transitioning to PFS-safe authorization. A MitM could spoof the source of packets, even while completing a puzzle interrogation. A weaker attacker could elicit a response to the first packet sent to a server. Given this, MinimaLT is designed to minimize amplification attacks, in which a request is smaller than its reply (to a spoofed source address). A connection request causes a connection acknowledgment or puzzle interrogation; both responses are smaller than the request.

5.3.2 After establishing a tunnel Given a tunnel, an attacker can easily forge a packet with garbage in place of ciphertext and send it to a service. This forces MinimaLT to decrypt the packet and verify its checksum, wasting processor time. However, MinimaLT's symmetric cryptography on established tunnels operates at line speed on commodity hardware (§5.2), so DoS would be equivalent to the attacker saturating a Gb/s link.

Low cost (small w) puzzles can be used to check that a client is still reachable. Puzzles can occur at other than connection establishment time, so they can require that a client perform work to keep a connection alive. We use control connection RPCs to pose and solve these puzzles to prevent an attacker from attempting RST-style mischief [23].

Creation of fictitious strangers Stranger services are vulnerable to further CPU attacks—attackers could generate false user identities that would fail authentication, but only after the server performed a public-key decryption. A server will apply the puzzle RPCs when connection rates exceed the limits discussed in §5.2.

An attacker could also generate verifiable authenticators and connect to a stranger-authorized service many times as different stranger users. This would cause a system to generate accounts for each stranger identity. However, this is no different from any other creation of pseudo-anonymous accounts; it is up to the system to decide how to allocate account resources to strangers. Perhaps the rate is faster because of the lack of a Captcha, but unlike many contemporary pseudo-anonymous services, a MinimaLT system can prune stranger accounts as necessary; the stranger's long-term resources (e.g., files on disk) will remain isolated and become available if the account is later regenerated because public keys remain temporally unique [55]. Of course, applications could impose additional requirements (e.g., a Captcha) before allowing a stranger to consume persistent resources like disk space.

5.4 Cryptographic security

Tunnel IDs and nonces are visible on the network, and follow a clear pattern for each client-server pair. Essentially the same information is available through a simple log of IP addresses of packets sent. Mobile clients automatically switch to new tunnel IDs when they change IP addresses, as discussed in §4.3.3. Ephemeral client public keys are visible when each tunnel is established, but are not reused and are not connected to any other client information.

Other information is boxed (encrypted and authenticated) between public keys at the ends of the tunnel. These boxes can be created and understood using either of the two corresponding private keys, but such keys are maintained locally inside MinimaLT hosts. The attacker can try to violate confidentiality by breaking the encryption, or violate integrity by breaking the authentication, but the cryptography makes these attacks very difficult; see below. The attacker can also try to substitute his public key for a legitimate public key, fooling the client or server into encrypting data to the attacker or accepting data actually from the attacker, but this requires violating integrity of previous packets: for example, before the client encrypts data to D', the client obtains D' from a boxed packet between D and C'.

This type of temporal data-flow analysis is conceptually straightforward. One might think that existing protocol-analysis tools are already powerful enough to formally verify the confidentiality and integrity properties of high-level protocols such as MinimaLT, assuming that the box mechanism is secure. However, the security properties of authenticated encryption using non-interactive DH were only very recently formalized (see [30]), and more work is required to develop a higher-level security calculus on top of these properties; note that replacing boxes with *unauthenticated* encryption would eliminate the security of typical box-based protocols.

For comparison, attempts to verify the security of TLS (such as [40]) have so far covered only limited portions of TLS. The unverified portions of TLS are more complex than the entire MinimaLT protocol. We also comment that there is an apparently neverending string of announcements of TLS security failures, evidently in the unverified portions of TLS; [12] traces these failures to various aspects of the cryptographic choices in TLS that are systematically avoided by NaCl, the cryptographic library used in MinimaLT.

The cryptographic details of NaCl are as follows. Encryption and authentication use the elliptic curve Curve25519 [8] to generate a 256-bit shared secret, the stream cipher Salsa20 [9] to expand the shared secret into a long pad used to encrypt data, and the message-authentication code Poly1305 [7] to produce a 128-bit authenticator of the ciphertext. Elliptic-curve cryptography has been extensively studied since 1985, and since 2005 has been the only public-key cryptography recommended by NSA for the protection of US government Secret information. Curve25519, which is also used in Apple's iOS operating system [3], meets the IEEE P1363 standard criteria [35] for elliptic-curve security, along with additional security criteria such as twist security; see [8]. Poly1305 is information-theoretically secure, with a forgery probability below 2^{-106} per byte; see [7]. Salsa20 has been analyzed in papers by 23 cryptanalysts, culminating in an attack on just 8 out of the full 20 rounds; Salsa20 is one of only four software ciphers recommended by the ECRYPT Stream Cipher Project [5]. The fastest attacks known against any of these cryptographic primitives use approximately 2^{128} operations. This means they are stronger than the RSA-2048 option chosen in OpenSSL for experiments. All of the implementations in NaCl are fully protected against cache-timing attacks, branch-prediction attacks, etc.; see [12].

5.5 Key isolation

We designed MinimaLT to facilitate strong key isolation. Since the semantics of MinimaLT include encryption and server/user authentication, it is natural to keep private keys under the control of MinimaLT, never releasing them to applications. We have done this in Ethos. (Ethos also provides a sign system call for this reason [51].)

5.6 Ongoing performance tuning

Using a single CPU core, MinimaLT transmits encrypted data at nearly one Gb/s and performs thousands of authentications per second. Future work will focus on increasing tunnel establishment rates by offloading public key operations to other CPU cores. We expect a roughly N-fold improvement in cryptography from using N cores, and thus expect Gb/s-speed tunnel establishment with 16 cores. When not under attack, MinimaLT would use far fewer cores.

6 Conclusions and future work

MinimaLT provides network confidentiality, integrity, privacy, server authentication, user authentication, and DoS protections with a simple protocol and implementation. A particular concern for protected networking is latency, as research has shown users are very sensitive to delay. MinimaLT combines directory services and tunnel establishment in a new way to minimize latency—even outperforming unencrypted TCP/IP. MinimaLT's first round trip is performed only once, at system boot time. The second is a protected analogue of a DNS lookup and is required under the same circumstances as DNS. Thus in the typical case, MinimaLT clients transmit encrypted data to an end server in the first packet sent.

MinimaLT establishes a tunnel which can be long-lived. Of course, the tunnel can be terminated at any time, but absent resource constraints MinimaLT is intended to maintain tunnels even across system suspends and network migration. This makes for a more reliable system as recovery code needs to be run less often.

Future work includes the aforementioned performance tuning, remote client software attestation, and building proxies which will enable MinimaLT to talk to legacy applications. We plan to soon release Ethos and our Linux MinimaLT implementation as open source software.

7 References

[1] Aiello, W., Bellovin, S. M., Blaze, M., Canetti, R., Ioannidis, J., Keromytis, A. D., and Reingold, O. Just Fast Keying: Key agreement in a hostile Internet. *ACM Trans. Inf. Syst. Secur. 7*, 2 (May 2004), 242–273.

[2] AlFardan, N., and Paterson, K. Lucky thirteen: Breaking the TLS and DTLS record protocols. http://www.isg.rhul.ac.uk/tls/, February 2013.

[3] Apple. iOS security, 2012. /home/djb/download/images.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf.

[4] Argyraki, K. J., Maniatis, P., Irzak, O., Ashish, S., and Shenker, S. Loss and delay accountability for the internet. In *ICNP* (2007), pp. 194–205.

[5] Babbage, S., Cannière, C. D., Canteaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B., Rijmen, V., and Robshaw, M. The eSTREAM portfolio, 2008. http://www.ecrypt.eu.org/stream/portfolio.pdf.

[6] Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. Recommendation for key management—part 1: General (revised), Mar. 2007.

[7] Bernstein, D. J. The Poly1305-AES message-authentication code. In *Fast Software Encryption* (2005), H. Gilbert and H. Handschuh, Eds., vol. 3557, Springer, pp. 32–49.

[8] Bernstein, D. J. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography* (2006), pp. 207–228.

[9] Bernstein, D. J. *The Salsa20 family of stream ciphers*, vol. 4986 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 84–97.

[10] Bernstein, D. J., and Lange, T. eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to/.

[11] Bernstein, D. J., Lange, T., and Schwabe, P. NaCl: Networking and cryptography library. http://nacl.cr.yp.to/.

[12] Bernstein, D. J., Lange, T., and Schwabe, P. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America* (2012), vol. 7533 of *Lecture Notes in Computer Science*, Springer, pp. 159–176.

[13] Bernstein, D. J., and Schwabe, P. NEON crypto. In *Workshop on Cryptographic Hardware and Embedded Systems* (2012), vol. 7428 of *Lecture Notes in Computer Science*, Springer, pp. 320–339.

[14] Birrell, A., and Nelson, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (1984), 39–59.

[15] Bittau, A., Hamburg, M., Handley, M., Mazières, D., and Boneh, D. The case for ubiquitous transport-level encryption. In *Proc. of the USENIX Security Symposium* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 26–26.

[16] Bonneau, J., Herley, C., van Oorschot, P. C., and Stajano, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. IEEE Symp. Security and Privacy* (2012), pp. 553–567.

[17] Card, S. K., Robertson, G. G., and Mackinlay, J. D. The information visualizer, an information workspace. In *Proc. ACM Conf. Human Factors in Computing Systems* (Apr. 1991), ACM, pp. 181–188.

[18] de Vivo, M., de Vivo, G. O., Koeneke, R., and Isern, G. Internet vulnerabilities related to TCP/IP and T/TCP. *SIGCOMM Comput. Commun. Rev. 29*, 1 (Jan. 1999), 81–85.

[19] Dierks, T., and Allen, C. RFC 2246: The TLS protocol version 1, Jan. 1999. Status: PROPOSED STANDARD.

[20] Dingledine, R., Mathewson, N., and Syverson, P. F. Tor: The

second-generation onion router. In *Proc. of the USENIX Security Symposium* (2004), pp. 303–320.

[21] Douceur, J. The Sybil Attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop* (March 2002).

[22] Duong, T., and Rizzo, J. Here come the ⊕ ninjas. In *Ekoparty Security Conference* (2011).

[23] Eckersley, P., von Lohmann, F., and Schoen, S. Packet forgery by ISPs: A report on the Comcast affair. `https://www.eff.org/files/eff_comcast_report.pdf`.

[24] Egevang, K., and Francis, P. RFC 1631: The IP network address translator (NAT), May 1994. Status: INFORMATIONAL.

[25] Electronic Frontier Foundation. HTTPS everywhere. `https://www.eff.org/https-everywhere`.

[26] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *Proc. ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2012), CCS '12, ACM, pp. 50–61.

[27] Floyd, S. Congestion control principles; RFC 2914, Sept. 2000.

[28] Ford, B. Directions in Internet transport evolution. *IETF Journal 3*, 3 (2007), 29–32.

[29] Ford, B. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 361–372.

[30] Freire, E. S., Hofheinz, D., Kiltz, E., and Paterson, K. G. Non-interactive key exchange. In *PKC 2013* (2013), Lecture Notes in Computer Science, Springer. `http://eprint.iacr.org/2012/732`.

[31] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proc. ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2012), CCS '12, ACM, pp. 38–49.

[32] Gettys, J., and Nichols, K. Bufferbloat: dark buffers in the internet. *Commun. ACM 55*, 1 (Jan. 2012), 57–65.

[33] Gummadi, P. K., Saroiu, S., and Gribble, S. D. King: estimating latency between arbitrary internet end hosts. In *Internet Measurement Workshop* (2002), pp. 5–18.

[34] Hiltgen, A., Kramp, T., and Weigold, T. Secure Internet banking authentication. *Security Privacy, IEEE 4*, 2 (March-April 2006), 21 –29.

[35] IEEE. *Standard specifications for public key cryptography.* IEEE, 2000. IEEE 1363-2000.

[36] Ioannidis, J., and Bellovin, S. M. Implementing pushback: Router-based defense against DDoS attacks. In *Proc. of the Symp. on Network and Distributed Systems Security (NDSS)* (San Diego, California, 2002), The Internet Society.

[37] Ioannidis, S., Keromytis, A. D., Bellovin, S. M., and Smith, J. M. Implementing a distributed firewall. In *Proc. ACM Conference on Computer and Communications Security (CCS)* (2000), ACM Press, pp. 190–199.

[38] Jackson, C., and Barth, A. ForceHTTPS: protecting high-security web sites from network attacks. In *International Conference on the World Wide Web* (New York, NY, USA, 2008), WWW '08, ACM, pp. 525–534.

[39] Jaeger, T., Butler, K., King, D. H., Hallyn, S., Latten, J., and Zhang, X. Leveraging IPsec for mandatory access control across systems. In *Proc. of the Second International Conference on Security and Privacy in Communication Networks* (Aug. 2006).

[40] Jager, T., Kohlar, F., Schäge, S., and Schwenk, J. On the security of TLS-DHE in the standard model. In *Crypto 2012* (2012), vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 273–293.

[41] Juels, A., and Brainard, J. G. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS* (1999).

[42] Keromytis, A. D., Ioannidis, S., Greenwald, M. B., and Smith, J. M. The STRONGMAN architecture. In *DARPA Information Survivability Conference and Exposition (DISCEX)* (2003), vol. 1, pp. 178–188.

[43] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computing Systems (TOCS) 10*, 4 (1992), 265–310.

[44] Langley, A. Transport Layer Security (TLS) Snap Start, June 2010.

[45] Langley, A., Modadugu, N., and Chang, W.-T. Overclocking SSL. In *Velocity: Web Performance and Operations Conference* (Santa Clara, CA, Jun 2010). `http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html`.

[46] Langley, A., Modadugu, N., and Moeller, B. Transport Layer Security (TLS) False Start, June 2010.

[47] Le Malécot, E., Hori, Y., and Sakurai, K. Preliminary insight into distributed SSH brute force attacks. *Proceedings of the IEICE General Conference* (2008).

[48] Liberatore, M., and Levine, B. N. Inferring the source of encrypted HTTP connections. In *Proc. ACM Conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2006), CCS '06, ACM, pp. 255–263.

[49] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proc. of the FREENIX Track* (Berkeley, CA, 2001), The USENIX Association, pp. 29–42.

[50] McGrew, D. RFC 5116: An interface and algorithms for authenticated encryption, 2008. Status: PROPOSED STANDARD.

[51] Petullo, W. M., and Solworth, J. A. Digital identity security architecture in Ethos. In *Proceedings of the 7th ACM workshop on Digital identity management* (New York, NY, USA, 2011), ACM, pp. 23–30.

[52] Petullo, W. M., and Solworth, J. A. Simple-to-use, secure-by-design networking in Ethos. In *Proceedings of the Sixth European Workshop on System Security* (New York, NY, USA, 2013), EUROSEC '13, ACM.

[53] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., and Raghavan, B. TCP fast open. In *Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2011), CoNEXT '11, ACM, pp. 21:1–21:12.

[54] Rescorla, E., and Modadugu, N. RFC 6347: Datagram transport layer security version 1.2, 2012. Status: PROPOSED STANDARD.

[55] Rivest, R. L., and Lampson, B. SDSI — a simple distributed security infrastucture. Tech. rep., MIT, Apr. 1996.

[56] Shields, C. What do we mean by network denial of service? In *IEEE Workshop on Information Assurance and Security (West Point, NY)* (June 2002).

[57] Song, D. X., Wagner, D., and Tian, X. Timing analysis of keystrokes and timing attacks on SSH. In *Proc. of the USENIX Security Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 25–25.

[58] Souders, S. Velocity and the bottom line. `http://programming.oreilly.com/2009/07/velocity-making-your-site-fast.html`, July 2009.

[59] Stark, E., Huang, L.-S., Israni, D., Jackson, C., and Boneh, D. The case for prefetching and prevalidating TLS server certificates. In *Proc. of the Symp. on Network and Distributed Systems Security (NDSS)* (San Diego, CA, 2012), Internet Society.

[60] Stewart, R. Stream Control Transmission Protocol, Sept. 2007.

[61] Vratonjic, N., Freudiger, J., Bindschaedler, V., and Hubaux, J.-P. The inconvenient truth about web certificates. In *The Workshop on Economics of Information Security (WEIS)* (2011).

[62] Weaver, N., Sommer, R., and Paxson, V. Detecting forged TCP reset packets. In *NDSS* (2009).

[63] White, J. E. A high-level framework for network-based resource sharing. In *National Computer Conference* (1976).

[64] Wobber, E., Abadi, M., Burrows, M., and Lampson, B. Authentication in the Taos operating system. In *Symposium on Operating System Principles (SOSP)* (1993), pp. 256–269.

APPENDIX

A Authentication/authorization hooks

We designed MinimaLT to provide a native network protocol with strong security properties. The interface to the MinimaLT protocol stack consists of two parts: (1) the networking API described in Appendix B and (2) authentication and authorization hooks used to provide system services to MinimaLT. We describe the latter part here.

MinimaLT requires two hooks into the host OS so that MinimaLT can perform protocol processing. These hooks are called on new connection requests. They restrict the incoming or outgoing connection, providing a bridge between MinimaLT and the OS' authorization monitor.

Service names MinimaLT follows the Unix sockets convention and identifies services with a string instead of a port number; both the create_0 and $\mathsf{createAuth}_0$ RPCs take as an argument such a service name. This allows for an inexhaustible range of mnemonic names for services. As a result, MinimaLT does not need to reuse ports (i.e., port 80 is often used for a wide range of web-based services); a service name remains bound to the service it describes. The cost is a slight increase in the amount of information needed to identify the service on the first packet (i.e., the connection type parameter to create_0 or $\mathsf{createAuth}_0$).

Server-side authorization When the MinimaLT implementation receives a create_0 or $\mathsf{createAuth}_0$ (with a valid authenticator), it invokes a hook into the host OS named mltIsIncomingAuthorized:

```
mltIsIncomingAuthorized(publicKey, serviceName)
```

which takes as parameters publicKey, the public key from $\mathsf{createAuth}_0$ (or nil in the case of create_0); and service, the service name. To service mltIsIncomingAuthorized, the OS consults a user database (either local or distributed) to ascertain the real-world identity of the user associated with publicKey (if not nil), decides whether to authorize access to the service serviceName, and returns true or false to MinimaLT.

If mltIsIncomingAuthorized returns false, then our MinimaLT implementation provides no response (at any network layer) to a request. With TCP/IP, this is possible only using weak, IP-address-based authentication; as we have shown, MinimaLT authentication is much stronger. Thus MinimaLT makes network mapping much more difficult. Since most hosts do not offer Internet-wide services, they present a minimal signature to attackers—on Ethos, even the equivalent to ping will respond only to authorized users.

Client-side authorization MinimaLT also authorizes outgoing connections. This can be used to restrict which services on which hosts a user/program pair may connect to. For example, an organization may wish to restrict mail clients so that they may connect only to a trusted service provider. The client-side authentication hook is:

```
mltIsOutgoingAuthorized(publicKey, serviceName)
```

Here publicKey is the receiving server's long-term public key, but otherwise an OS will implement this procedure in a manner similar to mltIsIncomingAuthorized.

B Ethos integration

One difficulty with network protocols is they often integrate poorly with related protections, resulting in overly complex APIs. Here we discuss the integration of MinimaLT with Ethos. Like distributed firewalls [37], which overload POSIX networking APIs (connect and accept), Ethos makes protections inescapable by adding transparent encryption and authentication to its networking system call semantics [52].

B.1 Anatomy of an ipc

Client applications invoke the ipc system call to initiate a network connection.

```
netFd = ipc(serviceName, host)
```

To service an ipc, Ethos first checks if the calling program and user are authorized to connect to the requested service/host pair. If authorized, Ethos next checks to see if a tunnel to the server host already exists. If not, then Ethos looks up the host's directory certificate and establishes the tunnel. Recall that Ethos has already created a tunnel to the directory service upon booting.

Ethos next looks up the user's key configuration. If the user selected a public key U for this service, then Ethos loads it and generates the authenticator x. Then Ethos creates a connection by invoking $\mathsf{createAuth}_0(c, s, U, x)$ where s is the service name and c is a new connection number within the tunnel to the server. Otherwise, Ethos invokes $\mathsf{create}_0(c, s)$ to attempt to create an anonymous connection. Once Ethos receives an ack_0 from the server, ipc returns a network file descriptor to the application. A variant of ipc, ipcWrite, allows as an additional parameter an application $\mathsf{serviceRequest}_c$ that is sent along with the first packet to the server.

Thus Ethos presents a very simple API to application developers, leaving less room for error than alternatives such as POSIX. The semantics of the ipc system call include encryption, server authentication, a user authenticator, and IP mobility. All of this is transparent to the application.

B.2 Anatomy of an import

Receiving a network connection on Ethos requires two system calls: advertise and import. advertise makes a service available to the network and returns a listening file descriptor. import takes a service file descriptor and returns a network file descriptor and remote user after receiving a network connection request from some client.

```
serviceFd = advertise(serviceName)
netFd, user = import(serviceFd)
```

Calling import waits to receive a $\mathsf{createAuth}_0$ or create_0 from some client. The following discussion ignores puzzles for simplicity. Ethos creates a tunnel upon receiving a tunnel establishment packet, if the encrypted packet decrypts properly (passing verification) and the TID is unique. Otherwise it finds a tunnel whose current TID (or next TID, set by $\mathsf{nextTid}_0$) matches the packet's TID, and checks that the packet decrypts properly under the current tunnel key (or the next tunnel key).

If Ethos receives a $\mathsf{createAuth}_0$, it validates the included authenticator. After doing so, it checks to see if the associated user is authorized to connect to the service. Alternatively, if Ethos receives a create_0, it checks to see if users may connect to the service anonymously. Ethos rejects and logs unauthorized connections within the kernel, so such users never interact with an application. Only if the user is authorized does Ethos reply with an ack_0 and return a network file descriptor and client user name to the application.

A client user might not have a local account on the Ethos server—this is always the case for strangers, as they are not known a priori. If a local account does not exist, then Ethos references its distributed user database. If the user account is not found there, then Ethos generates one, naming it after the client user's public key (such accounts are *sparse* in that they do not include identifying information other than their public key). Anonymous connections (create_0) are even more specialized because they do not provide a public key. Ethos generates a random name for these users; this identifier is not known to the client, so anonymous clients cannot create a second connection under the same identity.