Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS

David Naylor*, Kyle Schomp†, Matteo Varvello‡, Ilias Leontiadis‡, Jeremy Blackburn‡,
Diego Lopez‡, Konstantina Papagiannaki‡,
Pablo Rodriguez Rodriguez‡, and Peter Steenkiste*

*Carnegie Mellon University †Case Western Reserve University ‡Telefónica Research

ABSTRACT

A significant fraction of Internet traffic is now encrypted and HTTPS will likely be the default in HTTP/2. However, Transport Layer Security (TLS), the standard protocol for encryption in the Internet, assumes that all functionality resides at the endpoints, making it impossible to use in-network services that optimize network resource usage, improve user experience, and protect clients and servers from security threats. Re-introducing in-network functionality into TLS sessions today is done through hacks, often weakening overall security.

In this paper we introduce multi-context TLS (mcTLS), which extends TLS to support middleboxes. mcTLS breaks the current "all-or-nothing" security model by allowing endpoints and content providers to explicitly introduce middleboxes in secure end-to-end sessions while controlling which parts of the data they can read or write.

We evaluate a prototype mcTLS implementation in both controlled and "live" experiments, showing that its benefits come at the cost of minimal overhead. More importantly, we show that mcTLS can be incrementally deployed and requires only small changes to client, server, and middlebox software.

CCS Concepts

•**Security and privacy** → **Security protocols;** •**Networks** → **Middle boxes / network appliances;** *Session protocols;*

Keywords

TLS; SSL; HTTPS

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787482

1. INTRODUCTION

The increased personalization of Internet services and rising concern over users' privacy on the Internet has led to a number of services (e.g., Facebook, Twitter, and Google) offering access solely over HTTPS. HTTPS currently accounts for a significant portion of all Internet traffic (40% in [26], and estimated to grow at 40% every 6 months [27]). Transport Layer Security (TLS), which underlies HTTPS, has become the standard for end-to-end encryption on the web because it ensures (i) entity authentication, (ii) data secrecy, and (iii) data integrity and authentication. Moreover, it will likely be the default transport protocol for HTTP/2.

This is good news for privacy. However, TLS makes a fundamental assumption: all functionality must reside at the endpoints. In reality, Internet sessions are augmented by functional units along the path, providing services like intrusion detection, caching, parental filtering, content optimization (e.g., compression, transcoding), or compliance to corporate practices in enterprise environments. These functional units, often referred to as *middleboxes*, offer many benefits to users, content providers, and network operators, as evidenced by their widespread deployment in today's Internet.

While there are arguments that these services could (and should) be implemented on endpoints, we argue that this is often not optimal or even possible. First, although an in-network implementation might not be *required*, it might be *inherently more effective* than an endpoint-based one (e.g., thousands of users sharing an ISP cache). Second, there might be practical reasons why such a solution needs to reside in-network (e.g., virus scanners can always be up-to-date and immediately protect all clients). Third, certain features (e.g., intrusion detection or content-based routing decisions) simply cannot be implemented without network-wide visibility. Finally, in-network services can lead to increased competition, innovation, and choice for end users.

Recently, industrial efforts—one by Ericsson and AT&T [20] and one by Google [28]—have tried to provide a so-

lution for combining encryption with the richness of today's in-network services. However, as we will discuss, a good solution is still lacking and the topic is still active within the GSMA *ENCRY* and the IETF *httpbis*.

In this paper we present mcTLS, a protocol that builds on top of TLS to allow endpoints to explicitly and securely include in-network functionality with complete visibility and control. mcTLS: (i) provides endpoints explicit knowledge and control over *which* functional elements are part of the session, (ii) allows users and content providers to *dynamically* choose which portions of content are exposed to in-network services (e.g., HTTP headers vs. content), (iii) protects the authenticity and integrity of data while still enabling modifications by selected in-network services by separating read and write permissions, and (iv) is incrementally deployable.

We implemented mcTLS as a simple extension to the OpenSSL library. Our evaluation shows that mcTLS has negligible impact on page load time or data overhead for loading the top 500 Alexa sites and incorporating mcTLS into applications is relatively easy in many cases.

Our contributions are as follows: (i) a practical extension to TLS that explicitly introduces trusted in-network elements into secure sessions with the minimum level of access they need, (ii) a prototype implementation of mcTLS tested in controlled and live environments (our implementation is available online [1]), (iii) an efficient fine-grained access control mechanism which we show comes at very low cost, and (iv) strategies for using mcTLS to address concrete, relevant use cases, many of which can immediately benefit applications with little effort using mcTLS's most basic configuration.

2. MIDDLEBOXES AND ENCRYPTION

It is clear that there is both a growing interest in user privacy and a widespread use of in-network processing in the Internet. In this section, we describe middleboxes and why it is beneficial to keep them around as the Internet moves toward ubiquitous encryption. We then explain why it is difficult to do so with TLS.

2.1 Middleboxes

Middleboxes are services that run "inside" the network, sitting logically between the endpoints of communication sessions. A *client* (e.g., web browser) can connect to a *server* (e.g., web server) via one or more middleboxes that add value beyond basic data transport. Clients and servers are *endpoints*; *users* own/manage clients and *content providers* own/manage servers. The entire communication, across all parties, is a *session*; *connections* link individual hops in the session (e.g., a TCP connection between the client and a middlebox).

Our focus is on application level middleboxes, also called *proxies* or *in-path services*, which we loosely define as middleboxes that access application data, like intrusion detection systems (IDSes), content filters, and caching/compression proxies (see Table 1).

	Request		**Response**	
	Headers	*Body*	*Headers*	*Body*
Cache	○		●	●
Compression			●	●
Load Balancer	○			
IDS	○	○	○	○
Parental Filter	○			
Tracker Blocker	●		●	
Packet Pacer			○	
WAN Optimizer	○	○	○	○

(● = read/write; ○ = read-only)

Table 1: Examples of app-layer middleboxes and the permissions they need for HTTP. No middlebox needs read/write access to all of the data.

Middleboxes are sometimes viewed as undesirable. One reason is privacy concerns, which we address later. Another is that they violate the original Internet architecture, which places all functionality (transport and up) at the endpoints, a design motivated by the end-to-end principle [31]. However, the Internet has changed dramatically: both connectivity and content services are commercially supported, security is a major concern, performance expectations are much higher, the technology is more complex, and users are typically not in a position to manage it. As a result, the decision of where to place functionality depends on more than just technical concerns and, increasingly, "inside the network" is a good solution.

Middleboxes are useful: Providing processing and storage in the network has proven to be an effective way to help users, content providers, and network operators alike. For example, techniques such as caching, compression, prefetching, packet pacing, and reformatting improve load times for users [38, 18], reduce data usage for operators and users [3, 26, 13, 37, 29], and reduce energy consumption on the client [26, 12, 29, 14]. Middleboxes can also add functionality not provided by the endpoints, such as virus scanners in enterprises or content filters for children.

In-network may be better: First, functions such as caching and packet pacing are inherently more effective in the network [13, 12, 14]. Second, client implementations may be problematic because the client may be untrusted or its software, URL blacklists, virus signatures, etc. may be out-of-date (e.g., only a third of Android users run the latest version of the OS and over half are more than two years out of date [2]). Finally, users may trust a middlebox more than the application; for example, apps can unexpectedly leak personal information to a server [36], so users may want a middlebox to act as a watchdog.

They are widely used: In a 2012 survey of network operators, networks of all sizes reported having roughly as many middleboxes as L3 routers [33]. For web proxies in particular, 14% of Netalyzer sessions show evidence of a proxy [35] and all four major U.S. mobile carriers use proxies—connections to the top 100 Alexa sites are all proxied, with the exception of YouTube on T-Mobile [38]. In addition, all actors in the Internet use middleboxes. They are widely deployed in client networks (e.g., enterprise firewalls, cellular networks), and of the three IETF RFCs on using middleboxes with TLS, two are led by operators [20, 18] and one by a content provider [28]. Given this investment, middleboxes are unlikely to go away, so we need a clean, secure way to include them in encrypted sessions.

The Internet is a market-driven ecosystem: The Internet is not a centrally managed monopoly but is a market-driven ecosystem with many actors making independent decisions. For example, while servers can compress data, many only do so selectively [29]. Similarly, content providers may decide not to support device-specific content formatting but instead rely on third party providers, which can be selected by the client or content provider. For functions such as content filtering, clients may decide to pay for the convenience of having a single middlebox provider of their choice, instead of relying on individual content providers. Enterprise networks may similarly decide to outsource functionality [33]. Fundamentally, middleboxes give actors more choices, which leads to competition and innovation.

The bottom line is simple: just like end-to-end encryption, middleboxes are an integral, useful part of the Internet and they are here to stay.

2.2 Middleboxes *and* TLS

Given these trends, it is natural to want the best of both worlds. Before discussing how middleboxes and encryption can be used together today, let us take a closer look at Transport Layer Security (TLS) [11], the standard protocol for secure network communications.

What does TLS give us? TLS comprises two protocols, a *handshake protocol* for session establishment and a *record protocol* for data exchange, which together realize three security properties:

(1) Entity Authentication: During the handshake, the client authenticates the server by verifying that a valid certificate links the server's domain name and public key. Clients can also authenticate themselves to the server with certificates, but this is rarely used; client authentication typically happens in the application layer, e.g., using a password.

(2) Data Secrecy: The endpoints establish a symmetric *session key* during the handshake, which is used by the record protocol to encrypt/decrypt *records* (blocks of application data).

(3) Data Integrity & Authentication: The session key is also used to generate a message authentication code (MAC) for each record; a valid MAC indicates that (1) the data originated from the other endpoint (authenticity) and (2) the data was not changed in flight (integrity).

How do you add a middlebox to a TLS session? In short: you do not. By design, TLS supports secure communication between exactly two parties. Despite this, middleboxes are frequently inserted in TLS sessions, but this has to be done transparently to TLS. Consider an enterprise network that wants to insert a virus scanner in all employee sessions. A common solution is to install a custom root certificate on the client. The middlebox can then create a certificate for itself purported to be from the intended server and sign it with the custom root certificate. After the client connects to the middlebox, the middlebox connects to the server and passes the data from one connection to the other. We refer to this as *Split TLS*) and it gives rise to several problems:

(i) There is no mechanism for authenticating the middlebox. Even worse, the middlebox is completely transparent to the server, and while users can inspect the certificate chain to check who signed the certificate, very few do that or understand the difference. Moreover, even if they do, they have no information about what functions the middlebox performs.

(ii) The client has no guarantees beyond the first hop. While the connection to the middlebox is encrypted, the client cannot verify that TLS is being used from the middlebox to the server, whether additional middleboxes are present, or (depending on what application-level authentication is used) whether the endpoint of the session is even the intended server. The user needs to completely trust the middlebox, which he did not select and may not even know exists.

(iii) Middleboxes get full read/write access to the data stream. Middleboxes can read and modify any data transmitted and received over TLS sessions despite the fact that many middleboxes only need selective access to the data stream (Table 1).

Given these problems, it should not be a surprise that users are concerned about (transparent) middleboxes. One could even argue that using TLS with a middlebox is worse than not using TLS at all [7], since clients and servers are under the illusion that they have a secure session, while some of the expected security properties do not actually hold. In the next section, we propose a protocol based on TLS that explicitly supports middleboxes and addresses the above problems.

3. MULTI-CONTEXT TLS (mcTLS)

This section presents the design of multi-context TLS (mcTLS), which augments TLS with the ability to securely introduce trusted middleboxes. Middleboxes are trusted in the sense that they have to be inserted explicitly by either the client or the server, at both endpoints' consent. We first summarize our design requirements, then introduce the key ideas, and finally describe the key components of the protocol. A more detailed description of mcTLS is available online [1].

3.1 Protocol Requirements

First, we require mcTLS to maintain the properties provided by TLS (extended to apply to middleboxes):

R1: Entity Authentication Endpoints should be able to authenticate each other and all middleboxes. Similar to TLS usage today, we expect that clients will authenticate all entities in the session, but servers may prefer not to (e.g., to reduce overhead).

R2: Data Secrecy Only the endpoints and trusted middleboxes can read or write the data.

R3: Data Integrity & Authentication All members of the session must be able to detect in-flight modifications by unauthorized third parties, and endpoints must be able to check whether the data was originated by the other endpoint (vs. having been modified by a trusted middlebox).

Second, the introduction of middleboxes brings with it two entirely new requirements:

R4: Explicit Control & Visibility The protocol must ensure that trusted middleboxes are added to the session at the consent of both endpoints. Endpoints must always be able to see all trusted middleboxes in the session.

R5: Least Privilege In keeping with the principle of least privilege [32], middleboxes should be given the minimum level of access required to do their jobs [24, 19]. Middleboxes should have access only to the portion of the data stream relevant to their function; if that function does not require modifying the data, access should be read-only.

Finally, our protocol should meet all five requirements without substantial overhead, e.g., in terms of latency, data usage, computation, connection state, burden on users or administrators, etc.

3.2 Threat Model

A successfully negotiated mcTLS session meets the above requirements in the face of computationally bounded network attackers that can intercept, alter, drop, or insert packets during any phase of the session. Like TLS, mcTLS does not prevent denial of service.

We assume that all participants in an mcTLS session execute the protocol correctly and do not share information out-of-band. For example, the client could share keys with a middlebox not approved by the server, or middleboxes could collude to escalate their permissions. We do not consider such attacks because no protocol (including TLS) can prevent a party from sharing session keys out-of-band. Furthermore, such attacks are unlikely since at least two colluding parties would need to run a malicious mcTLS implementation. Major browsers and Web servers (especially open source ones) are unlikely to do this, since they would almost surely be caught. Mobile apps using HTTP would need to implement HTTP themselves instead of using the platform's (honest) HTTP library, which is unlikely. If it is essential to know that parties have not shared keys with unauthorized parties, some sort of remote attestation is the most promising solution.

Finally, even when all parties are honest, adding more entities to a session necessarily increases the attack surface: a bug or misconfiguration on any one could compromise the session. This risk is inherent in the problem, not any particular solution.

3.3 Design Overview

To satisfy the five design requirements, we add two key features to TLS:

(1) Encryption Contexts *(R2, R3, R5)* In TLS, there are only two parties, so it makes no sense to restrict one party's access to part of the data. With trusted middleboxes, however, the endpoints may wish to limit a middlebox's access to only a portion of the data or grant it read-only access. To make this possible, mcTLS introduces the notion of *encryption contexts*, or *contexts*, to TLS. An encryption context is simply a set of symmetric encryption and message authentication code (MAC) keys for controlling who can read and write the data sent in that context (§3.4). Applications can associate each context with a purpose (opaque to mcTLS itself) and access permissions for each middlebox. For instance, web browsers/servers could use one context for HTTP headers and another for content. We describe several strategies for using contexts in §4.2.

(2) Contributory Context Keys *(R1, R4)* The client and server each perform a key exchange with each middlebox after verifying the middleboxes' certificates if they choose to (R1). Next, the endpoints each generate half of every context key and send to each middlebox the half-keys for the contexts to which it has access, encrypted with the symmetric keys derived above. The middlebox only gains access to a context if it receives both halves of the key, ensuring that the client and server are both aware of each middlebox and agree on its access permissions (R4). The server may relinquish this control to avoid extra computation if it wishes (§3.6).

3.4 The mcTLS Record Protocol

The TLS record protocol takes data from higher layers (e.g., the application), breaks it into "manageable" blocks, optionally compresses, encrypts, and then MAC-protects each block, and finally transmits the blocks. mcTLS works much the same way, though each mcTLS record contains only data associated with a single context; we add a one byte context ID to the TLS record format. Record sequence numbers are global across contexts to ensure the correct ordering of all application data at the client and server and to prevent an adversary from deleting an entire record undetected. Any of the standard encryption and MAC algorithms supported by TLS can be used to protect records in mcTLS. (So, details like the order of encryption and MAC depend on the cipher suite; mcTLS makes no changes here.)

Building on [24, 25], mcTLS manages access to each context by controlling which middleboxes are given which context keys. For each context, there are four relevant parties, listed in decreasing order of privilege: *endpoints* (client and server), *writers* (middleboxes with write access to the context), *readers* (middleboxes with read-only access to the context), and *third parties* (blanket term for middleboxes with no access to the context, attackers, and bit flips during transmission). Changes by writers are *legal modifications* and changes by readers and third parties are *illegal modifications*. mcTLS achieves the following three access control properties:

(1) Endpoints can limit read access to a context to writers and readers only.
(2) Endpoints can detect legal and illegal modifications.
(3) Writers can detect illegal modifications.

Controlling Read Access Each context has its own encryption key (called $K_{readers}$, described below). Possession of this key constitutes read access, so mcTLS can prevent a middlebox from reading a context by withholding that context's key.

Controlling Write Access Write access is controlled by limiting who can generate a valid MAC. mcTLS takes the following "endpoint-writer-reader" approach to MACs. Each mcTLS record carries three keyed MACs, generated with keys $K_{endpoints}$ (shared by endpoints), $K_{writers}$ (shared by endpoints and writers), and $K_{readers}$ (shared by endpoints, writers, and readers). Each context has its own $K_{writers}$ and $K_{readers}$ but there is only one $K_{endpoints}$ for the session since the endpoints have access to all contexts.

Generating MACs

- When an **endpoint** assembles a record, it includes three MACs, one with each key.
- When a **writer** modifies a record, it generates new MACs with $K_{writers}$ and $K_{readers}$ and simply forwards the original $K_{endpoints}$ MAC.
- When a **reader** forwards a record, it leaves all three MACs unmodified.

Checking MACs

- When an **endpoint** receives a record, it checks the $K_{writers}$ MAC to confirm that no illegal modifications were made and it checks the $K_{endpoints}$ MAC to find out if any legal modifications were made (if the application cares).
- When a **writer** receives a record, it checks the $K_{writers}$ MAC to verify that no illegal modifications have been made.
- When a **reader** receives a record, it uses the $K_{readers}$ MAC to check if any *third party modifications* have been made.

Note that with the endpoint-writer-reader MAC scheme, *readers cannot detect illegal changes made by other readers*. The problem is that a shared key cannot be used by an entity to police other entities at the same privilege level. Because all readers share $K_{readers}$ (so that they can detect third party modifications), all readers are also capable of generating valid $K_{readers}$ MACs. This is only an issue when there are two or more readers for a context and, in general, readers not detecting reader modifications should not be a problem (reader modifications are still detectable at the next writer or endpoint). However, if needed, there are two options for fixing this: (a) readers and writers/endpoints share pairwise symmetric keys; writers/endpoints compute and append a MAC for *each* reader, or (b) endpoints and writers append digital signatures rather than MACs; unlike $K_{writers}$ MACs, readers can verify these signatures. The benefits seem insufficient to justify the additional overhead of (a) or (b), but they could be implemented as optional modes negotiated during the handshake.

3.5 The mcTLS Handshake Protocol

The mcTLS handshake is very similar to the TLS handshake. We make two simplifications here for ease of exposition: first, although the principles of the mcTLS handshake apply to many of the cipher suites available in TLS, we describe the handshake using ephemeral Diffie-Hellman with RSA signing keys because it is straightforward to illustrate and common in practice. Second, we describe the handshake with a single middlebox, but extending it to multiple middleboxes is straightforward. Table 2 defines the notation we use in this paper.

The purpose of the handshake is to:

- Allow the endpoints to agree on a cipher suite, a set of encryption contexts, a list of middleboxes, and permissions for those middleboxes.
- Allow the endpoints to authenticate each other and all of the middleboxes (if they choose to).
- Establish a shared symmetric key $K_{endpoints}$ between the endpoints.

Notation	Meaning
DH_E^+, DH_E^-	Entity E's ephemeral Diffie-Hellman public/private key pair
$DHCombine(\cdot,\cdot)$	Combine Diffie-Hellman public and private keys to produce a shared secret
PK_E^+, PK_E^-	Entity E's long-term signing public/private key pair (e.g., RSA)
$Sign_{PK_E^-}(\cdot)$	Signature using E's private key
S_E	Secret known only to entity E
$PS_{E_1\text{-}E_2}$	Pre-secret shared by entities E_1 & E_2
$S_{E_1\text{-}E_2}$	Secret shared between entities E_1 & E_2
$PRF_S(\cdot)$	Pseudorandom function keyed with secret S as defined in the TLS RFC [11]
$K_{E_1\text{-}E_2}$	Symmetric key shared by E_1 and E_2
K^E	Key material generated by entity E
$Enc_K(\cdot)$	Symmetric encryption using key K
$MAC_K(\cdot)$	Message authentication code with key K
$AuthEnc_K(\cdot)$	Authenticated encryption with key K
$H(\cdot)$	Cryptographic hash
$\|\|$	Concatenation

Table 2: Notation used in this paper.

- Establish a shared symmetric key $K_{writers}$ for each context among all writers and a shared symmetric key $K_{readers}$[1] for each context among all readers.

Handshake Below we explain the steps of a mcTLS handshake (shown in Figure 1), highlighting the differences from TLS. Note that it has the same 2-RTT "shape" as TLS.

❶ **Setup:** Each party generates a (public) random value and an ephemeral Diffie-Hellman key pair (the middlebox generates two key pairs, one for the client and one for the server). The endpoints also each generate a secret value.

❷ **Client Hello:** Like TLS, an mcTLS session begins with a `ClientHello` message containing a random value. In mcTLS, the `ClientHello` carries a `MiddleboxListExtension`, which contains (1) a list of the middleboxes to include in the session—we discuss building this list in the first place in §6.1—and (2) a list of encryption contexts, their purposes (strings meaningful to the application), and middlebox access permissions for every context. The client opens a TCP connection with the middlebox and sends the `ClientHello`; the middlebox opens a TCP connection with the server and forwards the `ClientHello`.

❸ **Certificate & Public Key Exchange:** As in TLS, the server responds with a series of messages

[1]Though we describe each as one key for simplicity, $K_{readers}$ and $K_{endpoints}$ are really four keys each (just like the "session key" in TLS): an encryption key for data in each direction and a MAC key for data in each direction. Likewise, $K_{writers}$ is really one MAC key for each direction.

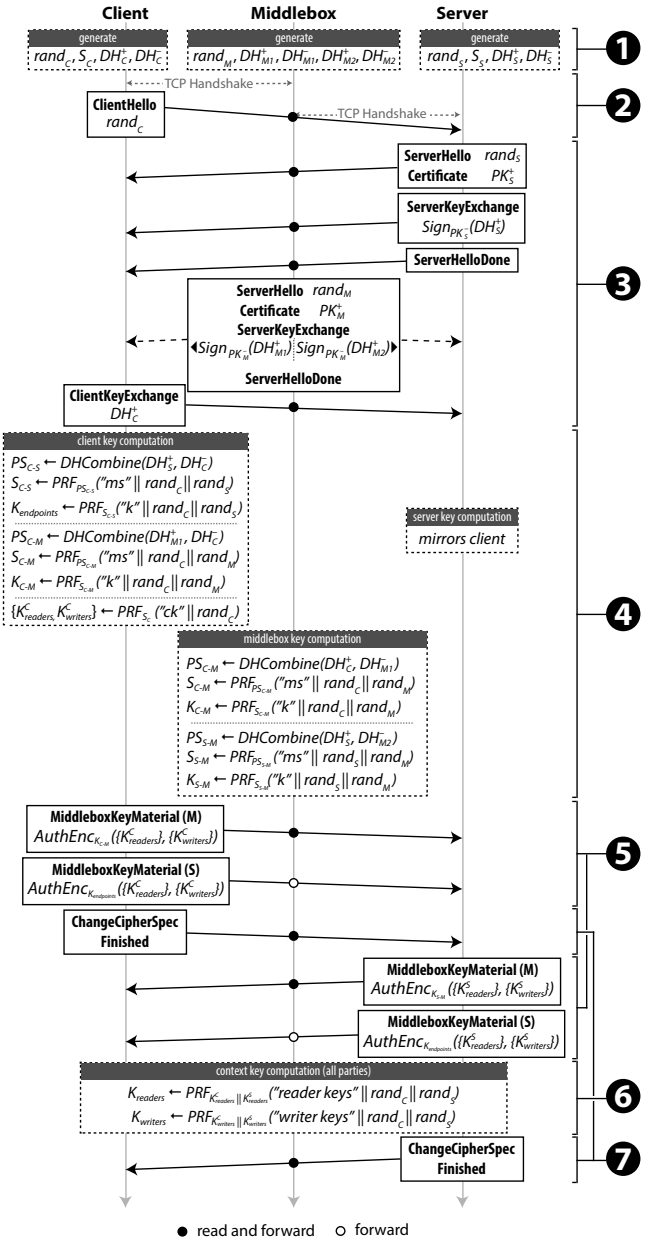

Figure 1: The mcTLS handshake.

containing a random value, its certificate, and an ephemeral public key signed by the key in the certificate. The middlebox does the same: it sends its random value, certificate, and ephemeral public key to both the client and the server. The client sends an ephemeral public key, which the middlebox saves and forwards to the server. The middlebox piggy-backs its messages on the `ServerKeyExchange` and `ClientKeyExchange` messages (indicated by dashed arrows). The ephemeral keys provide forward secrecy; the middlebox uses different key pairs for the client and the server to prevent small subgroup attacks [22].

	mcTLS			mcTLS (Client Key Dist.)			Split TLS		
	Client	*Middlebox*	*Server*	*Client*	*Middlebox*	*Server*	*Client*	*Middlebox*	*Server*
Hash	$12+6N$	0	$12+6N$	$10+5N$	0	$10+5N$	10	20	10
Secret Comp.	$N+1$	2	$N+1$	$N+1$	1	1	1	2	1
Key Gen.	$4K+N+1$	$(k \leq 2K)+2$	$4K+N+1$	$2K+N+1$	1	1	1	2	1
Asym Verify	$N+1$	$n \leq 1$	$n \leq N$	$N+1$	$n \leq 1$	0	1	1	0
Sym Encrypt	$N+2$	0	$N+2$	$N+2$	0	1	1	2	1
Sym Decrypt	2	2	2	1	1	2	1	2	1

(N = number of middleboxes; K = number of contexts)

Table 3: Cryptographic operations performed by the client, middlebox, and server during the handshake. Assumes no TLS extensions, a DHE-RSA cipher suite, and the client is not authenticated with a certificate.

❹ **Shared Key Computation:** The **client** computes two secrets ($S_{C\text{-}M}$ and $S_{C\text{-}S}$) using the contributions from the server and middlebox, which it uses to generate a symmetric key shared with the middlebox ($K_{C\text{-}M}$) and the server ($K_{endpoints}$). The client also generates "partial keys," $K^C_{writers}$ and $K^C_{readers}$, for each context, using a secret known only to itself.

The **server** follows the same procedure as the client, resulting in $K_{endpoints}$, $K_{S\text{-}M}$, $K^S_{writers}$, and $K^S_{readers}$. The server may choose to avoid this overhead by asking the client to generate and distribute complete context keys (§3.6).

When the **middlebox** receives the `ClientKeyExchange`, it computes $K_{C\text{-}M}$ and $K_{S\text{-}M}$ using the client's and server's ephemeral public keys, respectively; it will use these keys later to decrypt context key material from the client and server.

❺ **Context Key Exchange:** Next, for each context, the endpoints send the partial context keys to the middlebox ($K^C_{readers}$ and $K^S_{readers}$ if it has read access and $K^C_{writers}$ and $K^S_{writers}$ if it has write access). These messages are sent encrypted and authenticated under $K_{C\text{-}M}$ and $K_{S\text{-}M}$, ensuring the secrecy and integrity of the partial context keys. The middlebox forwards each message on to the opposite endpoint so it can be included in the hash of the handshake messages that is verified at the end of the handshake. The endpoints also send all of the partial context keys to the opposite endpoint encrypted under $K_{endpoints}$. The middlebox forwards this message (but cannot read it).

❻ **Context Key Computation:** The client indicates that the cipher negotiated in the handshake should be used by sending a `ChangeCipherSpec` message. Receipt of the `ChangeCipherSpec` message prompts all parties to generate context keys using $PRF(\cdot)$ keyed with the concatenation of the partial context keys. This "partial key" approach serves two purposes: (1) it provides contributory key agreement (both endpoints contribute randomness) and (2) it ensures that a middlebox only gains access to a context if the client and server both agree.

❼ **Finished:** The mcTLS handshake concludes with the exchange of `Finished` messages. As in TLS, the `Finished` message contains a hash of the concatenation of all handshake messages (including those directed to the middlebox): $PRF_{S_{C\text{-}S}}(finished\_label, H(messages))$. Verifying this message ensures that both endpoints observe the same sequence of identical handshake messages, i.e., no messages were modified in flight.

Details There are a few subtle differences between the mcTLS and TLS handshakes. We briefly highlight the changes here and argue why they are safe; for a more detailed security analysis, see [1].

- For simplicity, the middlebox cannot negotiate session parameters (e.g., cipher suite or number of contexts). A more complex negotiation protocol could be considered in future work if needed.
- The server's context key material is not included in the client's `Finished` message, since this would require an extra RTT. However, this key material is sent encrypted and MAC-protected, so an adversary cannot learn or modify it.
- The client cannot decrypt the context key material the server sends the middlebox and vice versa. This would require establishing a three-way symmetric key between both endpoints and each middlebox. Because the middlebox needs key material from *both* endpoints, one rogue endpoint cannot unilaterally increase a middlebox's permissions.
- The middlebox cannot verify the handshake hash in the `Finished` message because it does not know $K_{endpoints}$. We do not include per-middlebox `Finished` messages to avoid overhead. This means it is possible for the middlebox to observe an incorrect sequence of handshake messages. However, this is at most a denial of service attack. For instance, even though the middlebox cannot detect a cipher suite downgrade attack, the endpoints would detect it and terminate the session. Furthermore, context key material is sent encrypted and MAC-protected under keys each endpoint shares with the middlebox, so as long as at least one endpoint verifies the middlebox's certificate, an adversary cannot learn or modify the context keys.

3.6 Reducing Server Overhead

One concern (albeit a diminishing one [17, 4]) about deploying TLS is that the handshake is computationally demanding, limiting the number of new connections per second servers can process. We do not want to make this problem worse in mcTLS, and one way we avoid this is by making certain features optional. For example, similar to TLS, authentication of the entities in the session is optional—in some cases, the server may not care who the middleboxes are. Another burden for servers in mcTLS is generating and encrypting the partial context keys for distribution to middleboxes. Rather than splitting this work between the client and server, it can optionally be moved to the client: context keys are generated from the master secret shared by the endpoints and the client encrypts and distributes them to middleboxes ("client key distribution mode"). This reduces the server load, but it has the disadvantage that agreement about middlebox permissions is not enforced. (Note that this does not sacrifice contributory key agreement in the sense that both endpoints contribute randomness. The client generates the context keys from the secret it shares with the server; if client/server key exchange was contributory, the context keys inherit this benefit.) Choosing a handshake mode is left to content providers, who can individually decide how to make this control-performance tradeoff; servers indicate their choice to clients in the `ServerHello`.

Table 3 compares the number of cryptographic operations performed by mcTLS and the split TLS approach described in Section 2.2. We show numbers for mcTLS both without and with client context key distribution. If we consider a simple example with a single middlebox ($N = 1$), the additional server load using client key distribution mode is limited to a small number of lightweight operations (Hash and Sym Decrypt).

4. USING mcTLS

4.1 Using Contexts

Just as the architects of HTTP had to define how it would operate over TLS [30], protocol designers need to standardize how their applications will use mcTLS. From an application developer's perspective, the biggest change mcTLS brings is contexts: the application needs to decide how many contexts to use and for what. First we give the topic a general treatment and then follow up with some concrete use cases below.

There are two ways to think about contexts: as sections of the data stream or as configurations of middlebox permissions. For example, suppose an application wants to transfer a document consisting of three pieces, A, B, and C, via two middleboxes, M_1 and M_2. M_1 should have read access to the entire document and M_2 should read A, write B, and have no access to C. The application could allocate one context for each piece and assign the appropriate permissions (Figure 2 left), or it could create one context for each combination of permissions and use the appropriate context when sending each piece of the document (Figure 2 right).

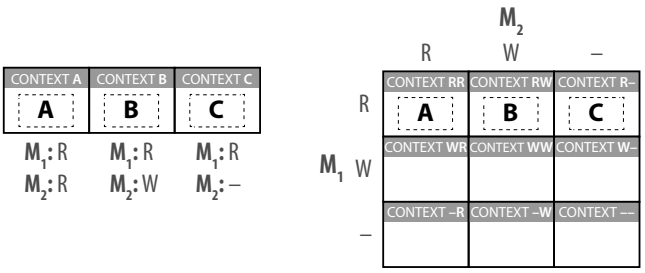

Figure 2: Strategies for using encryption contexts: context-per-section (left) and context-as-permissions (right).

Which model is appropriate depends on the use case: in the context-per-section model, n sections means n contexts. In the contexts-as-permissions model, m middleboxes means 3^m contexts. In practice, we expect at least one of these numbers to be small, since data in a session often is not of wildly varying levels of sensitivity and since most middleboxes need similar permissions (Table 1). For instance, in the case of HTTP, we imagine four contexts will be sufficient: request headers, request body, response headers, and response body. (Though you could imagine extreme cases in which each HTTP header has its own access control settings.)

Finally, the example above uses a static context assignment, but contexts can also be selected dynamically. An application could make two contexts, one which a middlebox can read and one it cannot, and switch between them to enable or disable middlebox access on-the-fly (for instance, to enable compression in response to particular user-agents).

4.2 Use Cases

Data Compression Proxy Many users—particularly on mobile devices—use proxies like Chrome's Data Compression Proxy [3], which re-scale/re-encode images, to reduce their data usage. However, Google's proxy currently ignores HTTPS flows. With mcTLS, users can instruct their browsers to give the compression proxy write access to HTTP responses. One step further, the browser and web server could coordinate to use two contexts for responses: one for images, which the proxy can access, and the other for HTML, CSS, and scripts, which the proxy cannot access. Context assignments can even change dynamically: if a mobile user connects to Wi-Fi mid-page-load, images might also be transferred over the no-access context since compression is no longer required.

Parental Filtering Libraries and schools—and sometimes even entire countries [8]—often employ filters to block age-inappropriate content. Such filters often depend on seeing the full URL being accessed (only 5% of the entries on the Internet Watch Foundation's blacklist

are entire (sub-)domains [26]). With mcTLS, IT staff could configure their machines to allow their filter read-only access to HTTP request headers, and user-owned devices connecting to the network could be configured to do the same dynamically via DHCP. The filter drops non-compliant connections.

Corporate Firewall Most companies funnel all network traffic through intrusion detection systems (IDS)/ firewalls/virus scanners. Currently, these devices either ignore encrypted traffic or install root certificates on employees' devices, transparently giving themselves access to all "secure" sessions. With mcTLS, administrators can configure devices to give the IDS—which users can now see—read-only access. Security appliances no longer need to impersonate end servers and users no longer grow accustomed to installing root certificates.

Online Banking Though we designed mcTLS to give users control over their sessions, there are cases in which the content provider really does know better than the user and should be able to say "no" to middleboxes. A prime example is online banking: banks have a responsibility to protect careless or nontechnical users from sharing their financial information with third parties. The server can easily prevent this by simply not giving middleboxes its half of the context keys, regardless of what level of access the client assigns.

HTTP/2 Streams One of the features of HTTP/2 is multiplexing multiple streams over a single transport connection. mcTLS allows browsers to easily set different access controls for each stream.

5. EVALUATION

mcTLS' fine-grained access control requires generating and distributing extra keys, computing extra MACs, and, possibly, sending a larger number of smaller records than TLS. In this section we evaluate this overhead.

Experimental Setup We built a prototype of mcTLS by modifying the OpenSSL[2] implementation of TLSv1.2. The prototype supports all the features of mcTLS's default mode, described in §3.4 and §3.5. We use the DHE-RSA-AES128-SHA256 cipher suite, though nothing prevents mcTLS from working with any standard key exchange, encryption, or MAC algorithm. In addition, though the `MiddleboxKeyMaterial` message should be encrypted using a key generated from the DHE key exchange between the endpoints and the middlebox, we use RSA public key cryptography for simplicity in our implementation. As a result, *forward secrecy* is not currently supported by our implementation. mcTLS requires some additions to the API, e.g., defining contexts and their read/write permissions, but these are similar to the current OpenSSL API.

[2]OpenSSL v1.0.1j from October 2014

Next we wrote a simple HTTP client, server, and proxy and support four modes of operation:

(1) `mcTLS`: Data transferred using mcTLS.
(2) `SplitTLS`: Split TLS connections between hops; middleboxes decrypt and re-encrypt data.
(3) `E2E-TLS`: A single end-to-end TLS connection; middleboxes blindly forward encrypted data.
(4) `NoEncrypt`: No encryption; data transferred and forwarded in the clear over TCP.

We instrumented the mcTLS library and our applications to measure handshake duration, file transfer time, data volume overhead, and connections per second.

We test in two environments. (1) *Controlled*: Client, middleboxes, and server all run on a single machine. We control bandwidth (10 Mbps unless otherwise noted, chosen from the median of SpeedTest.net samples) and latency with `tc`. (2) *Wide Area*: We run the client, middlebox, and server on EC2 instances in Spain, Ireland, and California, respectively. The client connects either over fiber or 3G. Unless otherwise specified, experiments in either environment consist of 50 runs for which we report the mean; error bars indicate one standard deviation. Middleboxes are given full read/write access to each context since this is the worst case for mcTLS performance.

5.1 Time Overhead

Handshake Time Figure 3 (left) shows the time to first byte as the number of contexts increases. There is one middlebox and each link has a 20 ms delay (80 ms total RTT). `NoEncrypt` serves as a baseline, with a time to first byte of 160 ms, or 2 RTT. Up to 9 contexts, `mcTLS`, `E2E-TLS`, and `SplitTLS` each take 4 RTTs. At 10 contexts, `mcTLS` jumps to 5 RTT and at 14 to 7.

The culprit was TCP's Nagle algorithm, which delays the transmission of data until a full MSS is ready to be sent. At 10 contexts, the handshake messages from the proxy to the server exceed 1 MSS and Nagle holds the extra bytes until the first MSS is ACKed. At 14 contexts the same thing happens to the middlebox key material from the client (+1 RTT) and the server (+1 RTT). Disabling the Nagle algorithm (not uncommon in practice [23]) solved the problem. We tried `E2E-TLS`, `SplitTLS`, and `NoEncrypt` without Nagle as well, but their performance did not improve since their messages never exceed 1 MSS.

Time to first byte scales linearly with the number of middleboxes, since in our experiments adding a middlebox also adds a 20 ms link (Figure 3 right). The latency increase and the extra key material to distribute exacerbate the problems caused by Nagle; disabling it once again brings `mcTLS` performance in line with `E2E-TLS` and `SplitTLS`. Finally, if middleboxes lie directly on the data path (which often happens), then the only additional overhead is processing time, which is minimal.

Takeaway: *mcTLS's handshake is not discernibly longer than `SplitTLS`'s or `E2E-TLS`'s.*

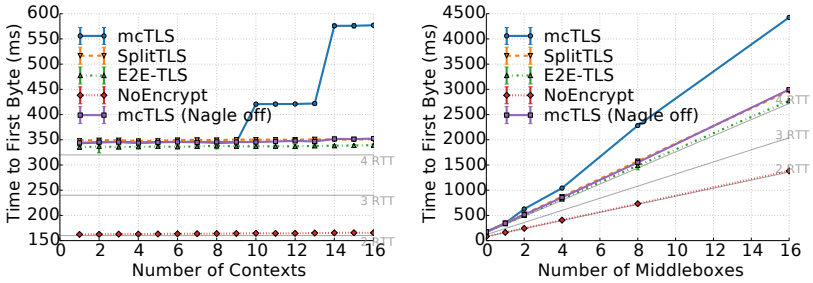

Figure 3: Time to first byte vs. # contexts (left) and # middleboxes (right).

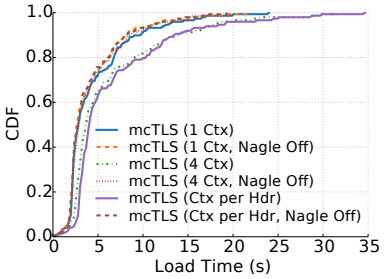

Figure 4: Page load time for different numbers of mcTLS contexts.

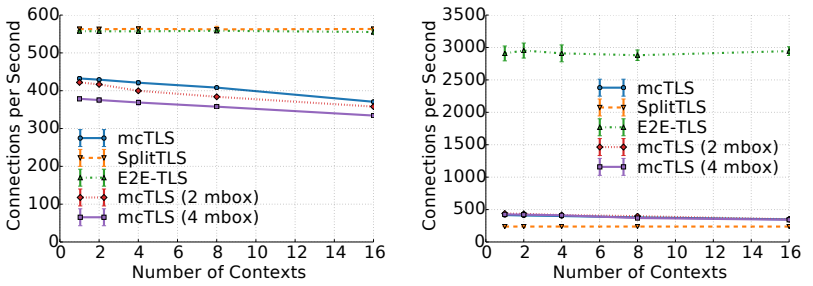

Figure 5: Load sustainable at the server (left) and middlebox (right).

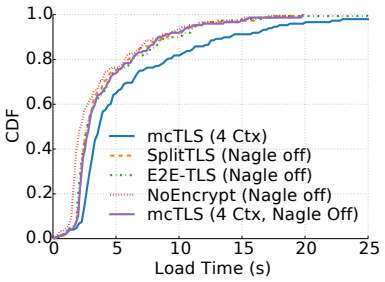

Figure 6: Page load time.

File Transfer Time Next we explore the timing behavior of each full protocol by transferring files through a single middlebox. To choose realistic file sizes, we loaded the top 500 Alexa pages and picked the 10th, 50th, and 99th percentile object sizes (0.5 kB, 4.9 kB, and 185 kB, respectively). We also consider large (10MB) downloads (e.g., larger zip files or video chunks).

The first four bar groups in Figure 7 show the download time for increasing file sizes at 1 Mbps; each bar represents 10 repetitions. As expected, the handshake overhead dominates for smaller files (<5 kB); all protocols that use encryption require an additional ∼17 ms compared to `NoEncrypt`. mcTLS is comparable to `E2E-TLS` and `SplitTLS`. We see the same behavior when downloading files at different link rates or in the wide area (last four bar groups). Handshake and data transfer dominate download time; protocol-specific processing makes little difference.

Takeaway: *`mcTLS` transfer times are not substantially higher than `SplitTLS` or `E2E-TLS` irrespective of link type, bandwidth, or file size.*

Page Load Time To understand how the microbenchmarks above translate to real-world performance, we examine web page load time. Though we have not yet ported a full-blown web browser to mcTLS, we approximate a full page load in our simple client as follows. First, we load all of the Alexa top 500 pages that support HTTPS in Chrome. For each page, we extract a list of the objects loaded, their sizes, and whether or not an existing connection was re-used to fetch each one (we cannot tell *which* connection was used, so we assign the object to an existing one chosen at random). Next, our client "plays back" the page load by requesting dummy objects of the appropriate sizes from the server. We make the simplifying assumption that each object depends only on the previous object loaded in the same connection (this might introduce false dependencies and ignore true ones).

First, we compare three mcTLS strategies: 1-Context (all data in one context), 4-Context (request headers, request body, response headers, response body), and ContextPerHeader (one context for each HTTP header, one for request body, and one for response body). Figure 4 shows the CDF of page load times for each strategy. The plot shows similar performance for each strategy, indicating that mcTLS is not overly sensitive to the way data is placed into contexts.

Next we compare `mcTLS` to `SplitTLS`, `E2E-TLS`, and `NoEncrypt` (Figure 6). We use the 4-Context strategy for `mcTLS`, since we imagine it will be the most common. `SplitTLS`, `E2E-TLS`, and `NoEncrypt` perform the same, while mcTLS adds a half second or more. Once again, Nagle is to blame: sending data in multiple contexts causes back-to-back `send()` calls to TCP. The first record is sent immediately, but the subsequent records are held because they are smaller than an MSS and there is unacknowledged data in flight. Repeating the experiment with Nagle turned off closed the gap.

Takeaway: *mcTLS has* no *impact on real world Web page load times.*

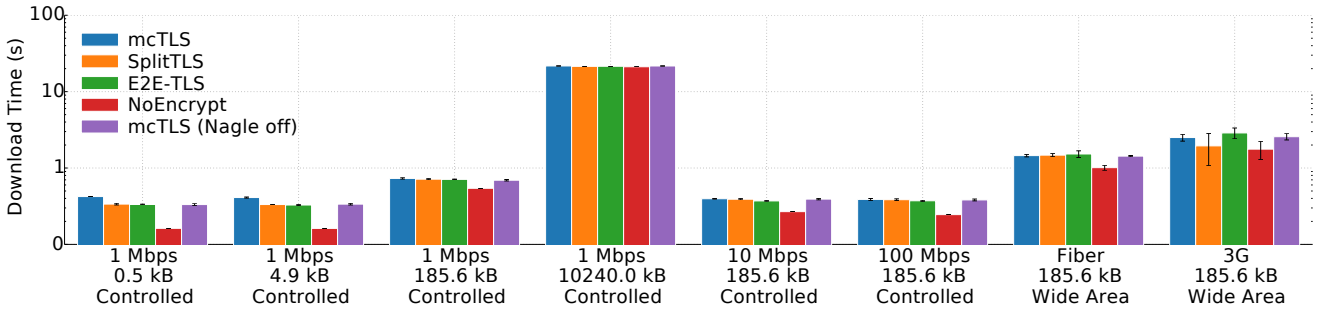

Figure 7: File download time for various configurations of link speed and file size.

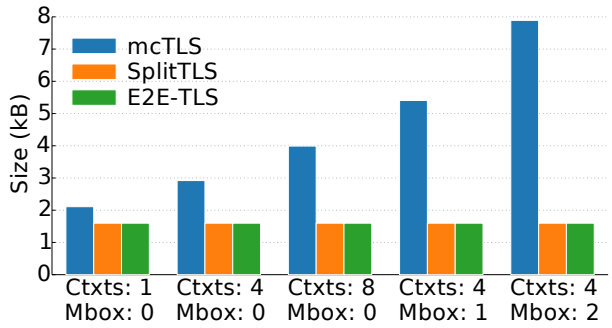

Figure 8: Handshake sizes.

5.2 Data Volume Overhead

Most of mcTLS' data overhead comes from the handshake, and it increases with the number of middleboxes and contexts (due to certificates and context key distribution). Figure 8 shows the total size of the handshake for different numbers of contexts and middleboxes. For a base configuration with one context and no middleboxes, the `mcTLS` handshake is 2.1 kB compared to 1.6 kB for `SplitTLS` and `E2E-TLS`. Note that the handshake size is independent of the file size.

Next, each record carries MACs (three in mcTLS, one in TLS). Their impact depends on the application's sending pattern—smaller records mean larger overhead. For the web browsing experiments in §5.1, the median MAC overhead for `SplitTLS` compared to `NoEncrypt` was 0.6%; as expected, `mcTLS` triples that to 2.4%.

Finally, padding and header overhead are negligible.

Takeaway: *Apart from the initial handshake overhead, which is negligible for all but short connections, mcTLS introduces less than 2% additional overhead for web browsing compared to* ***`SplitTLS`*** *or* ***`E2E-TLS`****.*

5.3 CPU Overhead

Figure 5 (left) shows the number of connections (only handshakes) per second the server can sustain. We see that the extra asymmetric encryption for distributing middlebox key material takes a toll. The `mcTLS` server handles 23% fewer connections than `SplitTLS` or `E2E-TLS`; that number drops to 35% fewer as the number of contexts, and therefore the number of partial context keys the server must encrypt, increases. We note two things: (1) key distribution optimizations, which we intend to pursue in future work, can shrink this gap, and (2) the server can reclaim this lost performance if the client handles key generation/distribution (§3.6).

The results for the middlebox are more interesting (Figure 5 right). First, `E2E-TLS` significantly outperforms `mcTLS` and `SplitTLS` (note the change in Y scale) because it does not participate in a TLS handshake. Second, `mcTLS` performs *better* than `SplitTLS` because in `SplitTLS` the proxy has to participate in *two* TLS handshakes. These results show it is not only feasible, but practical to use middleboxes in the core network.

Takeaway: *`mcTLS` servers can serve 23%–35% fewer connections per second than* ***`SplitTLS`****, but `mcTLS` middleboxes can serve 45%–75% more.*

5.4 Deployment

To begin understanding deployability, we built an extension to the Ruby SSL library that adds support for mcTLS with less than 250 lines of C code. Using the extension, we then built a 17 line Ruby web client with the same functionality as our C/OpenSSL-based evaluation client. While a bit more work is needed to make the extension more Ruby-like, the potential to easily write mcTLS-enabled mobile apps with developer-friendly tools like RubyMotion[3] is promising.

We also modified the OpenSSL `s_time` benchmarking tool to support mcTLS. Again, minimal changes were required: less than 30 new lines of C code were added, and about 10 lines were slightly changed. This means that relatively minor developer effort is required to gain the full benefits of mcTLS.

While supporting fine-grained access control requires the minimal effort of assigning data to a context and setting middlebox permissions for those contexts, many of the benefits of mcTLS are immediately available with just support from the HTTP client library and server. For example, HTTP libraries could use the 4-Context strategy by default, requiring no additional program-

[3]http://www.rubymotion.com/

ming or effort from application developers. Finally, we note that clients and servers can easily fall back to regular TLS if an mcTLS connection cannot be negotiated.

Takeaway: *Upgrading an application or library to mcTLS appears to be straightforward and easy.*

6. DISCUSSION

6.1 Middlebox Discovery

mcTLS assumes that the client has a list of middleboxes prior to initiating a handshake, which it includes in the `ClientHello`. Building this list is largely orthogonal to mcTLS itself; many existing mechanisms could be used, depending on *who* is trying to add a middlebox to the session. For example:

- **Users** or **system administrators** might configure the client (application or OS) directly (e.g., the user might point his browser toward Google's SPDY proxy). If users express interest in, e.g., a "nearby" data compression proxy, rather than a particular one, clients could discover available proxies using mDNS [10] or DNS-SD [9].
- **Content providers** could specify middleboxes to be used in any connection to its servers using DNS.
- **Network operators** can use DHCP or PDP/PDN to inform clients of any required middleboxes (e.g., virus scanners).

If a priori mechanisms like these are not flexible enough, the handshake could be extended to allow, e.g., on-path middleboxes to insert themselves (subject to subsequent approval by the endpoints, of course) during session setup. The costs and benefits of this are not immediately clear; we leave working out the details of more complex session negotiation for future work.

6.2 User Interface

The technical solution for adding middleboxes to secure communication sessions means little without suitable interfaces through which users can control it. The primary challenges for such an interface are:

- *Indicating to the user that the session is "secure."* Re-using the well-known lock icon is misleading, since the semantics of TLS and mcTLS differ.
- *Communicating to the user who can do what.* Which middleboxes can read the user's data? Which can modify it? What modifications do they make? Who owns the middleboxes? Who added them to the session and why?
- *Allowing users to set access controls.* Which sessions can a middlebox see? Within those sessions, which fields can it read or write? The difficulty is making such controls simple and scalable. For instance, asking users to set middlebox permissions for each domain they visit is not practical.

Designing a satisfactory interface atop mcTLS is a project in and of itself, one we cannot begin to do justice here.

	R1	R2	R3	R4	R5
mcTLS	•	•	•	•	•
(1) Custom Certificate					
(2) Proxy Certificate Flag	○			○	
(3) Session Key Out-of-Band	•	•		○	
(4) Custom Browser					
(5) Proxy Server Extension	○	○	○	○	

(• = full compliance; ○ = partial compliance)

Table 4: Design principle compliance for mcTLS and competing proposals.

7. RELATED WORK

There has been a lot of recent interest, particularly in industry, for including intermediaries in encrypted sessions. First we describe five previous proposals for doing so in the context of TLS; as shown in Table 4, none of them meets all five of our requirements. Then we discuss alternatives that replace TLS altogether.

(1) Custom Root Certificate Section 2.2 describes a common technique in which network administrators install a custom root certificate on the client.

Discussion: This technique does not meet any of our requirements. First, the server, and in many cases the client, is not aware of the existence of the middlebox (R4) so it clearly cannot authenticate it (R1). Second, the middlebox has full read and write access to all data in the session (R5). Finally, since the client has no control after the first hop, there is no guarantee about the secrecy, integrity, or authenticity of the data (R2, R3) or the identity of the server (R1).

(2) "I'm a proxy" Certificate Flag A 2014 IETF draft from Ericsson and AT&T proposes using the X.509 Extended Key Usage extension to indicate that a certificate belongs to a proxy [20]. Upon receiving such a certificate during a TLS handshake, the user agent would omit the domain name check (presumably with user permission) and establish a TLS session with the proxy, which would in turn open a connection with the server. Based on user preferences, the user agent might only accept proxy certificates for certain sessions.

Discussion: In this case, the client is made explicitly aware of the presence of the middlebox, so it can authenticate it (R1) and can control its use on a per connection basis (R4). The client still cannot authenticate the server and the server is unaware of the middlebox. R2, R3, and R5 remain unaddressed.

(3) Pass Session Key Out-of-Band Another IETF draft, this one from Google, assumes that the client maintains a persistent TLS connection with the proxy and multiplexes multiple sessions over that connection (much how Google's data compression proxy operates). After establishing an end-to-end TLS connection with the server (which the proxy blindly forwards), the client passes the session key to the proxy before transmit-

ting data on the new connection [28]. Again, the user agent can selectively grant the proxy access on a per-connection basis based on user preference.

Discussion: Compared with (1), this solution has the additional benefit that the client authenticates both the middlebox and the server (R1) and knows that the session is encrypted end-to-end (R2). R3, R4, and R5 are still partially or completely unaddressed.

(4) Ship a Custom Browser A fourth option is to modify the browser itself to accept certificates from certain trusted proxies. This is the approach Viasat is taking for its Exede satellite Internet customers [18], arguing that caching and prefetching are critical on high-latency links.

Discussion: This solution is essentially the same as (1), so it also fails all requirements. In addition, it has the drawback that a custom browser may not be updated quickly, is expensive to develop and maintain, and may be inconvenient to users.

(5) Proxy Server Extension The most promising approach so far is Cisco's TLS Proxy Server Extension [21]. The proxy receives a ClientHello from the client, establishes a TLS connection with the server, and includes the server's certificate and information about the ciphersuite negotiated for the proxy-server connection in a ProxyInfoExtension appended to the ServerHello it returns to the client. The client can then check both the proxy's and the server's certificate.

Discussion: The client must *completely* trust the middlebox to provide honest information about the server certificate and ciphersuite, so this solution only partially fulfills R1, R2, and R3. The proxy is not necessarily visible to the server, so only partial R4. Finally, the proxy has read/write access to all data (R5).

Other Approaches An alternative to TLS-based techniques is an extension to IPsec that allows portions of the payload to be encrypted/authenticated between the two end-points of a security association and leaves the remainder in the clear [16]. The authors target this architecture for securely enabling intermediary-based services for wireless mobile users. This solution leaves data for middleboxes completely unencrypted (R2); R1 and R3 are also violated. Furthermore, this approach does not allow explicit control of the data flow to different entities (R4).

Tcpcrypt [6, 5] is an alternative proposal for establishing end-to-end encrypted sessions. Compared with TLS, it reduces the overhead on the server, leaves authentication to the application, can be embedded in the TCP handshake, and uses a session ID to unambiguously identify the endpoints of a session. Similar to TLS, tcpcrypt supports communication between two endpoints only, but we believe that the concepts of encryption contexts and contributory context keys could be applied to it as well. However, because of mcTLS's increased handshake size, it may no longer be possible to embed the entire handshake in the TCP handshake.

An alternative to a transport-layer protocol, like TLS or mcTLS, is supporting trusted intermediaries at the network layer. The Delegation-Oriented Architecture (DOA) [34] and Named Data Networking (NDN) [15] do this with their own security mechanisms and properties. We chose to modify TLS due to its widespread use, making it the perfect vehicle for immediate experimentation and incremental deployment.

8. CONCLUSION

The increasing use of TLS by Internet services provides privacy and security, but also leads to the loss of capabilities that are typically provided by an invisible army of middleboxes offering security, compression, caching, or content/network resource optimization. Finding an incrementally deployable solution that can bring back these benefits while maintaining the security expectations of clients, content providers, and network operators is not easy. mcTLS does this by extending TLS, which already carries a significant portion of HTTP traffic. mcTLS focuses on transparency and control: (1) trusted middleboxes are introduced at the consent of both client and server, (2) on a per session basis, (3) with clear access rights (read/write), and (4) to specific parts of the data stream.

We show that building such a protocol is not only feasible but also introduces limited overhead in terms of latency, load time, and data overhead. More importantly, mcTLS can be incrementally deployed and requires only minor modifications to client and server software to support the majority of expected use cases. By using mcTLS, secure communication sessions can regain lost efficiencies with explicit consent from users and content providers.

Acknowledgments

Many thanks to the reviewers for their comments and to our shepherd, Sharon Goldberg, for going above and beyond. This research was funded in part by NSF under award number CNS-1345305, by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship 32 CFR 168a, and by the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

9. REFERENCES

[1] http://mctls.org.
[2] Dashboards—android developers. https://developer.android.com/about/dashboards/index.html. Accessed: Dec. 2014.
[3] V. Agababov, M. Buettner, V. Chudnovsky, et al. Flywheel: Google's data compression proxy for the mobile web. NSDI '15, pages 367–380, Oakland, CA, May 2015. USENIX Association.
[4] D. Beaver. HTTP2 Expression of Interest. http://lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html, 7 2012.

[5] A. Bittau, D. Boneh, M. Hamburg, et al. Cryptographic protection of tcp streams (tcpcrypt), Feb. 2014.

[6] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[7] I. Brown. End-to-end security in active networks. In *University College London PhD Thesis*, 2001.

[8] D. Cameron. The internet and pornography: Prime minister calls for action. https://www.gov.uk/government/speeches/the-internet-and-pornography-prime-minister-calls-for-action. Accessed: Jan. 2015.

[9] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), Feb. 2013.

[10] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), Feb. 2013.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[12] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices. MobiSys '10, New York, NY, USA, 2010. ACM.

[13] J. Erman, A. Gerber, M. Hajiaghayi, et al. To cache or not to cache: The 3g case. *Internet Computing, IEEE*, 15(2):27–34, March 2011.

[14] M. Hoque, M. Siekkinen, and J. K. Nurminen. On the energy efficiency of proxy-based traffic shaping for mobile audio streaming. CCNC, pages 891–895. IEEE, 2011.

[15] V. Jacobson, D. K. Smetters, J. D. Thornton, et al. Networking named content. CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[16] S. Kasera, S. Mizikovsky, G. S. Sundaram, and T. Y. C. Woo. On securely enabling intermediary-based services and performance enhancements for wireless mobile users. In *Workshop on Wireless Security, 2003*, 2003.

[17] A. Langley, N. Modadugu, and W.-T. Chang. Overclocking SSL. https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html, 6 2010.

[18] P. Lepeska. Trusted proxy and the cost of bits. http://www.ietf.org/proceedings/90/slides/slides-90-httpbis-6.pdf, 7 2014.

[19] V. Liu, S. Han, A. Krishnamurthy, and T. Anderson. An Internet Architecture Based on the Principle of Least Privilege. Technical Report UW-CSE-12-09-05, University of Washington, Sept. 2012.

[20] S. Loreto, J. Mattsson, R. Skog, et al. Explicit Trusted Proxy in HTTP/2.0. Internet-Draft draft-loreto-httpbis-trusted-proxy20-01, IETF Secretariat, Feb. 2014.

[21] D. McGrew, D. Wing, Y. Nir, and P. Gladstone. TLS Proxy Server Extension. Internet-Draft draft-mcgrew-tls-proxy-server-01, IETF Secretariat, July 2012.

[22] A. Menezes and B. Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. *International Journal of Applied Cryptography*, 2(2):154–158, 2010.

[23] J. C. Mogul and G. Minshall. Rethinking the tcp nagle algorithm. *SIGCOMM CCR*, Jan. 2001.

[24] C. Muthukrishnan, V. Paxson, M. Allman, and A. Akella. Using strongly typed networking to architect for tussle. Hotnets-IX, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[25] D. Naor, A. Shenhav, and A. Wool. Toward securing untrusted storage without public-key operations. StorageSS '05, pages 51–56, New York, NY, USA, 2005. ACM.

[26] D. Naylor, A. Finamore, I. Leontiadis, et al. The Cost of the "S" in HTTPS. CoNEXT '14, pages 133–140, New York, NY, USA, 2014. ACM.

[27] C. Nikolouzakis. Encrypted traffic grows 40% post edward snowden nsa leak. http://www.sinefa.com/blog/encrypted-traffic-grows-post-edward-snowden-nsa-leak. Accessed: Jan. 2015.

[28] R. Peon. Explicit Proxies for HTTP/2.0. Internet-Draft draft-rpeon-httpbis-exproxy-00, IETF Secretariat, June 2012.

[29] F. Qian, S. Sen, and O. Spatscheck. Characterizing resource usage for mobile web browsing. MobiSys '14, pages 218–231, New York, NY, USA, 2014. ACM.

[30] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000.

[31] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[32] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[33] J. Sherry, S. Hasan, C. Scott, et al. Making middleboxes someone else's problem: Network processing as a cloud service. SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[34] M. Walfish, J. Stribling, M. Krohn, et al. Middleboxes no longer considered harmful. OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[35] N. Weaver, C. Kreibich, M. Dam, and V. Paxson. Here be web proxies. In *Passive and Active Measurement*, pages 183–192. Springer, 2014.

[36] C. Wisniewski. Path and hipster iphone apps leak sensitive data without notification. https://nakedsecurity.sophos.com/2012/02/08/apple-mobile-apps-path-and-hipster-and-leak-sensitive-data-wi Accessed: May 2015.

[37] S. Woo, E. Jeong, S. Park, et al. Comparison of caching strategies in modern cellular backhaul networks. MobiSys '13, pages 319–332, New York, NY, USA, 2013. ACM.

[38] X. Xu, Y. Jiang, T. Flach, et al. Investigating transparent web proxies in cellular networks. PAM '15.