And Then There Were More: Secure Communication for More Than Two Parties

David Naylor[⋆‡], Richard Li[†‡] Christos Gkantsidis[‡], Thomas Karagiannis[‡], Peter Steenkiste[⋆]
[⋆]Carnegie Mellon University, [†]University of Utah, [‡]Microsoft Research

ABSTRACT

Internet communication today typically involves intermediary *middleboxes* like caches, compression proxies, or virus scanners. Unfortunately, as encryption becomes more widespread, these middleboxes become blind and we lose their security, functionality, and performance benefits. Despite initial efforts in both industry and academia, we remain unsure how to integrate middleboxes into secure sessions—it is not even clear how to *define* "secure" in this multi-entity context.

In this paper, we first describe a design space for secure multi-entity communication protocols, highlighting tradeoffs between mutually incompatible properties. We then target real-world requirements unmet by existing protocols, like *outsourcing middleboxes to untrusted infrastructure* and *supporting legacy clients*. We propose a security definition and present *Middlebox TLS (mbTLS)*, a protocol that provides it (in part by using Intel SGX to protect middleboxes from untrusted hardware). We show that mbTLS is deployable today and introduces little overhead, and we describe our experience building a simple mbTLS HTTP proxy.

CCS CONCEPTS

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

KEYWORDS

TLS, middleboxes, trusted computing, SGX

ACM Reference format:
David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. 2017. And Then There Were More: Secure Communication for More Than Two Parties. In *Proceedings of CoNEXT '17: The 13th International Conference on emerging Networking EXperiments and Technologies, Incheon, Republic of Korea, December 12–15, 2017 (CoNEXT '17),* 13 pages.
DOI: 10.1145/3143361.3143383

1 INTRODUCTION

Internet communication is no longer two endpoints exchanging messages over a dumb packet-forwarding core; our data is frequently processed by intermediary *middleboxes* like caches, compression proxies, intrusion detection systems, or virus scanners. For example, all four major U.S. mobile carriers use HTTP proxies [55] and a typical enterprise network has roughly as many middleboxes as it does routers and switches [47]. As the use of encryption online increases (as of 2014, nearly half of all Web flows used HTTPS [40]), these middleboxes become blind and can no longer perform their jobs, prompting both the academic community and industry to consider the question: *how do we integrate middleboxes into secure communication sessions?*

Because TLS—the standard secure communication protocol used in the Internet—is designed for exactly two parties, the current practice is to "split" the session into two separate TLS connections: the middlebox impersonates the server to the client and opens a second connection to the server. Doing so drastically weakens security, in part because the client cannot explicitly authenticate the middlebox or be sure that the middlebox will properly authenticate the server [23].

These weaknesses underscore the fact that, while the properties of TLS are well-understood in the two-party case, it is unclear how to define "secure" in the multi-party case. Recent work has proposed new protocols alongside new security definitions. For example, Multi-Context TLS (mcTLS) [41] allows endpoints to restrict which parts of the data stream the middleboxes can read or write and BlindBox [48] allows middleboxes to operate directly on encrypted data. However, these are largely point solutions that, while useful in certain scenarios, leave several real-world needs unmet.

In this paper, we focus on three practical requirements that are so far unaddressed. First, there is increasing interest in outsourcing middleboxes to third-party cloud providers [2, 11, 31, 47] or to ISPs [5, 9, 12]. This setting poses a new challenge: the owner of middlebox software and the owner of the hardware on which it runs are not the same. If the infrastructure is untrusted, existing protocols like "split TLS" and mcTLS cannot provide the standard security properties TLS does today because (1) session data and keys are visible in memory and (2) the endpoints cannot tell if the infrastructure provider ran the intended code. Second, in order to be realistically deployable, any new protocol should be reverse compatible with TLS (i.e., upgraded endpoints should be able to include middleboxes in sessions with legacy TLS endpoints). And third, the ability to discover middleboxes on-the-fly is a practical requirement in many contexts. For example, if a service provider places proxies in edge ISPs, directing each client to its local proxy using DNS (1) is an unnecessary configuration burden and (2) is brittle, since clients can use non-local DNS resolvers like OpenDNS.

Work done while David and Richard were interns at Microsoft Research.

CoNEXT '17, Incheon, Republic of Korea
© 2017 ACM. 978-1-4503-5422-6/17/12…$15.00
DOI: 10.1145/3143361.3143383

We make two primary contributions in this paper. First, we carefully articulate a **design space for secure multi-entity communication protocols** (and use it to place previous work in context). We describe the different properties that such a protocol might have and argue why some combinations are impossible to achieve at once, suggesting that the community needs to either carefully select which set of properties to support or develop different protocols for different use cases. Second, we present ***Middlebox TLS (mbTLS)***, a protocol for secure multi-entity communication that addresses the needs described above:

(1) mbTLS is immediately deployable. Unlike mcTLS or BlindBox, it interoperates with legacy TLS endpoints and provides in-band middlebox discovery.

(2) mbTLS protects session data from third party infrastructure providers. mbTLS leverages trusted computing technology, like Intel SGX [15, 26, 37], to isolate the middlebox execution environment from the third party infrastructure.

(3) mbTLS provides other important security properties unique to multi-party settings. For example, mbTLS guarantees that data visits its middleboxes in the order specified by the endpoints, prevents attackers from learning whether or not a middlebox modified a piece of data before forwarding it, and gives endpoints guarantees about what code a middlebox is running.

We implement mbTLS using OpenSSL and the Intel SGX SDK and evaluate its deployability and performance, showing that (1) mbTLS is immediately deployable, (2) mbTLS reduces CPU load on the middlebox and adds reasonable overhead on the server, and (3) running inside an SGX enclave does not degrade throughput.

Our hope is that mbTLS represents a significant and practical step toward bridging the gap between end-to-end security and the reality that middleboxes are not going away.

2 MULTI-PARTY COMMUNICATION

Most network communication sessions today involve more parties than just a client and a server. By and large, these additional entities fall into one of three categories:

Network-Layer Middleboxes *(e.g., firewall, NAT, layer 3 load balancer).* These process data packet by packet and do not need to reconstruct or access application layer data.

Application-Layer Middleboxes *(e.g., virus scanner, IDS, parental filter, cache, compression proxy, application layer load balancer).* These *do* need access to application layer data.

Application-Layer Delegates *(e.g., CDNs).* In contrast to middleboxes, which act as intermediaries between client and server *at communication time*, we use the term *delegate* for intermediaries that take on the role of the server during the session (though in terms of real-world relationships, they are still more naturally viewed as intermediaries). Content delivery networks (CDNs) are a good example; clients talk to directly to CDN servers; the origin server might not be involved at all.

As we move toward an Internet where encryption is ubiquitous, it is becoming clear that we do not have an adequate protocol for secure multi-entity communication, nor do we know exactly what properties one should provide. In the two-party case, it is well understood what security properties we want and how to achieve them; we have been using TLS for years. But in the multi-party case, there are still two key unanswered questions: *(1) what security properties should hold for sessions involving three or more parties?* and *(2) what are the best mechanisms to enforce those properties?*

The answers to these questions will be different for each of the three categories of intermediary. In this paper, we focus on *secure multi-entity communication for* ***application-layer middleboxes***. Even among just application-layer middleboxes, security needs are potentially diverse—for example, intrusion detection systems and compression proxies behave very differently and trust relationships differ between an administrator-mandated virus scanner and an opt-in compression service—which suggests there may not be a single one-size-fits-all solution. Our first step toward answering these questions is to articulate the design space.

2.1 Design Space

TLS Security Properties. TLS currently provides the following properties in the two-party case. Clearly we want these properties in the multi-party case as well, but it turns out there are multiple ways to extend the two-party definitions to the multi-party case.

Data Secrecy and *Data Authentication.* Only the endpoints can read and write (create, modify, delete, replay, or re-order) session data. Furthermore, with a modern cipher suite, communication is *forward secret* (the compromise of a long-term private key does not help an attacker access previous sessions' data). To extend these properties beyond two parties, the following questions arise.

> *Granularity of Data Access.* `yes/no` `RW/RO/None` `func. crypto`
> Do middleboxes have complete access to session data, or do they have some level of partial access? This could mean they can read/write some bytes but not others (e.g., HTTP headers but not HTTP bodies), as in mcTLS [41], or that they can perform a limited set of operations over encrypted data (e.g., search for patterns), as in BlindBox [48].

> *Definition of "Party."* `machine` `program`
> When a party is added to a session, is session data accessible to anyone with physical access to the machine, or only to the middlebox service software? This distinction becomes important when middleboxes are outsourced to third-party hardware (e.g., cloud providers or ISPs).

Entity Authentication. Each endpoint can verify that the other is operated by the expected entity by verifying that they possess a private key that a CA has certified belongs to that entity. To extend this property beyond two parties, the following question arises.

> *Definition of "Identity."* `owner` `code`
> When a party in a session verifies the "identity" of another party, what is it checking? That the machine is owned by the expected entity (e.g., this is a YouTube server)? That the machine is running the expected software and is correctly configured (e.g., Apache v2.4.25 with only strong TLS cipher suites enabled)? Both?

Other Security Properties. In the multi-party case, a number of new security properties arise.

Path Integrity. `yes` `no`
Does the protocol enforce that data must traverse middleboxes in a fixed order (and that they cannot be skipped)? Path order can impact security, especially when middleboxes perform filtering/sanitization functions.

Data Change Secrecy. `none` `value` `value + size`
Can the adversary learn anything about the communication by observing data before and after a middlebox? Protocols could offer no protection (adversary knows any time a middlebox makes a change), *value* protection (adversary does not learn when a middlebox changes a message so long as the size stays constant), or *value + size* protection (adversary does not learn about any changes).

Authorization. `0 endpts` `1 endpt` `both endpts` `endpts + mboxes`
Who gets to add a middlebox to a session (and decide what permissions it has)? Do both endpoints need to be made aware, so they can terminate the session if they do not approve? Only one? Should middleboxes be aware of other middleboxes?

Other Properties. Finally, there are a number of non-security properties that impact protocol deployability and usability.

Legacy Endpoints. `both upgrade` `1 legacy` `both legacy`
Do both endpoints need to be upgraded to a new protocol, or can one or both be legacy TLS endpoints?

In-Band Discovery. `yes` `yes + 1 RTT` `no`
Does the protocol allow endpoints to discover on-path middleboxes on-the-fly? If so, does adding discovered middleboxes add time to the handshake?

Computation. `arbitrary` `limited`
Does the protocol restrict what kinds of jobs middleboxes can perform (i.e., arbitrary computation vs. a limited set of operations, like pattern matching)?

2.2 Design Tradeoffs

Next we look at existing approaches in the context of this design space, highlighting how the mechanisms they introduce interact with the properties described above. Often, a mechanism that provides a particular property (denoted like `this`) along one dimension often eliminates options along another (denoted like ~~`this`~~).

TLS Interception with Custom Root Certificates [23, 27, 42] is the standard approach today. First, an administrator provisions clients with custom root certificates (this is easy in managed environments like corporate networks). Then, when the client opens a new connection, the middlebox intercepts it, impersonates the intended server by fabricating a certificate for that domain, and opens a second connection to the server. Though this means both clients can be legacy TLS clients [*Legacy:* `both legacy`], it also makes it impossible for clients to authenticate the server [*Authentication:* ~~`owner`~~]—they must trust the middlebox to do so (trust which, in practice, is often misplaced [23]).

Multi-Context TLS (mcTLS) [41] offers *access control*—endpoints can restrict which parts of the data middleboxes can access and whether that access is read/write or read only [*Data access:* `RW/RO/None`]. It does this by encrypting different parts of the data with different keys and only giving middleboxes certain keys. This requires that both endpoints run mcTLS, precluding legacy endpoints, since a legacy TLS endpoint only knows what to do with one key [*Legacy:* ~~`1 legacy`~~ ~~`both legacy`~~]. Furthermore, each endpoint generates part of the key material for each of these keys, ensuring that a middlebox only gains access if both endpoints agree [*Authorization:* `both endpts`]. This also prevents legacy support.

BlindBox [48] offers *searchable encryption*—pattern-matching middleboxes like intrusion detection systems can operate directly on encrypted data [*Data access:* `func. crypto`]. But searchable encryption only works for pattern-matching; it cannot support other middleboxes, like compression proxies, that perform arbitrary computation [*Computation:* ~~`arbitrary`~~]. It also requires that both endpoints use BlindBox's encryption scheme [*Legacy:* ~~`1 legacy`~~ ~~`both legacy`~~].

Middlebox TLS (mbTLS) ***(this paper).*** We will soon see this for mbTLS too: for example, mbTLS uses a different symmetric key for each "hop" in the session, allowing mbTLS to provide path integrity [*Path integrity:* `yes`], but making it impossible to support *two* legacy endpoints [*Legacy:* ~~`both legacy`~~].

The takeaway is this: ***there is no one-size-fits-all solution for secure communication with application-layer middleboxes***. Each protocol here gives up desirable properties in order to provide others. Different properties, and therefore different protocols, will lend themselves best to different use cases. For instance, mcTLS is ideal for read-only middleboxes since its access control mechanisms provide cryptographic guarantees that the middlebox will not modify data. And for pattern-matching middleboxes like IDSes, BlindBox provides better privacy guarantees than mbTLS or mcTLS. Unfortunately, mcTLS and BlindBox achieve these properties using mechanisms that make them harder to deploy; in this paper, our goal is a protocol that prioritizes deployability and operability.

3 MIDDLEBOX TLS

The solutions introduced in §2.2 serve niche needs while failing to address several common ones, making them harder to deploy and reducing the incentive to do so in the first place. In this section, we present ***Middlebox TLS***, or ***mbTLS***, a protocol for secure communication sessions that include application-layer middleboxes. We saw in §2.2 that it is hard to build a super-protocol incorporating all the good features from §2.1; instead, we target the following three common-case, real-world needs:

(1) Immediate Deployability: First, mbTLS needs to **interoperate with legacy endpoints**. BlindBox [48] and mcTLS [41] require both endpoints to be upgraded. Others protocols require that at least the client be upgraded [33, 34, 36], meaning servers cannot include middleboxes in a session with a legacy client. Realistically, however, it is not an option to wait until every client in the Internet is upgraded, especially since as many as 10% of HTTPS connections are *already* intercepted [23]. Second, **in-band middlebox discovery** is important for practical deployment in the use cases we target. For example, suppose a service provider places proxies

in edge ISPs. Directing clients to their local proxy using DNS (1) is an unnecessary configuration burden and (2) is brittle, since clients can use non-local DNS resolvers like OpenDNS. Another example is guest networks where administrators cannot feasibly configure every client device that joins (nor would those users want them to).

(2) Protection for Outsourced Middleboxes: There is an increasing interest in deploying middleboxes in third-party environments. This takes one of two forms. First, network functions can be outsourced to a cloud provider[1] that specializes in operating middleboxes, freeing network administrators from learning to operate specialized boxes and leveraging economy of scale to drive down costs [2, 11, 47]. Second, deploying middleboxes in client ISPs can help lower latency or bandwidth costs [5, 9, 12]. (For example, Google's Edge Network proxies connections using nodes in client ISPs [5].) In both cases, *the logical owner of the network function and the operator of the hardware on which it runs are different.* Since the middlebox infrastructure might not be trusted, mbTLS must **protect session data from the middlebox infrastructure** in addition to traditional network attackers.

(3) Middlebox Accountability: Endpoints may be more comfortable giving a middlebox access to their data if they have guarantees about its behavior. mcTLS and BlindBox provide this to an extent, but BlindBox only supports pattern matching and in mcTLS, once a middlebox is granted data access, it can do anything it wants. As a first step, mbTLS allows endpoints to **verify middlebox code identity**; in the future, combined with program analysis, this could provide guarantees about middlebox behavior.

3.1 Threat Model

Parties. To capture the threats implied by these requirements, we identify six distinct parties and label each as "trusted" or "untrusted," where trusted means authorized to read and write session data.

Client (C) [trusted]: The user, their machine, and the software they run (e.g., a web browser). We assume any other software running on the machine is trusted (i.e., its misbehavior is out of scope).

Service Provider (S) [trusted]: The company providing the online service, its servers, and the software it runs (e.g., a web server). We do not consider attacks by other software running on *S*'s servers or by malicious employees.

Third Parties (TP) [untrusted]: Anyone else with access to network traffic, such as ISPs or coffee shop Wi-Fi sniffers.

Middlebox Software (MS) [trusted]: The middlebox software that processes session data.

Middlebox Service Provider (MSP) [trusted]: The entity offering the middlebox service.

Middlebox Infrastructure Provider (MIP) [untrusted]: The entity providing the hardware on which the *MS* runs.

The *MIP* could be the *MSP* itself or a third party such as an edge ISP or a dedicated cloud middlebox service, in which case we assume this company, its employees, its hardware, and any other software running on its machines are *not* trusted. For example, suppose Google implemented its Flywheel proxy [14] using Apache httpd running on Amazon EC2. In this case, *MS* = Apache httpd, *MSP* = Google, and *MIP* = Amazon. By distinguishing between *MS* and *MIP*, we can require mbTLS to permit the *MS*, but not the *MIP*, to access session data [*Party:* `program`]

Adversary Capabilities. We assume an active, global adversary that can observe and control any untrusted part of the system. In the network, the adversary can observe, modify, or drop any packet and inject new ones. On the middlebox infrastructure, the adversary has complete access to all hardware (e.g., it can read and manipulate memory) and software (e.g., it can execute arbitrary code, including privileged code like a malicious OS). This includes the ability to modify or replace *MS* code sent by the MSP to be executed by the MIP. We assume the adversary is computationally bounded (i.e., cannot break standard cryptographic primitives) and cannot compromise trusted computing hardware (e.g., Intel SGX-enabled CPUs). Side channel attacks (e.g., based on traffic or cache access patterns), exploitable flaws in middlebox software, and denial of service are out of scope.

3.2 mbTLS Properties

Based on the design requirements above, we define "secure" for mbTLS by the following four security properties.

P1 **Data Secrecy.** **P1A** The adversary must not be able to read session data. [*Access:* `yes/no`] **P1B** Communication should be *forward secret* (the compromise of a long-term private key does not help an attacker access previous sessions' data). **P1C** The adversary should learn nothing more from observing ciphertext than it would if each "hop" were its own, independent TLS connection. [*Change secrecy:* `value`]

P2 **Data Authentication.** The adversary must not be able to modify, delete, or inject session data. This includes replaying or re-ordering data. More formally, if a session consists of an ordered set of nodes $N_1, \ldots, N_m$, then any data received by N_i must be a prefix of the data sent by N_{i-1}.

P3 **Entity Authentication.** Endpoints must be able to verify they are talking to the "right thing." This encompasses two properties. **P3A** Each endpoint can verify that the other endpoint is operated by the expected entity and that each *MS* is operated by the expected *MSP* (e.g., this proxy is operated by AT&T). (This requires the entity being verified to have a certificate.) [*Identity:* `owner`] **P3B** Each endpoint can verify that the other endpoint and each *MS* is running the expected software and that it is correctly configured (e.g., Apache v2.4.25 with only strong TLS cipher suites enabled).[2] (This requires the entity being verified to support secure execution environments—see below.) [*Identity:* `code`]

P4 **Path Integrity.** Each endpoint fixes an order for its middleboxes. It must not be possible for any other entity (including a middlebox or the other endpoint) to cause session data to be processed by middleboxes in a different order (including skipping a

[1] This trend is encouraged by maturing technology for running middlebox applications on commodity hardware (NFV) [24, 25, 35, 46], including commercial offerings [3, 7, 8].

[2] Note that this provides no guarantees about the behavior or bug-freeness of that software; for this, the code identity verification must be coupled with software analysis/verification, which is out of the scope of this paper.

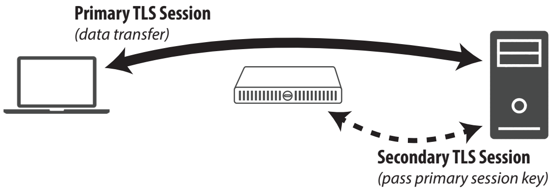

Figure 1: Naïve Approach. Establish a TLS session end-to-end and pass the session key to the middlebox over a secondary TLS session.

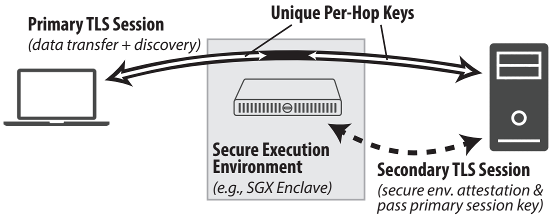
Primary TLS Session
(data transfer + discovery)
Unique Per-Hop Keys
Secure Execution Environment
(e.g., SGX Enclave)
Secondary TLS Session
(secure env. attestation & pass primary session key)

Figure 2: mbTLS Approach. Generate unique keys for each "hop" and run middleboxes in secure execution environments.

middlebox). More formally, if a session consists of an ordered set of nodes $N_1, \ldots, N_m$, then a record received by N_i was either already processed by N_{i-1} or was originated by N_{i-1}. [*Path integrity:* yes]

In addition to these security properties, we also have the following performance and deployability-related goals:

P5 **Legacy Interoperability.** mbTLS should work with legacy TLS endpoints (client or server) so long as one of the endpoints has been upgraded. [*Legacy endpoints:* 1 legacy]

P6 **In-Band Discovery.** mbTLS should discover on-path middleboxes during session setup. [*Discovery:* yes]

P7 **Minimal Overhead.** mbTLS should introduce as little overhead (compared to TLS) as possible. Importantly, mbTLS should not add any round trips to the TLS handshake.

3.3 Design Overview

Since TLS already provides many of the properties we want, one simple approach is the following: establish a regular TLS session between the client and the server, then pass the session keys to the middleboxes over separate, secondary TLS sessions (Figure 1) [43]. This provides many of the properties we want: data is encrypted and integrity-protected against changes from third parties (partial P1A), the communication is forward secret if a forward secure cipher suite is used P1B, the endpoints can verify one another's identities using certificates P3A, and a middlebox-aware endpoint can add middleboxes without support from the other endpoint P5.

However, using TLS in this way is insufficient in our threat model for three reasons: (1) it has no mechanism to provide path integrity since it was designed for two parties P4; (2) the same key is used for encryption on each "hop" in the session, making it simple for adversaries to compare records entering and leaving a middlebox to see if they changed P1C; and (3) the infrastructure provider can access session data in memory P1A, access key material in memory and use it to forge MACs P2, and potentially run software other than what was provided by the *MSP* P3B. Furthermore, TLS provides no discovery mechanisms P6.

We address these insufficiencies by introducing the following features (Figure 2), and we call the result ***Middlebox TLS (mbTLS)***.

- **In-Band Middlebox Discovery.** As long as one of the endpoints supports mbTLS, middleboxes can announce themselves and join the session (with endpoint approval) during the primary TLS handshake P6.
- **Secure Execution Environments.** Middleboxes can optionally run in a secure execution environment, like an Intel SGX enclave (see below),[3] which provides memory encryption, protecting session data and keys from an untrusted *MIP* P1A P2, and remote attestation, allowing endpoints to verify the software identity of the *MS* P3B. Endpoints may also run in secure environments to provide P3B.
- **Unique Per-Hop Keys.** Each "hop" uses its own symmetric keys for protecting session data. This prevents adversaries from delivering records to an out-of-sequence middlebox P4 and makes it impossible to tell when a middlebox forwards data without changing it P1C.

An Aside: Trusted Computing and SGX. Some features of mbTLS rely on trusted computing technology, like Intel's Software Guard Extensions (SGX) [15, 26, 37]. In particular, mbTLS uses two features provided by SGX—secure execution environments and remote attestation—though any trusted computing technology that offers these features, like Microsoft's Virtual Secure Mode (VSM) [21], would work as well. (Other technologies, like ARM TrustZone [1], offer similar functionality, but provide slightly different security guarantees.) We briefly describe these features now; if you are familiar with SGX, skip ahead to §3.4.

Secure Execution Environment. SGX allows applications to run code inside a secure environment called an *enclave*. An enclave is a region of protected memory; before cache lines are moved to DRAM, they are encrypted and integrity-protected by the CPU. As long as the CPU has not been physically compromised, malicious hardware or privileged software cannot access or modify enclave memory.

Remote Attestation. SGX can provide code running in an enclave with a special message, signed by the CPU, called an *attestation*, that proves to remote parties that the code in question is indeed running in an enclave on a genuine Intel CPU. The attestation includes a cryptographic hash of initial state of the enclave code and data pages (so the remote verifier can see that the expected code is running) as well as custom data provided by the enclave application (we use this to integrate attestation with the TLS handshake).

3.4 The mbTLS Protocol

At a high level, the endpoints do a standard TLS handshake, establishing a *primary TLS session*, which will eventually be used for data transfer. Each endpoint adds zero or more middleboxes to a session, which we refer to as *client-side* and *server-side* middleboxes and can be known a priori or discovered during the handshake P6. Each endpoint has no knowledge of the other's middleboxes (or if it has

[3] This will be come increasingly practical as more middleboxes are designed to run on commodity CPUs, on which features like SGX will soon be commonplace.

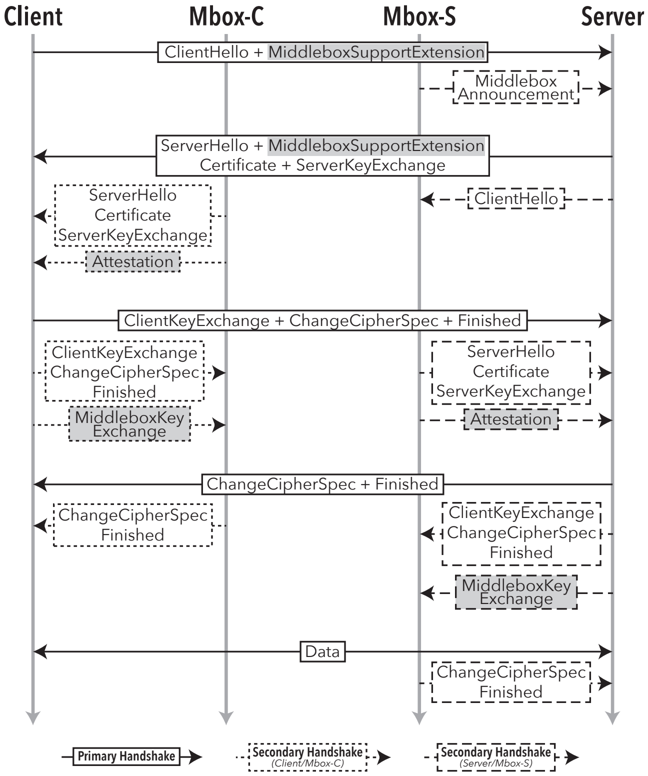

Figure 3: mbTLS Handshake. Note that it consists of multiple standard TLS handshakes, interleaved, with a few additional messages (shaded).

any at all) which means an mbTLS endpoint can inter-operate with legacy TLS endpoints P5. The endpoints simultaneously establish a *secondary TLS session* with each of their middleboxes. Once an endpoint has a secure channel to a middlebox (which can include verifying that the middlebox software is running in a secure execution environment, if the middlebox supports that), it sends the middlebox the key material it needs to join the primary session.

Data Transport. Our implementation creates a separate TCP connection for each hop. This is not strictly necessary—middleboxes that change the data stream could tweak TCP sequence numbers accordingly—but, since middleboxes operating on encrypted data must collect a complete TLS record before decrypting and operating on it, it makes sense to have reliable transport on each hop.

Control Messaging. mbTLS multiplexes the primary and secondary TLS sessions over the same TCP connections. Compared to opening secondary TCP connections, this reduces overhead P7 by (1) reducing TCP state on both middleboxes and endpoints, (2) keeping all handshake messages on the same path, and (3) keeping client-side middlebox discovery from adding a round trip. We introduce a new TLS record type (`Encapsulated`) to wrap secondary TLS records between a middlebox and its endpoint. These records consist of an outer TLS record header followed by a one byte *subchannel ID* and the encapsulated record. For details on mbTLS message formats, see Appendix A.

Middlebox Discovery. There are four ways a middlebox can become part of a session: client-side pre-configured, client-side discovered, server-side pre-configured, and server-side discovered.

Client-Side Middleboxes. If a client knows about a middlebox in advance (e.g., from user configuration), it opens a TCP connection directly to the middlebox. It then sends the primary `ClientHello`, which includes a new `MiddleboxSupport` TLS extension (top line in Figure 3) containing, among other things, a list of all middleboxes known a priori. The middlebox then opens a TCP connection to the next hop and forwards the `ClientHello`. Additionally, middleboxes on the default routing path can be discovered during the handshake P6. The `MiddleboxSupport` extension is a signal to these middleboxes that the client supports mbTLS; they optimistically split the TCP connection and, upon seeing the extension, join the handshake as described next.

In either case, when it sees the extension, the middlebox (1) forwards the `ClientHello` onward toward the server and (2) prepares to initiate its own secondary handshake with the client. In this secondary handshake, the middlebox plays the role of the server. The original, primary `ClientHello` serves double-duty as the `ClientHello` for the secondary handshake, allowing the middlebox to insert its own `ServerHello` right after forwarding the primary `ServerHello` toward the client, rather than waiting a round trip for a new secondary `ClientHello` P7. When the client receives the secondary `ServerHello`, if the middlebox was not configured by the client in advance, the client-side mbTLS library will ask the application for approval, which might in turn defer this decision to the user.

There may be multiple client-side middleboxes. Secondary handshake messages are sent in `Encapsulated` records, each middlebox with its own subchannel ID. Middleboxes wait until they see the primary `ServerHello`, buffer it, assign themselves the next available subchannel ID, inject their own secondary `ServerHello` into the data stream using that ID, and finally forward the primary `ServerHello`. This process ensures that each middlebox gets a unique subchannel ID with minimal coordination.

Server-Side Middleboxes. Server-side middleboxes can also be either pre-configured or discovered. In the pre-configured case, the server must somehow arrange for the handshake to traverse the (potentially off-path) middlebox. This could be done by changing the server's DNS entry to resolve to the middlebox or by asking the client—if it supports mbTLS—to include the middlebox in the `MiddleboxSupport` extension (e.g., by listing the middlebox in a new DNS record type).

In the discovery case, unlike the client, the server does not announce mbTLS support using the `MiddleboxSupport` extension for two reasons: first, the TLS spec forbids the server from including an extension in the `ServerHello` that the client did not include in the `ClientHello` [22]; relying on a `MiddleboxSupport` extension for the server would fail if the client does not also support mbTLS. Second, even if this were allowed, if server-side middleboxes waited to announce their presence until after the server's `ServerHello`, the middlebox-server handshake would finish after the primary handshake, lengthening the overall handshake to more than two RTTs (against P7).

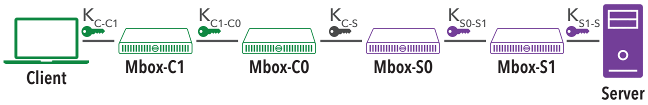

Figure 4: Unique Per-Hop Keys. Each hop encrypts and MAC-protects data with a different key—the client generates keys for the client-side hops (K_{C-C1} and K_{C1-C0}), the server generates keys for the server-side hops (K_{S0-S1} and K_{S1-S}), and the primary session key (K_{C-S}) bridges the sides.

Instead, server-side middleboxes optimistically announce themselves with a new `MiddleboxAnnouncement` message *before* they know if the server supports mbTLS. If it does not, then depending on its TLS implementation, it will either ignore the `MiddleboxAnnouncement` and the handshake will proceed without the middlebox, or the handshake will fail. (In either case, the middlebox will cache this information and not announce itself to this server again.) If the handshake fails, the client will need to retry. There is a potential danger that client software might interpret this to mean the server is running an out-of-date TLS stack and retry using an older version of TLS. We verified that Chrome and Firefox do *not* exhibit this behavior; Chrome will try a second TLS 1.2 handshake, and Firefox will not retry at all (meaning the user will need to manually click "Refresh"). Furthermore, in practice, we expect server-side middleboxes and servers will typically be under the same administrative control, in which case the middleboxes know that the server supports mbTLS and is expecting announcements.

Secure Environment Attestation. We have extended the TLS 1.2 handshake to optionally include a remote attestation (in addition to the standard certificate check). This extension is independent of mbTLS and could be used in standard client/server handshakes in the event that either endpoint wants to verify that the other runs in a secure environment. In the context of mbTLS, however, this capability will typically be used by endpoints to verify that (1) secondary TLS sessions with outsourced middleboxes terminate inside a secure execution environment **P1A** **P2** and (2) those middleboxes are running the expected software in the expected configuration **P3B**. The goal is to convince the endpoint that *only the middlebox application running in the enclave knows the TLS session key being established*. The main idea is the following: since the attestation includes the identity of the code, and we assume the code (application + mbTLS library) has been inspected and is trusted, then if the code asserts that it generated the secret key material for this handshake and did not export it, then the endpoint can trust this assertion.

The challenge becomes identifying "this handshake"—how can the endpoint be sure the attestation is fresh and not an adversary replaying an old attestation from a different handshake? This means, in addition to the code identity, the attestation must include some kind of *handshake identifier* to demonstrate freshness. For this, we use a hash of all handshake messages exchanged so far, much like TLS already does to authenticate the handshake in the `Finished` messages. Since this includes the client and server nonces, attestations will be unique to each handshake and the adversary cannot force them to repeat.

Unique Per-Hop Keys. At the end of an mbTLS handshake, the session looks like Figure 4. After finishing the secondary handshakes with its middleboxes, each endpoint generates a symmetric key for each hop on its side of the connection (e.g., the client generates K_{C-C1} and K_{C1-C0}). This prevents an adversary from causing records to skip a middlebox or traverse the middleboxes out-of-order **P4** and also prevents eavesdroppers from detecting whether or not a middlebox modified a record **P1C**. Endpoints distribute these keys to their middleboxes in `MiddleboxKeyExchange` records which are sent as (encrypted) data over the secondary TLS connections. (Just like the secondary handshake messages, secondary data records are sent in `Encapsulated` records in the same TCP stream.) At this point, the secondary TLS connections are not used again. The session key established during the primary handshake, K_{C-S}, serves as a "bridge" between the client-side and server-side middleboxes (or between a middlebox and a legacy endpoint **P5**).

3.5 Discussion

Session Resumption. mbTLS supports both ID-based and ticket-based session resumption. Each sub-handshake (the primary handshake and the secondary handshakes) is replaced with a standard abbreviated handshake; the only minor difference is that the session tickets for middleboxes should contain the session keys for the primary end-to-end session (in addition to the key for the secondary endpoint-middlebox session). A new attestation is not required, because only the enclave knows the key needed to decrypt the session ticket. The client stores a session ID or ticket for the server and each client-side middlebox. The server can either cache the session IDs/tickets for server-side middleboxes or ask the client to cache them and return them in its `ClientHello`.

Assuming the client sends application data first, it is possible for data to arrive at a server-side middlebox *before* the `ChangeCipherSpec` and `Finished` from the server. This is the same scenario seen in TLS False Start [32]. The middlebox can choose to wait for the server's `Finished`, thereby slightly delaying the handshake beyond 1 RTT, or go ahead and process the data if the cipher suite is whitelisted for use with False Start.

TLS 1.3. TLS 1.3 [44] significantly changes the TLS handshake compared to TLS 1.2 and earlier, shortening it from two round trips to just one. With minor modifications, mbTLS's handshake can be adapted to TLS 1.3. There is one caveat: when client-side middleboxes are present, data sent by the server in the same flight as the server `Finished` could be delayed, in the worst case, up to one round trip. In most cases, however, clients send data first.

Trust. How does an endpoint decide to trust a middlebox? While this is orthogonal to the design of mbTLS, we briefly describe three scenarios we anticipate. First, the user might sign up for a service (e.g., parental filtering from their ISP) and explicitly configure their browser to trust it. Second, client software might be pre-configured to trust middleboxes from a known set and prompt the user when one is discovered (e.g., Chrome could do this for Flywheel). Finally, service providers might rely on mbTLS to discover the closest instance of their own middleboxes; the server expects the middlebox and verifies it with certificate and attestation.

4 SECURITY ANALYSIS

4.1 Core Security Properties

We now revisit each security property from §3.2, arguing why mbTLS provides them. (Table 1 summarizes.)

P1 **Data Secrecy.**

P1A Decrypting session data requires access to one of the symmetric keys shown in Figure 4. The bridge key, K_{C-S}, is established during the end-to-end client-server TLS handshake in which the endpoints verify one another's certificates. Next, this key and the rest of the session keys (e.g., K_{C-C1}, K_{C1-C0}, etc.) are transferred to the middleboxes over individual secondary TLS connections; importantly, because these secondary connections terminate inside SGX enclaves (remote attestation proves to the endpoint that this is the case), the *MIP* cannot access the secondary session's key in memory, so only the *MS* (and not the *MIP*) learns the primary session keys.

P1B The bridge key (K_{C-S}) is the result of the (standard) primary TLS handshake, so if the primary handshake is forward secure, so is K_{C-S}. The other session keys (e.g., K_{C-C0}, K_{C0-C1}, etc.) are generated fresh for each session and sent to the middleboxes over (standard) secondary TLS connections. Therefore, if these secondary handshakes are forward secure, so are the non-bridge session keys.

P1C Since each hop uses its own independent encryption and MAC keys, after the handshake each hop effectively operates like its own TLS connection. In particular, this prevents an adversary from learning whether or not a middlebox modified a record (though it can still see the sizes and timings of each record, including whether a middlebox increased or decreased the size of the data).

P2 **Data Authentication.** Each record carries a *message authentication code (MAC)*, a small tag generated using the session key that identifies a piece of data. Unauthorized changes can be detected if the MAC does not match the data. Since only the endpoints and each *MS* know the session keys (see **P1A**), only these entities can modify or create records.

P3 **Entity Authentication.**

P3A First, the client and server can require one another's certificates in the primary handshake (though typically client authentication happens at the application layer). A certificate binds the server's public key to its identity, and that public key is used in the primary handshake to negotiate the shared bridge key, so after a successful handshake, the client is assured that any data encrypted with that bridge key can only be decrypted by the expected service provider (or middleboxes it chose to trust). Second, endpoints can also require certificates from middleboxes. Since the corresponding private keys are stored in the middleboxes' enclaves, inaccessible by the *MIP* (and remote attestation proves that this is the case), the endpoint is convinced it is talking to software supplied and configured by the expected *MSP*.

P3B Since our threat model assumes that the *SP* and all software running on its server is trusted, and in P3A we verified that the server possesses the *SP*'s private key, the client trusts that the machine is properly configured with the expected application software. The same logic applies to middleboxes if they are operated in-house; for outsourced middleboxes, endpoints can directly verify *MS* code and configuration with remote attestation.

P4 **Path Integrity.** This follows from the fact that mbTLS uses a fresh key for each hop. Suppose an adversary sniffs a record from the $C1 - C0$ link in Figure 4 and tries to insert it on the $S0 - S1$ link (thereby skipping middleboxes $C0$ and $S0$). The record will be encrypted and MAC'd with K_{C1-C0}, but $C1$ expects data secured with K_{S1-S0}, so the MAC check will fail and the record will be discarded. (Note, that an *endpoint* can inject, delete, or modify data anywhere in its portion of the path because it knows all the session keys on its side. We discuss potential security implications below.)

4.2 Other Security Properties

Endpoint Isolation. *Endpoints can only authenticate their own middleboxes, not those added by the other endpoint.* In fact, an endpoint likely does not even know about the other side's middleboxes. This follows from the way keys are generated and distributed. Checking a certificate or an attestation is only meaningful if the public key in the certificate is used for key exchange (then you trust that only the entity associated with that public key can decrypt what you send with the new symmetric key). Since endpoints don't do a KE with the other side's middleboxes, they have no means of authenticating one another, even if they exchanged certificates/attestations. This limitation seems reasonable; since the endpoints presumably trust one another or they would not be communicating in the first place, it is natural to trust the other endpoint to properly authenticate any middleboxes it adds to the session.

Path Flexibility. *It is not possible to interleave client-side and server-side middleboxes.* To support this, the endpoints would need to coordinate to generate/distribute keys to the interleaved portion of the path. This means (1) extra work for endpoints and (2) the endpoints would need to know about (some of) one another's middleboxes. This would also mean an endpoint could modify/inject traffic *after* the other endpoint's middleboxes, which could be a security problem if one of those middleboxes performs filtering or sanitization.

Middlebox State Poisoning. *It is not safe to use mbTLS with **client-side** middleboxes that keep **global state**.* Since endpoints know the keys for each hop on their side of the session, a malicious client can read and/or modify data on any of these hops without its middleboxes knowing. This is a problem when a middlebox shares state across multiple clients, like a Web cache does. A client with access to a link between the cache and the server could (1) request a page, (2) drop the server's response, and (3) inject its own response, thereby poisoning the cache for other clients.

One possible solution is to alter the handshake protocol so that middleboxes establish keys with their neighbors rather than endpoints generating and distributing session keys; this means each party only knows the key(s) for the hop(s) adjacent to it. The downside is the client has lost the ability to directly authenticate the server; instead, the client must trust its middleboxes to authenticate the server.

Security Property	Threat	Defense (TLS)	Defense (mbTLS)
Data Secrecy	P1A Data read on-the-wire by TP or MIP	Encryption	Encryption*
	P1A Data read in MS application memory by MIP	—	Secure Execution Environment
	P1B Old data decrypted by TP after a long-term key leaks	Ephemeral Key Exchange	Ephemeral Key Exchange
	P1C TP compares record entering and leaving MS to see if it was modified	—	Unique Per-Hop Keys
Data Authentication	P2 Records dropped, injected, or modified on-the-wire	MACs	MACs*
	P2 Data deleted, injected, or modified in RAM by MIP	—	Secure Execution Environment
Entity Authentication	P3A C establishes key with wrong software on S	Certificate	Certificate
	P3A C establishes key with software on hardware operated by someone other than S	Certificate	Certificate
	P3A C or S establishes key with MS software operated by someone other than MSP	—	Certificate*
	P3B C or S establishes key with wrong MS software	—	Remote Attestation
Path Integrity	P4 Records passed to middleboxes in the wrong order	—	Unique Per-Hop Keys

Table 1: Threats and Defenses. How mbTLS defends against concrete threats to our core security properties. For comparison, we include TLS where applicable. An asterisk indicates that defense also relies on the secure environment to safeguard the session key.

Bypassing "Filter" Middleboxes. It might appear that the fact endpoints know all the session keys on their side enables another attack: if a middlebox performs some kind of filtering function (e.g., a virus scanner, parental filter, or data exfiltration detector), this means the endpoint has the keys to access incoming data *before* it is filtered or inject outbound data afterward. However, if an endpoint is capable of reading or writing data "on the other side" of the filter (i.e., physically retrieve/inject packets from/into the network beyond the middlebox), then the filter was ineffective to begin with.

5 EVALUATION

We evaluate four critical aspects of mbTLS. First, our security analysis argues that mbTLS is **secure** (§4). Second, with a series of real-world experiments, we show that mbTLS is **immediately deployable** (§5.1). Third, we show mbTLS imposes **reasonable CPU overhead** for servers wishing to deploy it (and reduces it for middleboxes) (§5.2). Finally, we show that **SGX applications can support network I/O heavy workloads** (§5.3).

Prototype Implementation. We implemented mbTLS in OpenSSL (v1.1.1-dev) using the Intel SGX SDK for Windows (v1.7). Our prototype currently supports any cipher suites using `DHE` or `ECDHE` for key exchange and `AES256-GCM` for bulk encryption (these are not fundamental limitations of the protocol—any TLS cipher suite works in mbTLS). We also provide a support library for running our mbTLS implementation inside an SGX enclave. Our support library implements 8 libc functions directly in the enclave (3 of which are only used for debugging and can be removed in production builds) and exits the enclave for another 7 libc functions (4 of which are for debugging). The middlebox in the following experiments is a simple HTTP proxy that performs HTTP header insertion.

Testbed. Our local testbed comprises four servers running Windows Server 2016, with SGX-enabled Intel Core i7 6700 processors and an SGX-enabled motherboard. These are connected through Mellanox ConnectX-3 40Gbps NICs to a local Arista 7050X switch.

Network Type	# Sites	Network Type	# Sites
Enterprise	6	*University*	11
Residential	34	*Public*	1
Mobile	2	*Hosting*	56
Colocation Services	35	*Data Center*	19
Uncategorized	77	**Total**	241

Table 2: Handshake Viability. Number of distinct sites from which we performed mbTLS handshakes to our test server, broken down by network type. All handshakes were successful.

5.1 Deployability

We test two things through real-world deployments: (1) Do firewalls or traffic normalizers in the public Internet block mbTLS connections? (2) Can mbTLS interoperate with legacy endpoints?

Handshake Viability. Since mbTLS introduces new TLS extensions (`MiddleboxSupport`) and record types (`Encapsulated` and `MiddleboxAnnouncement`), we verify that existing filters, like firewalls, traffic normalizers, or IDSes, do not drop our handshakes. To do so, we connect to a middlebox and server running in Azure from clients located in various networks around the world. The middlebox is configured to be a client-side middlebox, so the new record types traverse the networks between the client and the data center. To test a diverse set of client networks, we do two things. First, we run our client through Tor, using 550 exit nodes located in 46 countries across 214 ASes. Using whois data, we categorized the networks by type. We chose not to use platforms such as PlanetLab (whose nodes are mainly located in university networks) or public clouds like Azure or EC2, as these environments are more homogeneous and typically are not heavily filtered. Second, to fill in gaps in the Tor experiment, we manually connect from different types of networks (namely public, mobile, and data center networks). Table 2 shows a breakdown of the distinct client networks from which we initiated a handshake. All handshakes were successful.

Legacy Interoperability. To demonstrate that mbTLS can communicate with legacy endpoints, we use a version of curl [4] modified

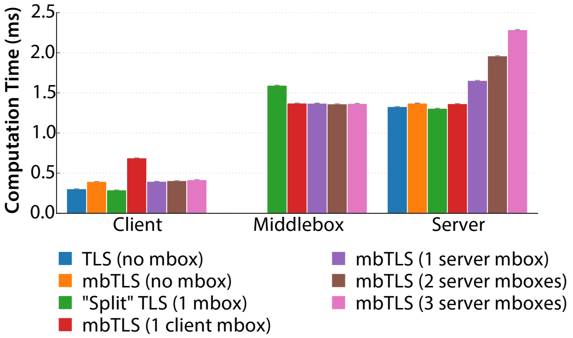

Figure 5: Handshake CPU Microbenchmarks. Each bar shows the time spent executing a single handshake (not including waiting for network I/O). Each bar is the mean of 1000 trials; error bars show a 95% confidence interval of the mean.

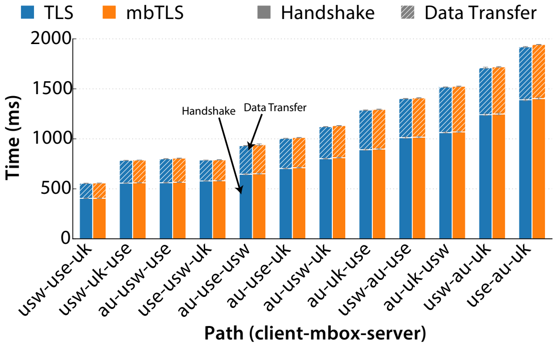

Figure 6: mbTLS vs. TLS Latency. Time to fetch a small object via one middlebox across various paths between data centers.

to use mbTLS to download the root HTML document for the top 500 Alexa sites that support HTTPS via our SOCKS HTTP proxy running in Azure. Only 385 sites in the Alexa top 500 support HTTPS; we successfully connected to 308 of these. Of the 77 that failed, 19 had invalid or expired certificates. Another 40 did not support `AES256-GCM`, the only encryption algorithm currently supported by our prototype (note that this is a limitation of our prototype, not the protocol). Another 13 failed due to redirects our SOCKS implementation did not properly handle. The remaining 5 failed for unknown reasons.

5.2 Performance Overhead

mbTLS does not modify the TLS record layer, so it has no impact on data transfer performance. On the other hand, it *does* change the handshake, so we (1) investigate the computational expense of performing a handshake and (2) empirically verify that mbTLS does not increase session setup latency.

Handshake CPU Microbenchmarks. To understand mbTLS's impact on CPU load, we measured the time it costs clients, middleboxes, and servers to perform TLS and mbTLS handshakes. CPU overhead is of particular concern for middleboxes and servers, who need to serve many connections simultaneously. As Figure 5 shows, without a middlebox, the TLS and mbTLS times are close (we suspect the slight difference is inefficiency in our implementation, not the protocol). Second, for the middlebox, an mbTLS handshake is *cheaper* than Split TLS because the middlebox only performs one TLS handshake, not two. Finally, the server's load is not impacted by client-side middleboxes and increases linearly with the number of server-side middleboxes. Note, however, that each server-side middlebox only adds approximately 20% of the original, no-middlebox handshake time; this is because, for each middlebox, the server performs one additional *client* TLS handshake, which is cheaper than a server handshake. (Key exchange was `ECDHE-RSA`; results were similar for `DHE-RSA`. Machine specs: Intel i7-6700K CPU at 4 GHz; 16 GB RAM; Windows 10.)

Handshake Latency. To confirm that our handshake protocol does not inflate session setup in practice (it should not, since it maintains the same four-flight "shape" as TLS), we perform several handshakes across data centers in Microsoft Azure. We deploy VMs in four regions (Australia, US West, US East, and UK) and test all permutations of a client-middlebox-server path across them (with no two VMs in the same DC). In each test, we compare the time to fetch a small object using mbTLS and TLS. For regular TLS, the middlebox simply relays packets, i.e., it does not perform split TLS—this is the worst-possible case to compare mbTLS against since the middlebox performs no handshake operations. Figure 6 summarizes the results, broken into handshake time and data transfer time. Each bar is the mean of 100 trials; error bars show a 95% confidence interval. We observe that mbTLS increases the handshake latency on average by 0.7% (1.2% in the worst case—a 10 ms increase out of 800 ms). This is likely due simply to handshake computations on the middlebox.

5.3 Network I/O in SGX

Finally, we investigate network I/O performance from the enclave. SGX imposes restrictions on what enclave code can do. Since only the CPU is trusted, interaction with the outside world is not permitted by default (notably, system calls are not permitted, since the OS is untrusted). When an enclave thread needs to make a system call, there are two high-level strategies: (1) it copies the arguments into unprotected memory, exits the enclave, executes the call, re-enters the enclave, and copies the result back into enclave memory (this boundary-crossing incurs a performance penalty); or (2) it places a request in a shared queue and another thread outside the enclave executes the call passes the result back into the enclave via a response queue. To borrow terminology from SCONE [16], these are *synchronous* and *asynchronous* system calls, respectively.

In a microbenchmark of repeated `pwrite()`s, SCONE found that, for small buffer sizes, asynchronous calls can be up to an order of magnitude faster. However, here we are concerned specifically with `send()`s and `recv()`s, so we performed a small experiment to test the enclave's impact on throughput. We configured four machines in our lab to be a middlebox, a server and two clients. The clients send a stream of random bytes, sent in encrypted chunks whose size we vary. The middlebox is configured with one of four behaviors: it either simply forwards the (encrypted) data to the server or it decrypts and re-encrypts it before forwarding, and it does this either

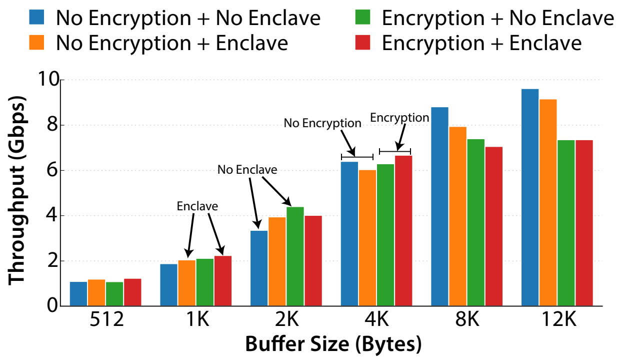

Figure 7: SGX (Non-)Overhead. Middlebox throughput with/without encryption and with/without SGX. Confidence intervals are within 1-5% of the average and differences between different scenarios for small buffers are not statistically significant.

outside or inside the enclave. We add connections from multiple client threads until the middlebox CPU is saturated.

Figure 7 shows that the enclave did *not* have a noticeable impact on throughput, suggesting that optimizations like asynchronous system calls are not necessary for applications with I/O heavy workloads. We suspect this is due to high I/O interrupt rates; overhead from interrupt handling overwhelms the overhead from crossing the enclave boundary. Even if a developer uses asynchronous system calls, under the impression that a thread will permanently live in the enclave, that thread will still leave the enclave whenever that core is interrupted. While pinning interrupts to a different core to avoid interrupting the enclave might help, you then pay the cost of transferring the received data from the interrupt-handling core to the enclave core. Figure 7 also shows that the throughput when the middlebox decrypts/encrypts data plateaus around 7 Gbps. This is yet another source of CPU overhead that helps outweigh any performance penalty from enclave transitions.

6 RELATED WORK

Middlebox as an Endpoint. In §2.2 we describe the current practice of intercepting TLS connections using custom root CA certificates. Other proposals include a 2014 IETF draft from Ericsson and AT&T [34] which would allow a proxy to intercept a TLS handshake and return a certificate identifying itself as such; if the client chooses, it can complete the handshake with the proxy and rely on the proxy to open its own TLS connection to the server. A related Cisco proposal [36] builds on this by introducing a `ProxyInfoExtension` which the proxy would use to pass the client information about the certificate and cipher suite used on the proxy-server connection. Finally, at least one ISP ships custom browsers with the certificates for its proxies built in [33]. Unfortunately, these approaches do not allow the client to authenticate the server or verify what cipher suite is used between the middlebox and the server.

Middlebox as a Middlebox. In contrast to the above approaches, which glue together separate TLS connections, several techniques maintain some form of end-to-end session. An IETF draft from Google has clients connect to servers directly and pass the session key to a proxy out-of-band over a separate TLS connection [43]. CloudFlare's Keyless SSL does much the same thing for server-side delegates [20, 52]. Both of these techniques expose data to the middlebox infrastructure and fail to provide path integrity (the same session key is used on each hop).

We discuss mcTLS [41] and BlindBox [48] in §2.2.

Finally, the security concerns for network-layer middleboxes, which operate on L3 and L4 header fields (e.g., NATs, firewalls, and load balancers), are orthogonal to the concerns mbTLS addresses. Systems like Embark [31], SplitBox [17], and others [29, 38, 49] allow network administrators to outsource network-layer middleboxes to a cloud provider or ISP without revealing private information about their networks.

Network Architectures. DOA [53] provides network support for routing a packet through one or more intermediaries, but does not by itself provide all of our security properties. ICING [39] is a mechanism for enforcing path integrity and is more general than ours. Our path integrity mechanism optimizes by taking advantage of the fact that mbTLS parties already share keys and each mbTLS record is already MAC-protected.

SGX. Protecting outsourced middleboxes with SGX was briefly discussed in [30], but without a concrete protocol or implementation. S-NFV [50] sketches a framework for implementing SGX-protected middlebox applications. PRI [45] details the design of an SGX-based IDS. Both focus on the middlebox application architecture rather than the protocol for including a middlebox in a communication session to begin with. There are also a number of TLS implementations designed to work in SGX enclaves [6, 10, 18, 54]. These libraries port unmodified TLS; we go a step further and extend the TLS handshake to include remote attestation, allowing one party to verify that the TLS session terminates inside an enclave. Finally, there is a rapidly growing body of work on how to build SGX-protected systems (or port existing ones) [13, 16, 19, 28, 51]. These are all orthogonal to this work and could be used in concert with mbTLS to build an SGX-protected middlebox.

7 CONCLUSION

In this paper we presented Middlebox TLS, or mbTLS, a protocol for secure multi-entity communication. Unlike previous solutions for integrating middleboxes into secure sessions, mbTLS (1) interoperates with legacy TLS endpoints and (2) can protect session data from untrusted middlebox infrastructure using trusted computing technology like Intel SGX. Our prototype implementation shows that mbTLS can indeed communicate with real, unmodified web servers and incurs reasonable overhead. Finally, we discuss the space of security properties for multi-entity communication and the trade-offs protocol designers must make among them.

Acknowledgements Many thanks to Cédric Fournet, Antoine Delignat-Lavaud, Felix Schuster, Manuel Costa, and Istvan Haller for their time, feedback, and expertise. We also thank the reviewers and our shepherd for their helpful comments and suggestions. This work was funded in part by NSF under award number CNS-1345305.

A MBTLS PROTOCOL DETAILS

This appendix describes the message formats and protocol constants used in our implementation of mbTLS. It follows the formatting conventions set forth in RFC 5246 (TLS 1.2) [22].

A.1 Record Protocol

mbTLS adds three new record types (bold):

```
enum {
  change_cipher_spec(20),
  alert(21),
  handshake(22),
  application_data(23),
  mbtls_encapsulated(30),
  mbtls_key_material(31),
  mbtls_middlebox_announcement(32),
  (255)
} ContentType;
```

Encapsulated. The MBTLSEncapsulated record is used to carry secondary handshake messages.

```
struct {
    uint8 subchannelID;
    opaque record[TLSPlaintext.length-1];
} EncapsulatedRecord;
```

Here, record is another complete TLS record. Because the inner record is the outer record's payload, which is limited to 2^{14} bytes, and because the subchannel ID uses 1 byte, the inner record's payload is limited to 2^{14-1} bytes. MBTLSEncapsulated records may only be sent during handshake or renegotiation.

Key Material. The MBTLSKeyMaterial record is used by endpoints to send symmetric key material to their middleboxes.

```
struct {
    uint32 key_len;
    uint32 iv_len;
    opaque clientWriteKey[key_len];
    opaque clientWriteIV[iv_len];
    opaque clientReadKey[key_len];
    opaque clientReadIV[iv_len];
    opaque serverWriteKey[key_len];
    opaque serverWriteIV[iv_len];
    opaque serverReadKey[key_len];
    opaque serverReadIV[iv_len];
} MBTLSGCMKeyMaterial;

struct {
    Version client_server_version;
    opaque client_to_server_sequence[8];
    opaque server_to_client_sequence[8];
    CipherSuite cipher_suite;  /* 2 bytes */
    select(cipher_suite) {
        case TLS_RSA_WITH_AES_256_GCM_SHA384:
          MBTLSGCMKeyMaterial;
    }
} MBTLSKeyMaterial;
```

The MBTLSKeyMaterial message is always sent encapsulated in a subchannel (i.e., in an MBTLSEncapsulated record). It contains the TLS version negotiated between the client and the server, the sequence number for client-to-server data (write sequence from client's perspective and read sequence from server's) and the sequence number for server-to-client data. It also contains key and IV material in a format dependent on the cipher suite.

Middlebox Announcement. The MBTLSMiddleboxAnnouncement message is used by middleboxes to alert the server to their presence.

```
struct {
} MBTLSMiddleboxAnnouncement;
```

The MBTLSMiddleboxAnnouncement message is always sent in an MBTLSEncapsulated record. The message is empty, and only serves to alert the server of the middlebox's presence. Middleboxes never send this message to a client.

A.2 Handshake Protocol

mbTLS adds one handshake protocol message (bold):

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    sgx_attestation(17),
    finished(20), (255)
} HandshakeType;
```

SGX Attestation. The SGXAttestation handshake message can optionally be used during the handshake for the server to send the client an SGX attestation (quote). This feature is independent of the rest of mbTLS. sgx_quote follows Intel's sgx_quote_t format.

```
struct {
    opaque sgx_quote<0..2^14-1>;
} SGXAttestation;
```

Middlebox Support Extension. mbTLS also adds one TLS extension, the MiddleboxSupportExtension:

```
struct {
    uint8 numHellos;
    uint16 helloLengths[numHellos];
    opaque clientHellos[numHellos];
    uint8 numMboxes;
    opaque middleboxes[numMboxes];
} MiddleboxSupportExtension;
```

The MiddleboxSupportExtension is sent by a TLS client in the ClientHello message. It indicates that the client supports mbTLS, inviting on-path middleboxes to announce themselves to the client. The extension carries one or more "optimistic" ClientHellos, to which the middleboxes may respond with ServerHellos, as well as a list of middleboxes known to the client a priori.

REFERENCES

[1] ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone. Accessed: January 2017.

[2] Aryaka. http://www.aryaka.com. Accessed: January 2017.

[3] Brocade Network Functions Virtualization. http://www.brocade.com/en/products-services/software-networking/network-functions-virtualization.html. Accessed: January 2017.

[4] curl. https://curl.haxx.se.

[5] Google Edge Network. https://peering.google.com. Accessed: January 2017.

[6] Intel Official SGX OpenSSL Library. https://software.intel.com/en-us/sgx-sdk/download. Accessed: June 2017.

[7] Juniper Architecture for Technology Transformation. https://www.juniper.net/assets/us/en/local/pdf/whitepapers/2000633-en.pdf. Accessed: January 2017.

[8] Network Functions Virtualization (Dell). http://www.dell.com/en-us/work/learn/tme-telecommunications-solutions-telecom-nfv. Accessed: January 2017.

[9] Telefónica NFV Reference Lab. http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab. Accessed: January 2017.

[10] TLS for SGX: a port of mbedtls. https://github.com/bl4ck5un/mbedtls-SGX. Accessed: June 2017.

[11] Zscaler. https://www.zscaler.com. Accessed: January 2017.

[12] AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf, 2013. Accessed: January 2017.

[13] Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017. USENIX Association.

[14] 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.

[15] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP '13*, volume 13, 2013.

[16] S. Arnautov, B. Trach, F. Gregor, et al. SCONE: Secure Linux Containers with Intel SGX. In *OSDI '16*, pages 689–703, GA, 2016. USENIX Association.

[17] H. J. Asghar, L. Melis, C. Soldani, et al. Splitbox: Toward efficient private network function virtualization. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 7–13, New York, NY, USA, 2016. ACM.

[18] P.-L. Aublin, F. Kelbert, D. O'Keeffe, et al. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves.

[19] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI '14*. USENIX – Advanced Computing Systems Association, Oct. 2014.

[20] K. Bhargavan, I. Boureanu, P.-A. Fouque, C. Onete, and B. Richard. Content delivery over tls: A cryptographic analysis of keyless ssl. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[21] A. de Zylva. Windows 10 Device Guard and Credential Guard Demystified. https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/. Accessed: January 2017.

[22] 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.

[23] Z. Durumeric, Z. Ma, D. Springall, et al. The Security Impact of HTTPS Interception. In *NDSS '17*, 2017.

[24] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *NSDI 14*, pages 543–546, Seattle, WA, 2014. USENIX Association.

[25] A. Gember-Jacobson, R. Viswanathan, C. Prakash, et al. Opennf: Enabling innovation in network function control. SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[26] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP '13*, page 11, 2013.

[27] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged ssl certificates in the wild. In *IEEE Symposium on Security and Privacy*, SP '14, pages 83–97, Washington, DC, USA, 2014. IEEE Computer Society.

[28] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549, GA, 2016. USENIX Association.

[29] A. R. Khakpour and A. X. Liu. First step toward cloud-based firewalling. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 41–50, Oct 2012.

[30] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A first step towards leveraging commodity trusted execution environments for network applications. In *HotNets-XIV*, pages 7:1–7:7, New York, NY, USA, 2015. ACM.

[31] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely outsourcing middleboxes to the cloud. In *NSDI '16*, pages 255–273, Santa Clara, CA, Mar. 2016. USENIX Association.

[32] A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. RFC 7918 (Informational), Aug. 2016.

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

[34] 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.

[35] J. Martins, M. Ahmed, C. Raiciu, et al. ClickOS and the Art of Network Function Virtualization. In *NSDI '14*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[36] 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.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, et al. Innovative instructions and software model for isolated execution. In *HASP '13*, page 10, 2013.

[38] L. Melis, H. J. Asghar, E. De Cristofaro, and M. A. Kaafar. Private processing of outsourced network functions: Feasibility and constructions. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks &#38; Network Function Virtualization*, SDN-NFV Security '16, pages 39–44, New York, NY, USA, 2016. ACM.

[39] J. Naous, M. Walfish, A. Nicolosi, et al. Verifying and enforcing network paths with icing. CoNEXT '11, pages 30:1–30:12, New York, NY, USA, 2011. ACM.

[40] 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.

[41] D. Naylor, K. Schomp, M. Varvello, et al. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 199–212, New York, NY, USA, 2015. ACM.

[42] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. Tls proxies: Friend or foe? In *IMC '16*, pages 551–557, New York, NY, USA, 2016. ACM.

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

[44] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-18, IETF Secretariat, Oct. 2016.

[45] L. Schiff and S. Schmid. PRI: Privacy Preserving Inspection of Encrypted Network Traffic. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 296–303. IEEE, 2016.

[46] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[47] 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.

[48] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *SIGCOMM '15*, pages 213–226, New York, NY, USA, 2015. ACM.

[49] J. Shi, Y. Zhang, and S. Zhong. Privacy-preserving network functionality outsourcing. *CoRR*, abs/1502.00389, 2015.

[50] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks &#38; Network Function Virtualization*, SDN-NFV Security '16, pages 45–48, New York, NY, USA, 2016. ACM.

[51] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.

[52] N. Sullivan. Keyless SSL: The Nitty Gritty Technical Details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/. Accessed: September 2016.

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

[54] wolfSSL. wolfSSL with Intel SGX. https://software.intel.com/en-us/sgx-sdk/download. Accessed: June 2017.

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