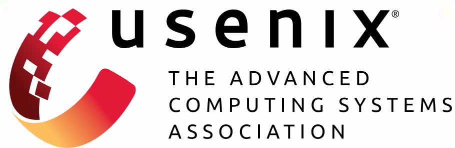



# mmTLS: Scaling the Performance of Encrypted Network Traffic Inspection

Junghan Yoon, *Seoul National University;* Seunghyun Do and Duckwoo Kim, *KAIST;* Taejoong Chung, *Virginia Tech;* KyoungSoo Park, *Seoul National University*

https://www.usenix.org/conference/atc24/presentation/yoon

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

# mmTLS: Scaling the Performance of Encrypted Network Traffic Inspection

Junghan Yoon[†] Seunghyun Do[#] Duckwoo Kim[#] Taejoong Chung[‡] KyougSoo Park[†]

*Seoul National University*[†] *KAIST*[#] *Virginia Tech*[‡]

## Abstract

Modern network monitoring TLS middleboxes play a critical role in fighting against the abuse by encrypted network traffic. Unfortunately, operating a TLS middlebox often incurs a huge computational overhead as it must translate and relay encrypted traffic from one endpoint to the other. We observe that even a simple TLS proxy drops the throughput of end-to-end TLS sessions by 43% to 73%. What is worse is that recent security enhancement TLS middlebox works levy an even more computational tax.

In this paper, we present mmTLS, a scalable TLS middlebox development framework that significantly improves the traffic inspection performance and provides a TLS event programming library with which one can write a TLS middlebox with ease. mmTLS eliminates the traffic relaying cost as it operates on a single end-to-end TLS session by secure session key sharing. This approach is not only beneficial to performance but it naturally guarantees all end-to-end TLS properties except confidentiality. To detect illegal content modification, mmTLS supplements a TLS record with a private tag whose key is kept secret only to TLS endpoints. We find that the extra overhead for private tag generation and verification is minimal when augmented with the first tag generation. Our evaluation demonstrates that mmTLS outperforms the nginx TLS proxy in the split-connection mode by a factor 2.7 to 41.2, and achieves 179 Gbps of traffic relaying throughput.

## 1 Introduction

Transport Layer Security (TLS) [18, 19] is increasingly popular in the modern Internet. According to a recent measurement study [49], 76% of the top one million Web sites support TLS and 63% of them support the latest version, TLS 1.3. TLS is often enabled by default not only for private Web browsing but even for large content transfer such as video streaming [4, 14, 15, 26, 28]. Encrypted network transfer in the Internet has become ubiquitous.

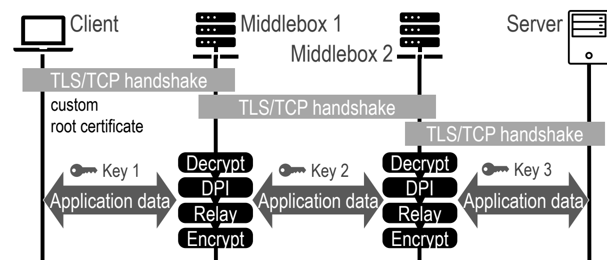



**Figure 1:** TLS middlebox operation with split-connection

While TLS ensures end-to-end private communication, it complicates security monitoring of network traffic at enterprises[1]. To gain the visibility into the encrypted traffic, an enterprise TLS middlebox intentionally operates as a man in the middle (MITM) – it impersonates as *any* site that a client intends to visit and inspects the stream while it relays the encrypted traffic between the two endpoints. This impersonation is commonly achieved by installing a custom root certificate on the client's system. Consequently, the client blindly trusts any site certificate signed by the custom root certificate managed by the TLS middlebox [35, 42, 46, 47].

Unsurprisingly, the status quo severely undermines the security of a TLS session as it does not guarantee end-to-end authentication nor content integrity. To address the problem, existing works [51, 57, 58] propose that the endpoints authenticate the middleboxes explicitly and that middleboxes should participate in the key exchange or session-key sharing process. They divide the middleboxes into "read-only" and "read-write" ones, and allow the endpoints (and/or middleboxes) to detect illegal modification by non-privileged attacker middleboxes. Unfortunately, these approaches present two problems. First, if a legitimate read-write middlebox gets compromised, an endpoint receiver has no way to tell if the modification is illegal. Even running them in trusted execution environments (TEE) [41, 57, 63, 68] is not entirely bullet-

[1] In this paper, we refer to any small ISP network operated by a school, a government, a hospital, a company, etc., as an "enterprise" network.

proof as the original software may have its own vulnerabilities or operators or developers can make a mistake [39,60,72]. Thus, we argue that employing read-write TLS middleboxes on the client side incurs a significant security issue. Second, they employ different session keys per each network path segment, so they should operate by split-connection as shown in Figure 1. However, the split-connection architecture is very costly as the middlebox not only relays the traffic between two TCP connections but it must translates the encrypted content from one endpoint to the other. We observe 54% to 71% throughput degradation when a TLS middlebox splits the connection between two endpoints.

We tackle this problem with mmTLS, a highly-scalable TLS traffic monitoring architecture without connection termination. Our key observation is that modern security inspection middleboxes on the client side (or enterprise side) are predominantly read-only [30,58,66]. Intrusion detection/prevention systems (IDSes/IPSes) [21, 23, 29], virus scanning [5, 6], parental filtering [8, 10], and data loss prevention (DLP) systems [11, 20, 25] typically run deep packet inspection (DPI) without content modification. Traditional read-write middleboxes for client-side, transparent Web caching and network traffic deduplication for general Web traffic[2] have largely disappeared thanks to the significant improvement in network bandwidth and its cost. Tracker/ad-blockers had been supported by middleboxes, but the functionality has moved to 3rd-party applications on clients since middleboxes could not catch up with fine-grained blocking [55]. Thus, in a typical enterprise environment where only read-only middleboxes are deployed, we do not need to support trustworthy content modification, and more importantly, we can significantly improve the performance of the monitoring middlebox if clients share the session keys with it. Then, the TLS middlebox simply forwards the packets between the endpoints without re-encryption. For security monitoring, it reassembles the encrypted packets in a flow, decrypts them, and monitors the plaintext traffic. The only extra tax beyond packet forwarding would be managing an out-of-band TLS session for session key delivery from a client as well as the key delivery overhead. However, one can minimize this overhead by leveraging SmartNIC as a scalable session key deliverer.

Session key sharing is trivially simple, and it naturally ensures end-to-end authentication even with in-network middleboxes. However, end-to-end content integrity can be compromised as all TLS middleboxes share the session keys – if compromised, they can modify the content without notice by the endpoints. As TLS 1.3 mandates authenticated encryption with additional data (AEAD) [19], one can encrypt the data and generate a tag with the same key. To address this problem, we propose using a second tag (called a *private tag*) whose key is kept secret only to endpoints. This requires every TLS record to include two tags – one for original message authentication and the other for detecting illegal content modification in the middle. While private tag generation would incur extra overhead at the sender, augmenting it with original tag generation substantially reduces the overhead to as small as 2% to 5% of the original cost. We show this scheme is directly applicable to AES-GCM, the most popular symmetric cipher for TLS 1.3 [49], as well as other popular ciphers in TLS 1.3.

Our evaluation demonstrates that mmTLS improves the TLS traffic monitoring performance by 2.4x to 4.6x over the split-connection architecture when they use persistent TLS connections. When all TLS connections are ephemeral, the performance benefit is even larger despite the higher key delivery cost– mmTLS achieves 63.5x throughput improvement for 1KB object download with ephemeral connections. This is because an extra TLS handshake in the split connection dominates the overhead, which makes the session key delivery overhead negligible.

The contribution of our work is summarized as follows. First, we analyze the performance overhead of existing TLS middleboxes. The primary reason for poor performance lies in the split connection and re-encryption, which degrades the performance by up to 71% over end-to-end TLS connections. Second, we propose mmTLS, a scalable TLS traffic monitoring framework on a single TLS/TCP connection with secure session key sharing. mmTLS also provides a TLS event programming library with which one can write a TLS middlebox with ease. The TLS event library is implemented by extending mOS [45], a high-performance TCP event monitoring stack on DPDK. Third, we present a set of optimization techniques that significantly improve the performance. This includes delivering the session keys via SmartNIC for multi-core scalability as well as generating the private tag with a minimal overhead from the first tag generation.

## 2 Background and Motivation

We present a brief background on TLS and explain the security and performance implications of TLS middleboxes.

### 2.1 TLS and TLS middleboxes

TLS ensures the following three security properties for end-to-end network communication. First, the network traffic between two endpoints should be private only to them (**confidentiality**). Second, an endpoint receiver should be able to detect if the content is delivered intact or modified in the middle (**content integrity** or **data authentication**). Third, an endpoint should be able to verify the identity of the other end of communication (**endpoint authentication**). While the growing demand for private network transfer has mainly driven the adoption of TLS, commodity hardware support for

[2] We see that the data deduplication market for storage backup is strong, but that does not require deploying transparent TLS middleboxes.

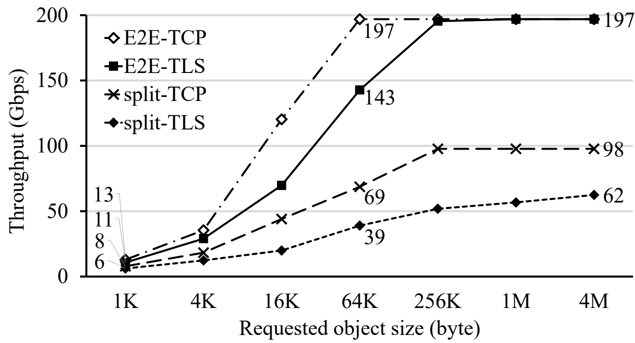
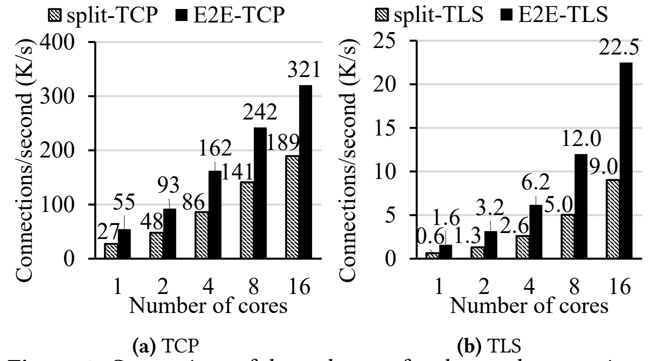



**Figure 2:** Comparison of throughputs of end-to-end (E2E) TLS connections vs. nginx TLS proxy with persistent connections




**(a)** TCP **(b)** TLS

**Figure 3:** Comparison of throughputs of end-to-end connections vs. nginx proxy with ephemeral connections. Clients download a 1 KB object at each connection. For E2E, the X-axis represents the number of cores in the server.

| Function (P) | CPU | Function (E) | CPU |
|---|---|---|---|
| SSL_write() | 32.2% | SSL_do_handshake() | 91.6% |
| SSL_read() | 31.6% | SSL_write() | 1.0% |
| aesni_gcm_encrypt() | 12.5% | SSL_read() | 0.6% |
| aesni_gcm_decrypt() | 10.4% | | |
| Others | 13.2% | Others | 6.8% |

**Table 1:** CPU usage breakdown by the functions of the nginx TLS proxy with persistent (P) and ephemeral (E) connections

crypto operations like AES-NI [43] or Intel QAT [44] makes the deployment a lot more practical.

However, TLS fundamentally conflicts with existing middlebox operations at enterprises. Enterprise networks often employ TLS middleboxes (e.g., IDSes/IPSes [21, 23, 29], virus scanning [5, 6], data loss prevention systems [11, 20, 25]) to inspect the plaintext content of network traffic to stop malicious content or to prevent leakage of corporate secrets. In older days, they ran network traffic deduplication or transparent Web caching middleboxes to save the costly network bandwidth. These middleboxes must break the confidentiality between the TLS endpoints while read-write middleboxes do not ensure end-to-end content integrity.

Unfortunately, the way that enterprise middleboxes operate stays at a stopgap – a TLS middlebox impersonates the other endpoint so that it can retrieve the session keys to decrypt the traffic. To enable this, the current practice installs custom root CA certificates at clients, which wreaks havoc on the assumptions and security guarantees by TLS. Due to impersonation, one endpoint cannot authenticate the other endpoint. Also, an endpoint cannot check if the TLS middlebox modifies the content in the middle. In fact, today's MITM-based TLS middleboxes fail to provide any of the end-to-end security guarantees of TLS [35, 37, 53, 61, 69].

## 2.2 Existing Works & Performance Penalty

There have been largely two directions that tackle the above problem. The first approach is to exploit searchable encryption. For instance, BlindBox [67] or Embark [50] enables middleboxes to search on the encrypted content. However, this approach depends on the order-preserving property of specific crypto algorithms, which limits the type of cipher suites. The other approach is to extend TLS to explicitly support middleboxes such as mcTLS [58], mbTLS [57], and maTLS [51] (called "TLS-extension"). These works suggest that the middleboxes share the session keys while they use a separate message authentication code (MAC) key for modification. They support authenticating endpoints as well as middleboxes if necessary, and they ensure that any illegal content modification by a non-privileged middlebox is detected at endpoints. While end-to-end confidentiality has to be breached, this is necessary for correct operation.

TLS-extension works look more promising, but they leave a few serious obstacles in deployment. First, they require invasive modification on the TLS handshake protocol. Modification on the TLS protocol is inevitable as TLS disallows any 3rd-party middlebox to inspect its traffic. However, existing proposals overly complicate key negotiation to enable each read-write middlebox to modify the content with a valid tag by its own key. Furthermore, any servers must negotiate the keys for unknown middleboxes at a random enterprise, which is impractical. Second, existing TLS-extension works do not alleviate the performance overhead from the MITM approach. Both approaches require split-connection as a middlebox employs different session keys for each network segment [51, 57]. Also, key derivation on a middlebox would be an extra cost for small-content transactions. Given that Moore's law for CPU ended [38] and that TLS is becoming ubiquitous, the increasing computational cost for TLS middleboxes remains a serious problem.

Figure 2 and 3 compare the throughputs of end-to-end (E2E) TLS connections (or plaintext TCP connections) with those of nginx TLS proxy [16] (or plaintext nginx TCP proxy) with persistent and ephemeral connections, respectively.[3] The purpose of the comparison is to understand how much performance is being wasted due to TLS/TCP-level session

[3]For more detailed setup, please refer to Section 5.

proxying. For persistent connections, a simple TLS proxy lowers the performance by 43% to 73% from E2E TLS connections. Table 1 shows that 86.8% of the CPU cycles are spent on TCP I/O and crypto operations for persistent connections. Thanks to the support for AES-NI, the CPU usage for crypto operations is relatively smaller than that for traffic relaying. For content size exceeding 64KB, we observe that removing crypto operations can improve the performance by about 1.5x. However, if we get rid of the split connection, the performance boost reaches 3.4x. Unfortunately, one cannot simply use a zero-copy API like `splice()` or `ktls` [56] as they use different TLS session keys. When the connections are all ephemeral, TLS proxying results in 58% to 63% performance loss. In this case, TLS handshake for key exchange (`SSL_do_handshake()`) is the main bottleneck as it consumes over 90% of the CPU cycles as in table 1. The TLS proxy needs to do the handshake twice in a serial manner, which ends up degrading the performance by over a half.

## 2.3 Read-write Enterprise TLS Middleboxes

The root cause for the complexity in TLS handshake extension works lies in the support for content modification by legitimate middleboxes. However, we argue that deploying read-write TLS middleboxes at an enterprise should be avoided as it is highly dangerous. A compromised read-write TLS middlebox can legitimately inject arbitrarily malicious content into any client at the enterprise. While running them within the TEE mitigates some risks, it is not entirely safe, as software bugs or operator/developer mistakes can still occur [39, 60, 72]. Fortunately, we observe that client-side read-write middleboxes are becoming rare, largely due to the reduced cost of network bandwidth. We do not worry about server-side read-write TLS middleboxes such as L7 load balancers and CDN reverse proxies as they operate as TLS endpoints with real certificates and they should be treated equally as origin servers for compromise protection.

Beyond security issues, read-write middleboxes inevitably incur a serious performance overhead as they adopt the split connection architecture which runs the TCP protocol between each network segment, reassembles the flow, decrypts, and re-encrypts the content at each hop. Our main argument is that these operations are unnecessary as client-side TLS traffic inspection suffices to be read-only, which does not require terminating a TCP/TLS connection.

# 3 mmTLS System Design

In this section, we present the design of mmTLS, a scalable TLS traffic monitoring architecture that ensures all end-to-end TLS properties (i.e., end-to-end authentication/content integrity) except confidentiality.

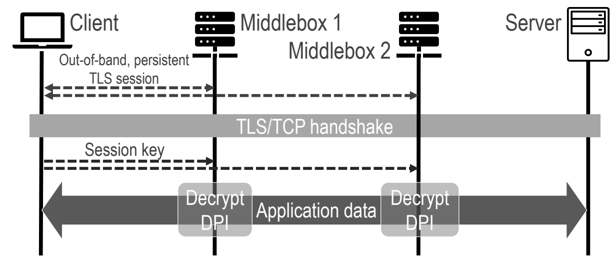



**Figure 4:** Operations of an mmTLS middlebox

## 3.1 Operating Assumptions

Our target environment consists of a chain of read-only middleboxes that transparently inspect the TLS traffic and may terminate or block a connection if the traffic turns out to be malicious. Our primary focus is on read-only middleboxes employed by enterprise networks. Nonetheless, non-enterprise or individual users can sign up for similar abuse-prevention services. These middleboxes can be provisioned on-premises or provided as online services like in security as a service (SECaaS) in cloud [9, 63, 66]. We assume that all traffic to clients cannot circumvent the middlebox with proper network configuration. We do not provide any special middlebox discovery method here as one can leverage SDN [34, 54, 64] or existing algorithms [57]. Even the server side can adopt mmTLS for read-only middleboxes such as packet-level or Web firewalls [27]. As mentioned, server-side read-write middleboxes are treated as origin servers as they run as TLS endpoints with real certificates.

We assume that mmTLS middleboxes are configured not to send any data to random destinations or that they are constantly monitored if they leak any information to an external entity. Clients and servers are assumed to be trustworthy — mmTLS does not detect if they are compromised and securing them is beyond the scope of this work. A compromised client may send wrong session keys or incorrect flow information to the middlebox, but that can be easily detected.

## 3.2 Overall Architecture

The high level idea of mmTLS is very simple. Read-only middleboxes in the network path do not need to split an end-to-end TLS connection – an endpoint that employs the middleboxes simply need to securely share its session keys with them. Despite sharing the session keys, mmTLS still ensures end-to-end content integrity even if some middleboxes between the two endpoints get compromised. Also, we design mmTLS to be incrementally deployable.

**Base design.** Figure 4 shows the overall operation of an mmTLS middlebox in the client side. A client first creates a persistent TLS session with each mmTLS middlebox. This extra TLS session authenticates the middlebox while it

| Solutions | TLS Handshake Modification | TLS Record Update | Additional Key Negotiation | Re-encryption | Detection of Malicious Content Modification by mbox | Framework Support for TLS Protocol Events |
|---|---|---|---|---|---|---|
| SplitTLS | no | no | for each middlebox | for each middlebox | no | no |
| mcTLS [58]<br>mbTLS [57]<br>maTLS [51] | yes (extensive) | yes (extensive) | for each middlebox | for each middlebox | not always (e.g., modification by compromised write mbox) | no |
| mmTLS | no | yes (limited) | no | no | always | yes (event-driven APIs) |

**Table 2:** Comparison of protocol modification and operating behavior with existing works. Note that all secure approaches require update on the TLS record format to monitor encrypted traffic.

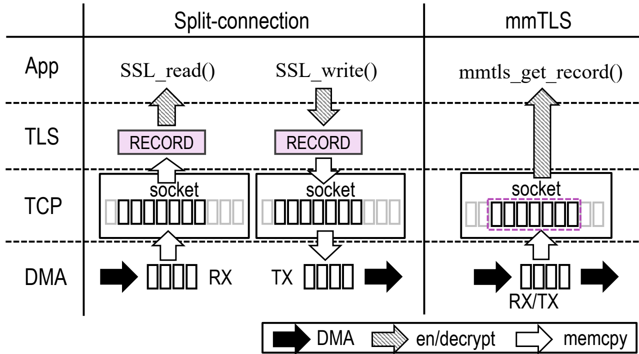



**Figure 5:** Performance benefit over split-connection middleboxes

serves as a secure channel for session key sharing. Previous works [51, 58] suggest that middleboxes should be authenticated by both endpoints, but we believe it is more logical for only the employer of the middlebox to authenticate them as the other endpoint might not fully understand what the middlebox does. Then, the client initiates a TLS handshake with a server, and both endpoints agree on the session keys by standard TLS handshake. After the key exchange, the client sends either real session keys or "null" keys to the middlebox over the secure channel. The "null" keys indicate that the middlebox must bypass the traffic as the client thinks that its plaintext content should not be exposed to even middleboxes (e.g., highly-sensitive data like bank account or private health information). This represents the *"inspect-my-own-data"* semantics – it expresses traffic receiver's (or sender's) permission (or refusal) to inspect "my" own incoming (or outgoing) data [4].

After session key sharing, the TLS traffic flows between the endpoints without any modification. The middlebox decrypts the traffic and performs its service while it relays the packets from one endpoint to the other without re-encryption (as in figure 4). The non-split nature of mmTLS is especially beneficial to performance as it eliminates redundant memory copies in stack processing as shown in figure 5. In case a client or a server sends the application data before the session keys are delivered to the middlebox, the middlebox can hold back the data until the keys arrive. Such key delivery delay could slow down the network communication, but we later show that this extra delay is negligible in most cases when compared to typical end-to-end delays in wide-area networks.

We note that session key sharing is hardly novel as previous works [33, 40] employ it for different architectures. However, this simple idea enables a scalable TLS traffic monitoring architecture for client-side middleboxes, which we will explain later. mmTLS easily supports a chain of monitoring middleboxes without exposing them in the handshake phase to the other endpoint like [51]. In addition, it naturally supports TLS session resumption since key sharing also happens after that. We argue that this design would allow the TLS handshake process to remain simple and to evolve independently of any middleboxes in the future.

**End-to-end content integrity verification.** Our base design poses a problem in case an mmTLS middlebox is compromised. A compromised middlebox with the session keys can modify the content arbitrarily without notice of the endpoint receiver. This is increasingly more problematic as AEAD is becoming commonplace (i.e., TLS 1.3 mandates it) – a single key (per direction) is used for both content encryption and tag generation.

To remedy the problem, mmTLS slightly extends the TLS handshake to generate two keys per direction[5] – one for the original session key (k1) and the other (k2) for generating an extra tag (called *private tag*) per TLS record. When preparing a TLS record, the sender not only encrypts the data and generates an original tag with k1, but also creates a private tag over the same record with a different key, k2. Note that k2 is used only for private tag generation and endpoints should not share it with read-only mmTLS middleboxes. Tag verification with k2 enables an endpoint receiver to detect any illegal modification by read-only middleboxes or any other attackers in the middle of the network path.

Figure 6 shows the process – a middlebox can verify the original tag with k1, but it cannot modify the data and generate a correct private tag as it does not own k2. If a compromised middlebox with access to k1 makes illegal modification of the data, a legitimate middlebox may not detect it. However, the endpoint receiver can eventually detect it as it

[4]Operators can configure the TLS middlebox to detect and block clients if they keep sending null keys for non-sensitive destinations.

[5]In real deployment, this part can be implemented as a new TLS extension, which makes it incrementally deployable.

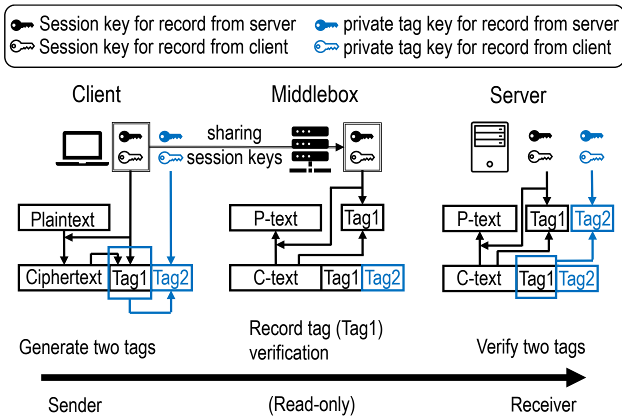



**Figure 6:** Private tag generation for end-to-end content integrity. Endpoints use a different private tag key per direction.

$Tag = MSB_t(GHASH(H, A, C) \oplus E(K, Y))$ where block size $t = 128$

$H = E(K, 0^{128})$ , $C = E(K, P)$ , and

$$Y = \begin{cases} IV || 0^{31} || 1 & \text{for } len(IV) = 96 \\ GHASH(IV || 0^{s} || 0^{64} || len_{64}(IV)) & \text{for otherwise} \end{cases}$$

where $s = (128 - (len(IV) \bmod 128)) \bmod 128$

**Figure 7:** Steps for private tag generation in AES-GCM. Note that $MSB_t$ refers to $t$ most significant bits, $E(K, x)$ represents encrypting $x$ with a key, $K$. Also, $H$, $C$, $P$, $Y$, and $IV$ mean hash key, ciphertext, plaintext, counter 0, and initial vector, respectively.

always verifies the private tag with k2.

The private tag requires the extension of TLS record format, but the update is fairly small and it is incrementally deployable as a standard TLS extension. If a sever does not support the private tag extension, middleboxes and clients can roll back to the legacy MITM mode. Even without the extension, middleboxes would benefit from higher performance in the same security level as in today's deployment. Table 2 compares the protocol modification and operating behaviors with existing solutions.

## 3.3 Efficient Private Tag Generation

Generating a private tag per each TLS record may incur a high overhead if it has to hash the entire content again. Instead, a traffic sender in mmTLS creates the MAC of only the original tag, which would reduce the overhead substantially. Figure 7 shows the mathematical steps for tag generation in AES-GCM which exploits the $GHASH()$ operation. For the private tag generation, we run the following operations with a private tag key, $K_2$.

$$Tag_2 = MSB_t(GHASH(H_2, A, Tag) \oplus E(K_2, Y)), \quad (1)$$

$$\text{where } H_2 = E(K_2, 0^{128}) \quad (2)$$

We believe the same technique as above can be applied to other TLS 1.3 cipher suites such as AES-CCM or

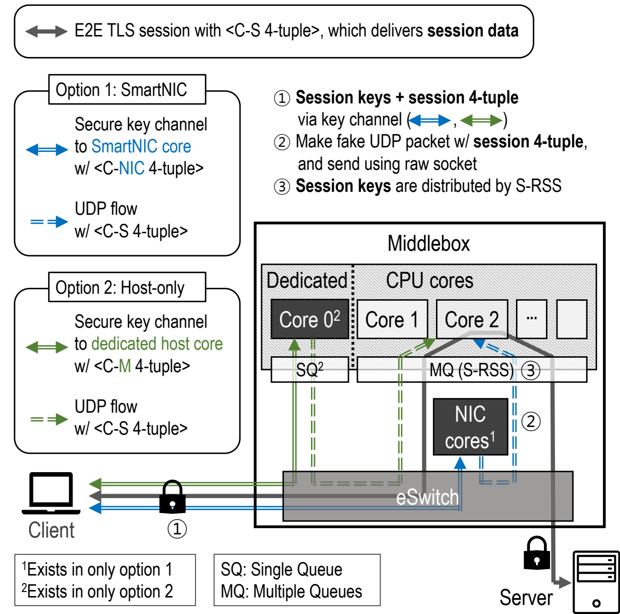



**Figure 8:** Key delivery to the right CPU core in the middlebox using (1) SmartNIC or (2) a dedicated CPU core

ChaCha20Poly1305. For tag verification, the endpoint receiver should verify the original tag first per each TLS record, and if it is correct, it should move on to verify the private tag as well. This two-step verification is necessary as verifying only the private tag would allow the attackers to plug in arbitrary TLS record content with some valid pair of (original tag, private tag). An mmTLS middlebox verifies only the original tag as it does not have the key for the private tag. We find that the cost for private tag generation and verification is minimal in practice – it incurs only 2~5% extra overhead depending on the record size as shown in Section 5.4.

## 3.4 Scalable TLS Session Key Delivery

An mmTLS middlebox monitors multiple concurrent TLS sessions in parallel on a multicore system by adopting the share-nothing architecture. It leverages symmetric receive-side scaling (S-RSS) [71] as both the upstream and downstream flows of a TCP/TLS connection need to map into the same CPU core. Each CPU core on an mmTLS middlebox handles only the TLS sessions that are steered to it by S-RSS.

However, one problem arises when a client delivers the session keys of its own TLS session to the mmTLS middlebox. Note that the client establishes a separate TLS session with the middlebox for session key delivery, and the same TLS session is used for delivering the session keys of multiple different end-to-end TLS sessions of the client that are monitored by the middlebox. So, blindly applying S-RSS to the key delivery packet on the mmTLS middlebox would forward it to a wrong CPU core that does not monitor the target TLS session operating with the keys. The CPU core that receives

| Event | Description |
|---|---|
| ON_TLS_SESSION_START | TLS session initiation |
| ON_TLS_SESSION_END | TLS session termination |
| ON_TLS_HANDSHAKE_START | TLS handshake initiation |
| ON_TLS_HANDSHAKE_END | TLS handshake finish |
| ON_TLS_NEW_RECORD | New TLS record arrival |
| ON_TLS_ERROR | TLS protocol parsing error |

**Table 3:** mmTLS built-in events for TLS session monitoring.

the packet may look up the right CPU core and deliver the keys to it, but that would require running S-RSS in software as well as a lock to safely deliver the keys.

mmTLS addresses the problem by re-routing the key-delivery packet to the right CPU core. When an mmTLS middlebox receives the key-delivery packet along with the four tuples of the monitored TLS session, it creates a fake UDP packet with the same four tuples of the monitored session and has it carry the session keys. Then, it sets the Ethernet address destined to the local NIC. This process allows the UDP packet to have the same four tuples as those of the target TLS session so that the packet would end up at the right CPU core for the session as it is steered by S-RSS on the NIC hardware. This avoids software S-RSS computation as well as any lock contention.

Figure 8 illustrates the localhost session key delivery mechanism of mmTLS. There are two implementation options. The first option is to leverage SmartNIC to manage a key delivery TLS session per client. When SmartNIC receives the session key information from a client (①), it creates and sends a UDP packet with the same four tuple to the NIC of the CPU side (② and ③). The session key information includes the "client random" and "server random" strings exchanged at the handshake to identify the right TLS session of the keys. The second option is to leverage a dedicated CPU core that handles only key delivery while the rest of the CPU cores run end-to-end TLS session monitoring.[6] We assume that the localhost communication on a middlebox is secure, but if not, one can choose to encrypt the UDP packet with a symmetric key.

## 3.5 Event-Driven TLS Traffic Monitoring

mmTLS provides an event-driven TLS application development framework for easy app-level customization. This framework enables developers to focus on high-level custom logic on the plaintext content rather than handling low-level TLS protocol parsing or record decryption. The underlying framework parses the TLS protocol and runs TCP flow reassembly for each session while it forwards the packets between the two endpoints.

Table 3 lists the current set of mmTLS events for which one can program a custom event handler. For example, ON_-TLS_NEW_RECORD arises whenever a new TLS record arrives (i.e., when a full TLS record is collected in the internal buffer). So, one can write a DPI application by registering an event handler (via mmtls_register_callback()) for ON_TLS_-NEW_RECORD. Inside the event handler, the application can call mmtls_get_record() to retrieve the fully-decrypted TLS record body or only the record size without decrypting the content. One can choose to decrypt only select TLS records as ON_TLS_NEW_RECORD is raised per each new TLS record and the record body is decrypted only by calling mmtls_get_record(). Furthermore, the application can skip an arbitrary number of bytes of application-level content (e.g., skipping the remaining HTTP response) by mmtls_pause_monitor() without TLS record decryption. The application developer can also get the detailed TLS session information such as the TLS version, the negotiated cipher suite, the four tuples of the session, etc., with mmtls_-get_tlsinfo(). With the mmTLS events and APIs (shown in figure 23 in Appendix), one can write an efficient but flexible TCP session monitoring application.

When packets arrive at an mmTLS middlebox, the middlebox runs TCP/TLS protocol parsing with the packets and reassembles the flow data for application-level content. Events are raised while running this task, and registered event handlers are invoked in the order of event generation. In case an

[6] One can even distribute the key delivery TLS sessions across the CPU cores, but the implementation would be challenging as we use DPDK [12].

```
1  static void
2  cb_new_record(mmtls_t mmctx, int cid, int side)
3  {
4      int len;
5      uctx *c = mmtls_get_uctx(mmctx, cid);
6      /* get plaintext record */
7      if (mmtls_get_record(mmctx, cid, side,
8                           c->buf + c->off, &len) < 0)
9          goto error_case;
10     c->off += len;
11     if (!IsHTTPHdrReceived(c) && ParseHTTPRespHdr(c))
12         len = c->off - c->bodyOff;
13     if (IsHTTPHdrReceived(c)) {
14         /* DPI on the response body */
15         if (DPI(c->dpiCtx,
16                 c->buf + c->bodyOff, len) < 0)
17             goto error_case;
18         c->bodyOff += len;
19     }
20     /* monitor only first 10KB of response */
21     if (c->bodyOff > 10000)) {
22         mmtls_pause_monitor(mmctx, cid, side,
23                             c->contLen - c->bodyOff);
24         PrepareNewResponse(c);
25     }
26     return;
27 error_case:
28     free_ctx(c);
29     mmtls_reset_session(mmctx, cid);
30 }
```

**Figure 9:** Code snippet for a simple TLS middlebox that inspects the first 10KB of HTTP 1.1 responses. We omit the logic for error handling and buffer overflow handling for brevity.

event handler detects malicious content in the TLS record (or over the collected data combined with previous records), the application can choose to block the packets and terminate the session (by `mmtls_reset_session()`). If not, the packets are forwarded to the other endpoint. Note that packets arriving out of order in a flow are still forwarded to the other endpoint without being blocked for full-reassembly of the in-order data. This is necessary as the mmTLS middlebox does not terminate the TCP connection and the other endpoint must send an ACK/SACK properly to inform the sender of (possibly) lost packets. In fact, this is more efficient than split TCP connections as all packets are delivered directly to the other endpoint without getting blocked nor buffered for later delivery.

A potential concern might be that a part of malicious content could reach the other endpoint as out-of-order packets are forwarded without raising the `ON_TLS_NEW_RECORD` event. However, this is OK as the TLS standard dictates that the TLS layer should not deliver a partially-decrypted TLS record to the upper layer without tag verification [19].[7] Later when a full record is collected, the middlebox would detect the malicious content and terminate the connection without forwarding the packets. Thus, the other endpoint would not have a chance to collect a full TLS record for decryption.

Finally, mmTLS events support `ON_TLS_ERROR` when TLS protocol parsing fails. Then, the application can check the type of the error by `mmtls_get_error()`. Figure 9 shows an example of application code that runs DPI and pattern matching on the first 10KB of the response.

## 4 Implementation

We briefly explain the implementation of the mmTLS framework and porting existing TLS applications.

**mmTLS middlebox programming framework.** We implement the mmTLS framework with mOS [45] running on Intel DPDK. mOS enables event-driven TCP flow monitoring by allowing the developer to write event handlers for built-in and user-defined TCP stack events. mmTLS extends mOS to support TLS events with its built-in TCP events. In addition to TCP connection setup (`ON_TLS_SESSION_START`) and teardown events (`ON_TLS_SESSION_END`), mmTLS supports TLS handshake events to closely monitor the handshake process (`ON_TLS_HANDSHAKE_START` and `ON_TLS_HANDSHKE_END`). mmTLS supports `ON_TLS_NEW_RECORD` by leveraging the `MOS_ON_CONN_NEW_DATA` event of mOS. While `MOS_ON_CONN_NEW_DATA` is generated for new arrival of in-ordered data of a flow, mmTLS ensures to raise `ON_TLS_NEW_RECORD` for each fully-collected TLS record. mmTLS exploits `MOS_ON_PACKET_IN` of mOS to stall application-level TLS packets until the keys for the TLS session arrive.

[7]The endpoint receiver must terminate the connection with a `bad_record_mac` alert if tag verification fails.

The current mmTLS implementation supports all cipher suites[8] of TLS 1.3 and all AEAD symmetric ciphers supported by TLS 1.2. Supporting symmetric ciphers is simple in mmTLS as it actively follows the TLS handshake process. For fair performance comparison with mcTLS [58], we added support for AES-CBC with TLS 1.2 as well. mmTLS optimizes the existing mOS implementation by benefiting from TCP segmentation offload (TSO) and Large Receive Offload (LRO) for better TCP flow processing performance. It shares the DMA buffer for packet RX and TX operations to avoid memory copy in packet forwarding. The mmTLS middlebox programming framework consists of 2,330 lines of C code. We also implement a simple DPI application using Hyperscan [70], which requires 696 lines of C code.

**Session key forwarder.** The mmTLS framework receives session keys from each client, and delivers them to the right CPU core that monitors the end-to-end TLS session. We implement it as an `epoll`-based OpenSSL server that uses `sendmmsg()` to send multiple keys over different UDP packets to the right CPU cores. It consists of 750 lines of C code, and the same code runs on both SmartNIC and dedicated CPU core as shown Figure 8. mmTLS uses a dedicated RX queue per core for UDP packets delivering session keys to avoid any packet loss due to interference by regular TLS traffic. For SmartNIC implementation, we use NVIDIA Bluefield-2 [17], but the system can run on stripped-down SmartNIC as well.

**mmTLS clients.** We port Chromium Web browser [7] and two HTTP benchmark tools (h2load [59] and ab [31]) to support mmTLS. For session key sharing, we use an existing OpenSSL API, `SSL_CTX_set_keylog_callback()`, to extract the session keys at TLS handshake. For key derivation in the client, we use HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [13] and Pseudorandom Function (PRF) [24], for TLS 1.3 and 1.2, respectively. For the session key delivery TLS session, we use TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 of TLS 1.3 as the cipher suite, and conduct middlebox authentication at each session. We build the above functions into a common library of 630 lines that client applications link to for easy porting.

**Private tag generation.** We update OpenSSL 3.2.0 to reflect the extra tag generation/verification cost into TLS record en/decryption. The update includes 1) adding `aesni_set_encrypt_key()` and `aesni_encrypt()` to `tls13_set_crypto_state()` for initializing $H = E(K, 0^{128})$ and $E(K, Y)$ during TLS handshake, and 2) adding `gcm_ghash_avx()`, which performs `xor` and field multiplication (`gcm_gmult_avx()`) with initialized $H$, to `tls13_cipher()` for additional $GHASH()$ operation which generates/verifies *private tag* upon en/decrypting each record. The update requires 141 extra lines of code. We link nginx to the updated OpenSSL library to reflect the cost on an endpoint server.

[8]AES-GCM, AES-CCM, and ChaCha20-Poly1305.

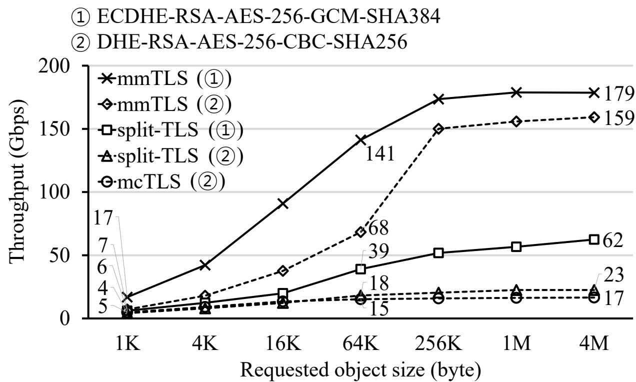



**Figure 10:** Comparison of throughputs for persistent connections

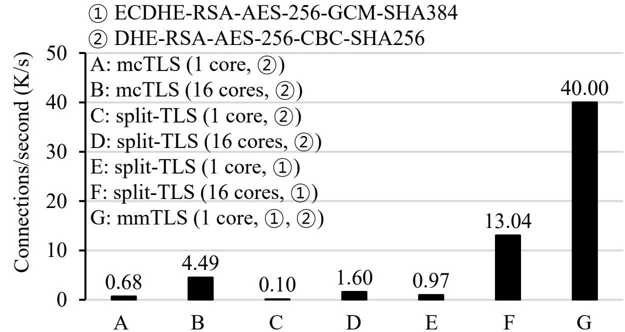



**Figure 11:** Comparison of throughputs for ephemeral connections for downloading a 1KB object per connection

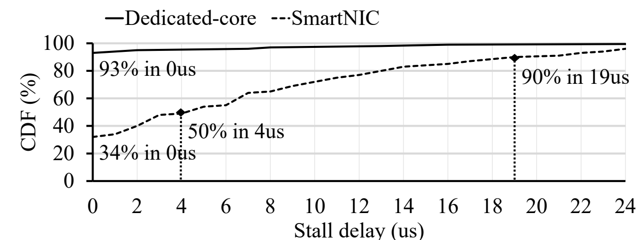



**Figure 12:** Stall delay of key arrival in mmTLS middlebox when download 1KB object

## 5 Evaluation

We evaluate mmTLS on the following aspects – performance benefit over existing solutions, delay overhead for out-of-band session key delivery, multicore scalability and private tag generation overhead. We also evaluate if mmTLS improves the performance of real-world applications, and how easy it is to develop an TLS application.

### 5.1 Experiment Setup

**Comparison target.** We compare the performance of an mmTLS middlebox against existing solutions. For a baseline MITM middlebox, we use the nginx TLS proxy (version 1.24.0), and we call it split-TLS. We also compare the performance with mcTLS [58], an TLS-extension-based middlebox architecture. We could not compare with mbTLS [57] nor maTLS [51] as the source code of the former is unavailable and the latter supports only a single TLS handshake with a content that is represented in a single TLS record. For gauging the performance overhead incurred by a middlebox, we compare with the performance of direct end-to-end TLS connections (E2E-TLS) as well.

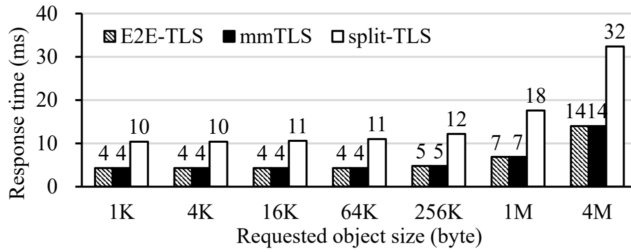
**Test setup and scenario.** We use TLS 1.3 with TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 as a default cipher suite for evaluation, as it is currently the most popular [49]. For comparison with mcTLS [58], we use TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 of TLS 1.2. We configure mcTLS to use a read-only context, which makes an mcTLS middlebox verify a reader tag for each record. Unless mentioned otherwise, all experiments decrypt each record but do not run DPI (or pattern matching) on the plaintext content to focus on the performance by different architectures. The middlebox platform has a 16-core Xeon Gold 6326 CPU @ 2.90GHz CPU with a Bluefield-2 dual-port 100GbE NIC, which is used for session key forwarding by default. For servers, we use two machines equipped with 24-core Xeon Gold 6342 CPU @ 2.80GHz, and 16-core Xeon Gold 6326 CPU @ 2.90GHz, respectively. Both servers employ a dual-port ConnectX-6 100GbE NIC. We use HTTP/1.1[9] nginx web server (version 1.24.0) for the endpoint servers. For clients, two machines equipped with two 22-core Xeon E5-2699 v4 CPUs @ 2.20GHz and the other two machines equipped with a 16-core Xeon E5-2683 v4 CPU @ 2.10GHz. All client machines are equipped with a ConnectX-5 100GbE NIC. On clients, we run h2load [59] for persistent connection tests, and ab [31] for ephemeral connection tests. For test scenarios, clients request fixed-sized objects from the nginx servers in HTTPS/1.1.

### 5.2 Throughput and Delay Performance

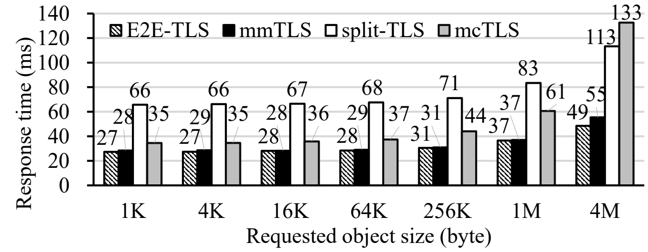
**Persistent connections.** We compare the throughputs with persistent connections for delivering contents whose size ranges from 1KB to 4MB. We assume that TLS middleboxes decrypt the first 64 KB of the content for inspection as many IDSes limit the region for DPI. We employ 4096 concurrent TLS sessions for the experiments, and we confirm that all 16 cores of the middlebox platform are fully utilized. Figure 10 shows that the mmTLS middlebox achieves 170+ Gbps for contents larger than 64KB with AES-GCM 256-bit keys. It outperforms split-TLS (①) by 2.7x ~ 4.6x while it outperforms mcTLS (②) by 1.5x ~ 9.6x. mmTLS (②) achieves much higher throughput for larger files because the middlebox decrypts only the first 64 KB of the response[10]. The

[9] nginx currently does not support HTTP/2 for backend connections.

[10] ②'s performance should be similar to that of ① for 64+ KB files, but our testbed could not make the mmTLS middlebox a bottleneck with ② due to the bottleneck of endpoint servers.



(a) ECDHE-RSA-AES-256-GCM-SHA384



(b) DHE-RSA-AES-256-CBC-SHA256

**Figure 13:** Comparison of delay performance for LAN connections with two cipher suites

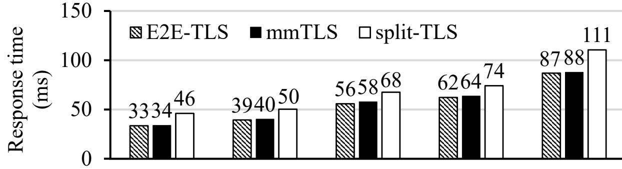


**Figure 14:** Comparison of response times for popular news sites

performance benefit reflects the architectural difference – nginx and mcTLS adopt split TCP connections, so they have to reassemble the flow data from one endpoint, decrypt and re-encrypt the data, and send it to the other endpoint. mcTLS suffers from poorer performance as it `forks` a process for each TLS session. In contrast, mmTLS reassembles the flow data without blocking the packets nor re-encrypting the data.

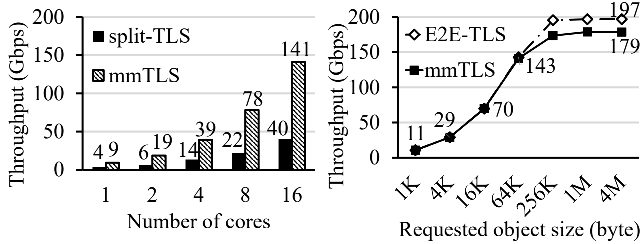
**Ephemeral connections.** mmTLS uses explicit key sharing via secure out-of-band channel, so one may wonder if asynchronous key delivery impacts the performance. To answer it, we evaluate the throughput of ephemeral connections with a small content. In this experiment, clients request 1KB objects but they perform a TCP connection setup and a TLS handshake for every request. We measure the peak TLS handshake throughput when the CPU of a middlebox is fully saturated. Figure 11 shows the results. The peak throughput of mmTLS is **41.2x** larger than that of split-TLS when they both employ a single CPU core with the cipher suite ①. The split-TLS throughput is only 970 TCP/TLS sessions per second as it needs to perform the TLS handshake twice in sequence. In contrast, mmTLS achieves a much higher throughput as it simply forwards the handshake packets between the two endpoints. In fact, even the 1-core throughput of an mmTLS middlebox is 3.1x larger than that of 16-core with split-TLS. The throughput gap from mcTLS with ② is even larger, as mcTLS uses DHE as the key exchange algorithm, which is much slower than ECDHE. ECDHE employs elliptic curve cryptography that operates on a smaller key size while DHE suffers from heavyweight modular exponentiation. The CBC mode that mcTLS adopts is also slower than the GCM mode as its encryption is serialized per each block. Also, mcTLS suffers from the `fork()` overhead per connection. As a result, mmTLS outperforms mcTLS by **58.8x** as the throughput of mmTLS does not change. mmTLS outperforms split-TLS by **400.0x** with ② as split-TLS with ② is much slower than with ①.

Our evaluation confirms that the extra TLS handshake by split-TLS or mcTLS is substantially more heavyweight than the asynchronous session key delivery by mmTLS. Figure 12 shows the distribution of stall delays of application-level packets due to key delivery. It shows that mmTLS with SmartNIC does not delay the application packets at all for 34% of the time, and the median and $90^{th}$% stall delays are 4 us and 19 us in our setup, which would minimally affect the end-to-end latency in the WAN. When using a dedicated core as the session key forwarder, the stall delay becomes even smaller but at the cost of throughput loss by one core.

**Response Time.** We compare the response times for downloading objects of various sizes. In our testbed, the average RTTs are 83us (between a client and a middlebox) and 40us (between a middlebox and a server), respectively. We report the average response time for downloading 100 times. Figures 13a and 13b show that the response times of E2E-TLS and mmTLS are almost the same with object size up to 1 MB (with at most 2 ms extra delay) regardless of the cipher suites. The 6 ms latency blowup for 4MB objects reflects the extra packet forwarding overhead by the mmTLS middlebox. In contrast, split-TLS suffers from 2.3~2.8x and 2.2~2.4x larger response times for each cipher suite, respectively. We observe that mcTLS outperforms split-TLS as key exchanges for the middlebox are converged into one E2E handshake. However, the response time becomes much worse when the content size grows up to 4 MB.

The delay blowup for a small object in split-TLS and mcTLS is mainly due to the extra key exchange process. When we chain multiple middleboxes, split-TLS adds 5~7 ms of extra delay per each middlebox for ephemeral connections. For persistent connections, the per-hop delay of split-TLS is 0.5~0.6 ms per middlebox due to content re-encryption and TCP operations. In contrast, we can see that an mmTLS middlebox incurs less than 0.2 ms of extra delay per middlebox regardless of ephemeral or persistent connections.

We run the same experiments for five popular news sites in the real world as shown in figure 14. The content sizes of the target objects are 204KB, 230KB, 638KB, 1.94MB, and 2.06MB (from A to E in figure 14), respectively. The round trip times to the sites are 5.6ms, 5.5ms, 5.2ms, 5.5ms, and




**(a)** Multi-core scalability **(b)** Performance penalty

**Figure 15:** Comparison of multi-core scalability over split-TLS and and performance penalty over E2E TLS on the same spec machine with an endpoint server.

3.5ms, respectively. They all use ECDHE with X25519 curve for key exchange, and RSA for authentication. For a symmetric cipher, E uses AES_256_GCM_SHA384 while all the other sites use AES_128_GCM_SHA256. Figure 14 shows a similar trend as our earlier experiments.[11] We see that the extra delays by split-TLS over E2E-TLS for WAN connections are similar to those in the LAN for similar object sizes. While split-TLS incurs at least 11 ms of extra delay over E2E-TLS, the extra delay by mmTLS is at most 2 ms.

## 5.3 Scalability & Middlebox Overhead

**Multicore performance scalability.** We evaluate the performance scalability over multiple CPU cores. In this experiment, clients request 64KB objects with 4096 concurrent, persistent connections. From figure 15a, we observe that the throughputs of both mmTLS and split-TLS middleboxes scale to the number of employed CPU cores, but the single-core performance of mmTLS is 2.5x larger than that of split-TLS, achieving 141 Gbps at 16 cores. We could not measure the throughput of ephemeral connections as we do not have enough client-server machines to saturate many CPU cores of an mmTLS middlebox. Instead, we report that the session key forwarder on SmartNIC (or on a dedicated CPU core) can handle 338K (or 298K) connections per second, which would be the bottleneck in the ephemeral-connection case. We believe we can improve the performance of the forwarder further by switching to AF_XDP [2] for UDP packet creation and sending.

**Overhead by an mmTLS middlebox.** We evaluate if an mmTLS middlebox limits the throughput that can be achieved by a server running on the same hardware platform. Figure 15b compares the E2E-TLS performance with that of the same server when an mmTLS middlebox is deployed for persistent connections. The figure shows that the mmTLS middlebox does not constrain the performance of the server up to 64KB. If the content size exceeds 64KB, the mmTLS middlebox becomes a bottleneck due to internal flow data reassembly operations.

[11]We could not compare with mcTLS as the server needs to be updated.

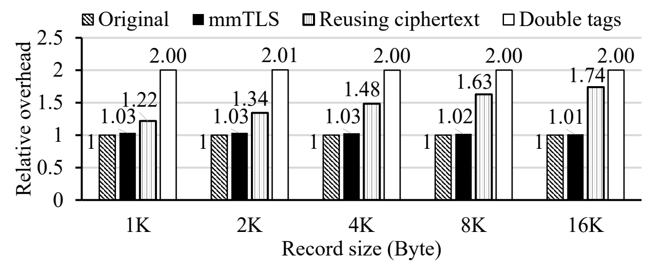



**Figure 16:** Microbenchmark for private tag generation

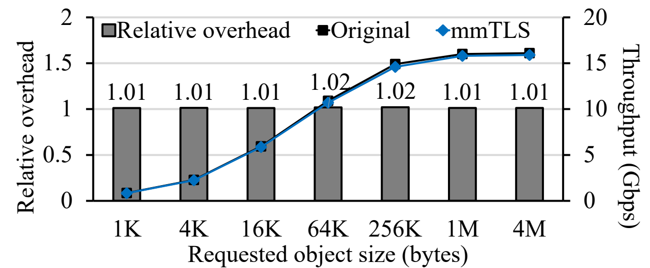



**Figure 17:** Throughput of end-to-end TLS with private tag generation extension on sender side

## 5.4 Private Tag Generation Overhead

Figure 16 compares the relative private tag generation overhead over different TLS record sizes in AES-GCM with 256bit keys. "Double tags" refers to running the cipher again with a different key for the extra tag while "reusing ciphertext" refers to sharing the ciphertext from the original key for computing the hash value. mmTLS refers to our scheme while "original" refers to the existing scheme without private tag generation. "Original" serves as the baseline. As we see, our logic incurs only 1~3% overhead from the original scheme while "reusing ciphertext" shows 22% to 74% of extra overhead. This implies that $GHASH()$ incurs more overhead than encryption for large records while the overhead by our logic is minimal as the input size is small (16B).

We measure the relative overheads and the throughputs of E2E-TLS connections with and without the private tag generation extension. Figure 17 shows that the relative overhead on the server is at most 2%. This is because the portion of the private tag generation computation becomes smaller due to other operations. Note that our logic provides the same benefit to an endpoint receiver as well since the cost for private tag verification is the same as the generation.

## 5.5 Real-world Application Performance

**Page Load Time with Chromium.** We measure the page load times (PLTs)[12] of the front page at five popular news sites with mmTLS-ported Chromium in our campus and compare with E2E-TLS. Figure 18 shows that the PLTs range from 1.58 to 4.11 seconds, but the difference from E2E-TLS is at most

[12]Unlike in benchmark tests, page load time in the browser includes parsing DOM, executing scripts, and downloading embedded objects.

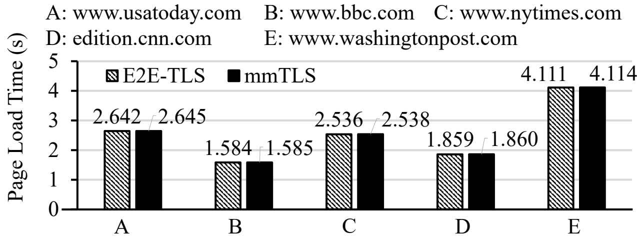



**Figure 18:** Comparison of page load times for WAN connections

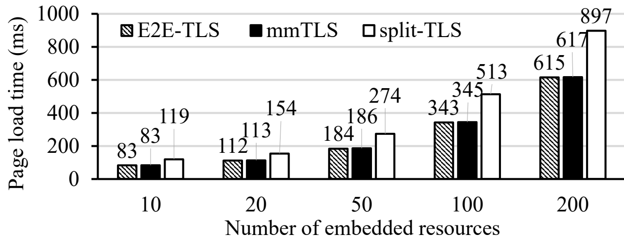



**Figure 19:** Comparison of page load time for LAN connections

3 ms. We could not measure the PLT with split-TLS as the setup by Chromium cannot fetch the objects from multiple different domains. Instead, we measure the PLT with a single server in the LAN that serves a page with embedded objects. Figure 19 compares the PLTs of pages with a different number of embedded objects whose size is 136 KB. Similar to WAN experiments, mmTLS exhibits similar PLTs with E2E-TLS, but we find split-TLS suffers from 38 to 50% larger PLTs.

**Snort ruleset matching.** We compare the DPI performance with a popular Snort ruleset of 10K entries (Snort3-community-rules) [22]. We implement multi-string pattern matching with Hyperscan [70] for an mmTLS middlebox as well as for the nginx TLS proxy. We use the same experiment setup as ① in figure 10, and have each connection request 1MB objects. We control the amount of content for DPI from 16KB to 128KB as typical IDSes do not inspect the entire content. Figure 20 shows the results where mmTLS achieves 2.7x to 3.1x higher throughputs over the split-TLS architecture. This confirms that a DPI application benefits from the architectural efficiency of mmTLS.

## 5.6 Popular TLS Cipher Suites

The mmTLS events and APIs are simple and flexible, so one can easily use it to measure the properties of TLS sessions as well. We write a simple mmTLS middlebox that records the cipher suite chosen by a server site. One can get the information in the event handler of `ON_TLS_HANDSHAKE_END` as shown in figure 22. The entire source code is only 62 lines. Figure 21 shows the distribution of TLS cipher suites for 1000 sites listed at the Alexa Top 1M list [3]. We see that 81.7% of the sites use TLS 1.3 and 99.8% of the sites use AES-GCM, which confirms the dominant symmetric cipher employed in practice. 72.1% of the sites that use AES-GCM use 128-bit keys rather than 256-bit keys.

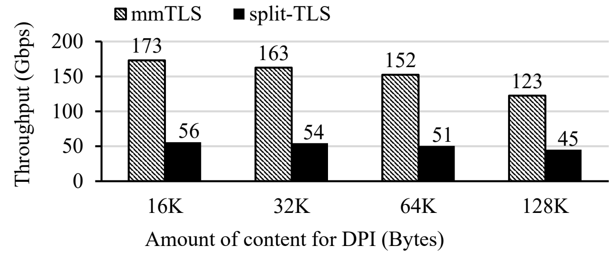



**Figure 20:** Comparison of throughput of DPI applications

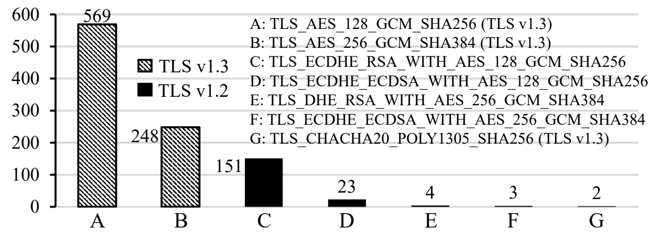



**Figure 21:** Counts of cipher suites and TLS version among 1000 available HTTPS web sites from Alexa top 1M.

## 6 Related Work

We briefly explain the related works here.

**Analysis of MITM middleboxes.** A number of studies have consistently raised security issues with the MITM middleboxes. Carnavalet et al. [35] and Waked et al. [69] present frameworks to analyze and evaluate vulnerabilities of TLS proxies. Huang et al. [42] and Durumeric et al. [37] present methods to detect MITM attacks, and HTTPS interceptions, respectively. Mani et al. [53], and O'Neill et al. [61] conduct measurement studies on open proxies, and TLS proxies in terms of security issues, respectively. Carnavalet et al. [36] also present extensive survey and analysis on MITM middlebox and TLS interception.

**Frameworks for high performance middleboxes.** mOS [45] and microboxes [52] provide a highly scalable and flexible framework for developing TCP connection monitoring middleboxes. Both frameworks support TCP flow reassembly, but they do not support easy development of TLS middleboxes for monitoring encrypted traffic. MiddleClick [32] extends Click [48] to allow monitoring and modifying individual TCP packets without terminating a TCP connection. However, the framework does not support reassembly of TCP packets, so one has to implement it in the upper layer to monitor the content over multiple packets especially if they arrive out of order.

**Secure TLS middlebox architecture.** There has been a long thread of works that endeavor to enhance the security and accountability of TLS middleboxes. For instance, mcTLS [58] aims to impose restrictions on the behavior of middleboxes by introducing two additional MAC keys for `read` and `write` operations, which ensures that only authorized middleboxes can utilize one of these keys for their permitted access. This context-dependent access control

```
static void
cb_handshake_end(mmtls_t mmctx, int cid, int side)
{
    session_info info;
    uctx *c = mmtls_get_uctx(mmctx, cid);
    mmtls_get_tls_info(mmctx, cid, &info,
        SNI|VERSION|CIPHER_SUITE);
    fprintf(c->logfile,
        "server name:%s version:%d cipher:%d\n",
        info.sni, info.version, info.cipher_suite);
}
```

**Figure 22:** Code snippet for a simple TLS middlebox that logs server name, TLS version, and negotiated cipher suite for each session.

mechanism enables the endpoints to be aware of the presence of middleboxes and empowers them to detect any unauthorized modifications made to the transmitted content.[13] mbTLS [57] was proposed to facilitate interoperability with legacy TLS endpoints while safeguarding session data from untrusted middleboxes through the use of Intel SGX technology. However, its implementation necessitates an additional in-band secondary handshake between the endpoint and each middlebox for every end-to-end session. This introduces computational overhead and can potentially create a bottleneck within the middlebox infrastructure. maTLS [51] introduces *middlebox certificates* to enable middleboxes to participate in the TLS session, thus enhancing visibility and auditability. However, like mcTLS and mbTLS, it increases overhead during the handshake and end-to-end response time by splitting the TLS connection into multiple segments.

In common, these extensions focus on security issues in the middleboxes using split-connection method. However split-connection method has a fundamental issue in terms of performance. Also, since they consider the general use cases of middleboxes including read-write middleboxes that usually lie on the server-side network, there is no opportunity to improve the performance of read-only middleboxes that are usually used for traffic monitoring in the client-side network. In addition, they all require intrusive modification on the existing TLS protocol, so lower the deployability.

**Session key sharing.** Similar to TLS-extension works, the IETF key-share extension [1] proposes in-band session key sharing with TLS middleboxes. However, in-band session key sharing is not only invasive, but it incurs a high overhead as middleboxes must participate in the handshake. mmTLS avoids the issue by out-of-band session key sharing. LOCKS [33] also adopts out-of-band key sharing with Bro [62], but it does not ensure content integrity nor supports a generic TLS middlebox platform. EndBox [40] proposes running TLS middleboxes on a client with shared keys from their modified TLS libraries. However, their middleboxes are based on Click [48] that does not support TCP flow reassembly, so it would be challenging to verify TLS records if packets are lost or delivered out of order. Likewise, it fails to support general DPI applications that inspect the content spanning over multiple packets. In summary, we note that key sharing itself is not our contribution, but none of the existing works propose a high-performance TLS traffic monitoring framework that allows easy middlebox development.

**Searchable encryption.** BlindBox [67] and Embark [50] focus on a monitoring middlebox that mainly locates in the client-side network. They present such a novel method for pattern matching: instead of decrypting the traffic to be inspected, they encrypt the patterns and send them to monitoring middleboxes via secondary secure channel. Because the monitoring middleboxes are read-only, and unaware of session keys, they can be run on the public clouds. However, since these kinds of method are tied to specific encryption algorithms, such as order-preserving encryption, they are also not practical to be used in real world.

## 7 Conclusion

We have presented mmTLS, a novel TLS monitoring middlebox framework that scalably inspects the traffic without splitting the TLS/TCP connection. While existing solutions improve the security aspect of MITM middleboxes, they suffer from poor performance due to redundant TLS handshakes, content reassembly and relaying over two connections, and content re-encryption. mmTLS breaks away from the constraints and operates with event-driven programming library that allows developers to focus on the core logic of content monitoring rather than dealing with low-level TLS/TCP protocol parsing and encryption. We have demonstrated that mmTLS outperforms split-TLS by up to 4.6x, achieving 179 Gbps of throughput with a popular TLS 1.3 cipher suite. It also improves the TLS handshake performance by 41.2x over split-TLS as it avoids participating in the handshake process. We have shown that porting a popular browser to using mmTLS is not difficult, and that a DPI application with a real-world ruleset on mmTLS achieves 100+ Gbps. We believe that mmTLS is incrementally deployable and we plan to release the source code of mmTLS for practical use.

## Acknowledgments

We appreciate the insightful feedback and suggestions from USENIX ATC 2024 reviewers. This work is in part supported by the ICT Research and Development Program of MSIT/IITP, Korea, under [2022-0-00531, Development of in-network computing techniques for efficient execution of AI applications], [RS-2024-00349594, Development of networked systems technologies leveraging SmartNIC], [Next-generation Cloud-native Cellular Network Center] and the New Faculty Startup Fund from Seoul National University.

[13] There is a standardized specification driven by ETSI, named Middlebox Security Protocol (MSP) [65]. It extends mcTLS to support AEAD, and other operations such as `delete`.

## References


[1] A Method for Sharing Record Protocol Keys with a Middlebox in TLS. https://www.ietf.org/archive/id/draft-nir-tls-keyshare-02.txt.

[2] AF_XDP – the Linux kernel documentation. https://docs.kernel.org/networking/af_xdp.html.

[3] Alexa Top Websites - Last Save. https://www.expireddomains.net/alexa-top-websites/.

[4] Amazon Prime. https://www.amazon.com/amazonprime.

[5] Avast. https://www.avast.com/c-router-malware.

[6] AVG. https://www.avg.com/en/signal/remove-router-virus.

[7] Chromium. https://www.chromium.org/chromium-projects/.

[8] Circle. https://meetcircle.com/.

[9] Cloud-Native Network Firewall Service. https://www.fortinet.com/products/public-cloud-security/cloud-native-firewall.

[10] ConnectSafely. https://www.connectsafely.org/parentalcontrols/.

[11] CrowdStrike. https://www.crowdstrike.com/.

[12] DPDK: Data Plane Development Kit. https://www.dpdk.org/.

[13] HMAC-based Extract-and-Expand Key Derivation Function RFC. https://tools.ietf.org/html/rfc5869.

[14] Hulu. https://www.hulu.com/.

[15] Netflix. https://www.netflix.com/.

[16] NGINX Docs – NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[17] Nvidia BlueField DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[18] RFC 5246. https://www.ietf.org/rfc/rfc5246.txt.

[19] RFC 8446. https://www.rfc-editor.org/rfc/rfc8446.

[20] SentinelOne. https://www.sentinelone.com/.

[21] Snort. https://www.snort.org/.

[22] snort3-community-rules. https://www.snort.org/downloads/community/snort3-community-rules.tar.gz.

[23] Suricata. https://suricata.io/.

[24] The Transport Layer Security (TLS) Protocol Version 1.2. https://tools.ietf.org/html/rfc5246.

[25] Trellix. https://www.trellix.com/.

[26] Twitch. https://www.twitch.tv/.

[27] What Is a Web Application Firewall (WAF)? https://www.akamai.com/glossary/what-is-a-waf.

[28] Youtube. https://www.youtube.com/.

[29] Zeek. https://zeek.org/.

[30] Amer AlGhadhban and Ahmad Showail. SALMA: A Novel Middlebox Infrastructure System Based on Integrated Subnets. *Systems*, 10(5), 2022.

[31] Apache. ApacheBenchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

[32] Tom Barbette, Cyril Soldani, and Laurent Mathy. Combined stateful classification and session splicing for high-speed NFV service chaining. *IEEE/ACM Transactions on Networking*, 29(6):2560–2573, 2021.

[33] Michael Bierma, Aaron Brown, Troy DeLano, Thomas M. Kroeger, and Howard Poston. Locally Operated Cooperative Key Sharing (LOCKS). In *Proceedings of the International Conference on Computing, Networking and Communications (ICNC)*, 2017.

[34] Martin Casado, Michael Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2007.

[35] Xavier de Carné de Carnavale and Mohammad Mannan. Killed by Proxy: Analyzing Client-end TLS Interception Software. In *Proceedings of Annual Network and Distributed Systems Security (NDSS)*, 2016.

[36] Xavier de Carné de Carnavalet and Paul C. van Oorschot. A Survey and Analysis of TLS Interception Mechanisms and Motivations: Exploring How End-to-End TLS is Made “end-to-Me” for Web Traffic. *ACM Comput. Surv.*, 55(13s), jul 2023.

[37] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. The security impact of https interception. In *Proceedings of the Annual Network and Distributed Systems Security (NDSS)*, 2017.

[38] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[39] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Computing Surveys*, 54(6), 2021.

[40] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, and Christof Fetzer. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[41] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the Asia-Pacific Workshop on Networking (APNet)*, 2018.

[42] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged ssl certificates in the wild. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE SP)*, 2014.

[43] Intel. AES-NI. https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html.

[44] Intel. Intel QAT. https://github.com/intel/QAT_Engine.

[45] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2017.

[46] Jeff Jarmoc. SSL/TLS Interception Proxies and Transitive Trust. 2012.

[47] Adrian Kingsley-Hughes. Gogo in-flight Wi-Fi serving spoofed SSL certificates. ZDNet, 2015.

[48] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[49] F5 Labs. The 2021 TLS telemetry report. https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report.

[50] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[51] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted "Taekyoung" Kwon. maTLS: How to Make TLS middlebox-aware? In *Proceedings of Annual Network and Distributed Systems Security (NDSS)*, 2019.

[52] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[53] Akshaya Mani, Tavish Vaidya, David Dworken, and Micah Sherr. An Extensive Evaluation of the Internet's Open Proxies. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[54] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[55] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroSP)*, 2017.

[56] Mikhail Isachenkov and Timo Stark. Improving NGINX Performance with Kernel TLS and SSL_sendfile(). F5 NGINX Blog News, 2021. https://www.nginx.com/blog/improving-nginx-performance-with-kernel-tls/.

[57] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[58] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego Lopez, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[59] nghttp2. H2load. https://nghttp2.org/documentation/h2load-howto.html.

[60] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A Survey of Published Attacks on Intel SGX, 2000.

[61] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. TLS Proxies: Friend or Foe? In *Proceedings of the Internet Measurement Conference (IMC)*, 2016.

[62] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 1998.

[63] Rishabh Poddar, Chang Lan, Raluca Ada Popa, , and Sylvia Ratnasamy. Safebricks: Shielding Network Functions in the Cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[64] Zafar Ayyub Qazi, Cheng-Chun Tu, Rui Miao Luis Chiang, Vyas Sekar, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[65] Tony Rutkowski and Roger Eriksson. Redefining Network Security: The Standardized Middlebox Security Protocol (MSP). https://www.etsi.org/images/files/ETSIWhitePapers/ETSI_WP-43-MSP.pdf, 2021.

[66] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[67] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[68] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proceedings of the ACM Symposium on SDN Research (SOSR)*, 2018.

[69] Louis Waked, Mohammad Mannan, and Amr Youssef. To Intercept or Not to Intercept: Analyzing TLS Interception in Network Appliances. In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[70] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A Fast Multi-Pattern Regex Matcher for Modern CPUs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[71] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceedings of the ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[72] Yahui Zhang, Min Zhao, Tingquan Li, and Huan Han. Survey of attacks and defenses against SGX. In *IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, 2020.

# Appendix A

```c
/*----------------------------------------------------------------------------*/
/* Initialize/destory mmtls variables and memory buffer pools, etc. */
int mmtls_init(const char *fname, int num_cpus);
int mmtls_destroy();
/*----------------------------------------------------------------------------*/
/* Create mmtls a thread context for a given CPU core */
mmctx_t mmtls_create_context(int cpu);
/*----------------------------------------------------------------------------*/
/* Wait until an mmtls thread context joins */
void mmtls_app_join(mmctx_t mmctx);
/*----------------------------------------------------------------------------*/
/* Register/deregister an event handler for a given mmtls thread */
int mmtls_register_callback(mmctx_t mmctx, event_t event, mmtls_cb cb);
int mmtls_deregister_callback(mmctx_t mmctx, event_t event);
/*----------------------------------------------------------------------------*/
/* Pause/resume the ON_TLS_NEW_RECORD event for a content length */
int mmtls_pause_monitor(mmctx_t mmctx, int cid, int side, int len);
int mmtls_resume_monitor(mmctx_t mmctx, int cid, int side);
/*----------------------------------------------------------------------------*/
/* Get a plaintext record, a length, and a record type
 * If buf is NULL, only the length and type are returned */
int mmtls_get_record(mmctx_t mmctx, int cid, int side, char *buf, int *len, uint8_t *type);
/*----------------------------------------------------------------------------*/
/* Set/get a user context for a session
 * The context points to an opaque user data structure */
int mmtls_set_uctx(mmctx_t mmctx, int cid, void *uctx);
void *mmtls_get_uctx(mmctx_t mmctx, int cid);
/*----------------------------------------------------------------------------*/
/* Reset/terminate the session */
int mmtls_reset_session(mmctx_t mmctx, int cid);
/*----------------------------------------------------------------------------*/
/* Get an error code
 * Can be called after ON_TLS_ERROR_CALLBACK and mmtls_get_record() */
int mmtls_get_last_error(mmctx_t mmctx, int cid);
/*----------------------------------------------------------------------------*/
/* Get a set of information related to a given session
 * Information can be determined by bitmask */
int mmtls_get_tls_info(mmctx_t mmctx, int cid, session_info *info, uint16_t bitmask);
```

**Figure 23:** mmTLS API list