Selective HTTPS Traffic Manipulation At Middleboxes for BYOD Devices

Xing Liu
Indiana University
xl45@indiana.edu

Feng Qian
Indiana University
fengqian@indiana.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

***Abstract*—HTTPS has become a vital component of the WWW ecosystem. However, today's application-layer middleboxes in the cloud are largely "blind" to HTTPS traffic. We propose a novel system infrastructural solution, called** CloudEye**, that allows middleboxes to selectively manipulate HTTPS traffic. A key design philosophy of** CloudEye **is to hide all the complexity from client and server applications (thus being transparent to them) and to have middlebox-related functions managed by a dedicated OS service.** CloudEye **provides control of what information the middlebox can access through new techniques such as HTTPS tags and shadow connections, without changing the TLS/SSL or HTTP protocol.** CloudEye **is secure and easy to use. We implemented its prototype on Linux/Android, and demonstrated its low overhead and rich use cases on off-the-shelf mobile devices and cloud servers.**

I. Introduction

HTTPS is the secure version of HTTP. It consists of HTTP over a TCP connection encrypted by Transport Layer Security (TLS) or Secure Sockets Layer (SSL)[1]. HTTPS traffic is growing at an unprecedented rate. It accounts for 40% of the overall Internet traffic [32], and a recent report estimates its growth to be 40% every six months [1]. This is partly attributed to increased concern about Internet privacy. Also, new web protocols such as HTTP/2 [21] and QUIC [11] use encryption by default or mandatorily.

Meanwhile, middleboxes play a crucial role in the Internet. They are *ubiquitous*: they are widely used in enterprise networks [38] and all major cellular networks [45]. Middleboxes are *diverse*: there are not only network-layer middleboxes such as NAT, but also application-layer middleboxes such as Intrusion Detection System (IDS), web caches, virus scanner, and web page optimizer. Middleboxes are also *useful*: functions such as caching, IDS, URL filtering, and compression [41] are inherently needed to be, or more effective if carried out on a middlebox, as opposed to being performed on endhosts with limited storage and computation power.

The trend of "HTTPS everywhere" facilitates privacy and security for the WWW. It however makes application-layer middleboxes (proxies) difficult or impossible to operate given that they need to inspect or modify the traffic. Currently, the most widely adopted approach of in-network manipulation of HTTPS traffic is to use a man-in-the-middle (MITM) proxy [8], [12][2], which splits an end-to-end TLS session into two sessions. This approach is indeed used in practice. A recent large-scale measurement [34] found that about 1 in 250 TLS connections is proxied (*i.e.,* using MITM), and the vast majority of MITM cases are for legitimate purposes (in particular, used by corporate networks). Such a vanilla MITM approach would enable a middlebox to perform its function over TLS (HTTPS), but it completely breaks the end-to-end security by giving the proxy full control of inspecting and modifying *all* traffic in an HTTPS session.

Privacy-preserving TLS traffic inspection is needed by corporations, governments, law enforcement, *etc.* Recently, the industry (AT&T, Cisco, Google, *etc.*) has proposed various schemes such as Explicit Trusted Proxy [3], TLS Proxy Server Extension [14], and Explicit Proxies for HTTP/2.0 [2]. These proposals to a certain extent improve the interaction between endhosts and MITM proxies (*e.g.,* allowing a client to explicitly authenticate a proxy, as to be detailed in §II), but none has addressed the fundamental limitation of the "all-or-nothing" access paradigm of MITM proxies.

Also recently, researchers proposed a protocol named Multi-context TLS (mcTLS) [33]. It breaks TLS's "all-or-nothing" security model by allowing endpoints to explicitly control what parts of the data in a TLS session can be read or written by middleboxes. Its basic idea is to grant different *encryption contexts i.e.,* symmetric encryption and MAC keys to middleboxes to achieve permission and access control. mcTLS is a complex protocol *requiring changing all entities in the HTTPS ecosystem*: the protocols and their APIs, the client and server applications, and the middleboxes. Moreover, mcTLS is not "plug-and-play". It exposes the complexity of (from developers' perspective) creating encryption contexts, and (from servers' perspective) managing contexts with middleboxes deployed by different ISPs. It even remains unclear how the complex interface of mcTLS should look like. Even the authors of [33] admit that "designing a satisfactory interface atop mcTLS is a project in and of itself".

This paper attempts to address the problem of HTTPS traffic manipulation at middleboxes. This is in particular an issue in the enterprise BYOD (Bring Your Own Device) context. Many companies today allow employees to use their personal devices (*e.g.,* smartphones and laptops) to access

[1]We use "TLS" to refer to TLS and SSL unless otherwise noted.

978-1-5090-6501-1/17/$31.00

[2]We use "middlebox" and "proxy" interchangeably in this paper.

privileged company data and web applications. To ensure security and performance, companies have strong incentives to deploy dedicated middleboxes to perform tasks such as virus detection, compression, and URL filtering [34], and may even make it mandatory that users' traffic is inspected by such middleboxes. Meanwhile, employers also desire to respect employees' privacy and to reduce the workload of their middleboxes by not inspecting everything. We thus propose a novel system infrastructural solution, called CloudEye, that allows middleboxes to selectively manipulate HTTPS traffic without changing client or server applications. CloudEye has the following features.

• *Ready to deploy.* CloudEye is deployed only at client hosts as an operating system (OS) service, as well as on middleboxes. One key philosophy of CloudEye is to hide all the complexity from applications (thus being transparent to them) and have middlebox-related functions managed centralized at the system layer. In addition, CloudEye does *not change anything on the server side*, and leaves the decision to the client-side policy regarding what data should be examined by the middlebox. Moreover, CloudEye *does not change the TLS protocol*, as doing so can be tricky and can lead to new security vulnerabilities (numerous exploited vulnerabilities told us designing and implementing a security protocol right is notoriously difficult). All above features are desirable to corporate network operators as the deployment will be dramatically simplified. Then CloudEye immediately works for all servers that speak TLS.

• *Access control.* Different from enterprise mobility management (EMM [27]) and mobile device management (MDM [6]) that provide access controls on what users can and cannot do *on the device*, CloudEye specifies what information within an HTTPS session a client exposes to the *in-network* middlebox. For example, the client can selectively allow the middlebox to inspect a subset of HTTP transactions within the *same* TLS session; the client can expose only certain information in HTTP requests, such as URL and host names, to the middlebox; the client can also give permission of scanning only requests or only responses (or even part of them) to the middlebox. To achieve the above goals, we introduce novel primitives such as HTTPS tags and shadow HTTPS connections as to be detailed in §III. Note a limitation of CloudEye is it cannot provide access control that is as fine-grained as mcTLS does. But this stems from the inherent tradeoff between the server transparency requirement and the granularity of access control. From a practical point of view, oftentimes the systems with the best tradeoff get deployed eventually. We thus believe CloudEye strikes the right tradeoff.

• *Easy to configure and use.* CloudEye is a practical software framework. Companies can roll out CloudEye using standard feature sets such as EMM that are capable of securely setting up system software (*e.g.,* VPN) on BYOD devices. BYOD users cannot alter the software and its policies without knowing the password set by the employer. IT departments can then pre-configure the middlebox access control policies on employees' BYOD devices as exemplified in Figure 1.

(1) Domains always visible to middleboxes: `*.mycompany.com`.
(2) Domains never visible to middleboxes: `facebook.com`, `twitter.com`, `youtube.com`.
(3) Exposed HTTP header fields: URI (first 32 bytes), `Hostname`, `user-agent`.
(4) Object types allowing middleboxes DPI: `*.exe`, `*.msi`, `*.dmg`, `*.apk`, `*.zip`, `*.tar.gz`.
(5) Other objects types: do not allow middlebox DPI.

Fig. 1: An example access control policy for middlebox.

• *Security and Performance.* CloudEye is in general as secure as existing HTTPS and TLS protocols. The added system support and introduced attack surface will be minimal (§IV). We have implemented a CloudEye prototype on commodity Linux/Android systems. Experiments indicate that CloudEye incurs very small runtime overhead even on low-end mobile devices. We also demonstrate through case studies how real middlebox functions can be easily built on CloudEye (§V).

II. Related Work

Besides mcTLS [33], we describe other related work here. The *de-facto* approach of examining HTTPS traffic is to use a man-in-the-middle (MITM) proxy [8], [12]. The client needs to be configured to trust the in-network proxy's certificate. The MITM approach is easy to set up, and is used by systems such as Meddle [37], Beyond the Radio [44], and APLOMB [39]. However, it breaks the end-to-end security of TLS by giving the proxy full control of inspecting and modifying the traffic that may carry users' sensitive data such as passwords.

TLS traffic manipulation at middleboxes becomes a hot topic due to the increasing popularity of HTTPS. A recent IETF draft [3] introduces an X.509 extension to distinguish an MITM proxy with a regular TLS server. This allows the client to become aware of the proxy and thus to explicitly authenticate it. Cisco also proposes a TLS Proxy Server Extension [14] that allows the client to know (via the proxy) servers' certificate and cipher suite information, so that the client can reject TLS sessions with servers that it does not trust. However, using either extension, the proxy still has full access to *all* traffic over a TLS session. Another proposal from Google [2] allows the client to pass TLS session keys to the proxy in a separate secure channel. The proxy is not MITM, but can (only) use the session key to decrypt the TLS traffic. This approach gives the client more control of what it wants the proxy to see. But similar to [3] and [14], the proxy still can inspect everything within the same TLS session. It also requires modifying the client application.

Recently, researchers propose to perform Deep Packet Inspection (DPI) directly over encrypted TLS traffic [40], by leveraging the concept of searchable encryption [42]. Although the direction is promising, this approach is not very practical due to its computation complexity. For DPI involving thousands of signatures, establishing an HTTPS session may take several minutes.

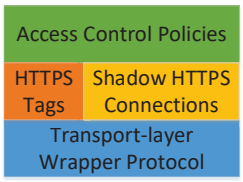

Fig. 2: Components of CloudEye.

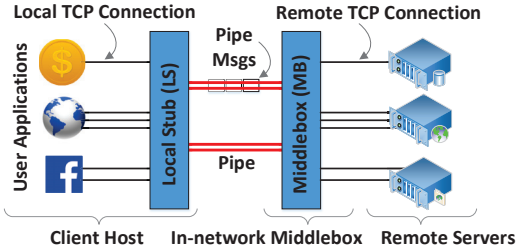

Fig. 3: Multiplexing-based transport-layer infrastructure.

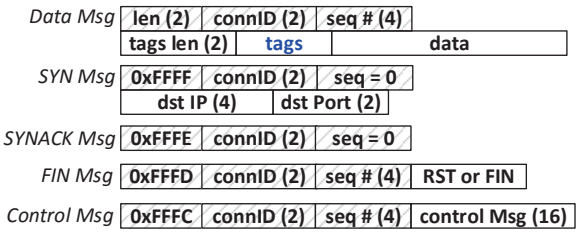

Fig. 4: Pipe message format with the common header shaded.

III. The CloudEye Design

We propose an architecture called CloudEye, a system solution that supports practical HTTPS traffic manipulation at middleboxes. CloudEye provides the following key features.

• CloudEye enables a middlebox to perform many functions on HTTPS traffic *without decrypting it*. To realize this, CloudEye introduces a new primitive called *tags* that are automatically attached to HTTPS traffic based on pre-defined rules. Tags give "least privileges" to middleboxes that can manipulate HTTPS by inspecting tags instead of decrypting the traffic.

• For middlebox functions involving deep packet inspection (DPI) that do require decrypting the traffic, CloudEye provides a novel approach allowing a middlebox to *selectively* examine certain HTTP transactions within a persistent HTTPS connection without servers' support. To realize this, the client host creates a *shadow* HTTPS connection over which the middlebox is a man-in-the-middle, and then selectively redirects HTTP transactions that the middlebox is allowed to examine to the shadow connection. This idea can further be generalized to allow the middlebox to inspect only part of an HTTPS transaction (§VII).

• CloudEye does not require changes to the TLS protocol, and it requires no modification to the client and server applications. To realize this, CloudEye is provided as a client-side OS service that can be transparently applied to any application. Such transparency significantly facilitates the deployment of CloudEye, which can be deployed via standard application and OS update. It also eliminates the burden of developers who do not want to (and should not) tailor their applications based on functions provided by a particular middlebox inside a particular ISP or corporate network. As we will show, the OS service can be easily configured to support all popular middlebox functionalities. At a high level, our solution pushes the control and some network functionality back to the client side. We believe this is a desirable tradeoff and matches the recent trend to push network functionalities (*e.g.,* SDN) back to the client [20], [29].

CloudEye consists of four components shown in Figure 2. At the bottom is a lightweight transport-layer wrapper that transparently intercepts and encapsulates application traffic into customized tunnels (§III-A). Built upon them, CloudEye's key innovations, HTTPS tags (§III-B) and shadow connections (§III-C) are supported. CloudEye provides an interface for specifying the access control policies (§III-E).

A. The Transport-layer Infrastructure

We first describe the transport-layer infrastructure of CloudEye. As depicted in Figure 3, it consists of two key components: a *local stub* (LS) and a *middlebox* (MB). The LS resides on the client BYOD device, and the MB is deployed in the corporate network. Deployed as a system-level service, the LS is the key to realizing the transparency requirement, making CloudEye be able to "plug-and-play" for all applications. The LS and MB split an end-to-end TCP connection into three segments: *(i)* a *local connection* between client applications and the LS, *(ii)* one or more *pipes* between the LS and MB, and *(iii)* a *remote connection* between the MB and the remote server. The pipes serve as a transport-layer wrapper between the LS and MB. Therefore pipes are below the SSL/TLS layer and have nothing to do with encryption.

Pipes allow LS and MB to communicate using a customized protocol, to enable features of HTTPS tags and shadow connections to be described later. There are many ways to realize pipes. For example, one can use existing proxying protocols such as SOCKS5 [30]. In our current prototype, we modified and extended the TM^3 proxying protocol [35] previously developed by us. TM^3 performs transport-layer multiplexing by encapsulating application TCP data (both user payload and control messages such as SYN and FIN) into customized frames called *pipe messages*, which are carried by TCP connections (served as pipes) between the LS and MB. This is conceptually similar to multiplexing performed by some web protocols such as HTTP/2, which multiplexes web objects instead of application TCP connections.

Pipes are bidirectional and *long-lived*, and they are by default shared by all applications via multiplexing. They therefore facilitate centralized traffic management at the LS. The user-side policy determines which applications' traffic will go through CloudEye. The pipe message formats are shown in Figure 4. The Data, SYN, SYNACK, and FIN messages encapsulate user payload, TCP SYN, SYN-ACK, and FIN/RST, respectively. The connID field distinguishes different user TCP connections (since we are doing multiplexing). Since pipe messages created from one TCP connection can be concurrently delivered by multiple pipes in an arbitrary order, each pipe message of the same TCP connection needs a sequence number that increases *per message*. The *Control* message is used by the LS and MB to exchange control-plane information. Most fields in Figure 4 are self-explanatory. The "tags" fields will be described shortly.

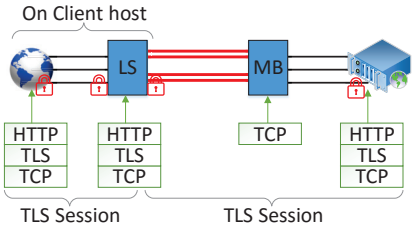

Fig. 5: Protocol stacks at different entities. Lock icons indicate places where encryption/decryption is performed.

B. HTTPS Tags: Manipulating HTTPS Traffic without Decryption at Middlebox

In CloudEye, a very important function of the LS is to inspect HTTPS traffic. Instead of using an in-network MITM proxy that breaks the end-to-end security of HTTPS, we deploy the MITM proxy on the client host. As shown in Figure 5, the LS intercepts and accepts a TLS handshake from a client application. It then establishes a new TLS session with the remote server. Note that, however, the MB is *not* an MITM so it cannot decrypt the TLS traffic. By moving the MITM to the client host, the end-to-end security provided by TLS is still preserved (assuming the LS is not compromised, see §IV for discussions of security aspects of CloudEye). Note that client-side MITM has also been leveraged by certain anti-virus products such as Kaspersky for securing web browsing, while the key innovation of CloudEye is its new primitives for selective and flexible HTTPS traffic manipulation to be described next. Another concern is on-device MITM may incur high overheads due to encryption and decryption. We quantify this in §V and demonstrate this approach is entirely feasible even on low-end mobile devices.

If the MB cannot decrypt TLS traffic, how can it perform middlebox functions? Our key observation is, performing many middlebox functions does not require decrypting the traffic. Instead, the MB only needs to know certain properties of the traffic. For example, in order to prioritize the delivery of business over non-business traffic, the MB only needs to know whether the traffic is business-related or not – just one-bit information. We found that many middlebox functions can be performed in a similar way, as to be shown in §VI.

CloudEye uses *tags* to deliver characteristics of HTTPS traffic to MB. As shown in Figure 4, each pipe message can carry tags of up to 64KB, although we expect for most tags, their sizes will be far smaller than that. As an example, consider a middlebox that performs URL filtering. The client can put a URL into a tag, which is examined by the MB to determine whether the URL points to a malicious page. Note multiple tags can be contained in a single pipe message, and it is entirely normal that a message does not contain a tag. Sensitive tags such as URLs will be encrypted between LS and MB, as to be discussed in §IV. Since the traffic volume of tags is very small, the encryption overhead is expected to be negligible.

A question to be addressed by CloudEye is *who should generate the tags, and how*. One option is to let applications directly pass tags to the LS. Doing so, however, requires modifying apps' source code. This also makes it difficult to precisely attach a tag to a particular pipe message, which is not visible to the application layer. To overcome these limitations, in CloudEye, tags are automatically generated by the LS. Specifically, the LS examines decrypted TLS traffic (recall that the LS is MITM so it has to perform decryption), and matches them against predefined rules (translated from user-specified policies, see §III-E). Consider the two examples below.

Rule 1: `if (http.request.host contains "mycompany") t1←1;`

Rule 2: `if (http.request) t2←http.host + "/" + http.uri;`

`t1` and `t2` are tag names. In Rule 1, if the data of a pipe message belongs to an HTTP request header and the `host` field contains "mycompany", then `t1`, whose value is set to 1, is added to the tag field. In Rule 2, if the data belongs to an HTTP request header, then its requested URL (hostname+URI) is put into the tag field of `t2`. Also note if an HTTP header spans across multiple pipe messages, only the first message will be tagged. The mappings between tag names and their semantics can be either statically configured or dynamically negotiated between the LS and MB by exchanging the *Control* messages when the pipes are set up.

Clearly, the LS can only tag uplink traffic containing HTTP requests (this leads to very small overhead of tag generation as most traffic is downlink). We however make two observations here. First, HTTP requests already contain vital information (URL, host name, cookie, user-agent, user submitted data, *etc.*) that can be leveraged by the MB to perform a wide range of middlebox functions. For example, URL is used by many commercial systems such as VirusTotal [16] and Google Safe Browsing API [4] to detect malicious web content. In CloudEye, a URL can be inserted into a tag. If the URL hits the blacklist, the MB will reset the connection or send a warning back to the client.

Our second observation is, by leveraging HTTP's request/response traffic patterns, response traffic associated with a request can be identified without decrypting the traffic (new HTTP protocols such as HTTP/2 make this slightly more complex due to multiplexing, see §III-D). The MB can then manipulate the (encrypted) response traffic at the transport layer. For example, for traffic prioritization, the tag is the HTTP transaction's priority, based on which the MB will prioritize the corresponding response traffic without decrypting it. We give more examples in §VI.

C. Shadow Connection: Selectively Decrypting HTTPS Traffic

As described in §III-B, the MB can perform many functions by examining tags attached to HTTP requests. The MB can also associate tagged requests with responses, and manipulate HTTP response traffic at the transport layer. However, for middlebox functions using DPI, such as virus scan, compression, and web page optimization, the MB has to perform decryption.

CloudEye provides a way to allow MB to selectively decrypt HTTPS transactions. For example, the client can make the MB

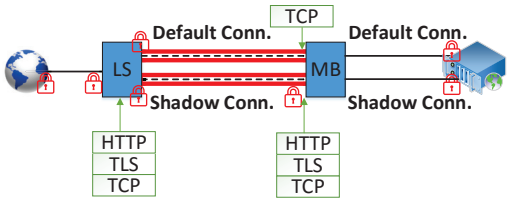

Fig. 6: The default connection and shadow connection.

be able to only decrypt `.js` and `.css` files and compress them. Our proposed solution is shown in Figure 6. The basic idea is to let the MB serve as *both* an in-network MITM proxy and a transport-layer proxy. The LS transparently redirects a subset of transactions (per the client-side policy) of an application HTTPS session to the in-network MITM proxy. These HTTPS transactions can therefore be decrypted and manipulated by the MB.

As described in §III-A, the LS intercepts HTTPS sessions from applications. For each incoming HTTPS session, the LS establishes a corresponding connection (encapsulated in pipes) to the MB, which then relays it to the server, as denoted by "*default* conn." in Figure 6. Since HTTP/1.1 and HTTP/2 use persistent connection, each local TCP connection (and therefore its HTTPS session) usually carries multiple HTTP transactions. The new thing here is, as an MITM, the LS can also transparently establish a *shadow* connection (also encapsulated in pipes) with the MB, when needed. The shadow connection is relayed to the server by the MB as well.

For a default connection, the MB acts as a transport-layer proxy, so the MB can only manipulate the traffic at the transport layer using tags as described in §III-B. For a shadow connection, however, the MB acts as a TLS MITM proxy, and can therefore decrypt and modify the HTTPS transactions. By default, only the default connection is used, but the client side can set up policies (similar to the tag policies) specifying which HTTP transactions are directed to the shadow connection as exemplified below. Note a shadow connection is established in a "lazy" manner, *i.e.,* only when a rule is triggered, to save resources.

```
if (http.request.uri endwith ".exe")
send_to_shadow_connection;
```

As described above, shadow connections strategically perform what we call "HTTP session splitting" *i.e.,* directing a subset of objects within an HTTP session to a different (shadow) connection. This can result in a discrepancy between the client's and server's views: the client application thinks it only issues one TCP connection, while the server thinks there are two. This does *not* affect the correctness of our scheme due to the *statelessness* nature of HTTP(S) transactions. In fact, HTTP session splitting and merging are already implicitly used by today's off-the-shelf HTTP(S) proxies, which may create more (or less) connections to the server (compared to those created by the client) based on their own connection management algorithms when forwarding the traffic. Note that HTTP session splitting does not affect the request-response causality whose logic occurs at the client/server applications: the reception of Request (Response) X triggers the transmission of Response (Request) Y. Also note that the MB serves as both an MITM proxy (for shadow connections) and a TCP proxy (for default connections). Therefore, from the server's perspective, both types of connections will have the same source IP address so the server believes they are indeed from the same client.

D. Handling HTTP/2 Transactions

So far we assume CloudEye runs below HTTP/1.1, the state-of-the-art HTTP protocol. Recently, new web protocols such as HTTP/2 [21] and QUIC [11] have been proposed. In particular, as the next-generation web protocol, HTTP/2 has been standardized in 2015. HTTP/2 encapsulates HTTP transactions into *streams* each carrying one HTTP transaction, and multiple streams can be multiplexed over one TCP connection. In HTTP/2, TLS-based encryption is enabled by default. Due to multiplexing, HTTP/2 allows multiple concurrent outstanding requests over one TCP connection, causing possibly out-of-order responses.

We first note that shadow connections (§III-C) work for both HTTP/1.1 and HTTP/2, because they can be decrypted by the MB. However, using tags (§III-B) in HTTP/2 may encounter difficulties. Let us consider three scenarios. First, for transactions without a tag, since the MB just blindly forwards them, they can still be multiplexed over the same connection. Second, for all transactions with the *same* tag, they usually can also be multiplexed. This is because the MB applies the same function (*e.g.,* traffic blocking and prioritization) on them so there is no need to separate them. Third, consider transactions with *different* tags that are originally multiplexed in a connection between the application and the LS. Due to multiplexing, these transactions may be intertwined together (sometimes even in the same TLS record), making it impossible to precisely associate a request with its response without decrypting the traffic, a basic requirement for using tags. We propose a simple solution to handle this case. The LS transparently creates additional HTTP/2 connections to the server (this is allowed by the HTTP/2 specification [21]). These additional connections are replicas of the default connection shown in Figure 6 in that the MB cannot decrypt them. The LS then re-distribute the (originally multiplexed) transactions with different tags to these connections in a way that on each connection, only *one* outstanding request per-unique-tag is allowed. In this way, we essentially "emulate" HTTP/1.1's traffic pattern on HTTP/2 *i.e.,* using concurrent connections each carrying HTTP/2 transactions *sequentially* for a particular tag. This also works for other multiplexing protocols such as QUIC.

The above scheme also strategically leverages HTTP session splitting described in §III-C[3]. It essentially makes HTTP/2 behave similarly to HTTP/1.1 for transactions with different tags. A question here is, does doing so eliminate the performance benefits brought by HTTP/2's multiplexing? No! Recall

[3]An HTTP/2 proxy also implicitly uses HTTP session splitting: it blindly forwards traffic between a single HTTP/2 connection (client–proxy) and multiple HTTP/1.1 connections (proxy–server).

that CloudEye's transport architecture (§III-A) has already performed multiplexing at the transport layer between the LS and MB, *i.e.,* the last mile that is usually the performance bottleneck. This eliminates the need for multiplexing at the application layer. In fact, our prior study [35] has shown that strategic multiplexing at the transport layer actually outperforms the multiplexing scheme of HTTP/2 (SPDY). This justifies why we choose a multiplexing-based transport-layer infrastructure. Note that HTTP/2's other features such as binary frames and header compression are still retained on the replica connections created by the LS.

E. Middlebox Access Control Policies

Deployed as an OS service, CloudEye is transparent to the application layer. Its efficacy thus relies on the client-side access control policies. We expect enterprise IT departments to distribute several default policies for users to use. Some basic policies such as URL filtering may be enforced by the IT whereas some policies such as content-based scanning (DPI) can be optionally opted-in by employees. A policy may look like Figure 1 (written in the natural language). The policy is then automatically translated into low-level rules of tags and shadow connections as exemplified in §III-B and §III-C. The policy of CloudEye can also change depending on whether the BYOD device is connected to the corporate network (either physically or via a VPN).

IV. Security Analysis

CloudEye introduces new opportunities for deploying more intelligent services at middleboxes by exposing to them tags and/or a chosen subset of the HTTPS contents. To facilitate the discussion CloudEye's security aspect, let us first consider a VPN client, which is now a standard feature in any modern operating system. A VPN client is already a (Layer 2/3) MITM in that it is responsible for intercepting any outgoing traffic, encrypting them using the public key of the VPN server, and decrypting any incoming traffic from the VPN server. We thus do not consider intercepting traffic a fundamental security hazard as it is a standard feature in modern OSes and is already widely used by today's BYOD devices. One potential issue is CloudEye may introduce additional attack surface and privacy concerns. However, we argue that compared to a VPN client, the proposed LS has marginally more complexity: the logic of tags and shadow connections are simple and easy to reason, and the information carried by tags is no more sensitive compared to the regular HTTPS traffic. Also note that the pipes are transport-layer data channels and thus have no visibility of the sensitive data.

Despite the above, we do want to discuss any new attacks that the full-fledged MITM may cause. First, an attacker may compromise the MITM feature to maliciously intercept all traffic locally. This requires an attacker to compromise the LS, a newly introduced system service. This is indeed possible, but is nothing fundamental to our solution itself, as a VPN client can introduce similar security vulnerabilities as well. Even a conventional TLS library may be hijacked by an adversary [24]. Second, private keys used by the LS are stored locally on the device. An attacker can potentially extract the key and use it to perform network-level MITM attacks, as the corresponding public key and certificate are trusted by the device. While possible, this is as difficult as compromising the LS. We can also store the private key on secure hardware (*e.g.,* TPM [18]) to minimize the chance of having it be stolen. Third, managing the private keys can be tricky. If different devices all share the same private key, leaking it on one device will affect other devices, a common pitfall in doing OS-level man-in-the-middle as reported by [7]. In contrast, our solution requires a *unique* root certificate (and private key) for any LS on a new BYOD device. Therefore, the leak of its private key can affect only a single device.

Even though unlikely, it is theoretically possible that an employee (*i.e.,* a malignant BYOD user) intentionally tries to evade the monitoring of MB. Fortunately, there are already OS features such as Android enterprise features [26] that are used to support the Enterprise Mobility Management (EMM [27]) and Mobile Device Management (MDM) functionalities, where a user cannot alter the policy without knowing the password set by the employer.

The functions at the MB are inherited mostly from those at existing middleboxes such as compression, intrusion detection, and URL filtering. There could be concerns where the LS may collude with the MB and route all traffic through the shadow connection to make it visible to the MB. To prevent this issue, one can make the LS implementation open-sourced and have it verified publicly to ensure that there is no backdoor. It is further possible to remotely attest [31] the LS to ensure that there is no tampering of its code. We consider such malicious intent to be highly unlikely as once detected, the reputation of the vendors and the enterprise ISP will be hammered.

Assuming neither the LS nor MB is compromised, the confidentiality and integrity guarantee of CloudEye would be the same as they are in standard HTTPS, as CloudEye in no way reduces the strength of the underlying cryptographic system (recall CloudEye itself does not modify TLS or HTTPS). One potential issue arises from the tags inserted into pipe messages. Even though tags are always encrypted, their sizes can still leak information. This itself leaks only a small amount of information and can be mitigated through padding.

V. Implementation & Performance Evaluation

We have built a prototype of CloudEye on Linux and Android. We developed the transport-layer wrapper protocol (§III-A) by realizing and extending the TM^3 proxying protocol. Based on that, we then implemented a TLS protocol stack using OpenSSL's cryptographic functions as building blocks. We also implemented tags (§III-B) and shadow connection support (§III-C), as well as a dynamic certificate generator. Our implementation efforts involve about 8,000 lines of C/C++ code. Similar to existing TLS libraries, all of CloudEye's components run at user level except for a lightweight kernel module that intercepts application connections and forwards them to the LS. This component can be replaced with a custom

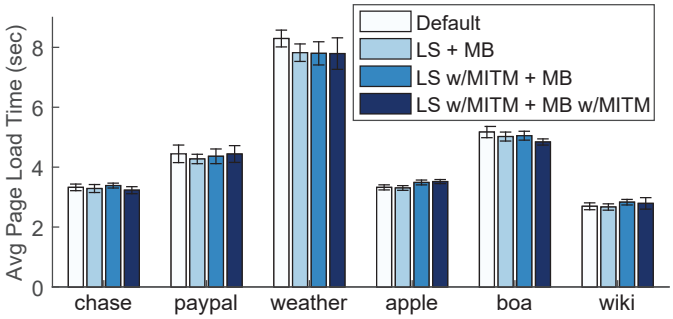

Fig. 7: Page load time (PLT) across six landing pages. Tested on Chrome browser on SGS3 smartphone.

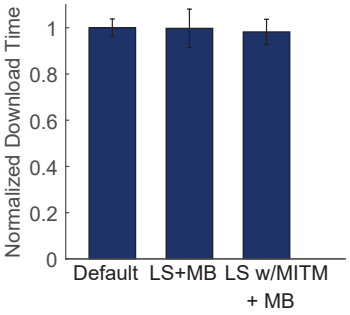

Fig. 8: Download time of a 32MB file, under three configurations.

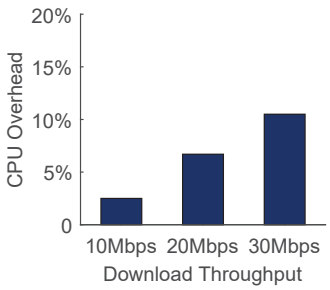

Fig. 9: Additional CPU overhead due to MITM at LS side.

local VPN module which we leave as future work. There are several limitations of our currently prototype. It only supports TLS/1.0 [23] with a limited number of cipher suites using the RSA handshake. Also, it supports only HTTP/1.1. We are working on adding HTTP/2 support to CloudEye. Furthermore, the access control policies are currently hardcoded. Designing and implementing a full-fledged policy language demonstrated in §III-E is our planned work.

A. Performance Evaluation

Our CloudEye prototype allows us to answer an important question: what is CloudEye's performance and incurred overhead on commodity devices? The overhead mostly comes from the MITM proxy at the LS, which performs additional TLS handshake, encryption, and decryption. To understand the "upper bound" of CloudEye's performance impact, we have deployed the LS on an off-the-shelf low-end smartphone (Samsung Galaxy S III running Android 4.0.4). The MB runs on a commodity Ubuntu server with Linux kernel version 3.19. The phone and the MB are connected with high-speed Wi-Fi (>30Mbps TCP throughput with <3ms RTT). The MB and servers are either co-located (when using our own servers) or connected over wired networks that are usually well-provisioned (when using real servers). This setting ensures that the local computation dominates the performance overhead. In all experiments below, every uplink pipe message carries a one-byte dummy tag unless otherwise noted.

Web browsing. We use the Chrome browser on Android to load six popular websites' landing pages (mobile version, cold cache, all using HTTPS), whose sizes range from 200KB to 1MB. The path quality (bandwidth, latency, and loss) between the MB and the web servers are measured to be good. Figure 7 measures each page's page load time (PLT) across 20 measurements. For each website, we consider four configurations. "Default" is the baseline where CloudEye is not used. "LS+MB" corresponds to the scenario where CloudEye is only performing transport-layer forwarding *i.e.,* no TLS encryption/decryption is performed. "LS w/MITM + MB" is the HTTPS tag case where the MITM is enabled only at the LS. "LS w/MITM + MB w/MITM" represents the configuration of shadow connections where both the LS and MB are performing MITM and all traffic is forwarded through the shadow connection. As shown in Figure 7, all four configurations yield very similar PLT. Enabling MITM on the LS slightly increases the PLT by no more than 6% due to the additional crypto-heavy TLS handshakes. Also using LS slightly outperforms the default scenario, because the multiplexing scheme employed by LS is known to improve the web performance [35]. To ensure the above results are not due to the MB–server link becoming the bottleneck, we also repeat the experiments by replicating the pages and hosting them on our own web server that is co-located with the MB (using the Google Web Page Replay tool [5]). We observe qualitatively similar results.

Large file download. We measure the download time for a 32MB file hosted on an Apache web server co-located with the MB. As shown in Figure 8, the three configurations (no CloudEye, transport-layer forwarding only, and LS-side MITM) show no qualitative difference of download time, with the average TCP throughput being at around 35Mbps.

CPU utilization. We measure the additional CPU overhead incurred by MITM at the LS (the overhead of transport-layer forwarding is negligible) at different download throughput controlled by Dummynet [13]. As shown in Figure 9, the CPU overhead is non-trivial but small: 2.5%, 6.7%, and 10.5% when downloading at 10, 20, and 30Mbps, respectively. They also translate into very small energy footprint, as the energy incurred by 10% of CPU utilization is much lower than that incurred by the wireless radio at a high data transmission rate [22]. Note CloudEye does not incur any computation overhead when there is no network traffic.

Protocol overhead. We measured the bandwidth overhead caused by the transport-layer wrapper protocol by running more than 10 diverse Android apps over CloudEye. Since the protocol headers are very small (Figure 4), their incurred protocol overhead is less than 1%. Recall we assume every uplink pipe message contains a 1-byte dummy tag. Clearly, the protocol overhead will increase when the tags carry more information. However, we expect the overhead still to be small because tags are only applied to uplink traffic (HTTP requests) whose volume is much smaller than downlink traffic. The protocol overhead of shadow connections is even lower.

VI. Case Studies of Middlebox Functions

CloudEye enables a wide range of middlebox functions. Some of them only need tags (*e.g.,* traffic prioritization, URL-based malware detection, traffic shaping, multipath

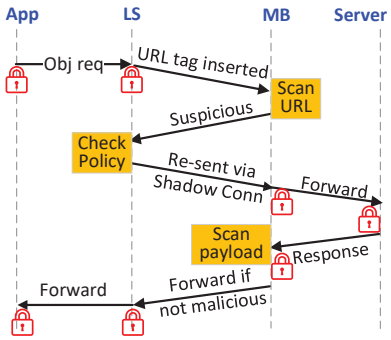

Fig. 10: Selective Inspection by IDS.

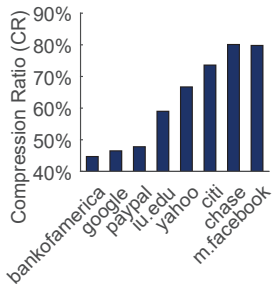

Fig. 11: Compression Ratio for browsing traffic across 8 HTTPS sites.

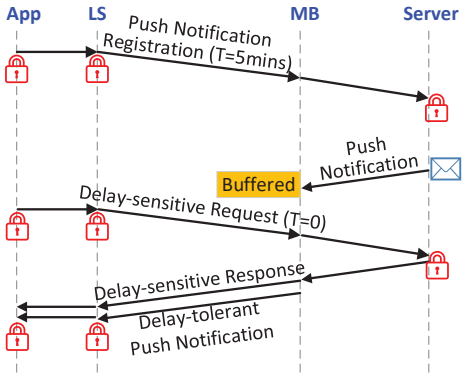

Fig. 12: Delay-tolerant traffic shaping.

transport [10]) while some need shadow connections (*e.g.,* compression [41], intrusion detection, DPI-based virus scan, web page optimizer [45]) as the MB has to selectively decrypt the traffic. Next we conduct three case studies.

A. Selective Inspection by IDS

Consider an Intrusion Detection System (IDS) that performs malware detection over HTTPS in an enterprise network. One obvious solution is to run IDS at an MITM proxy. This however results in poor privacy where user cookies and even passwords will be fully disclosed to the MB unnecessarily. To protect users' privacy, we leverage CloudEye's tags and shadow connections to allow an in-network MB to selectively perform intrusion detection over HTTPS. To strike a balance among security, privacy, and performance, we designed an adaptive scheme of performing *two-step* scan: all requests (except those on the whitelist) will undergo a lightweight URL-based scan, and suspicious contents will be deeply scanned by exposing their payload if allowed by access control policies. As shown in Figure 10, the request URL is inserted as a tag so the MB can consult the reputation database on the likelihood of the content from the URL being malicious. Many reputation-based systems are discussed in the literature [36], [28] and are offered as commercial services [4], [16]. We used the VirusTotal public API [16] in our implementation. Doing such URL-based "pre-filtering" helps reduce the workload of the IDS service. If a URL is deemed to be suspicious by the MB, a control message will be sent to the LS indicating the need to decrypt the response to inspect the object payload. If the access control policy allows, the LS will resend the request through the shadow connection so the MB will be able to see both the request and response in cleartext. Finally, the MB will buffer the response and release it only after determining that the object is not malicious based on the DPI-based scan. During the above process, the MB can decrypt only the relevant information (*i.e.,* the suspicious transaction and nothing else) per the client policy, satisfying the goal of exposing the least amount of information. In addition, the whole process is transparent to both the client application and the server. Note although the LS sends the same request twice (one through the default and the other through the shadow connection), the first request is intercepted and discarded by the MB so the server only receives the second request.

B. Selective Compression

We implemented a middlebox function that compresses web contents selectively. Based on access control policies, the LS uses the MB as either an MITM proxy (for HTTP sessions with objects to be compressed by the MB) or a TCP proxy (for HTTP sessions whose objects are not allowed to be inspected by the MB). If an HTTP session contains both types of objects, shadow connections will be used to separate them.

A design decision we need to make is to select the compression algorithm. A traditional HTTP compression middlebox (*i.e.,* an HTTP(S) proxy) performs object-level compression. Doing so however cannot capture the cross-object redundancy such as the same non-cacheable object being downloaded multiple times. In CloudEye, pipes, which cover the last mile that is usually the performance bottleneck, provides more flexibility in the compression algorithm selection. Here we apply cache-based redundancy elimination (RE) [43]. To compress downlink traffic, the MB-side RE algorithm replaces byte sequences that have appeared in previously observed packets (pipe messages in our case) with pointers. The byte sequence to pointer mappings are stored in *fingerprint caches* that are synchronized between the MB and LS. Then the LS-side RE restores the original byte sequences using the fingerprint cache. Note that RE is a general network-layer packet stream compression scheme that can be applied to any traffic. In our implementation, we direct to-be-compressed traffic to a dedicated pipe and apply the MODP RE algorithm [43][4].

We use MODP-enabled CloudEye to visit eight popular HTTPS websites (mobile version, cold cache) to measure their *compression ratios* (CR), defined as the volume of compressed traffic divided by the volume of traffic sent by the original servers. Note encryption/decryption has little impact on the traffic volume. For each website, we obtained three traces (each about 5 minutes) by visiting it using the built-in browser of the SGS3 phone as normal users (*e.g.,* checking credit

[4]Parameters: cache size = 128K, sampling rate = 1:8. Adjusting these does not qualitatively change the results.

card deals and logging onto the bank account for a bank website). The reported CR is the average across the three traces. The results are shown in Figure 11. Surprisingly, the CR ranges from as low as 45% to 88%, indicating significant traffic redundancy for many popular mobile HTTPS websites. This is attributed to several reasons: (1) the HTTP object-level compression is under-utilized; (2) many cacheable objects are configured to be non-cacheable or to have short expiration time, leading to cross-object redundancy; (3) the session-level compression supported by TLS is rarely used. Deploying the compression service can effectively reduce the transmission time when the last-mile wireless link has poor quality.

C. Energy-efficient Traffic Shaping

HTTP(S) has become a general-purpose protocol used in many scenarios such as smartphone apps and push notifications. Here we consider delaying delay-tolerant HTTPS responses and transferring them in *bundles*. It is well known that this helps reduce the mobile device energy consumption in wireless networks. For example, in cellular networks, due to their radio resource management policy, the most efficient way to transfer data is to group multiple transfers into one single burst (or as few bursts as possible). Doing so saves the radio energy, as fewer bursts reduce frequent radio on/off switches and the total "tail time", a timeout for shutting down the radio interface after data transmission/reception [19]. Also, batching non-urgent notifications makes smartphones less distracting during working hours, as it is known that smartphones may significantly reduce the workplace productivity [17]. The employer can provide some incentives to let employees sign up such a service.

CloudEye allows the MB to perform the above traffic shaping on encrypted data. To illustrate this, we developed a middlebox function shown in Figure 12. The idea is to batch multiple delay-tolerant transfers under the constraint of meeting user-specified deadlines, and to piggyback delay-tolerant transfers with delay-sensitive transfers. Specifically, the LS can attach a timespan T as a tag to an HTTPS request. T specifies the maximum duration by which the response (*e.g.,* an email notification or weather update) can be delayed. A request without a tag is assumed to have T=0. Upon the reception of a request with tag T at time t_0, the MB will immediately send the request to the server, but will hold off sending the (encrypted) response to the client until either the deadline ($T + t_0$) is reached, or the deadline of another transfer is reached so the response can be piggybacked (*i.e.,* delivered together) with that transfer. In the example illustrated in Figure 12, the client first sends an HTTPS request with tag T=5 minutes to register a push notification. When an (encrypted) notification arrives, it is first buffered at the MB. Later, it is piggybacked with a delay-sensitive response (carried by another TCP connection) with T=0 that arrives within 5 minutes. The above middlebox function and its tag can further be modified to allow multiple delay-tolerant transfers with different T to be delivered over the same connection.

We deploy the above middlebox function on an Amazon EC2 server (as the MB) and an SGS3 smartphone. The phone is hooked to a Monsoon power monitor [9] measuring the energy consumption in two scenarios: (1) the scenario in Figure 12, and (2) an unoptimized scenario where the push transfer and the delay-sensitive transfer are delivered separately. Both transfers (less than 1KB) are delivered over a commercial U.S. cellular carrier under normal signal strength. The overall device energy consumption of the above two scenarios are 830 uAh and 1737 uAh, respectively (589 uAh and 1227 uAh respectively for cellular radio energy). The results imply the effectiveness of the traffic shaping approach.

VII. Discussions

Access Control at a Finer Granularity. Despite allowing selective HTTPS traffic manipulation at middleboxes, CloudEye does not provide access control that is as fine-grained as mcTLS does. But as described in §I, there is an inherent tradeoff between the server transparency requirement and the granularity of access control. Nevertheless, we do ask ourselves the following question: without modifying servers, can we achieve even finer-grained access control than what shadow connections currently offer (selectively examining HTTP transactions within an HTTPS session)? The answer is yes. Below lists some of the possibilities.

- Tags can be used to expose small portions of an HTTP request, such as URL. We already discussed this in §III-B.
- After a default TLS session is established, the LS can give the TLS session key to the MB (through an encrypted tag). Since uplink and downlink streams use different session keys, giving only the uplink (downlink) session key to the MB allows the MB to only inspect HTTPS requests (responses).
- In the above method, when the LS does not want MB to inspect the TLS traffic from a certain point on, the LS can issue a *renegotiation request* (a standard feature supported by TLS) to renegotiate new session keys. This allows the MB to selectively inspect, for example, certain HTTP responses but not their requests.

We realize that the above approaches may incur some overheads, such as the delay caused by TLS renegotiation. We are studying and evaluating them in our on-going work.

Client Authentication and HPKP. Client authentication allows the server to authenticate a client during a TLS handshake. The client-side private key and its associated certificate can be either static or dynamically generated [15]. HTTP Public Key Pinning (HPKP) provides a mechanism allowing a client to only accept a list of certificates "pinned" by the server [25]. CloudEye's shadow connections do not support client-side authentication or HPKP unless the client gives its private key to the MB. Note this is a limitation of *any* MITM approach. Also note these two mechanisms are very rarely used in practice and not supported by all browsers.

Connection Management Overhead. Recall in §III-C that shadow connections (using the MB as MITM) are created on demand by the LS. This works fine when the MB inspects a small fraction of objects. But if the MB needs to inspect

most objects, then a lot of shadow connections will be created, leading to high connection management overhead at the server side. In this case, we can swap the meanings of "shadow" and "default" connections (*i.e.,* by default the LS creates MITM connections and forks non-MITM connections on demand). The worst case happens when *every* connection contains *some* (but not all) objects to be inspected by the MB so for each original connection, the LS needs to create two connections anyway. But this is unlikely to happen in practice.

VIII. Concluding Remarks

Enterprise network operators need ready-to-deploy solutions to selectively manipulate HTTPS traffic at their middleboxes. CloudEye addresses this challenge by offloading an essential part of access control and middlebox function management to endhosts as OS services, and by introducing new primitives such as HTTPS tags and shadow connections to allow flexible access control. Through CloudEye, enterprise IT can explicitly control what the middlebox can see by defining high-level access control policies. We believe CloudEye strikes the right tradeoff among the access control granularity, transparency, usability, and performance, while maintaining the security provided by HTTPS.

Acknowledgment

We thank our anonymous reviewers for their comments. This work was supported in part by NSF Award #1618898.

References

[1] Encrypted traffic grows 40% post Edward Snowden NSA leak. http://www.sinefa.com/blog/encrypted-traffic-grows-post-edward-snowden-nsa-leak.

[2] Explicit Proxies for HTTP/2.0 draft-rpeon-httpbis-exproxy-00. https://tools.ietf.org/html/draft-rpeon-httpbis-exproxy-00.

[3] Explicit Trusted Proxy in HTTP/2.0 draft-loreto-httpbis-trusted-proxy20-01. https://tools.ietf.org/html/draft-loreto-httpbis-trusted-proxy20-01.

[4] Google Safe Browsing. https://developers.google.com/safe-browsing/.

[5] Google Web Page Replay Tool. https://github.com/chromium/web-page-replay/.

[6] IBM MaaS360 Mobile Device Management. https://www.ibm.com/security/maas360/mobile-device-management/.

[7] Lenovo Is Breaking HTTPS Security on its Recent Laptops. https://www.eff.org/deeplinks/2015/02/further-evidence-lenovo-breaking-https-security-its-laptops.

[8] mitmproxy: an SSL-capable man-in-the-middle proxy. http://mitmproxy.org/.

[9] Monsoon Power Monitor. http://www.msoon.com/.

[10] MPTCP proxy mechanisms. https://tools.ietf.org/html/draft-wei-mptcp-proxy-mechanism-02.

[11] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic.

[12] SSL MITM Proxy by Stanford Crypto Group. https://crypto.stanford.edu/ssl-mitm/.

[13] The dummynet project. http://info.iet.unipi.it/ luigi/dummynet/.

[14] TLS Proxy Server Extension draft-mcgrew-tls-proxy-server-01. https://tools.ietf.org/html/draft-mcgrew-tls-proxy-server-01.

[15] Transport Layer Security (TLS) Channel IDs draft-balfanz-tls-channelid-01. https://tools.ietf.org/html/draft-balfanz-tls-channelid-01.

[16] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

[17] Your smartphone is making you a workplace slacker. https://www.cbsnews.com/news/your-smartphone-is-making-you-a-workplace-slacker/.

[18] N. Aaraj, A. Raghunathan, and N. K. Jha. Analysis and Design of a Hardware/Software Trusted Platform Module for Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 2009.

[19] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[20] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-host Network Functions. In *SIGCOMM*, 2015.

[21] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[22] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. In *SIGMETRICS*, 2015.

[23] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, 1999.

[24] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The Matter of Heartbleed. In *IMC*, 2014.

[25] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469, 2015.

[26] Google. Android enterprise feature list. https://developers.google.com/android/work/requirements/features.

[27] Google. Enterprise Solution Sets for Android Management. https://developers.google.com/android/work/requirements.

[28] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting spammers with snare: Spatio-temporal network-level automatic reputation engine. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[29] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. In *NDSS*, 2016.

[30] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, 1996.

[31] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[32] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafo, K. Papagiannaki, and P. Steenkiste. The Cost of the "S" in HTTPS. In *CoNEXT*, 2014.

[33] D. Naylor, K. Schompy, M. Varvelloz, I. Leontiadisz, J. Blackburnz, D. Lopezz, K. Papagiannakiz, P. R. Rodriguezz, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *SIGCOMM*, 2015.

[34] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. TLS Proxies: Friend or Foe? In *IMC*, 2016.

[35] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck. TM3: Flexible Transport-layer Multi-pipe Multiplexing Middlebox Without Head-of-line Blocking. In *CoNEXT*, 2015.

[36] Z. Qian, Z. M. Mao, Y. Xie, and F. Yu. On Network-level Clusters for Spam Detection. In *NDSS*, 2010.

[37] A. Rao, J. Sherry, A. Legaut, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: Middleboxes for Increased Transparency and Control of Mobile Traffic. In *CoNEXT Student Workshop*, 2012.

[38] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[39] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing As a Cloud Service. In *SIGCOMM*, 2012.

[40] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *SIGCOMM*, 2015.

[41] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: NetworkAware Compaction for Accelerating Mobile Web Transfers. In *Mobicom*, 2015.

[42] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proc. of IEEE Security & Privacy*, 2000.

[43] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *SIGCOMM*, 2000.

[44] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson. Beyond the Radio: Illuminating the Higher Layers of Mobile Networks. In *Mobisys*, 2015.

[45] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *PAM*, 2015.