The Case for Session Sharing: Relieving Clients from TLS Handshake Overheads

Jens Hiller*, Martin Henze§, Torsten Zimmermann*, Oliver Hohlfeld‡, Klaus Wehrle*
*Communication and Distributed Systems, RWTH Aachen University, Aachen, Germany
Email: {hiller, zimmermann, wehrle}@comsys.rwth-aachen.de
§Cyber Analysis & Defense, Fraunhofer FKIE, Bonn-Bad Godesberg, Germany
Email: martin.henze@fkie.fraunhofer.de
‡Computer Networks and Communication Systems, Brandenburg University of Technology, Cottbus, Germany
Email: oliver.hohlfeld@b-tu.de

***Abstract*—In recent years, the amount of traffic protected with Transport Layer Security (TLS) has significantly increased and new protocols such as HTTP/2 and QUIC further foster this emerging trend. However, protecting traffic with TLS has significant impacts on network entities. While the restrictions for middleboxes have been extensively studied, addressing the impact of TLS on clients and servers has been mostly neglected so far. Especially mobile clients in emerging 5G and IoT deployments suffer from significantly increased latency, traffic, and energy overheads when protecting traffic with TLS. In this paper, we address this emerging topic by thoroughly analyzing the impact of TLS on clients and servers and derive opportunities for significantly decreasing latency of TLS communication and downsizing TLS management traffic, thereby also reducing TLS-induced server load. We propose a protocol compatible redesign of TLS session management to use these opportunities and showcase their potential based on mobile device traffic and mobile web-browsing traces. These show promising potentials for latency improvements by up to 25.8% and energy savings of up to 26.3%.**

***Index Terms*—TLS Performance, Efficient Secure Communication, TLS Session Resumption, Network Security, Security and Privacy, Mobile Networking, 5G and IoT**

I. Introduction

The nature of Internet traffic is currently shifting towards a regime where encryption is widely applied at the transport layer for many classes of traffic to counter the threat of pervasive monitoring [7]. This shift is reflected by the drastic increase of HTTPS traffic in recent years. In the core (IXP), HTTPS traffic was reported to increase by a factor of 6 from 2011 to 2013 [26]. At the edge (ISP), HTTPS traffic was reported to increase by a factor of 2 and accounts for 44.3% of the connections in 2014 [22]. Browser telemetry data shows a further grow from about 40% to around 60% from 2015 to 2017 [8]. This rise of TLS traffic will continue, e.g., strengthened by efforts in enforcing TLS by all HTTP/2 capable browsers [24], [38] and the rise of new transport protocols such as QUIC which use TLS by default [16]. Likewise, with the advent of 5G as the next generation wireless access technology, more and more *mobile* clients will use TLS. Furthermore, already the steadily growing Internet of Things yields millions of new devices that employ TLS to secure communication [17].

While the rise of TLS provides confidentiality and authenticity to a large class of applications, it also fundamentally challenges network entities. These challenges have been widely studied for on-path middleboxes that have to cope with the fact that packet payload now becomes inaccessible to network functions along the data path such as intrusion detection, traffic filtering, image compression, or content caches [10], [23], [29], [37]. In contrast, addressing the impact of TLS on clients and servers has been mostly neglected so far. Especially for mobile clients, the prevalence of TLS encrypted traffic involves latency, traffic, and energy overheads.

Current TLS architectures are not fully optimized to reduce these overheads for clients and servers and thus provide yet unused optimization potential. That is, the TLS-induced traffic overhead and processing overheads can be reduced, end-to-end latency improved, and resulting energy consumption decreased. One example dimension comprises the fact that many mobile applications often establish separate connections to the same services, each involving the establishment of separate TLS sessions. Without compromising security properties, these individual sessions can be shared at the operating system level. This sharing omits unnecessary handshakes and thus reduces the latency, traffic, and energy overheads.

In this paper, we identify and address the emerging topic of optimizing the use of TLS with respect to latency, traffic, and energy overheads, especially considering the needs of mobile clients. To this end, we empirically study the potential for TLS optimizations for mobile apps and mobile websites before we propose a redesigned TLS session management. Our redesign unlocks this currently unused potential to reduce traffic, latency, energy consumption, and processing overhead of TLS, especially on mobile clients. It enables client applications to reuse session state *across* applications to optimize traffic—an optimization that further benefits from independent enhancements of session management on the server side leveraging network softwarization, e.g., deployed by content delivery networks (CDNs). This redesign can be applied easily to all major mobile operating systems without requiring changes to applications. Concretely, our main contributions are as follows:

- In an empirical study on Android apps and mobile websites, we demonstrate potential traffic, latency, energy, and

 DOI: 10.1109/LCNSymposium47956.2019.9000667

processing improvements of advanced TLS session management. We analyze the TLS connections established by the top 500 Android applications and TLS connections created by browsing the Alexa top 5 000 mobile websites. Our results show that we can reduce average latency of TLS handshakes by up to 195 ms (23.1%) by only modifying clients. Small additional adaptions to TLS session handling on the server side allow us to decrease average latency even by up to 218 ms (25.8%). Our analysis shows significant potential for reducing TLS handshake traffic, latency for application data, and energy consumption especially for mobile devices such as smartphones.

- Motivated by the identified potential, we show how efficient session management for TLS could be easily realized on a current smartphone. We illustrate our proposal by focusing on Android and iOS as the two dominant mobile operating systems. We outline how to easily integrate our approach into these mobile operating systems without major changes. Thereby, we optimize latency and energy overhead without compromising security.
- Complementing improvement on the client side, we show how efficient TLS session management can be supported on the server side, especially when requests are load-balanced to many servers (e.g., in the context of CDNs). To this end, we make the case for sharing of session state between servers of the same provider offered by network softwarization approaches that are often already in place.

With this paper, we aim to open the discussion on the emerging topic of TLS session management optimizations and remark that further optimization potential is available beyond the mobile web, e.g., in the context of the Internet of Things.

II. Related Work

Our work aims at reducing latency and energy for TLS and complements (orthogonal) approaches to improve the performance of TLS. These approaches can either be combined with our proposal to share TLS sessions across applications or require larger changes, hence limiting their deployability.

Tcpcrypt [4] is an alternative to TLS with the goal to improve performance. However, it is incompatible to the widely deployed TLS protocol and requires changes at both servers and clients, severely limiting its deployability. In contrast, adopting our session sharing approach only at a client, i.e., without modification of servers, already significantly improves the performance of TLS connections.

Orthogonal to our proposal to share TLS sessions across applications, *TLS resumption across hostnames* [33] proposes to resume a (single) TLS session for *several* domains as long as they are covered by the same (multi-domain) certificate. A similar approach has been proposed to reduce connection establishment times for QUIC connections [31]. These approaches require support at both, client and server side, while our approach to share TLS sessions across applications already works with changes only at the client. Still, TLS resumption across hostnames and our approach to share sessions across applications complement each other nicely: Session sharing across applications enables even more connections to benefit from TLS resumption across hostnames and vice versa.

From a different perspective, the *TLS false start* mechanism [18] reduces latency for full TLS 1.2 handshakes by one RTT. To this end, the client starts sending application data before receiving the last handshake message from the server. However, web browsers typically disable this mechanism due to incompatibilities and only apply it if servers signal support using a TLS extension [11]. In contrast, our session sharing additionally saves authentications and does not require server support as it leverages session resumption which, being an often used part of the TLS standard, does not incorporate incompatibilities. Furthermore, TLS false start cannot further improve TLS 1.3 which already uses the improved sending behavior of TLS false start. With session sharing, we can also decrease latency by one RTT for TLS 1.3 by enabling the use of the early data mechanism, which is only available to session resumptions, for a larger number of connections.

Following a similar approach, *TCP Fast Open* [5] enables the start of data transmission already with the TCP SYN packet, thus reducing the latency for TCP connection establishment by one RTT. As this transmitted data can be a TLS message, TCP Fast Open can be combined with our approach to even further decrease latency for the establishment of TLS sessions. To prevent third parties from tracking users via the cookies used by TCP Fast Open, servers can use a cross-layer solution to securely communicate cookies to clients via the subsequently established TLS connection, and generate a new cookie for the next TCP connection each time [34].

From a different perspective and specifically focusing on resource constraints in the Internet of Things, the two approaches for *antedated encryption* and *data authentication with templates* [13] accelerate processing of TLS transmissions on Internet of Things devices through the preprocessing of cryptographic operations. As these optimizations become relevant after the establishment of TLS connections, they especially benefit from the reduced latency, traffic, and energy overheads achieved by our approach to share TLS sessions across applications.

III. State-of-the-Art Usage of TLS

We start our analysis of the potential of redesigning TLS session management by reviewing the current application of TLS. To set up a secure channel, the client triggers a TLS handshake which involves additional overhead in terms of public key cryptography and multiple round-trips for the exchange of messages. Specifically, client and server use this handshake to negotiate cryptographic algorithms and establish secret material. Additionally, the server typically presents a cryptographic certificate to prove its authenticity to the client (in rare cases, a client certificate is used to likewise authenticate the client to the server). Only after completing the full TLS handshake, client and server can securely communicate over the established connection. The information negotiated during a TLS handshake is called a *TLS session*.

To reduce the involved overhead for connection establishment, established sessions can be re-used by the same application only (and *not* be shared between multiple applications). This reuse mechanism is called *session resumption* and it currently allows a single application to leverage existing session information to significantly shorten the handshake for subsequent, and also parallel TLS connections with the same server. It thus considerably reduces cryptographic processing and allows for earlier transmission of application data which reduces overall transmission time [6] and energy consumption [3]—which is why we propose to extend session management beyond current application boundaries. An application can leverage session resumption to efficiently maintain multiple parallel TLS connections to the same server or for lightweight revival of a previous connection, e.g., when the device changes between cellular and WiFi networks. To this end, the application includes a session identifier in the first handshake message, enabling the server to select the session to resume. While no strict session lifetime is enforced, the standard suggests a validity period of one day [6] which may be relaxed based on security requirements [28] (we observed validity periods up to three days in our Android and website measurements).

Session resumption involves further costs since it requires client and server to store their respective session state. As this might become prohibitive for servers with many client sessions [30], TLS allows servers to securely offload their session state to clients [28]. To this end, the server encrypts the offloaded state and retrieves it from the client in the first message of the session resumption handshake.

While session resumption provides clear benefits, we argue that its potential is not fully reached yet since it is bound to per-application sessions only and does not benefit from session sharing gains between applications and different servers of the same provider. Especially mobile applications, which often employ similar services, would significantly profit from TLS session sharing *across* these applications. We will show these gains in Section IV by analyzing popular Android apps and mobile websites.

A. Further Evolution of TLS

In addition to the benefits provided by session resumption today, the new TLS 1.3 standard [25] enables the secure transmission of application data already with the first message of a session resumption handshake (referred to as early data or 0-RTT data). To protect the 0-RTT data, the former session setup derives early data secrets for a future resumption with 0-RTT data. Likewise, the server can include application data already in its first handshake message (0.5-RTT data). In contrast to 0-RTT data, the server protects 0.5-RTT already with fresh secrets derived with a public key share sent in the first message of the client. Thus, 0.5-RTT data is already available in a *full handshake* and not limited to session resumption handshakes. However, data sent by servers typically relies on input of clients, e.g., the request of a specific website. Hence, a server can typically leverage 0.5-RTT data only if the client transmits its request as 0-RTT data, which is limited to session resumption. Thus, session resumption unleashes the full potential of 0.5-RTT data as it enables clients to send requests as 0-RTT data such that the server can use the opportunity to send the response as 0.5-RTT data.

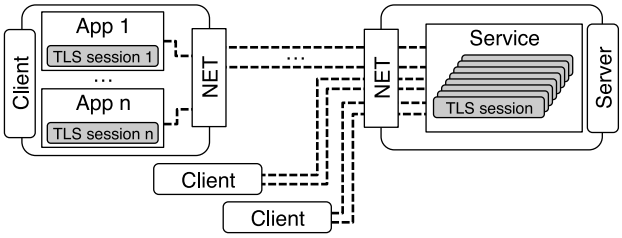


Fig. 1. Inefficient session management of clients is responsible for a huge number of unnecessary full TLS handshakes (dashed lines) and sessions.

In summary, by employing 0-RTT data, TLS 1.3 enables clients to receive protected responses for requests within 1 round-trip, compared to 2 round-trips with today's session resumption, or even 3 round-trips without session resumption. Hence, the ability to use session resumption for many connections, as enabled by our proposal to share TLS sessions between different applications, will be even more beneficial with the increasing deployment of TLS 1.3.

B. Current Limitations of Employing TLS

With the increasing prevalence of TLS, *mobile clients* are challenged by the resulting increase in flow completion times, traffic overhead, and energy consumption. A full TLS handshake, e.g., can add up to 906 ms of latency for application data on smartphones [3].

While session resumption today allows for reducing the overhead introduced by TLS *within* one application, the same does not hold when considering simultaneous TLS connections *across* different applications. This results from the predominant paradigm of implementing TLS as a library in the context of individual applications. Corresponding restricted access to session state prevents the benign sharing of session state among different applications to benefit from the advantages of reusing session state *across* applications.

Hence, as depicted in Figure 1, if multiple applications on the same client establish secure connections to the identical server, they have to execute many full handshakes. Up to 79.4% of these handshakes are redundant (see Section IV), thus resulting in unnecessary processing overhead for authentication mechanisms, generation of superfluous traffic, and increased delay before the transmission of application data can take place. Involved energy overheads additionally decrease operational time of mobile devices before they require recharging. Besides disadvantages for mobile clients, which trigger session resumption, limiting session resumption to connections within the same application also puts unnecessary load on servers.

IV. Session Sharing: Unused Potential

TLS session resumption as a mechanism to speed up the establishment of recurring connections is currently limited to *within-application* session resumption and does not yet leverage benefits by *between-application session sharing*. To overcome this inefficient session handling, which is especially disadvantageous to mobile clients as we will show next, we propose to share TLS sessions *across* applications (see Section V for the realization on Android and iOS). This enables us to unleash the full potential of session resumption and, hence, provide significant benefits, especially for mobile clients. Since session sharing lowers the connection setup RTTs, it will ultimately contribute to the efforts to raise adoption of TLS by considerably increasing performance and decreasing involved traffic and energy overheads.

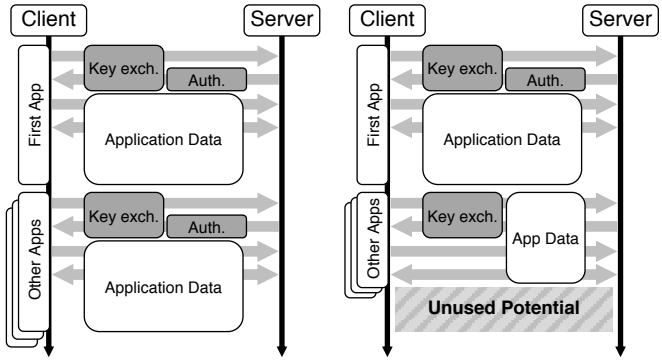
A. Conceptual Benefit Analysis

We start our analysis of the benefits of *between-application session sharing* by showing the optimization potential *conceptually* in Figure 2, before we describe our measurement study in the remainder of this section. Recall that today, even when connecting to the same service, each application performs its own full TLS handshake. The cost of performing handshakes particularly dominates short-lived connections that prevail in current web and mobile app traffic patterns [15], [21]. In contrast, with our proposed session sharing concept, only the first application that connects to a service executes this rather expensive handshake (see Figure 2b). Further applications that connect to the same service can then leverage the existing session for session resumption to benefit from an improved latency by one RTT (depicted as *unused potential* in Figure 2b), decreased traffic, and reduced energy consumption due to the omitted full handshake. While this may suggest only minor improvements *per-connection*, the *overall* savings become relevant in the presence of many short-lived connections, especially for mobile clients. To further explore this potential, we next focus on an empirical analysis of the optimization potential by evaluating application traffic.

B. Empirical Benefit Analysis

Especially mobile applications and websites repeatedly employ the same services, e.g., when including social network functionality [12]. We therefore show the potential of session sharing for *i)* mobile apps and *ii)* mobile websites.

1) Benefits for Mobile Applications: Different mobile applications often access the same services (e.g., social networks, analytic services, advertising networks, or cloud storages) [12], but are required to perform individual TLS handshakes. We show that these individual TLS handshakes are often unnecessary and can be optimized by proper TLS session management *without* compromising security properties. Thus, we demonstrate the potential of *session sharing across applications* to reduce traffic and processing overhead, improve end-to-end latency, and decrease energy consumption based on traffic generated by Android applications.



(a) Separate TLS usage (b) Session Sharing

Fig. 2. With separate TLS usage (a), each application performs its own handshake (here we show TLS 1.3). With our session sharing approach (b) only the first application executes a full handshake. Later on, other applications can leverage the existing session to use session resumption which saves the authentication and decreases the latency by one RTT. Session sharing also achieves this reduction for TLS 1.2, which, compared to TLS 1.3, typically requires one additional RTT before starting the key exchange.

a) Measurement setup: To showcase the applicability and potential of session sharing, we analyzed the TLS connections of the top 500 free Android applications from the Google Play store. We note that a single user typically does not use all these apps. However, our analysis already showed benefits when considering only the top 10 applications. For our measurements, we instrumented a Nexus 7 (2013) operating Android 6.0.1 to execute each application for 30 seconds while supplying random user input with Google's Application Exerciser Monkey [2]. During the execution, we captured and analyzed the network traffic of the device to obtain the number of full TLS handshakes as well as the number of *within-app* session resumptions. To further determine the amount of connections that could employ session resumption when sharing sessions *across* application, we grouped the TLS connections of all applications by IP address and server name indication (SNI). The SNI is exchanged during the TLS handshake to indicate the target domain of the connection [36]. Thus, it enables us to distinguish TLS connections to different services that are reachable via the same IP. As we observed differences in TLS connections established during the first start of an application after installation compared to subsequent starts (which we attribute to application initialization), we executed each application two times and only consider the second run for our analysis. To quantify energy consumption and latency of TLS handshakes, we rely on measurements performed by Ariyapala et al. [3].

b) Significantly more usage of session resumption: Our measurements, which we summarize in the left half of Table I, comprise a total of 4 995 TLS handshakes of which 80.5% are full handshakes. Session sharing on the client side would drastically reduce this ratio to 30.8%, decreasing delay by 168 ms, and saving 20.4% of energy on average. Still, these results are limited by content delivery networks (CDNs), which often serve client requests from different servers, e.g., to balance

	Top 500 Android applications			Alexa top 5 000 mobile websites		
	Unmodified Client + Server	Modified Client only	Modified Client + Server	Unmodified Client + Server	Modified Client only	Modified Client + Server
Full Handshakes [#]	4 021 (80.5%)	1 538 (30.8%)	828 (16.6%)	57 719 (81.7%)	17 029 (24.1%)	12 539 (17.7%)
Resumptions [#]	974 (19.5%)	3 457 (69.2%)	4 167 (83.4%)	12 935 (18.3%)	53 625 (75.9%)	58 115 (82.3%)
∅ Energy [μAh]	51.29	40.80	37.80	51.54	39.39	38.05
∅ Latency [ms]	840.44	671.96	623.79	844.48	649.29	627.75

TABLE I
SESSION SHARING SIGNIFICANTLY INCREASES THE AMOUNT OF SESSION RESUMPTIONS FOR ANDROID APPLICATIONS (LEFT) AS WELL AS FOR MOBILE WEB-BROWSING (RIGHT), THUS DECREASING AVERAGE LATENCY AND ENERGY CONSUMPTION PER HANDSHAKE. LATENCY AND ENERGY CONSUMPTION ARE BASED ON MEASUREMENTS FROM [3].

load. Typically, session resumption in this setting requires multiple full handshakes for the same service (one per server) as session secrets are only available at the host that established the session. To overcome this limitation, modifications on the server side can enable different servers of a CDN to still resume the same session (see Section V-B). Using the SNI and autonomous system as indicators for connections to the same CDN, we estimate that server-side session sharing further reduces full handshakes up to 16.6% and thereby decreases average latency by additional 48 ms (217 ms in total).

Notably, adoption of these modifications by CDNs is extremely likely as the increase in TLS traffic also challenges *servers* by imposing overhead which can significantly be reduced by session sharing at the *client* side. Additionally, the increased latency of TLS (compared to unencrypted traffic) can significantly decrease the revenue of web services. Amazon, e.g., observed that a 100 ms increase in latency reduced their sales rate by 1% [9], [20]. Support of session sharing at clients by CDNs can decrease this latency and thus increase revenue. In fact, Twitter and Cloudflare, e.g., already support server-side session sharing to enable *within-app* session resumption across their servers (see Section V-B).

c) Detailed analysis of CDN influence: To highlight the influence of CDNs on the benefit of sharing TLS sessions across applications, we illustrate the number of applications that connect to a specific service (i.e., distinct SNI) via TLS for the 10 most often used services in Figure 3. Here, each colored area represents the number of applications connecting to a single host (IP address) of this service, while the full bar represents the total number of applications that connect to this service across all hosts (i.e., using any of the IP addresses that provide access to the service). We observe that Flurry (data.flurry.com, a mobile app analytics service) serves all our requests from a single host (the bar has a single area) although we conducted our measurements across three days, and thus expected changing hosts. Consequently, Flurry enables us to already leverage the full potential of session sharing by only adapting clients. Also for Facebook (graph.facebook.com), which served our device with two hosts, clients almost draw full benefits without specific server support.

On the other extreme, some services employ many hosts, e.g., we observed 89 hosts that serve requests to the crash reporting and statistics service Crashlytics (the bar for

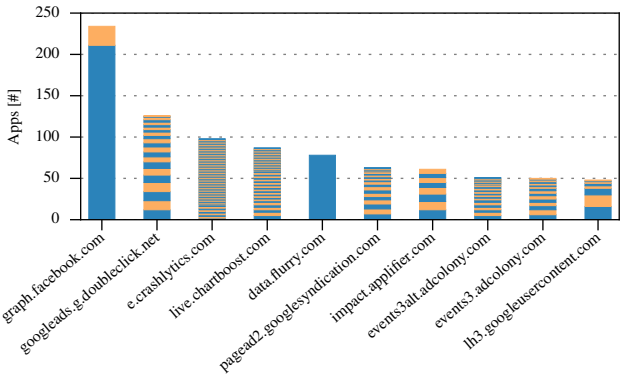


Fig. 3. Top 10 web services used by the top 500 free Android applications via TLS. Services (DNS names) hosted by multiple hosts (IPs) depicted by colored areas: bar height denotes the number of apps connecting to a host.

e.crashlytics.com comprises 89 areas). In this case, session sharing requires server support to optimize 104 of 105 full handshakes (notably, some applications unnecessarily perform multiple full handshakes), while missing CDN support limits the application of session sharing to only 16 of those handshakes. However, we again highlight that adoption of support for session sharing by CDNs is likely as for Crashlytics, e.g., this saves 99.0% of full handshake related processing and traffic overhead while missing support reduces these benefits to only 15.2%. This also affects average latency which decreases by 51 ms without server support, but by 329 ms with server support, significantly increasing revenue [9], [20]. We remark that services that use measures to almost always serve a specific client from the same host, e.g., as we observe with Facebook, sparsely benefit from server support. However, in this case, clients already achieve maximal benefits from session sharing even without server support.

Apart from the extremes, for most services with multiple hosts, session sharing already excels without CDN support, e.g., 126 applications connect to Google's advertising network Doubleclick (googleads.g.doubleclick.net) which served the applications in our measurements from 22 different hosts. Here, session sharing applies to 85.8% of full handshakes without server support (99.3% with server support) and thereby decreases average latency by 282 ms (328 ms).

d) Not sharable sessions are seldom: As we detail in Section VI, we cannot share sessions with authenticated clients (i.e., using client certificates) across applications due to security reasons. However, only 5 out of a total of 4 995 TLS connections in our measurements employed this optional feature. Further, corresponding services anyway do not benefit from session sharing as each is used by exactly one application. Hence, excluding sessions with client authentication from session sharing does not negatively affect its huge potential.

2) Benefits for Mobile Websites: More and more mobile applications encompass browser functionality, e.g., to browse newspaper articles. To account for this evolution, we further measured the TLS usage during web-browsing. To this end, we instrumented Chrome to visit the mobile versions of the Alexa top 5 000 websites [1].

We depict the result of our analysis in the right half of Table I. Overall, we observed 70 654 TLS handshakes of which 81.7% are full handshakes. Session sharing across applications again drastically decreases this to 24.1% without and 17.7% with support by CDN servers. Thereby, average latency decreases by 195 ms and 217 ms, respectively. These latency decreases are more drastic as compared to pure application traffic because of the lower fraction of session resumptions in the unmodified case. Again, client authentication has no relevant impact as only 8 connections applied it.

Figure 4 shows the number of websites employing the 10 most popular web-services (distinct hostnames) via TLS. Again, Facebook serves a client from only a few hosts which enables clients to leverage full benefits of session sharing without server support. Other services, e.g., the ad network Adnxs, heavily distribute connections of a single client to multiple hosts such that average latency reduces by 154 ms for client only support, but by 268 ms with server support. However, the majority of web-services moderately distributes client connections such that session sharing provides considerable benefits without server support, e.g., Google Analytics with 10 hosts shows an average latency reduction by 337 ms with server support of which already 331 ms can be achieved without server support.

Our analysis of TLS handshakes in both use cases, i.e., mobile applications and browsing on mobile devices, shows the huge potential of session sharing across applications with respect to latency and energy consumption. Depending on the setup as well as the support by clients and servers, session sharing across applications allows us to reduce the number of necessary full TLS handshakes by 79.4% for apps and by 78.3% for browsing, respectively. Thereby, session sharing removes the superfluous overhead introduced by the traditional isolated way applications use TLS sessions today. Encouraged by these empirical observations, we next illustrate how session sharing can be realized for both clients and servers.

V. Towards Realizing Session Sharing

In light of the identified potential of sharing TLS sessions across applications, we now outline approaches for realizing session sharing on the client side, specifically for mobile devices using Android and iOS. We then discuss how to amplify benefits of session sharing through support on the server side, especially considering CDNs.

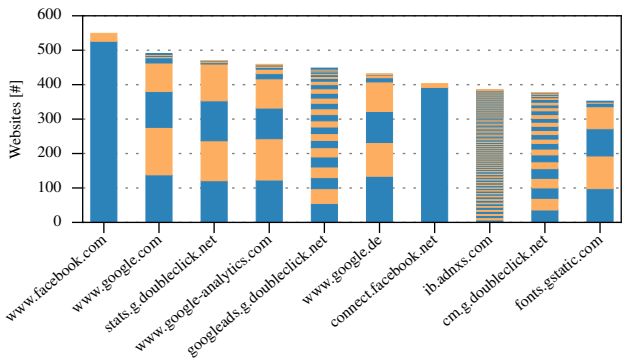


Fig. 4. Top 10 web-services used by Alexa top 5000 websites via TLS. Again, multiple hosts per service depicted by colored areas.

A. Sharing TLS Sessions Across Applications

Our analysis highlights that session sharing provides considerable benefits, even if only clients support it. Hence, we first propose an approach that clients can apply without any support by servers, which significantly eases deployment. To realize the potential of session sharing across applications without the need to modify each single application, we leverage the fact that mobile applications typically employ the TLS API of the mobile operating system to use the TLS implementation shipped with the operating system. That is, as we show in Figure 5, we only need to adapt Android's *Security Provider* and the *App Transport Security* (ATS) of iOS to enable millions of applications to benefit from the advantages of session sharing across applications without any modifications to the applications itself. To this end, we propose to integrate the already existing TLS implementation, which is currently linked to applications at runtime, as a system component, which we refer to as the *TLS manager*. We can then modify the security provider (Android) respectively app transport security (iOS) to use this system component.

Introducing the TLS manager enables us to realize faster connection establishment through session sharing across applications, which we realize based on session resumption that is already present in TLS. As for every TLS implementation, each application still has its own TLS connection. However, in the case of resumable sessions with the same server (also from other applications), the TLS manager can use session resumption to speed up the connection establishment. This extension can be implemented easily, since session resumption is already available in the TLS implementations. Only client authentication requires us to slightly modify the TLS implementation to enforce session sharing restrictions, i.e., sessions with (seldom used) client authentication must not be resumed across applications to prevent impersonation of another application (see Section VI). An optimized version could still

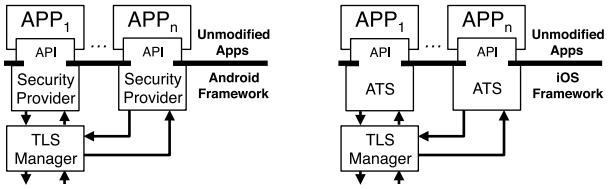
allow for resumption in the case of client authentication if an application can prove possession of the same client certificate as already used in the session. Notably, benefiting from session sharing improvements only requires us to moderately modify the management of TLS sessions. Importantly, our realization does not require server-side changes since we leverage session resumption which is already widely supported by servers [27].

Additionally, applications still use the unmodified API of the security provider (Android) respectively app transport security (iOS) to obtain TLS session information from the TLS manager, e.g., server certificates and employed cryptographic algorithms. Notably, these APIs already prevent applications to access session secrets that would allow them to tamper with all connections based on this session, i.e., also connections of other applications. Realizing the TLS manager as a system component only further prevents malicious applications to extract session secrets from their memory. Thus, our proposed scheme does not compromise security since session secrets are sealed by the TLS manager and applications continue to communicate via separate connections, for which, however, we drastically decrease handshake overheads.

B. Amplification with Support by CDNs

As shown by our measurements, session sharing already excels without specific support by servers, but shows even better results when session sharing works across different servers used to respond to requests, e.g., in the case of content delivery networks (CDNs). That is, joint usage of session state between servers that load balance their processing allows for increased usage of session resumption (a challenge that already exists *without* our optimized session management). To leverage this unused potential, different servers of a CDN must be able to resume the same sessions. Twitter and Cloudflare, e.g., already support this to enable *within-app* session resumption across their servers [14], [19]. To this end, they employ their CDN management network to either synchronize TLS sessions across servers or synchronize the secrets that protect offloaded session tickets (see Section III) to decrypt and verify tickets on any of their servers. One promising approach to synchronize TLS sessions or corresponding secrets across different servers leverages the capabilities of network softwarization, e.g., using software-defined networking [35].

To also leverage this support for session sharing *across* applications, the TLS manager on a client must select the correct session for different servers of a service. As our measurements show, applications already provide the service name to the Android security provider which sends it as server name indication (SNI) in the first handshake message. Likewise, the app transport security of iOS obtains this information. Servers use the SNI to multiplex connections to the correct service. The TLS manager, however, can leverage it to determine the correct session by comparing the SNI provided by the application with the SNIs used by existing TLS sessions. Alternatively, it can compare the provided SNI with server certificates of active TLS sessions. This additionally allows us to reuse sessions for different subdomains under control of the same entity, e.g., www.facebook.com and connect.facebook.net, if the certificate is valid for both. This is a sensible decision as the certificate provides a strong indication that the same provider operates both services.



(a) Android (b) iOS

Fig. 5. Session sharing can be realized by adapting the Android Security Provider and the iOS App Transport Security (ATS) already used by apps.

It is important to note that selecting a session on the client side that the server cannot resume (e.g., because the session state has not been synchronized) neither affects security (see below) nor considerably decreases handshake performance. Specifically, if encountering a session identifier or ticket that it cannot resume, a server proceeds with a full handshake (without additional messages). Thus, a failed resumption adds an only small overhead for transmitting the session identifier or ticket in the first handshake message of the client, on average 32 Byte for a session identifier or 187.98 Byte for a session ticket in our Android measurements.

VI. Security Considerations

Given the huge potential and promising feasibility results for sharing TLS sessions across applications, we discuss security implications of our proposed session sharing scheme. Overall, our approach does not compromise the security properties of TLS connections since the TLS manager handles all session secrets such that applications have no access to them and thus cannot tamper with connections of other applications. We note that applications continue to communicate via separate connections with own security keys, but the TLS manager establishes these keys with session resumption instead of full handshakes to save corresponding overheads. These separate connections with their own keys prevent that a malicious application can obtain secret keys of its own connection, e.g., with a chosen-ciphertext attack, to then use these keys to decrypt the communication of other applications. Furthermore, the separate connections enable the TLS manager to easily associate data streams with the correct application for data forwarding. Although applications have no access to the session secrets, they still are able to check security parameters of an established TLS session or present them to users. To this end, the TLS manager provides access to non-critical session information, e.g., server certificates or negotiated cryptographic algorithms, via the API of the security provider (Android) or app transport security (iOS), respectively.

Next, we discuss the security implications of the centralized handling of TLS connections at the TLS manager. Importantly, the TLS manager can realize session sharing based on the well-

established TLS implementation shipped with the (mobile) operating system. Thus, we argue that the centralization of TLS handling does not introduce new security risks with respect to the implementation of TLS or cryptographic functionality. Specifically, if a security vulnerability affects the TLS implementation of the operating system, this affects all applications regardless of whether they rely on the TLS implementation directly or indirectly via the TLS manager. Furthermore, the strong coupling of connections and applications as discussed above and straightforward checks that identify situations that do not allow for session sharing (see below) minimize the risk for security vulnerabilities introduced by the TLS manager.

Considering services operated on the server side, e.g., provided by a CDN, a client may try to resume a session that the server has no knowledge about (e.g., because the session state has not been synchronized with this server). This neither decreases the performance (see Section V-B) nor compromises security as session identifiers provide no value without the corresponding session [6] and tickets are protected during offloading as, already during *within-app* session resumption, any party on the communication path gets access to them [28]. Session resumption identifiers, however, allow for tracking of users across multiple connections as they link all connections to a single session [32], which is a general problem of TLS session resumption. Session sharing across applications can facilitate this problem as it increases the amount of connections per session. However, application developers can alternatively realize this tracking with device-specific identifiers. Content providers without access to the application, e.g., CDNs, might learn which applications share a session based on the requested content, but lack access to on-device information for more detailed tracking. For applications that provide personally identifiable information to such third parties, we recommend to selectively opt-out from session sharing.

Importantly, session sharing is not applicable to sessions that use client authentication. Sharing such a session would enable other applications to establish an authenticated session without possession of the corresponding certificate which is typically bound to a single application. However, the use of client authentication is almost non-existent for mobile app and website traffic (see our empirical evaluation in Section IV). To ensure that session sharing is not used for sessions that use client authentication, our TLS manager keeps track of such sessions and creates a new session for each separate application. To offer further control over the security of TLS sessions to applications, we envision that applications can completely opt-out from session sharing, e.g., if they link sessions to further state, which is, however, unusual as not covered by the TLS standard.

Regarding joint use of sessions on the server side, e.g., in a CDN, we note that a single misconfigured server may compromise security of all TLS sessions in the complete cluster of servers. However, Twitter and Cloudflare, e.g., already synchronize sessions among their servers to increase benefits from *within-application* session resumption. Leveraging this existing support to increase benefits of session sharing *across* applications does not increase the requirements on secure handling of sessions by servers. Furthermore, session sharing at clients already proves valuable without server support.

Consequently, our approach to share TLS sessions across applications does not compromise security properties on the client side, while achieving the same level of security for synchronizing TLS sessions across different hosts on the server side as in today's deployment for supporting *within-application* session resumption across different hosts.

VII. Conclusion and Further Potential

The rising amount of TLS traffic increases the amount of securely communicated data, but also leads to increases in latency, traffic, and energy overheads, which is especially problematic for mobile clients in emerging 5G and IoT deployments. In this paper, we revealed potential for (mobile) clients to significantly decrease these overheads through a protocol-compliant redesign of TLS session management. Without compromising TLS security properties, our approach enables the sharing of TLS sessions *across* applications to increase the potential for benefiting from session resumptions. We have shown the potential of this approach to reduce TLS-induced traffic, end-to-end latency, and energy overheads of (mobile) clients both conceptually and with empirical analysis of TLS usage by mobile applications and mobile websites. Given the tremendous benefits, we discussed the envisioned realization of session sharing for Android and iOS without requiring changes to existing mobile applications. We further discussed how support on the server side, offered by network softwarization approaches that are often already in place, can yield even higher gains when different servers handle requests, e.g., in the context of content delivery networks.

For future work, we envision a realization of TLS session sharing for mobile operating systems to enable us to analyze its benefits in real user environments. Specifically, we want to analyze the effect of session sharing based on the individual application sets of users and their real behavior opposed to our analysis based on Google's Application Exerciser Monkey.

However, the benefits of sharing TLS sessions across applications are not limited to mobile clients and broadly open the space for further optimizations. For example, non-mobile clients can benefit from TLS improvements in light of transitioning from stand-alone to cloud services. Furthermore, we have shown that also servers significantly benefit from session sharing due to decreased management traffic, less cryptographic processing, and increased revenue, whose evaluation opens the need for further research. Moreover, the improvements of the rising TLS 1.3, especially its early data mechanism which will be applicable more often due to session sharing, promise further interesting directions for researching efficient secure communication. Thus, we believe that the potential identified in this paper provides an important first step towards optimizing the costs of secure communication for a broad class of applications—both at the server- and client-side—and thus offers exciting new research potential.

VIII. Acknowledgments

This paper has received funding from the CONNECT project as part of the Electronic Components and Systems for European Leadership Joint Undertaking. Our work in the CONNECT project has received support from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 737434 as well as the German Federal Ministry of Education and Research (BMBF) under project funding reference no. 16ESE0154. The authors would further like to thank the German Research Foundation (DFG) for the kind support within the Cluster of Excellence Internet of Production (IoP) under project ID 390621612 and for the funding as part of the CRC 1053 MAKI. This paper reflects only the authors' views and the funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Alexa Internet, Inc., "Alexa – Actionable Analytics for the Web," http://www.alexa.com/, accessed: August 5, 2019.

[2] Android Developers, "UI/Application Exerciser Monkey," http://developer.android.com/tools/help/monkey.html, accessed: August 5, 2019.

[3] K. Ariyapala, M. Conti, and C. M. Pinotti, "CaT: Evaluating cloud-aided TLS for smartphone energy efficiency," in *IEEE CNS*, 2015.

[4] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh, "The case for ubiquitous transport-level encryption," in *USENIX Security*, 2010.

[5] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413, Internet Engineering Task Force, 2014.

[6] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Internet Engineering Task Force, 2008.

[7] S. Farrell and H. Tschofenig, "Pervasive Monitoring Is an Attack," Internet Engineering Task Force (IETF), RFC 7528, May 2014.

[8] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring HTTPS adoption on the web," in *USENIX Security*, 2017.

[9] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: The virtue of gentle aggression," in *ACM SIGCOMM*, 2013.

[10] T. Fossati, V. K. Gurbani, and V. Kolesnikov, "Love all, trust few: on trusting intermediaries in HTTP," in *ACM SIGCOMM HotMiddlebox*, 2015.

[11] D. Goodin, "False Start's sad demise: Google abandons noble attempt to make SSL less painful," http://arstechnica.com/business/2012/04/google-abandons-noble-experiment-to-make-ssl-less-painful/, accessed: August 5, 2019.

[12] M. Henze, J. Pennekamp, D. Hellmanns, E. Mühmer, J. H. Ziegeldorf, A. Drichel, and K. Wehrle, "CloudAnalyzer: Uncovering the Cloud Usage of Mobile Apps," in *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*, 2017.

[13] J. Hiller, M. Henze, M. Serror, E. Wagner, J. N. Richter, and K. Wehrle, "Secure Low Latency Communication for Constrained Industrial IoT Scenarios," in *IEEE LCN*, 2018.

[14] J. Hoffman-Andrews, "Forward Secrecy at Twitter," https://blog.twitter.com/2013/forward-secrecy-at-twitter-0, accessed: August 5, 2019.

[15] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance," in *ACM SIGCOMM*, 2013.

[17] H. Kolamunna, J. Chauhan, Y. Hu, K. Thilakarathna, D. Perino, D. Makaroff, and A. Seneviratne, "Are Wearables Ready for HTTPS? On the Potential of Direct Secure Communication on Wearables," in *IEEE LCN*, 2017.

[16] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force (IETF), Internet-Draft draft-ietf-quic-transport-20, April 2019.

[18] A. Langley, N. Modadugu, and B. Moeller, "Transport Layer Security (TLS) False Start," IETF Internet-Draft draft-bmoeller-tls-falsestart-01, IETF, 2014, accessed: August 5, 2019. [Online]. Available: http://tools.ietf.org/internet-drafts/draft-bmoeller-tls-falsestart-01.txt

[19] Z. Lin, "TLS Session Resumption: Full-speed and Secure," https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure/, accessed: August 5, 2019.

[20] G. Linden, "Make data useful," http://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt, 2006, accessed: August 5, 2019.

[21] F. Michelinakis, G. Kreitz, R. Petrocco, B. Zhang, and J. Widmer, "Passive Mobile Bandwidth Classification Using Short Lived TCP Connections," in *IFIP WMNC*, 2015.

[22] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The cost of the "s" in HTTPS," in *ACM CoNEXT*, 2014.

[23] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS," in *ACM SIGCOMM*, 2015.

[24] M. Nottingham, "HTTP/2 Implementation Status," https://www.mnot.net/blog/2015/06/15/http2_implementation_status, accessed: August 5, 2019.

[25] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force (IETF), RFC 8446, August 2018.

[26] P. Richter, N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger, "Distilling the internet's application mix from packet-sampled traffic," in *PAM*, 2015.

[27] I. Ristic, "Internet SSL Survey 2010," in *Black Hat Abu Dhabi*, 2010.

[28] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig, "Transport layer security (TLS) session resumption without server-side state," RFC 5077, Internet Engineering Task Force, 2008.

[29] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "BlindBox: Deep packet inspection over encrypted traffic," in *ACM SIGCOMM*, 2015.

[30] R. Stevens and H. Chen, "Predictive Eviction: A Novel Policy for Optimizing TLS Session Cache Performance," in *IEEE GLOBECOM*, 2015.

[31] E. Sy, "Surfing the Web quicker than QUIC via a shared Address Validation," https://arxiv.org/abs/1903.09466, 2019, accessed: August 5, 2019.

[32] E. Sy, C. Burkert, H. Federrath, and M. Fischer, "Tracking Users Across the Web via TLS Session Resumption," in *ACSAC*, 2018.

[33] E. Sy, M. Moennich, T. Mueller, H. Federrath, and M. Fischer, "Enhanced Performance for the encrypted Web through TLS Resumption across Hostnames," https://arxiv.org/abs/1902.02531, 2019, accessed: August 5, 2019.

[34] E. Sy, T. Mueller, C. Burkert, H. Federrath, and M. Fischer, "Enhanced performance and privacy for TLS over TCP fast open," http://arxiv.org/abs/1905.03518, 2019, accessed: August 5, 2019.

[35] M. Wichtlhuber, R. Reinecke, and D. Hausheer, "An SDN-based CDN/ISP collaboration architecture for managing high-volume flows," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 48–60, 2015.

[36] T. Wright, S. Blake-Wilson, J. Mikkelsen, M. Nystrom, and D. Hopwood, "Transport Layer Security (TLS) Extensions," RFC 4366, Internet Engineering Task Force, 2006.

[37] Z. Zhou and T. Benson, "Towards a safe playground for HTTPS and middle boxes with QoS2," in *ACM SIGCOMM HotMiddlebox*, 2015.

[38] T. Zimmermann, J. Rüth, B. Wolters, and O. Hohlfeld, "How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push," in *IFIP Networking*, 2017.