LURK: Practical & Secure Server-Controlled TLS Delegation

Abstract

The well-known Transport Layer Security (TLS) is supposed to be executed end-to-end, just between one client and one server. But, in practice, *TLS delegation* often takes place: i.e., *middleboxes* proxy, inspect and even modify the traffic between the said client and server. Recently, tech giants like Akamai, together with standardisation bodies (IETF, ETSI), as well as academia, have given prominent attention to numerous ways of achieving TLS delegation.

We propose LURK: a suite of designs for TLS delegation where the TLS-server is aware of the middlebox. We comprehensively discuss how our designs balance (provable) security and competitive performance. We implement and test LURK. We cryptographically prove and formally verify the security of LURK.

ACM Reference Format:

. 2019. LURK: Practical & Secure Server-Controlled TLS Delegation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Decades ago, Internet protocols were designed such that the application logic operated only at the endpoints. Yet, nowadays, before it gets to the end user, data is often processed by a series of intermediaries, such as Content Delivery Networks (CDNs). But, the application-data to be processed is encrypted, most often within a *TLS* session. In this case, processing of such in-transit data by 3rd-parties is often referred to as *SSL inspection* or ***TLS delegation***. TLS delegation for CDNs generally works as follows; the CDN's *edge servers* hold a valid X.509 certificate for the domain(s) delivered and an associated private key impersonating the content-owning servers on behalf of which the delivery is made.

However, whilst this type of CDN-ing over TLS is common, there are many other more complex cases. For instance, leveraging work of the IETF CDN Interconnection Working Group (CDNI) [9], the Open Caching Working Group [33] has specified an architecture, to enable the delegation of content in between CDNs. An *upstream CDN (uCDN)* is a CDN that is willing to delegate content to a *downstream CDN (dCDN)*. However, in this case of advanced CDN-ing, placing trust on possibly unknown dCDNs to hold private keys impersonating the content-owner is a big ask.

But, nowadays, edge computing takes new forms fueled by massive IoT and the rising 5G-fulled. At this end, we find service-based architectures, where the proxies are called sidecars and are interconnected via TLS based on short-lived private keys. An example is Istio [20], where one service-component is in charge of frequently triggering private-key rotation (e.g., Istio Citadel).

So, performing TLS delegation by sharing a private-key on behalf of the original content-owner is inappropriate for inter-CDNing, as well as for service-based architectures such as Istio. An option for them may be *delegated credentials* [3]. Therein, delegation is performed on a TLS specific structure and validated by the end-points. Whilst client-side integration makes this mechanism viable for updatable end-points (e.g., browsers), it is sadly not at all fitting for legacy, hard-to-update end-points, such as TLS Clients that only support TLS1.2 (e.g., playstations).

Last but not least, tech giants (Akamai, Cloudflare, Telefonica, Ericsson) and standardisation bodies (ETSI, IETF) alike have recently devoted considerable attention to proxied-infrastructure security, including TLS delegation, under the umbrella term of middlebox security (see https://portal.etsi.org).

So, all in all, there are many calls for a (backwards-compatible) TLS-solution that: a). places less trust on the proxies; b) provides an easy/agnostic integration with the client-side; c). is secure and efficient. To this end, attempts like mcTLS [30] require impractical integration and are insecure [4], whilst *KeylessSSL* [43] is more practical, but still insecure [5]. Cherry-picking from these, in this paper, we discuss the design, implementation, efficiency and security analysis of such a suite of TLS-delegation solutions called *LURK*.

Main Contributions

1. We propose *LURK (Limited Usage of Remote Key)*: a suite of new designs for practical and provably secure server-controlled TLS delegation, in which the mediating party has limited and remote access to end-server. Herein, we discuss $LURK1.2$: specific versions of LURK based on TLS1.2. $LURK1.2$ has different variants, meeting in the middle between the (insecure) KeylessSSL [43] and the

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(provably secure but inefficient) 3(S)ACCE-K-SSL [5], to balance between security and practicality.

2. We implement the more efficient variants of $LURK1.2$ and test them in practice. We report our findings.

3. In $LURK1.2$, we built in a new "freshness mechanism" of avoiding the replay attacks, i.e., attain perfect forward secrecy in $LURK1.2$.[1]. To this end, we encode $LURK1.2$ in RSA mode in the automatic protocol-verifier ProVerif and we formally prove that our "freshness mechanism" does indeed aid to ensure perfect forward secrecy in $LURK1.2$ in RSA mode.

4. In the long version of this paper [27], we provide cryptographic proofs that $LURK1.2$ attains attains all the expected security properties for a (proxied) authenticated key-exchange; for the proofs, we use the recent 3(S)ACCE formal security model for proxied AKE [5].

Why $LURK1.2$? Whilst we are actively working on LURK based on TLS 1.3 [11], LURK as presented herein is based on TLS 1.2 and denoted $LURK1.2$. The aim of $LURK1.2$ is to provide the necessary agility required during this transition from TLS 1.2 to TLS 1.3. Concretely, TLS 1.3 [36] was standardised by IETF in August 2018, and has seen fast adoption by large companies (Facebook, Google, Microsoft, Akamai) as well by standardisation bodies such as 3GPP. Yet, the requirements by NISTSP800-52 rev2 are to support TLS1.3 by January 2024. Meanwhile, TLS 1.2 remains the de-facto version of TLS used worldwide, in more and more proxied versions, and LURK aims to prevent such bespoke TLS 1.2-communications operate in a weak way. In fact, many services rely on so-called legacy devices, such as video on demand being provided by Customer Premises Equipment (CPE); for these, it has also been reported that the migration process from TLS 1.0 to TLS 1.2 is expected to start in the next two/three years and be completed by 2030; and, their migration to TLS 1.3 is expected to take even more, with EMS/EtM/LTS within ten years. One of $LURK1.2$'s aim is to bridge the security and practicality gap of proxied-TLS solutions especially for this type of devices.

Moreover, as lesser reasons, we also aimed to propose a secure and efficient version of KeylessSSL, and KeylessSSL was based on TLS1.2. So, with this in mind and the space constraints, herein we discuss just $LURK1.2$ (and we leave LURK based on TLS1.3 for the long version [27] and, even more so, for future lines of this work [11]).

Structure

In Sec. 2, we present our designs of LURK using TLS 1.2, with different variants. In Subsec. 2.2, we include a comprehensive discussion of our design choices.

In Sec. 3, we present our Python implementation of Variant 1 of $LURK1.2$. In Section 4, we evaluate the latency of LURK vs TLS. In Subsec. 5.1, we state the security requirements of $LURK1.2$; the accompanying cryptographic proofs of these are given in the long version of the paper [27], using the 3(S)ACCE formal model from [4]. In Subsec. 5.2, we use formal verification to show that our new "freshness mechanism" does aid to ensure perfect forward secrecy in $LURK1.2$ in RSA mode. In Sec. 6, we compare with the most relevant alternatives, notably KeylessSSL [43]. In Sec. 7, we include future directions and conclusions. In the long version of the paper [27], we discuss elements of our work on LURK based on TLS 1.3, as well as include more details on all the aforementioned aspects.

2 LURK: Server-Controlled TLS Delegation

LURK enables server-controlled TLS delegation transparently to the TLS Client. In LURK, the record layer of TLS remains unchanged. The modifications to TLS affect just the part called the *handshake* : i.e., the session-key establishment. To this end, the standard TLS Server is split into two services: 1). a *Cryptographic Service* ("Crypto Service" for short), denoted S, which performs cryptographic operations associated to the private key of the TLS Server; 2). a *LURK Engine* ("Engine" for short), denoted E, which performs the remainder of the TLS server-side process. The Crypto service and the LURK Engine can be colocated[2] services or not. These two services communicate over a secure channel, providing confidentiality, integrity and mutual authentication.

$LURK1.2$ instantiates the above with TLS 1.2. Please see Appendix B for details on TLS 1.2.

2.1 The $LURK1.2$ Designs

$LURK1.2$ is parametric in a security parameter s, as well as on a pseudorandom function φ called the *freshness function* that is re-chosen at each session between the Engine and the Crypto Service. Notation-wise, $\varphi(\cdot)$ is a shortcut for $\varphi_k(\cdot)$ with k being fresh key indistinguishable from random, exported from an authenticated key-exchange protocol (e.g., EAP-TLS or other "TLS-like" protocol [8]) pre-run between the Engine and the Service.

$LURK1.2$ **in RSA Mode.** Fig. 1 presents $LURK1.2$ for TLS1.2 in RSA-mode. As per Fig. 1, the Client initiates the TLS handshake. Thereafter, the LURK Engine generates the server-nonce N_E in a different manner than in TLS 1.2. Instead, it first starts by generating a nonce N_i at random with a length parametric in a security parameter.

Second, the LURK Engine applies the φ and returns $N_E = \varphi(N_i)$ as the "TLS server random" to the Client, and deletes N_i off its memory. Third, the TLS Client then sends the client key-exchange message KE_C containing the encrypted pre-master secret pmk, alongside the client-finished messages Fin_C. Fourth, the LURK Engine

[1] Since TLS 1.2 RSA mode does not ensure forward secrecy, placing a mediating party in between the client and the server can lead to replay attacks. This was shown of KeylessSSL in its variant based on TLS 1.2 in RSA mode, and a repair was proposed via the 3(S)ACCE-K-SSL design [5]. Our replay-counteracting mechanism differs from the proposition that [5] made to mend replay attacks in Keyless SSL.

[2] In CDN-ing, the LURK Engine would be hosted by the CDN at the edge, while the Crypto Service by the content owner.

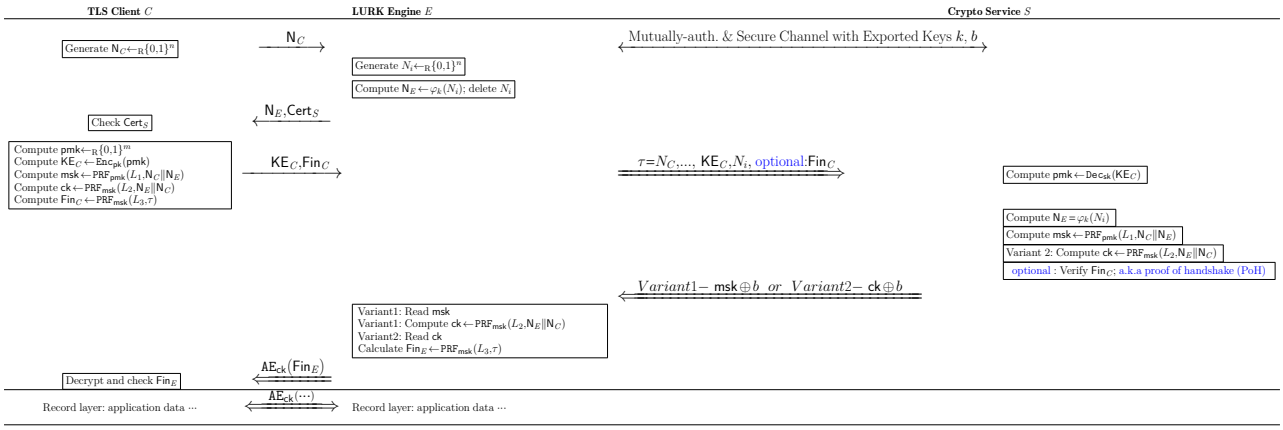


Figure 1. $LURK1.2$ based on TLS1.2 in RSA mode: Two Variants

forwards these (Fin_C being optional), along with N_i and all elements of the transcript τ to the Crypto Service. Then, the Crypto Service computes N_E as $\varphi(N_i)$, retrieves the pmk, eventually verifies Fin_C and computes the master secret msk. Note that the optional Fin_C is also referred as the *Proof of Handshake (PoH)*.

Henceforth, $LURK1.2$ branches out in two variants. In Variant 1, the Crypto Services sends back the master-secret msk to the LURK Engine, whereas in Variant 2 – the channel key ck is sent back. Either message, msk or ck is sent encrypted with another exported key b. Then, the protocol between the Engine and the Client follows the normal TLS interaction and record-layer communication.

$LURK1.2$ **in RSA Extended Mode.** $LURK1.2$ in RSA Extended mode only differs from $LURK1.2$ in RSA in that the master secret msk is is generated using the transcripts of the handshake instead of the nonces N_C and N_E.

$LURK1.2$ **in DHE Mode.** W.r.t. $LURK1.2$ in DHE mode, we also propose two variants.

In the first variant presented in Fig. 2, the LURK Engine generates the DHE keypair $(v, g^v \mod p)$ and sends the shared secret KE_E to the Crypto Service together with N_C and N_i, as well as an optional *Proof of Ownership of* v, denoted $PoO(v)$; the latter being a non-interactive proof of knowledge of the secret exponent v.

The Crypto Service only accepts a specific data-structure for the messages and declines the communication otherwise. Then, the Crypto Service verifies the optional PoO, computes and signs the hash of sv before returning the signature to the LURK Engine. From here on, the LURK Engine continues the TLS handshake with the Client as expected. After the use of the DHE keypair and the N_i nonce, the LURK Engine deletes them off its memory.

In the second variant, presented in Fig. 6 – found in Appendix A, the Crypto Service executes more operations on behalf of the Engine than in Variant 1. Namely, the Crypto Service generates the ephemeral DHE exponent v, so it generates the pmk value and it only returns to the LURK Engine the channel key.

Notes: A detailed specification of $LURK1.2$ w.r.t. network layer, packet formats, options, etc., is available at [1]. Section 3 will detail on some of this.

2.2 $LURK1.2$'s Security and Efficiency-driven Design Choices

$LURK1.2$**'s Mechanism Anti-replay Attacks: the Freshness Function** φ**.** This mechanism enables $LURK1.2$ in RSA mode to protect against replays and provide perfect forward secrecy (PFS). In simple terms, if an adversary $\mathscr{A}$ gathers plaintext information from a handshake, then $\mathscr{A}$ will get N_E and KE_C but not the N_i sent on the secure channel between the Engine and the Crypto Service. If later on, at time t, $\mathscr{A}$ corrupts the Engine E, N_i has been deleted from E's memory at the end of its use and so cannot be retrieved by $\mathscr{A}$. With φ being a PRF and as such non-invertible, $N_E = \varphi(N_i)$ cannot be guessed by $\mathscr{A}$. So, $\mathscr{A}$ cannot present an old query to the Crypto Service.

We note that this PFS-enforcing mechanism in $LURK1.2$ is more communication-effective than the repairs made to KeylessSSL via 3(S)ACCE-K-SSL in [5] (i.e., in $LURK1.2$, the Crypto Service does not need to send N_E to the Engine at the beginning of the handshake).

We will further discuss the PFS guarantees of $LURK1.2$ in RSA mode in Section 5.

$LURK1.2$ **Limiting Signing Oracles & Cross-protocol Attacks via Specific Checks & Proofs of Membership.**

In the DHE mode, some precautions are required to prevent the Crypto Service be turned into a "signing oracle" that would sign any input, including inputs outside the scope of the TLS handshake (cross-protocol attacks). These precautions follow. Firstly, we impose that the Engine send the individual elements of his side of the handshake and it is the Crypto Service who is computing the hash of these (as opposed to allowing the Engine to send the hash directly); this prevents chosen plaintext be signed, as well as enables specific checks to be performed by the Crypto Service. The latter includes checking nonce format,

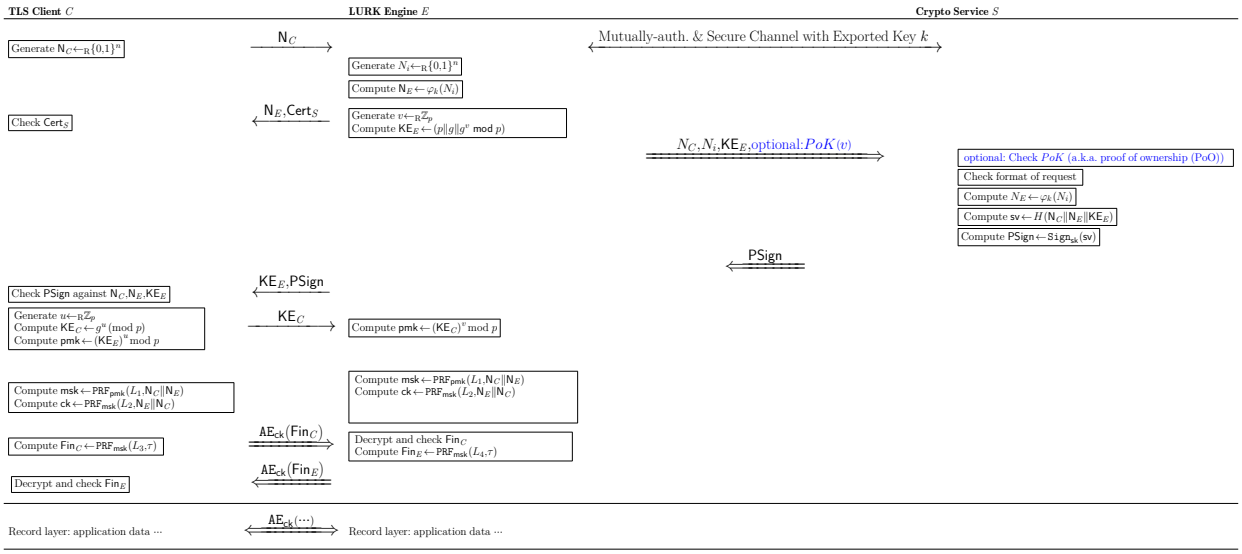


Figure 2. $LURK1.2$ based on TLS1.2 in DHE mode: Variant1

the format of the Elliptic Curve (EC) Diffie-Hellman parameters (compatibility with the pre-agreed curve), as well as the optional *PoO* (proof of ownership). The PoO is a non-interactive knowledge proof that the Engine E indeed knows the private counterpart v of the to-be-signed public-key g^v, i.e., an interactive proof such as Schnorr made non-interactive via the Fiat-Shamir transformation [13].

We will further discuss these matters in Section 3.

$LURK1.2$: Session Resumption vs. Accountability. *Session resumption* consists in producing a new channel key ck by the Engine without contacting the Crypto Service, using a previously generated msk and new nonces generated by the Client and the Engine itself.

Variant 1 of $LURK1.2$ in RSA mode returns the msk, so the LURK Engine may, in theory, perform session resumption. However, LURK does *not* provide an extension that would enable session resumption. Engines. In Variant 2 of $LURK1.2$ in RSA mode, the channel key ck is returned to the TLS Engine E, and so –in this version– session resumption is not possible. So, if we were to modify LURK so that we explicitly opened for session resumption, then Variant 1 of $LURK1.2$ in RSA mode could become more communication-efficient than Variant 2.

Yet, unlike Variant 1, Variant 2 of $LURK1.2$ in RSA mode achieves a strong security requirement called *accountability* [5]. Simply put, accountability equates to the Crypto Service being able to audit the activity of the LURK Engine. Accountability in Variant 2 is due to the very fact that the Crypto Service computes the TLS channel-key used by the Client and LURK Engine (as opposed to just the msk in Variant 1).

Protection Against Malicious Crypto Services. In practical settings, the ephemeral secret v of the Engine may not always be regenerated, e.g, see Section 6.4 of RFC7525. As such, if the Engine operated with a static v, plus a malicious Crypto Service learnt this value v at a given time and thereafter became malicious, then the said Crypto Service could impersonate the Engine and inject unwanted messages to the Client. This can not only compromise the Client's security, but breaks the assumptions of the collaborative setup whereby the Engine always mediates the LURK connections between the Client and the Service. This is why we do not send the ephemeral secret v, from the Engine to the Crypto Service.

3 System Implementation

`pylurk`. We implemented Variant 1 of $LURK1.2$ in RSA and DHE modes, in Python. We yielded a client/server library, including both the Engine and the Crypto Service, called ***`pylurk`*** [35], closely following our low-level specifications from [1]. These specifications include aspects of the network interfaces and packet deliveries, in addition to the design presented in Section 2.

We adopt a modular approach where each LURK extension is programmed as a module. Specifically, `pylurk` defines the interface with each module such that $LURK1.2$ packets are steered from the core $LURK1.2$ module to the appropriate module based on the $LURK1.2$ header (i.e., $LURK1.2$ extension, version, and method).

To define packets, we use Python Construct 2.8 (https://pypi.org/project/construct/2.8.22/), which parses binary format into containers (and vice versa). Containers are Python objects that easy to manipulate (i.e., datatypes similar to dictionaries). `pylurk` performs object-manipulations and processing using the container datatypes. Specific data structures are defined in each protocol extension, but the parsing and/or the conversion to/from a binary format is performed by the core $LURK1.2$ module.

Network Interfaces in `pylurk` . Between the $LURK1.2$ Engine and the Crypto Service support UDP, TCP, TCP+TLS, HTTP and HTTPS. The Crypto Service is based on the socketserver module [40] for TCP and UDP implementations, on `http.server` [18] for the HTTP implementation, and `SSL` [42], as in TLS/SSL wrapper for socket objects, for the packet protection. Note that TCPServer is originally designed to close the TCP session after each request; instead, we implement a version that handles long-term TCP sessions. This enables a client and a server (here the Engine and Crypto Service) to handle a TCP session eventually protected by TLS, and use that session to transport many $LURK1.2$ requests. However, we left the `http.server` class unchanged. Hence, when HTTP is used in combination with TLS, a TLS exchange is performed for each TCP session per request, which is non-optimal. UDP+DTLS remains unimplemented as we could not find a suitable DTLS library in Python. Furthermore, UDP+DTLS would end up in a stateful protocol, removing the main characteristics of UDP. As a result, we followed the similar design as DoT [19] and DoH [16], and limited our scope for the CDN use case to TCP+TLS and HTTPS. UDP is left to the usage of $LURK1.2$ in a TEE (Trusted Execution Environment) or in containerized environments, where exchanges are performed on a given platform without exposure to the network.

Cryptographic Primitives in `pylurk`. These are mostly based on `Cryptodome` [10], while specific elliptic curve operations, e.g., the proof of ownership (PoO) of the DHE exponent, rely on `tinyec` [45]. The Crypto Service can enforce the specific TLS primitives anc TLS ciphers to be used. To this end, we currently support, e.g., the use of SHA256, SHA384 and SHA512 for generating the master secret. For efficiency reasons, the freshness function φ is implemented with SHA256. Other options can easily be added, including where φ is a PRF instance keyed on a fresh value established at each new session (cf. Section 2). Alongside, in RSA mode, we enforced the following ciphers: TLS_RSA_WITH_AES_128_GCM_SHA256 and TLS_RSA_WITH_AES_256_GCM_SHA384. For DHE mode, namely ECDHE, we enforced the use of secure hash functions in the signature scheme (SHA256 and SHA512) for both RSA and ECDSA. Similarly, we implement secure elliptic curves (secp256r1, secp384r1, secp521r1) for the generation of ECDHE and ECDSA signature. In RSA mode, we encrypt the premaster secret using a 2048-bit public key. Again, other options can easily be added to the implementation.

Lastly, in the implementation for $LURK1.2$ in RSA mode, during the last message between the Service and the Engine, we simply send the message (e.g., msk), not its encryption under the exported key b; this is because the channel is already secure and said encryption is simply needed for strong 3(S)ACCE provable-security results (see the long version of this paper [27]), but adds nothing to practical security.

4 Performance Evaluation

We now investigate the performance of $LURK1.2$ vs. that of a classical TLS1.2 handshake, and study how different design and implementation choices in $LURK1.2$ impact its overall performance. For all experiments, we use the Variants 1 of $LURK1.2$ in the `pylurk` [35] implementation (Section 3). Our prototype runs on Xubuntu 18.04, on an Intel i7-2820QM CPU (2.3GHz) with 16GB RAM. All our results are derived by averaging over 50 iterations.

Recall that $LURK1.2$ splits the TLS Server in two services, and thus adds extra communication to the TLS handshake. Hence, we evaluate the overhead in terms of latency and computing resources (i.e., CPU load) of a $LURK1.2$ session (with different parameters) vs. a TLS session, considering: (i) the transport stack used between the LURK Engine and the Crypto Service; (ii) the main $LURK1.2$ additons to TLS1.2, i.e., the Proof of Handshake (PoH), the Proof of Ownership (PoO) and the Perfect Forward Secrecy (PFS) mechanisms (cf. Section 2).

4.1 Latency

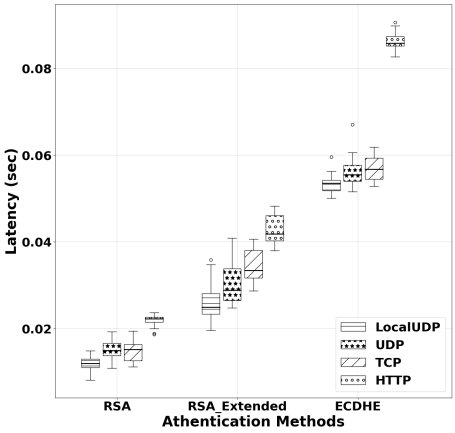
For a given configuration, the measured latency is:

$$l_{LURK} = p_{req} + RTT + p_{resp},$$

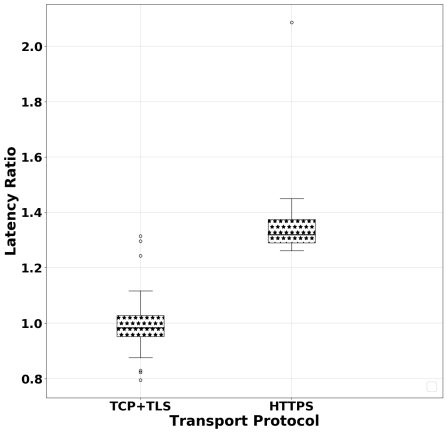
where p_{req} and p_{resp} represent the latency introduced by the treatment of the request and response, respectively, at various layers such as application (parsing, building, processing the LURK messages) and transport (handling HTTP, TCP, TLS with associated interruption or processing). RTT is the round-trip time between the Lurk Engine and the Crypto Service. We measure RTT on a local network, approximating the latency within a data center.

The overhead of $LURK1.2$ compared to a standard TLS handshake can roughly be approximated as the latency between the LURK Engine and the Crypto Service for a UDP exchange performed on the local host: i.e., $l_{TLS} \approx l_{LURK}^{LocalUDP}$. In fact, the use of UDP on a local host between the LURK Engine and the Crypto Service minimizes the networking latency as it uses a virtual interface with a minimal transport protocol (UDP). In that sense, communications are expected to be close to Inter-Process Communications (IPC), and thus the latency overhead of $LURK1.2$ vs. standard TLS can be estimated as: $\delta = \frac{l_{LURK}}{l_{TLS}}$.

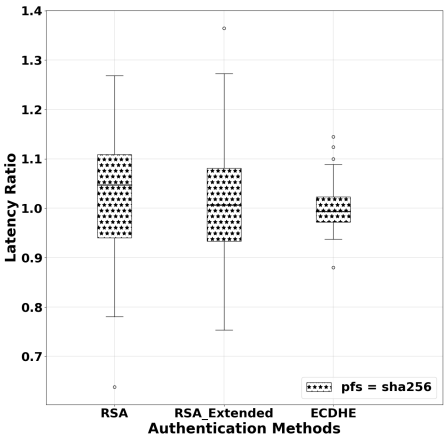
Figure 3a shows the latency in seconds for different $LURK1.2$ modes (i.e., RSA, RSA-extended and ECDHE) for different transport configurations (i.e., local UDP, UDP, TCP, HTTP). Figure 3b depicts the latency ratio of having $LURK1.2$ in RSA mode over TCP+TLS and HTTPS, compared to $LURK1.2$ in RSA mode over TCP, HTTP. In these cases, the PRF function in TLS and the φ freshness function we introduced in $LURK1.2$ are set to SHA256. Figures 3c – 3f show the latency ratio of $LURK1.2$ with particular options enabled vs. the average-times of a reference implementation without those options in place. We also consider a particular instantiation of φ freshness function, the PRF used in generating the master secret, the use of a PoH, the use of a specific PoO vs. the



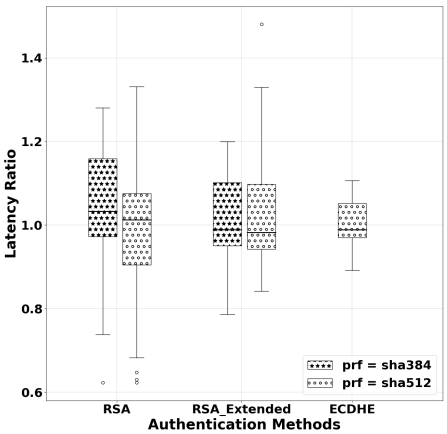
(a) $LURK1.2$ over insecure data (Latency in seconds)



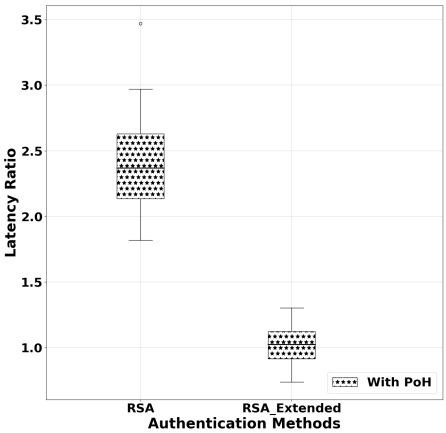
(b) Overhead of TCP+TLS/HTTP vs. TCP/HTTP for $LURK1.2$ in RSA mode



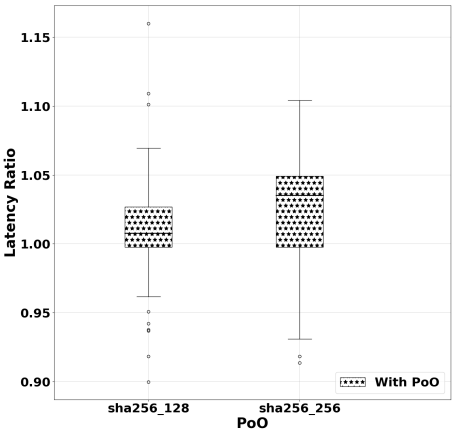
(c) Overhead of using a freshness function in $LURK1.2$



(d) Overhead of setting TLS' PRF to SHA384, SHA512. instead of SHA256. in $LURK1.2$



(e) Overhead of adding Proof of Handshake



(f) Overhead of adding Proof of Ownership

Figure 3. Latency measurements for $LURK1.2$

respective lack of such choices. The measurements shown in Figures 3c – 3f are performed over UDP.

4.1.1 On transport protocols

As mentioned in Section 3, a single TCP session is established between an Engine and the Crypto Service for all the requests. This explains the marginal latency introduced by TCP compared to UDP. In contrast, as we leave the implementation of http.server class intact, thus, establishing a new TCP session per request, we observe a significant increase in the latency when using HTTP in comparison to TCP (Figure 3a). Similarly, the additional latency overhead introduced in HTTPS in comparison to TCP+TLS is a result of the TCP sessions established per request (Figure 3b). Note that the reported latency accounts the time needed to treat the LURK request and response in addition to the RTT time of the message exchange between the LURK Engine and the Crypto Service.

While UDP provides optimal performance, the lack of delivery control makes it a poor candidate for $LURK1.2$. Further, we identify no clear benefit from using HTTP instead of TCP, as for instance, the use of HTTP generates larger payloads. TLS does not impose measurable latency. As a result, from these tests, we recommend that the Engine and the Crypto Service be connected via a long term TCP session protected by TLS.

Now, we recount the specific latency increases observed. TCP increases the latency relative to UDP by 1.02 for $LURK1.2$ in RSA mode, 1.16 for $LURK1.2$ in RSA-Extended mode, and by 1.02 for $LURK1.2$ in ECHDE mode. HTTP increases the latency over TCP by 1.46 in the case of $LURK1.2$ in RSA mode, by 1.25 for $LURK1.2$ in RSA-Extended mode and by 1.50 for $LURK1.2$ in ECDHE mode. Also, Figure 3b shows that, for $LURK1.2$ in RSA mode, the additional costs added onto TLS (e.g., via the introduction of the freshness function) are negligible for TCP+TLS; however, for HTTPS, $LURK1.2$ (vs. TLS) increases the latency by a factor of 1.3.

With TCP+TLS, the overhead of using $LURK1.2$ over the standard TLS1.2 is estimated to be: $\delta_{RSA} = 1.27$, $\delta_{RSAExt.} = 1.24$, $\delta_{ECDHE} = 1.05$. Although these results can, in principle, be affected by various implementation choices, e.g., the choice of the PRF inside TLS, the choice to add a PoH or PoO, etc., we show later in this section, that the costs of these choices are negligible, except for the use of a PoH with $LURK1.2$ in RSA mode.

In short, the measured impact of $LURK1.2$ introducing a Crypto Service and an Engine has a limited overhead over TLS in ECDHE mode, but not so for running TLS in RSA mode. This is expected given that $LURK1.2$ in RSA mode implied more changes to TLS1.2 in RSA mode (e.g., the use of the freshness function, more interaction between the Engine and the Crypto Service), whereas $LURK1.2$ in DHE mode is close to TLS1.2 in DHE mode.

4.1.2 On TLS modes

Clearly, the latency of $LURK1.2$ varies with the underlying TLS mode. In Figure 3a, for the particular case of local UDP connections, we note that $LURK1.2$ in RSA Extended has a 2.09-time greater latency than in RSA mode, whilst in ECDHE mode this is in 4.48-times greater than in RSA mode. From Figure 3a, we note that $LURK1.2$ in RSA Extended and ECDHE modes respectively require 1.93 and 3.71 more time than in RSA mode in case of UDP (non-local connections). Regarding TCP, $LURK1.2$ increases the times by a factor of 2.2 and 3.73, in RSA Extended and ECDHE modes, respectively. The difference between RSA and RSA Extended is due to the additional processing and communication of the full TLS handshake, while the difference between ECDHE and RSA is mostly due to the cryptographic operations involved[3] (e.g., more costly mathematical computations in ECHDE). To this end, we are aware of the fact that the parts of our implementation dealing with elliptic curves is sub-optimal, especially through the use of the `tinyec` [45] library. However, beyond performance degradation explained by as per the above, other factors need to be considered, such as the length of necessary data and the networking stack.

4.1.3 Other choices

Figure 3c shows the latency ratio of φ being set to SHA256 vs. φ being non-existent. The measured ratio is 1.016, 0.99 and 1.00 for RSA, RSA Extended and ECDHE modes, respectively. So, the impact of φ on the overall latency is negligible. Figure 3d depicts the RTT-degradation when TLS1.2's PRF is being set to SHA384 and SHA512, compared to the more-standard SHA256; this choice has negligible impact on the overall latency. Figure 3e shows that our added PoH has negligible impact (1.066) on RSA-Extended, given that a full handshake-transcript is already provided. In contrast, our added PoH increases the latency by 2.39 for RSA mode. However, note that the latency of $LURK1.2$ in RSA mode with added PoH is comparable to that of TLS 1.2 in RSA-Extended mode. Figure 3f depicts the impact of our added PoO of the DHE exponent over the average ECDHE latency. The impact observed is relatively negligible.

4.2 CPU Overhead

We load the Crypto Service with a rate of 100 requests per second, with a number of blocking clients operating in parallel. The results, shown in Fig. 4, confirm that the use of TLS over TCP has little impact on the performance of just TCP itself, which is due to an efficient TLS library.

HTTP and HTTPS seem to perform better than TCP, especially for $LURK1.2$ in RSA-Extended mode. A possible explanation is that HTTP libraries are optimized to handle input/output with the kernel and that some additional work may be needed when TCP is used. More specifically, the TCP implementation reads the LURK

[3]OpenSSL tests have always illustrated a significant latency overhead of TLS with ECDHE mode over TLS in RSA mode.

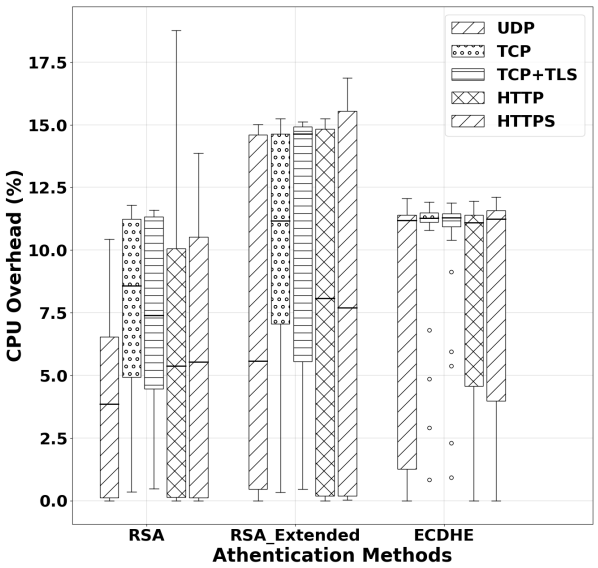


Figure 4. CPU Overheads of the Crypto Service in different $LURK1.2$ modes

Header to determine the length of the LURK request, and then reads the remaining bytes of the LURK request, increasing the interactions between the user and the kernel.

CPU consumption for $LURK1.2$ in ECDHE mode remained quite stable for different transport protocols. This is in part due to the fact that the additional processing required for handshakes is quite minimal compared to the cryptographic operations. But, processing the handshakes has a significant impact on the CPU overhead in the case of using $LURK1.2$ in RSA and RSA-Extended modes; heavier handshakes to process also explains the additional CPU overhead observed for $LURK1.2$ in RSA-Extended mode compared to RSA mode. Concretely, the Crypto Service in RSA-Extended mode requires 1.39 times more resources than for $LURK1.2$ in RSA mode. In ECDHE mode, it requires 2.08 times more than for $LURK1.2$ in RSA mode.

5 Formal Security Proofs & Analyses

The security analysis of $LURK1.2$ is in three parts. First, in Subsection 5.1, we present the formal security properties asked of $LURK1.2$. Then, in Subsection 5.2, we use formal security analysis to show that the freshness mechanism does indeed enforce forward secrecy in $LURK1.2$ in RSA mode. The third part, which contains cryptographic proofs for Variants1 and 2 of $LURK1.2$ in RSA mode and for Variant 1 of $LURK1.2$ in DHE mode, is given in the long version found at [27], due to space limitations.

5.1 $LURK1.2$'s Security Goals

TLS is a 2-party authenticated key exchange (AKE) protocol and LURK is a 3-party AKE protocol. The security of AKEs like TLS, i.e., AKEs with a key confirmation step, is formalised via the authenticated and confidential channel establishment (ACCE) model [21]. Meanwhile, [5] put forward *3(S)ACCE*, an ACCE-based model with formalisms and security requirements for "server-side delegated authenticated key-exchanges"; So, we now use the 3(S)ACCE model in order to describe LURK's cryptographic requirements.

Threat Model. ACCE models are session-based: i.e., Clients, Engines and Services are *parties* which have multiple *instances/sessions* running, and the security definitions rest on "agreements" and no "bad" event occurring in the interleaving of these sessions, even in the presence of an adversary. Our attacker is a 3(S)ACCE adversary who can compromise the LURK Engine, as well as the different end-points, i.e., the Client and the Crypto Service. Not all 3 parties can be compromised in the same LURK execution. The attacker also controls the network, within the realms of the type of channel (i.e., he cannot change a secure channel into an insecure one).

Security Requirements for $LURK1.2$.

Entity Authentication (EA) [5]. An *EA attacker* can corrupt parties (i.e., making them do arbitrary actions), open new sessions, probe the results of sessions, send its own messages. We say that there is a *EA attack* by an EA attacker if there exists a session of type X ending correctly, but there is no honest session of type Y that was started with X. In the above, X,Y can be either Client or Crypto Service and X is different from Y. In most cases, we are interested in the case where X is "Client" and Y is "Crypto Service", i.e., the EA views the authentication of the Crypto Service being forged to a given Client. We say a *server-side delegated authenticated key exchange achieves entity-authentication* if there is no EA attack onto the protocol.

Channel Security. We say a *server-side delegated authenticated key exchange achieves channel security* if no channel attacker can find the channel key of a session belonging to a party it did not corrupt. Notably, the attacker can corrupt a LURK Engine at a time t and thus learn its full state at that time, and use it henceforth to find the channel key of sessions that took place before time t; this type of attack is known as an attack against *perfect forward secrecy (PFS)*. It is well-known that TLS1.2 in RSA mode does not achieve perfect forward secrecy, and nor does [5] KeylessSSL [43] in RSA mode.

Accountability [5]. We say a *server-side delegated authenticated key exchange achieves accountability* if the Crypto Service is able to compute the channel keys used by the Client and the middle party, which in our case is the LURK Engine. This gives the Crypto Service powers to audit the activity of the LURK Service at the record layer, should this be required.

LURK Requirements. The $LURK1.2$ designs are expected to achieve channel security, entity authentication and accountability. We demand that the first two requirements hold for all $LURK1.2$ versions; meanwhile, we view accountability as an optional security requirement, which one can consider trading off for the sake of efficiency.

These requirements are stated as theorems and proven formally, in the 3(S)ACCE model, in the long version of the paper [27].

5.2 Symbolic Verification of $LURK1.2$ in RSA Mode.

Herein, we use the ProVerif [6] symbolic verifier to formally show that the bespoke way in which $LURK1.2$ in RSA mode introduces and uses the function φ as a freshness mechanism does indeed attain channel security *with perfect forward secrecy (PFS)*.

Weak LURK. To reach our goal, we also model and check a modified version of $LURK1.2$ in RSA mode, in which the freshness function is not present. In this version, the Engine chooses the nonce N_E directly and sends it to the client and the Crypto Service, without locally generating N_i and inputting it to the freshness function φ to compute N_E. These differences, which yield what we refer to as *"weak-LURK"*, are presented in Figure 5.

Symbolic Formalisation of $LURK1.2$'s Requirements. First, recall that if an attacker corrupting the Engine can get hold of the master secret msk from an old session and he has observed the handshake of said session, then the attacker can compute the channel key for that session. This would be failing the property of channel security with perfect forward secrecy. So, we need to formalise the aforementioned property of channel security (with perfect forward secrecy) in ProVerif. This would be expressed by encoding that a master secret msk cannot be learnt by an attacker who corrupts the Engine. More generally, in symbolic-verification tools, this would be a *secrecy* property, which allows one to verify that particular sensitive data are never inferred by the attacker in any protocol execution.

Second, we are also interested in seeing if ProVerif would find an actual replay attack whereby an attacker who corrupts the Engine learns not any msk but specifically an *old* msk. In ProVerif, this can be done via a *correspondence* property, which allows one to verify associations between stages in protocol executions, such as links between event occurring. That is, we formalise the verification of channel security (with perfect forward secrecy) via a secrecy property w.r.t. (old) master secrets, together with a correspondence property which checks if it is possible to re-query the Crypto Server on past cryptographic material such as old client or server random values. We check these properties in "weak LURK" vs (Variant 1 of) $LURK1.2$ in RSA mode, both encoded in ProVerif.

Symbolic Analysis of Channel Security with PFS in "Weak LURK". Our results show that "weak LURK" fails to achieve security against corrupted Engines performing replay attacks. Failure of the anti-replay properties implies perfect forward secrecy failure: acting as an Engine, the attacker is able to query Crypto Service and retrieve old master secrets. This is also confirmed by violating the secrecy property over the client master secret.

Symbolic Analysis of Channel Security with PFS in $LURK1.2$. As opposed to "weak LURK", $LURK1.2$ introduces the freshness mechanism. We model the freshness function φ as a pseudorandom function which cannot be inverted by the attacker. The results of the analysis show that, in $LURK1.2$, the nonce N_i can be accessed only by legitimate parties and as such an "old" N_i cannot be replayed to the Crypto Service.

This formally proves that $LURK1.2$ employing the freshness mechanism is resilient against replay attacks from corrupted Engines.

Link with KeylessSSL's Insecurities. As by product, our ProVerif-based demonstration that Weak LURK fails to ensure perfect forward secrecy (PFS) is also a new and automatic way of showing that KeylessSSL [43] in RSA mode has a replay attack and does not attain PFS; this was only shown with "pen and paper" before, in [5].

To this end, the two analyses above also prove that our freshness mechanism represents a viable alternative to the solution proposed in [5] to patch the KeylessSSL protocol's PFS problems (which was to have the end-server generate the server random for the middlebox).

Analysis of Channel Security with PFS: Summary. Table 1 gives details on the property-encoding and summarises the results of the verification. Our ProVerif code can be found at [34]. We also encoded Variant 1 of $LURK1.2$ in DHE mode and checked that this attains channel security. However, we elude the details of this verification here, due to space constraints.

6 Comparison with Alternative Approaches

6.1 Client-Invisible, Server-Controlled TLS Delegation

By server-controlled and client-invisible delegation, we mean that the TLS client is unaware of the middlebox and the latter is mandated/commissioned to delegate traffic by the TLS server.

Up to date, there are seven comprehensive such mechanisms, all used for or by CDNs. In Table 3, we summarise these as well as LURK, from the viewpoint of: a). the changes needed to the TLS Client; b). the important credentials over which the TLS Server (content owner) maintains control during the delegation; c) the ability of the TLS Server (content owner) to audit the delegated TLS session.

From LURK's stance, as we envisage this used with legacy clients, the TLS Client must not be updated. For security reasons, the ability to audit the middlebox is clearly also vital. As such, we see from Table 3, that LURK is a competitive solution on this desirable space of secure, backwards-aware TLS delegation.

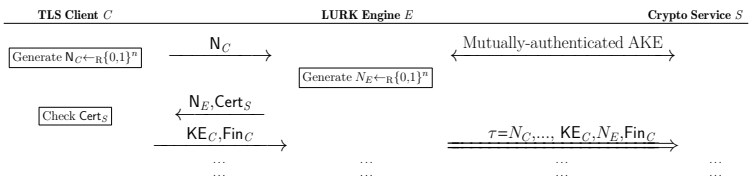


Figure 5. "Weak LURK": $LURK1.2$ in RSA mode stripped of the freshness mechanism

"Weak LURK", without the freshness mechanism			
Channel Security with PFS	**Code Excerpt**	**Result**	**Comments**
Secrecy	**query secret** *mastersecret* msk **query secret** *server_random* N_E	**False** **False**	Both properties fail. Attack: an attacker assuming the role of the Engine obtains an old master secret.
Antireplay / Correspondence	**query** *encpremaster* ,*srvrand* N_E :bitstring; **inj-event** keyserver_received_encpremaster on *encpremaster*,*srvrand* $N_E \Longrightarrow$ **inj-event** edge_resent_encpremaster on *encpremaster*,*srvrand* N_E.	**False**	The query asserts that reception by the Crypto Service of a distinct pair of encrypted premaster secret with a server random N_E occurs only once in the same protocol run. The property fails. Attack: trace showing a replay of the same encrypted premaster $Enc(\mathsf{pmk})$ and same server random N_E in two distinct instances of the Crypto Service process.
$LURK1.2$ **in RSA mode (Variant 1 encoded), with freshness mechanism**			
Channel Security with PFS	**Code Excerpt**	**Result**	**Comment**
Secrecy	**query secret** *clearserverrand* N_i	**True**	This server random N_i may be accessed only by legitimate parties.
Antireplay / Correspondence	**query** *srvRand* N_i:bitstring; **inj-event** keyserver_recvd_srvrnd_clear on *srvRand* N_i $\Longrightarrow$ **inj-event** edge_sent_srvrnd_clear on *srvRand* N_i.	**True**	The query asserts that reception by the Crypto Service of one given server random N_i occurs only once, as a result of an Engine transmitting this value. The property holds, guaranteeing that queries to the Crypto Service with old N_is are not possible.

Table 1. ProVerif Analysis of Channel Security of "Weak LURK" and $LURK1.2$ in RSA Mode

We now detail Table 3. Liang et al. [25] show that CDN providers are depending on TLS delegation, yet that TLS delegation is not appropriately handled. To this end, Liang et al. measured that 19 out of 20 CDNs and found that only 31.2 % of web sites on CDN using HTTPS present a Valid Certificate. Nonetheless, we recount all TLS delegation mechanisms for CDN-ing.

First, Liang et al. [25] showed that *sharing the private key* and names is a common practice for CDNs. In this way, the content owner gives up all its identity credentials with no control over them, which constitutes an obvious security threat.

Second, another way to provide collaborative delivery is to delegate using certificates or equivalent. The content owner may issue a signing intermediary CA to delegate the emission of keys owned by the CDN associated to the content owner name. However, *the X.509 Name Constraints* certificate extension [7] does not apply to the Subject Alternative Name (SAN), but only to the Common Name (CN) in the Distinguished Name (DN) while certificate validation considers DNS names in DN/CN or in SAN. As a result, there is not enough control in the certificates that may be validated.

Third, one has the option of *delegated credentials* [3]. This is similar in essence to certificate delegation but the delegation is performed on a TLS specific structure and validated by the Client. Client integration does not make the mechanism viable for legacy TLS 1.2 Clients.

Fourth, *Short-Term, Automatically-Renewed (STAR)* certificates [38], [37] describes a method where the domain name owner or the content owner authorides the Certificate Authority (CA) to renew the certificate when requested by the CDN - using using ACME [2]. For both Delegated credentials and STAR, the content-owner will regain control of the identity/ credentials after the delegation expires, however, during the delegation, the content-owner has little control or audit-powers over the CDN machines.

Fifth, the *DANE* design [17] takes advantage of DNSSEC to provide keys used to establish the TLS session. Although an elegant solution, there is currently not enough support for DANE by browser vendors.

Sixth, Gilad et al. [15] and Levy et al. [24] present an alternative, called *Stickler*, which involves decryption by the browser, that is at the application layer. With Stickler, upon downloading the home page, the content-origin provides a Loader. The Loader is sent over the secure TLS channel and can retrieve the JavaScript (RootJS) from the proxy, validating the software. The software is then

able to retrieve the signed objects from the mirror and checks them.

Seventh, KeylessSSL [5] is said –by their proprietors Cloudflare– not perform delegation but split the TLS into services and provide the ability for the content owner to keep the control of the identity credentials while other part of the delivery is let to the CDN. As no changes are required on the Client, it can be of use with legacy devices. But, in 2017, Bhargavan et al. [5] used a provable-security approach to show several vulnerabilities on KeylessSSL; they also advanced an alternative design, called *3(S)ACCE-K-SSL*, that provably achieve stronger security goals, albeit via a much less efficient design.

6.2 Client-visible TLS Delegation

On the client side, CDN-ing (as per the above) is not explicitly signalled. In other words, the CDN provider is assimilated to the content owner from the client's perspective. While this might be acceptable with one-to-one configurations, automated CDN collaborations like those envisioned by CDNI [9] seem to introduce a federated platform for content where the TLS termination is hardly controlled by the Client or the content owner, but where the TLS communication is composed of multiple intermediaries. In this context, the client and content owner may be willing to have a closer view on the different intermediaries. Indeed, multiple initiatives have been taken to have middleboxes partake in TLS sessions with an explicit agreement and negotiation of all parties involved [29], [32]. We recount these initiatives below.

First, *SplitTLS* [41] is commonly seen as the simplest architecture where the middlebox impersonates the endpoint. The client side requires to trust the root certificate of the middlebox that impersonates all servers. On the server side, such architecture could be interpreted as a TLS front end or a security gateway.

Second, *Explicit Trusted Proxy* [26] moves a step forward and lets the client indicate the use of a proxy but did not provide additional control on the proxy.

Third, *TLS ProxyInfo* [28] and *TLS Keyshare Extension* [31] ensure that both endpoints are aware of the existence of the proxy, while enabling client to authenticate the server. Yet, arguably, a shared key does not provide sufficient accountability or control on what is actually performed by the middlebox.

Fourth, *multi-context TLS (mcTLS)* [30] allows for the endpoints and middboxes to establish different access-level keys (read/write keys) per middlebox and per different data-fragments (e.g., HTTP headers, body).

Fifth, [4] showed mcTLS to be insecure and proposed a new provably-secure but less efficient design, in the same vain of visible and accountable proxying over TLS.

Sixth, *Transport Layer Middlebox Security Protocol (TLMSP)* [12] also improves on mcTLS by adding more measures to evaluate the transformations on data performed by each middlebox. Yet this design does not enable incremental deployment.

Mechanism	Auth	Content	E2E	Impact on TLS
Split TLS (client)	–	–	–	Root Cert.
Split TLS (server)	–	–	–	
Explicit Trusted Proxy	–	–	–	TLS ext.
TLS ProxyInfo	x	–	x	TLS ext.
TLS Keyshare	x	–	x	TLS ext
mcTLS	x	data, action (read, write)	x	New setup
TLMSP	x	data, actions, path order, modification	x	New setup
mbTLS	x	TEE	x	New setup
maTLS	x	certification	x	New setup,
Blindbox, Embark	x	encryption	x	New setup

Table 2. Mechanisms for Client-Visible TLS Delegation

Seventh, *Middlebox TLS (mbTLS)* enables middleboxes to leverage SGX to attest processing performed by them, while *middlebox aware TLS (maTLS)* [23] uses a specific certification model.

Eighth, *BlindBox* [39] and *Embark* [22] adopt a different approach where middleboxes operate over encrypted content.

Table 2 recounts most of the aforementioned initiatives w.r.t the main challenges each attempts to overcome: 1). the ability to authenticate end points as well as middleboxes (Auth); 2). the ability to restrict or control operations performed by each intermediary node (Content); 3). the ability for one endpoint to evaluate the overall security of the channel (E2E). Ensuring these capabilities impact the complexity of the establishment of the TLS session; this determines whether it can be implemented via a TLS extension (TLS ext.), like LURK is, or via more complex settings (New setup).

Mechanism	Changes to TLS client	Content-Owner Controls	Delegated-Session Audit by Content-Owner
Shared long-term private key [25]	–	–	–
X.509 Name Constraints [7]	X.509 parsing	long-term private key, name	–
Delegated Credentials [3]	TLS ext.	name	–
STAR [38]	–	name	–
DANE [17]	DNSSEC	name	–
Stickler [24]	browser plugin	long-term private key, name, content	–
KeylesSSL [43]	–	long-term private key, name	–
3(S)ACCE-K-SSL [5]	–	long-term private key, name	yes
LURK	–	long-term private key, name	yes

Table 3. Client-Invisible, Server-Controlled TLS Delegation for CDNs

6.3 Keyless SSL [43] and 3(S)ACCE-K-SSL [43]

As we can see from Table 3, LURK aligns itself most with KeylessSSL [43] and 3(S)ACCE-K-SSL [5]. In fact, these CDN-driven architectures appeared first in a patent by Akamai [14]: i.e., TLS-delegation systems where the TLS long-term private key stays on the server-side and the

associated certificate goes with the middlebox, who can therefore impersonate the server in a way invisible to the client. And, in 2015, Cloudflare commercialised a version of this, in a product called *KeylessSSL* [43]. However, KeylessSSL obviously required modifications to the *TLS handshake* (i.e., the secure-channel establishment part of TLS). Also, the resulting three-party "TLS-like" protocol arguably raises questions w.r.t. what it should guarantee and what it does actually guarantee. To this end, in 2017, Bhargavan et al. [5] used a provable-security approach to show several vulnerabilities on KeylessSSL: e.g., forward-secrecy attacks, signing oracle attacks or cross-protocol attacks, etc. So, Bhargavan et al. [5] advanced an alternative design, called *3(S)ACCE-K-SSL*, that provably achieves stronger security goals than KeylessSSL, albeit with reduced design-efficiency.

By cherry-picking just the security guarantees achievable in the real-world[4] and by some different design choices[5], LURK offers a more efficient design than 3(S)ACCE-K-SSL. Concretely, just to fix the forward-secrecy attack in KeylessSSL, the 3-(S)ACCE-K-SSL design requires 3 RTTs, which is prohibitive for the provided benefit. LURK addresses this concern by providing similar level of security with a single RTT. What is more, unlike 3(S)ACCE-K-SSL, we implemented and extensively tested LURK's performance, to aid further still with particular option/implementation choices.

Note that 3-(S)ACCE-K-SSL aims to achieve a strong property called content-soundness, for which they require one certificate per every content-unit (e.g., 1 HTML page, 1 HTTP header, etc.) delivered. This is arguably un-achievable in real life. Yet, the content-soundness property is interesting in that it cryptographically certifies each content-unit that the middlebox is allowed to deliver; but, in practice, the solutions are weaker, based on CDN configurations and access-control policies.

Last but not least, we offer several variants of LURK, each with different options (e.g., $LURK1.2$ Variant 1 can support session ID resumption if need be, whereas $LURK1.2$ Variant 2 does not and does attain accountability like 3-(S)ACCE-K-SSL). To this end, it could be considered that our LURK Variant 1 is a secure version of KeylessSSL, whereas our LURK Variant 2 is an even more secure, being real-life alternatives to 3-(S)ACCE-K-SSL.

Here are some further comparisons with KeylessSSL.

On Decrypting Oracles When RSA authentication is used by the TLS client to authenticate the TLS server, the TLS client provides an encrypted premaster secret that is decrypted by the TLS server and used to generate the master secret. With KeylessSSL, the decryption is performed by the Crypto Service, while the computation of the master secret is performed by the Engine[6]. Such design makes the key server subject to chosen ciphertext attack as an attacker that gains access to the Crypto Service is able to gather information by obtaining the decryption of chosen ciphertexts. Opening such attacks raises at least three concerns: 1) placing the Crypto Service under a DoS-like attack, 2) opening the Crypto Service to cryptanalysis attacks (cipher text attacks) to recover the key, 3) opening the Crypto Service to replay attacks with old premaster and compromise an old session. These problems were signaled and discussed at length in [5].

With LURK, even in its weaker version 1, the Crypto Service outputs the master secret which prevents cipher text attack. In addition, a freshness mechanism protects LURK against replay attacks. As a result, the Crypto Service makes these attacks unfeasible by design.

In $LURK1.2$ we implement two ways for the Crypto Service to generate the master secret: RSA and Extended RSA. This makes LURK a bit more complex than KeylessSSL, but we do avoid the aforementioned worries w.r.t. KeylessSSL's Crypto Service returning the premaster secret.

As per the above, the LURK Engine needs to send different parameters depending on the authentication method used to establish the TLS session. In the case of RSA Extended, the full handshake needs to be provided which results in a significant increase of the payload. As per Section 4, hen the Crypto Service and the LURK Engine are located in the same data center, in the worst case, RSA extended mode has 2.09-time greater latency than in RSA mode. With TCP_TLS, this means an observed latency is 20 ms, which is unlikely to be perceived by the end user.

When the Crypto Service and the LURK Engine are not located in the same data center, the experimented latency will be the one associated between the two sites. While RSA and Extended RSA will experiment that same propagation time that is directly associated to the distances, they will experiment a difference depending of the transmission which is directly related to the capacity of the link. In the light of KeylessSSL measurements provided by Cloudflare [44] between a data center in London and San Francisco the observed propagation can be roughly be estimated to 10,840 ms. A low entry bar of 10 Gbps inter-data center connectivity would make a few additional kilo bytes unnoticed. As a potential drawbacks introduced by LURK over KeylessSSL seems insignificant compared to the additional security provided by LURK.

On LURK's Proof of Handshake The proof of handshake consists in attesting that the LURK request occurs in a context of a TLS exchange which is based on the computation of the Finished message. In TLS RSA mode, the handshake-exchange can be entirely computed by an attacker acting as both the TLS client and the TLS server. To this end, the PoH also proves that the TLS client knows

[4] We do not require 3(S)ACCE-K-SSL's content-soundness.

[5] To add PFS to $LURK1.2$ in RSA mode, we do not run the whole handshake on behalf of the Engine (which 3(S)ACCE-K-SSL did to repair KeylessSSL).

[6] In KeylessSSL, the terminology is not that of a Crypto Service and an Engine, but we use this for easiness.

	RTT	PFS/Anti-Replay	PoO	PoH	Decrypting-oracle protection	Signing-oracle Protection
LURK 3(S) ACCE-K-SSL	1 3	ϕ 2-ACCE	yes no	yes yes	yes yes	yes yes
Keyless SSL	1	–	no	–	no	no

Table 4. Novelty of LURK versus 3(S) ACCE-K-SSL and KeylessSSL

the premaster secret, which makes such an attack meaningless, without any need to decrypt the premaster secret.

On Signing Oracles ECDHE (Elliptic Curve Diffie Hellman) authentication in TLS includes the following operations: 1) generation of some ingredients including the public part of the TLS-server DH public key and the TLS randoms; 2) these ingredients are hashed and then signed. With KeylessSSL only the signing operation is performed by the Crypto Service. Because the Crypto Service receives the output of the hash, it is not able to check whether it is actually signing TLS parameters of a given session. Instead it is blindly signing some random bits of a given length. This is discussed in [5].

Unlike in KeylessSSL, the LURK's Crypto Service performs both the hashing and the signing operation. I.e., it is the ingredients of the hash and not the hash that the LURK Engine provides to the Crypto Service. This enables the Crypto Service to check the input parameters and validate these parameters. E.g., typically only specific curves may be provided. Upon receiving a public value, the Crypto Service needs to check the public value is on the specified curve.

To enforce further checks, LURK adds also the Proof of Ownership (PoO) that proves the knowledge by the LURK Engine of the private key.

So, like in RSA mode, in DHE mode LURK provides larger parameters to the Crypto Service than KeylessSSL does. Yet, for the same reasons as above, the efficiency impact visible to the end-user is limited and do not overweight the extra security provided by LURK.

The table below summarises the comparisons made between these 3 designs

7 Conclusions & Future Work

Our suite of designs, called $LURK1.2$, aim to offer *provably secure* server-controlled TLS delegation, in a manner that achieves competitive performance. Our drive for this was motivated in real-life use-cases calling for server-controlled TLS delegation, such as complex CDN-delegations and service-to-service platforms. On the one hand, one can see LURK as a way to better the security of KeylessSSL [43], in a spirit similar to that of the recent 3(S)ACCE-KSL protocol in [4]. On the other hand, unlike the 3(S)ACCE-KSL scheme, we do not require that LURK attains the expensive, content-soundness requirement w.r.t. TLS-delegation, which –in turn– does away with the need for an arguably infeasible PKI infrastructure. Meanwhile, in some of its variants, LURK attains all other relevant security requirements of 3(S)ACCE-KSL, i.e., channel security, entity authentication and accountability; for these, in the long version [27], we provide cryptographic proofs in a suited 3-party authenticated key-exchange formal model. Moreover, we use protocol-verification (in ProVerif) to show that design-mechanisms that specifically separate LURK from KeylessSSL while achieving their intended, specific goals, i.e., enforce forward secrecy. Our studies focus on LURK instantiated with TLS 1.2, as this is still the most widely used version of TLS and will likely remain so for longer, especially for legacy devices. Our specifications go down to the API level, providing details down to network and packet level for the communications within the TLS delegation. This delegation, in LURK, is envisaged as a modular design, where the middle entity and the end-server operate in a service-to-service fashion. Lastly, our Python implementation and performance-testing of LURK show that it is a competitive solution for TLS-delegation. All in all, in this paper, our LURK constructions show that server-controlled TLS delegation is possible with both provable guarantees of real-world security and competitive efficiency.

W.r.t. future directions, we are actively working towards LURK based on TLS 1.3 [11]. In the long version of this paper [27], there are more details on this.

Also, the primary objective of our implementation, `pylurk`, was to build an initial testbed. Immediate future work involves, for instance, the extension of the interface to gRPC to better fit containerised environments. In addition, the integration of Curve25519 and Curve448 for both signatures (Ed25519, Ed448) as well as ECDHE (X25519, X448) are expected to be supported. One parallel line focuses on a C implementation of the Crypto Service, in line with the most notable TLS libraries.

A $LURK1.2$ in DHE mode –Variant2

In this section, we present Variant2 of $LURK1.2$ in DHE mode. This is in fact the same as the 3(S)ACCE-K-SSL design [5], with the exception that we do not require that each fragment of data delivered by the Engine be certified by a X.509 certificate. As such, our handshake is lighter with fewer verifications, but our design cannot achieve the content-soundness property that 3(S)ACCE-K-SSL can achieve. However, due to the similarities mentioned, we do not include cryptographic proofs for Variant2 of $LURK1.2$ in DHE mode, as for channel security, entity authentication and accountability these would be same as for the 3(S)ACCE-K-SSL design in [5].

B TLS 1.2

We show the TLS1.2 handshake (i.e., the secure-channel establishment part of TLS 1.2) in Figure 7.

TLS 1.2 Handshake in RSA Mode. The client C sends a nonce N_C to the server S, which responds with its own

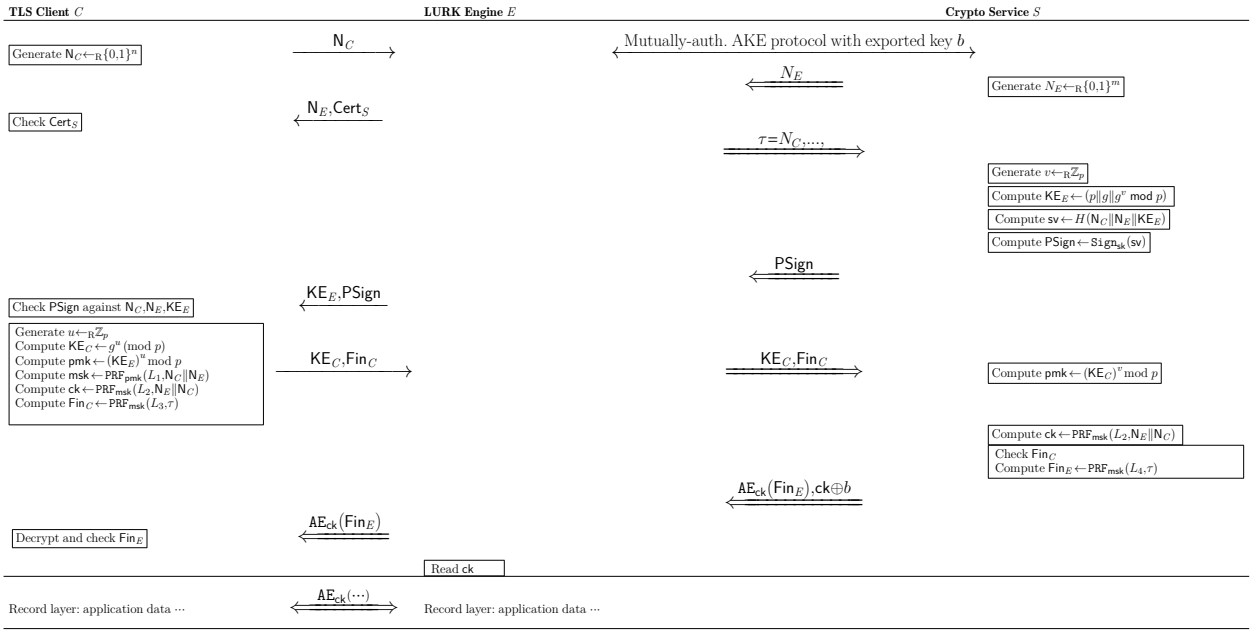


Figure 6. *LURK*1.2 based on TLS1.2 in DHE mode: Variant 2; in line with the 3(S)ACCE-K-SSL design [5]

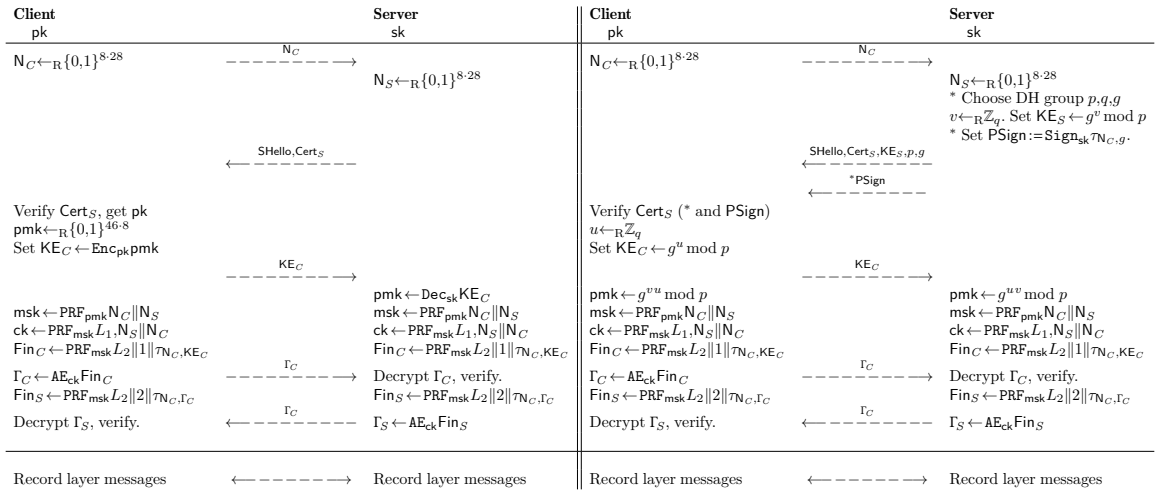


Figure 7. TLS1.2 handshake. Left: in RSA mode; Right: in DHE mode.

nonce N_S and a certificate Cert_S containing an RSA public key. The client then generates and sends a *pre-master secret* pmk encrypted under the server's public key. The server decrypts pmk and the client and server both compute a *master secret* msk using pmk and the two nonces. To complete the handshake, both client and server use msk to MAC the full handshake transcript and send (in an encrypted form) these MACs to each other in *finished messages* (Fin_C,Fin_S). At the end of the handshake, both client and server derive channle keys ck from msk and the two nonces, used for authenticated encryption of record-layer data.

TLS 1.2 in RSA mode does not provide *forward secrecy*, which means that if an adversary records a TLS1.2-RSA connection and later compromises the server's private key, it can decrypt the pmk, derive the connection keys, and read old application data.

TLS 1.2 Handshake in RSA Mode. The client and server first exchange nonces and the server certificate as in RSA mode. Then the server chooses a Diffie-Hellman group p,q,g (represented by an elliptic curve or by an explicit prime field) and generates a keypair (v,$g^v \bmod p$). It signs the nonces, the group, and its Diffie-Hellman public value with its certificate private key and sends them to the client, which then generates its own keypair (u,$g^u \bmod p$). Both

client and server then compute the pre-master secret pmk as $g^{uv} \bmod p$. The rest of the protocol and computations (msk,ck,Fin_C,Fin_S) proceed as in the RSA mode.

C The 3(S)ACCE Model, Other Security Details

C.1 The (S)ACCE Models [21]

We briefly describe the authenticated and confidential channel establishment (ACCE) security model. We use the notations Brzuska et al. [8].

Parties and instances. The ACCE model considers a set $\mathscr{P}$ of parties, which can be either *clients* $C \in \mathscr{C}$ or *servers* $S \in \mathscr{S}$. Parties are associated with private keys sk and their corresponding, certified public keys pk. The adversary can interact with parties in concurrent or sequential executions, called sessions, associated with single party *instances*. We denote by π_i^m the m-th instance (execution) of party P_i. Each instance is associated with the following attributes:

- the instance's **secret**, resp. **public keys** $\pi_i^m.\mathsf{sk} := \mathsf{sk}_i$ and $\pi_i^m.\mathsf{pk} := \mathsf{pk}_i$ of P_i. In unilaterally-authenticated handshakes, clients have no such parameters, thus we set $\pi_i^m.\mathsf{sk} = \pi_i^m.\mathsf{pk} := \perp$.
- the **role** of P_i as either the *initiator* or *responder* of the protocol, $\pi_i^m.\rho \in \{init,resp\}$.
- the **session identifier**, $\pi_i^m.\mathsf{sid}$ of an instance, set to $\perp$ for non-existent sessions.
- the **partner identifier**, $\pi_i^m.\mathsf{pid}$ set to $\perp$ for non-existent sessions. This attribute stores either a party identifier P_j, indicating the party that P_i believes it is running the protocol with (in unilateral authentication, clients are associated with a label "Client").
- the **acceptance-flag** $\pi_i^m.\alpha$, originally set to $\perp$ while the session is ongoing, but which turns to 1 or 0 as the party accepts or rejects the partner's authentication.
- the **channel-key**, $\pi_i^m.\mathsf{ck}$, which is set to $\perp$ at the beginning of the session, and becomes a non-null bitstring once π_i^m ends in an accepting state.
- the **left-or-right** bit $\pi_i^m.\mathsf{b}$, sampled at random when the instance is generated. This bit is used in the key-indistinguishability and channel-security games.
- the **transcript** $\pi_i^m.\tau$ of the instance, containing the suite of messages received and sent by this instance, as well as all public information known to all parties.

The definition of ACCE security heavily relies on the notion of *partnering*. Two instances π_i^m and π_j^n are said to be **partnered** if $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid} \neq \perp$.

Games and adversarial queries. In ACCE security, the adversary interacts with parties by calling *oracles*. It can generate new instances of P_i by calling the $\mathsf{NewSession}(P_i,\rho,pid)$ oracle. It can send messages by calling the $\mathsf{Send}(\pi_i^m,M)$ oracle. It can learn the party's secret keys via $\mathsf{Corrupt}(P_i)$ queries, and it can learn channel keys (for accepting instances) by querying $\mathsf{Reveal}(\pi_i^m)$. A $\mathsf{Test}(\pi_i^m)$ query outputs either the real channel keys $\pi_i^m.\mathsf{ck}$ computed by the accepting instance π_i^m or random keys of the same size. As opposed to standard AKE security, in the ACCE game, the adversary is also given access to two oracles, $\mathsf{Encrypt}(\pi_i^m,l,M_0,M_1,H)$ and $\mathsf{Decrypt}(\pi_i^m,C,H)$, which allow some access to the secure channel established by two instances. The output of both these oracles depends on the hidden bit $\pi_i^m.\mathsf{b}$ for any instance π_i^m.

The adversary's *advantage* to win is defined in terms of its success in two security games, namely *entity authentication* and *channel security*, the latter of which is subject to the following freshness definition.

Session freshness. A session π_i^m is *fresh* with intended partner P_j, if, upon the last query of the adversary $\mathscr{A}$, the uncorrupted instance π_i^m has finished its session in an accepting state, with $\pi_i^m.\mathsf{pid} = P_j$, for an uncorrupted P_j, such that no Reveal query was made on π_i^m,π_j^n.

ACCE Entity Authentication (EA). In the EA game, the adversary queries the first four oracles above and its goal is to make one instance, π_i^m of an uncorrupted P_i *accept maliciously*. That is, π_i^m must end in an accepting state, with partner ID P_j, also uncorrupted, such that no other unique instance of P_j partnering π_i^m exists. The adversary's advantage in this game is its winning probability.

ACCE Security of the Channel (SC). In this game, the adversary $\mathscr{A}$ can use all the oracles except Test and must output, for a fresh instance π_i^m, the bit $\pi_i^m.\mathsf{b}$ of that instance. The adversary's advantage is the absolute difference between its winning probability and $\frac{1}{2}$.

Mixed-ACCE Entity Authentication (mEA) [5]. In the mEA game, specific to proxied AKE, the adversary queries the first four oracles above and its goal is to make one instance, π_i^m of an uncorrupted P_i *accept maliciously*. That is, π_i^m must end in an accepting state, with partner ID P_j also uncorrupted, such that no other unique instance of P_j partnering π_i^m exists. Furthermore, let flag_i^m denote the mode-flag for the instance π_i^m. Furthermore, if $\mathsf{flag}_i^m = 0$, then P_i *must* be a client only. The adversary's advantage in this game is its winning probability.

C.2 The 3(S)ACCE Model [5]

In [5], an adaptation of the (S)ACCE model for 3 parties was introduced. It was called 3(S)ACCE and it covers also the case where a middle party collaborates in the exchange as per LURK, that is in a server-mandated manner. However, 3(S)ACCE also covers the case where the client is aware of the middle party. This does not concern the case of LURK.

3(S)ACCE introduces several new notions compared to ACCE which are instrumental in the formalisation: pre-channel keys (i.e., the equivalent in pmk in TLS), modification/additions of ACCE attributes (e.g., the partner

attribute returns as set of instances), new notion of freshness for sessions, new adversarial oracles to account for the corruption of the middle party, etc. Some of these directly view the 3(S)ACCE security notion of content soundness that does not concern us.

3(S)ACCE Partnering. One essential modification from the (S)ACCE model to the 3(S)ACCE is concerning the notion of partnering of sessions. We do not detail all the intricacies of 3(S)ACCE partnering, but we summarise its crux. For LURK, there are 4 instances of parties that form one partnering: a Client instance, one Engine instance (for the left-side communication), another Engine instance (for the right-side communication), a Service instance. This type of partnering, allows [5] to re-use 2-party security definitions for authentication and channel security.

Accountability is a new security notion introduced in [5] specifically for server-mandated collaborative delivery. Since the Client has no way of distinguishing the Engine from the Service, the Service is given enough cryptographic material of the handshake such that it is able to audit the secure channel established between the Client and the Engine. The aim is that in this way one makes sure that the Engine is unable to "hurt" the Client.

Without giving details of all of the oracles (as they will be clear from the context and the ACCE definition above), we do re-count below all the 3(S)ACCE security definitions that concern us.

Main 3(S)ACCE Security Definitions [5].

Entity Authentication (EA) [5]. In the *entity authentication game*, the adversary $\mathscr{A}$ can query the new oracle $\mathsf{RegParty}$ and traditional 2-ACCE oracles. Finally, $\mathscr{A}$ ends the game by outputting a special string "Finished" to its challenger. The adversary *wins* the EA game if there exists a party instance π_i^m *maliciously accepting* a partner $P_j \in \{\mathscr{S},\mathscr{E}\}$, according to the following definition.

Definition 1 (Winning condition – EA game)**.** *An instance π_i^m of some party P_i is said to* maliciously accept *with partner $P_j \in \{\mathscr{S},\mathscr{E}\}$ if the following holds:*

- $\pi_i^m.\alpha = 1$ *with* $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name} \neq$ "Client"*;*
- *No party in* $\pi_i^m.\mathsf{PSet}$ *is corrupted, no party in* $\pi_i^m.\mathsf{InstSet}$ *was queried in* Reveal *queries;*
- *There exists no unique* $\pi_j^n \in P_j.\mathsf{Instances}$ *such that* $\pi_j^n.\mathsf{sid} = \pi_i^m.\mathsf{sid}$*;*
- *If* $P_i \in \mathscr{C}$*, there exists no party* $P_k \in \mathscr{E}$ *such that:* $\mathsf{RegParty}(P_k,\cdot,P_j)$ *has been queried, and there exists an instance* $\pi_k^\ell \in \pi_i^m.\mathsf{InstSet}$.

The adversary's advantage, denoted $\mathsf{Adv}_\Pi^{\mathsf{EA}}\mathscr{A}$, is defined as its winning probability *i.e.*:

$$\mathsf{Adv}_\Pi^{\mathsf{EA}}\mathscr{A} := \mathbb{P}\mathscr{A} \text{ wins the EA game},$$

where the probability is taken over the random coins of all the N_P parties in the system.

Channel Security (CS) [5]. In the *channel security game*, the adversary $\mathscr{A}$ can use all the oracles (including $\mathsf{RegParty}$) adaptively, and finally outputs a tuple consisting of a fresh party instance π_i^j and a bit b'. The winning condition is defined below:

Definition 2 (Winning Conditions – CS Game)**.** *An adversary $\mathscr{A}$* breaks the channel security *of a* 3SACCE *protocol, if it terminates the channel security game with a tuple $\left(\pi_i^j, b'\right)$ such that:*

- π_i^m *is fresh with partner* P_j*;*
- $\pi_i^m.\mathsf{b} = b'$.

The advantage of the adversary $\mathscr{A}$ is defined as follows:

$$\mathsf{Adv}_\Pi^{\mathsf{SC}}\mathscr{A} := |\mathbb{P}\mathscr{A} \text{ wins the SC game} - \frac{1}{2}|,$$

where the probability is taken over the random coins of all the N_P parties in the system.

Accountability (Acc) [5]. In the *accountability* game the adversary may arbitrarily use all the oracles in the previous section, finally halting by outputting a "Finished" string to its challenger. We say $\mathscr{A}$ *wins* if there exists an instance π_i^m of a client P_i such that the following condition applies.

Definition 3 (Winning Conditions – Acc)**.** *An adversary $\mathscr{A}$* breaks the accountability *for instance π_i^m of $P_i \in \mathscr{C}$, if the following holds simultaneously:*

(a) $\pi_i^m.\alpha = 1$ *such that* $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ *for an* uncorrupted $P_j \in \mathscr{S}$*;*
(b) There exists no instance $\pi_j^n \in P_j.\mathsf{Instances}$ *such that* $\pi_j^n.\mathsf{ck} = \pi_i^m.\mathsf{ck}$*;*
(c) There exists no probabilistic algorithm Sim *(polynomial in the security parameter) which given the view of P_j (namely all instances* $\pi_j^n \in P_j.\mathsf{Instances}$ *with all their attributes), outputs* $\pi_i^m.\mathsf{ck}$.

The adversary's advantage is defined as its winning probability, *i.e.*:

$$\mathsf{Adv}_{v2-LURK1.2-RSA}^{\mathsf{Acc}}\mathscr{A} := \mathbb{P}\mathscr{A} \text{ wins the Acc game},$$

where the probability is taken over the random coins of all the N_P parties in the system.

D LURK's Cryptographic Analysis

In what follows, we state our provable-security results w.r.t. $LURK1.2$. Using the 3(S)ACCE model in [5], we present the formal theorems and proofs of these statements in Appendix C.

Entity-Authentication Result. If TLS 1.2 is secure w.r.t. unilateral entity authentication, if the protocol between the Engine and the Service is a secure AKE protocol with exported keys indistinguishable from random, if the two protocols ensure mixed entity authentication, if the signature and hash in TLS1.2 DHE mode are secure in their threat models and, respectively, if the encryption in TLS1.2 RSA mode is secure , then Variant 1 of $LURK1.2$ in DHE mode and respectively, Variants 1 and 2 of $LURK1.2$ in RSA mode are entity-authentication secure in the 3(S)ACCE model.

This is formalised and proven in Theorem 1 in Subsection D.1.

Channel Security Result. If TLS 1.2 is secure w.r.t. unilateral entity authentication, if the protocol between the Engine and the Service is a secure AKE protocol with exported keys indistinguishable from random, if the two protocols ensure mixed entity authentication, if the signature in TLS1.2 DHE mode is secure in its threat models plus, respectively, if the encryption in TLS1.2 in RSA mode is secure and the freshness function is a PRF, then Variant 1 of $LURK1.2$ in DHE mode and, respectively, Variants 1 and 2 of $LURK1.2$ in RSA mode attain channel security in the 3(S)ACCE model.

This is formalised and proven in Theorem 2 in Subsection D.1.

Accountability Result. If TLS 1.2 is secure w.r.t. unilateral entity authentication, if the protocol between the Engine and the Service is a secure AKE protocol with exported keys indistinguishable from random, if the two protocols ensure mixed entity authentication, and the freshness function is a PRF, then Variant 2 of $LURK1.2$ in RSA mode attains accountability in the 3(S)ACCE model.

This is formalised and proven in Theorem 3 in Subsection D.1.

D.1 Cryptographic Proofs

. We now present different security proofs for LURK, using the 3(S)ACCE model in [5] and recalled in Section C.

D.2 Entity Authentication Proofs

We now prove the entity-authentication security of $LURK1.2$ in all variants.

Theorem 1. *Let P be the unilaterally-authenticated TLS 1.2 handshake (as seen by the Client) and P' be the AKE protocol between Engine and Crypto Service which, at each session, exports a key k indistinguishable from random. Assume that P and P' are together mEA-secure.*

We denote by $\mathsf{n_P}$ the number of parties in the system.

Consider a (t,q)-adversary $\mathscr{A}$ against the EA-security of the protocol $LURK1.2$, running at most t queries and creating at most q party instances per party, where $\mathscr{A}$'s advantage is written $\mathsf{Adv}^{\mathsf{EA}}_{\Pi}\mathscr{A}$.

If such an adversary exists, then there exist adversaries $\mathscr{A}_1$ against the SACCE security of P, $\mathscr{A}_2$ against the ACCE security of P', $\mathscr{A}_3$ against the mEA security of P and P', $\mathscr{A}_4$ against the AKE security of P' with exported key k, $\mathscr{A}_5$ –in the TLS-DHE mode– against the existential unforgeability (EUF-CMA) of the signature used to generate PSign and $\mathscr{A}_6$ against the hash function H, or $\mathscr{A}_7$ –in the TLS-RSA mode– against the channel security of P, each adversary running in time $t' \sim O(t)$ and instantiating at most $q' = q$ instances per party, such that

- *For Variant1 of $LURK1.2$ in DHE mode:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{EA}}_{\Pi}\mathscr{A} \;\leq\;& 2\mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2 + \\ & 2\mathsf{n_P}^3\cdot\mathsf{Adv}^{mEA}_{P,P'}\mathscr{A}_3 + \\ & \mathsf{n_P}\cdot\mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}\mathscr{A}_5 + \mathsf{n_P}\cdot\mathsf{Adv}^{\mathsf{Coll.Res}}_{H}\mathscr{A}_6 + \\ & \mathsf{n_P}^3\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2 + 2\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_4.\end{aligned}$$

- *For Variants1 and 2 of $LURK1.2$ in RSA mode:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{EA}}_{\Pi}\mathscr{A} \;\leq\;& 2\mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2 + \\ & 2\mathsf{n_P}^3\cdot\mathsf{Adv}^{mEA}_{P,P'}\mathscr{A}_3 + \\ & \mathsf{n_P}^2\cdot\mathsf{Adv}^{\mathsf{SC-SACCE}}_{P}\mathscr{A}_7 \\ & \mathsf{n_P}^3\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2 + \\ & \mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_4\end{aligned}$$

Proof. Our proof has the following hops:

Game $\mathbb{G}_0$: This game works as the EA-game recalled in Section C.

Game $\mathbb{G}_1$: This is the same game as the EA-game, except that the adversary can no longer win if its winning instance π_i^m belongs to a Service.

In the EA definition, the only way the adversary can win if the party P_i is a Service is if the accepting instance π_i^m for which $\mathscr{A}$ wins has to accept for $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ with P_j is an Engine. Since such an attacker must guess the identity of the Service that will maliciously accept, and the Engine that is being impersonated, we have that

$$|\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_0}\text{ wins}\right] - \mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_1}\text{ wins}\right]| \leq \mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2.$$

Game $\mathbb{G}_2$: This game behaves as $\mathbb{G}_1$, except we now rule out the possibility that the party P_i, holding the "winning" instance, is an Engine. If that is the case, then its partner party P_j can only be a Service. In a similar way to the above, we can reduce this to the ACCE-EA security of P', namely,

$$|\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_1}\text{ wins}\right] - \mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_2}\text{ wins}\right]| \leq \mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2.$$

Game $\mathbb{G}_3$: In this game, the adversary may only win against an instance π_i^m of a client.

In this game, we rule out the possibility of the adversary winning in a direct Client-Service handshake. More formally, we rule out the possibility that $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ such that P_j is a Service and P_i is a client, such that there exists an instance π_j^n such that $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid}$. In other words, $\mathbb{G}_3$ corresponds to $\mathbb{G}_0$ with the restriction that P_i is a client and the targeted instance π_i^m has the related partnering: $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ with P_j being a Service and such that there exists some Engine P_k and an instance π_k^p such that π_k^p and π_i^m are 2-partnered (they have the same session ID).

So, the advantage of the adversary in $\mathbb{G}_3$ is basically building on the advantage of $\mathscr{A}_3$ (with $\mathscr{A}_3$ playing in the mEA game and as being interested in the sessions where he queries with the flag flag_i^m being 0, since we are in the case of P is a client). Next, we will show more clearly that

$$|\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_3}\text{ wins}\right] - \mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_2}\text{ wins}\right]| \leq \mathsf{n_P}^3\cdot\mathsf{Adv}^{mEA}_{P,P'}\mathscr{A}_3 + \Delta,$$

where Δ is obtained as per the below.

We first prove that, not only is an Engine the real partner and a Service is the intended partner, but it also holds that: there exists a matching instance π_k^ℓ such that π_k^ℓ and π_j^n are also 2-partnered, and furthermore, the session key $\pi_i^m.\mathsf{ck}$ is computed as expected from the pre-master secret pmk of π_j^n and the transcript of π_i^m. In this case, we recall that the partnering in $\mathsf{3SACCE}$ gives π_i^m these "party-partners" $\pi_i^m.\mathsf{PSet}=\{P_i,P_j,P_k\}$ and these "instance-partners" $\pi_i^m.\mathsf{InstSet}=\{\pi_i^m,\pi_j^n,\pi_k^p,\pi_k^\ell\}$.

Impersonation Succeses in $\mathbb{G}_3$:

• **TLS-DHE** To begin with, we focus on the transcript of π_i^m.

We now rule out the possibility that the client accepts P_x as if it were P_k, which is bounded, first by the collision-resistance of the hash function H, and secondly, by the unforgeability in the signature PSign: $\mathsf{n_P}\cdot\mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}\mathscr{A}_5$, accounting for getting which party the signature is generated for.

• **TLS-RSA** In this setting, the equivalent security is guaranteed by the fact that the encryption is under the public key of the purported partner P_k of π_i^m.

The only adversarial success-option is the case of having a party that is not P_k decrypt the encrypted pre-master key.

To this end, we can build a reduction to the SACCE security of the underlying protocol P, such that the adversary can learn the secret bit of instance π_i^m (by learning the pre-master secret and then computing the channel key). The probability $\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_3}\ \mathsf{wins}\right]$ is increased by $\mathsf{n_P}^2\cdot\mathsf{Adv}^{\mathsf{CS-SACCE}}_{P}\mathscr{A}_7$.

We now resume our proof on $\mathbb{G}_3$, fixing the three parties P_i,P_j,P_k. (This implies a factor of $\mathsf{n_P}^3$ in all the added advantages below).

We reduce the remaining winning probability in our EA game in our protocol to mEA-security assumption with respect to P and P'. The adversary $\mathscr{A}_{\mathbb{G}_3}$ is fed information by the adversary $\mathscr{A}_3$ which plays the mEA game with respect to the P and P' protocols. Whenever $\mathscr{A}_{\mathbb{G}_3}$ queries information for Client-Engine sessions, the queries made via $\mathscr{A}_3$ are with $\mathsf{flag}_i^m=0$. Whenever $\mathscr{A}_{\mathbb{G}_3}$ queries Engine-Service information, the queries made via $\mathscr{A}_3$ are with $\mathsf{flag}_k^l=1$. So, the probability $\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_3}\ \mathsf{wins}\right]$ is increased by the factor $\mathsf{n_P}^3\cdot\mathsf{Adv}^{mEA}_{P,P'}\mathscr{A}_3$[7].

W.r.t. our current EA game, we also note that the EA definition further stipulates that no Reveal query can be made on the instances $\pi_i^m.\mathsf{InstSet}$ partnered with π_i^m. W.r.t. the our current EA game and the mEA-game, the simulation for $\mathsf{RegParty}$, $\mathsf{NewSession}$, $\mathsf{Corrupt}$, Reveal clearly work with no issue, as in the 2(S)ACCE, TLS cases. The difference (between the our 3-party EA setting and the 2-party mEA setting) occurs for the Send oracle, since in order to simulate correctly the record-layer transcript of the Engine-Service session between π_k^ℓ and π_j^n. Here, on this Engine-Server side, we need to reduce to the capabilities of adversary $\mathscr{A}_2$ who is challenging the security of the ACCE protocol P'. The adversary $\mathscr{A}_2$ will query Reveal on this session (this is allowed in the EA game) and simulate the rest.

In particular, in RSA mode, to simulate sending $\mathsf{ck}\oplus k$ or $\mathsf{msk}\oplus k$, the adversary $\mathscr{A}_3$ (who can challenge the security of the inner P' in the mEA game) chooses at random a value r and sends r, sending this to the Engine. Thus, in RSA mode, the probability that $\mathscr{A}_{\mathbb{G}_3}$ win is increased is augmented by $\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{2\text{-}ACCE}}_{P'}\mathscr{A}_2+\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_4$. The above simulation for $\mathbb{G}_3$ is perfect. In particular, note that with protocol P' not containing a key-confirmation step and k is indistinguishable from random. So, sending r simulates perfectly sending $\mathsf{msk}\oplus k$ or $\mathsf{ck}\oplus k$. In DHE mode, there is nothing to simulate, and the probability that $\mathscr{A}_{\mathbb{G}_3}$ win is increased is augmented by $\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{2\text{-}ACCE}}_{P'}\mathscr{A}_2$

If the adversary $\mathscr{A}_{\mathbb{G}_3}$ wins for some session π_i^m, then $\mathscr{A}_3$ (in the mEA game with the flag flag_i^m being 0) verifies if there exists a unique instance π_k^p such that $\pi_i^m.\mathsf{sid}=\pi_k^p.\mathsf{sid}$. If this instance does not exist, this $\mathscr{A}_3$ will have π_i^m as its own winning instance. Otherwise, if the adversary $\mathscr{A}_{\mathbb{G}_3}$ does not win, it must be that $\mathscr{A}_4$ will find an instance π_j^n of P_j holding pmk (in RSA) or $\big(p,g,KE_S;Cert_E;PSign\big)$ (in DHE mode) corresponding to $\pi_i^m.\mathsf{ck}$, but such that there exists no matching, unique π_k^ℓ, also holding that pmkor $\big(p,g,KE_S;Cert_E;PSign\big)$, so that π_k^ℓ,π_j^n are 2-partnered. In this latter case, $\mathscr{A}_4$ wins.

This concludes the proof and, step-by-step, we yielded the indicated bound.

□

D.3 Channel Security Proofs

Theorem 2. *Let P be the unilaterally-authenticated TLS 1.2 handshake (as seen by the Client), and P' be the AKE protocol between Engine and Crypto Service which, at each session, exports a key k indistinguishable from random.*

Consider a (t,q)-adversary $\mathscr{A}$ against the SC-security of the protocol $LURK1.2$ running at most t queries and creating at most q party instances per party. We denote by $\mathsf{n_P}$ the number of parties in the system, and denote $\mathscr{A}$'s advantage by $\mathsf{Adv}^{\mathsf{SC}}_{\Pi}\mathscr{A}$.

If such an adversary exists, then there exist adversaries $\mathscr{A}_1$ against the SACCE security of P, $\mathscr{A}_2$ against the ACCE security of P', $\mathscr{A}_3$ against the AKE security of P' with exported key k and either: $\mathscr{A}_4$ against the existential unforgeability (EUF-CMA) of the signature algorithm used to generate PSign (for TLS-DHE), or $\mathscr{A}_4$ against the channel security of P (for TLS-RSA), each adversary running in time $t'\sim O(t)$ and instantiating at most $q'=q$ instances per party, $\mathscr{A}_5$ against the PRF φ, such that

[7]Note that in this bound we only give the dominant fact, since we do not count specifically the C-E sessions of P as per the queries with flag_i^m being 0, even though we are in the case of P is a client.

- *For Variant1 of LURK1.2 in DHE mode:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{SC}}_{\Pi}\mathscr{A} \;\leq\; & \left(2\mathsf{n_P}^2+2\mathsf{n_P}^3\right)\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2 \\ & +\mathsf{n_P}^2\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1+\mathsf{n_P}\mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}\mathscr{A}_4 \\ & +\mathsf{n_P}^3\left(\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_3+\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1\right).\end{aligned}$$

- *For Variants 1 and 2 of LURK1.2 in RSA mode:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{SC}}_{\Pi}\mathscr{A} \;\leq\; & \left(2\mathsf{n_P}^2+2\mathsf{n_P}^3\right)\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2+\left(\mathsf{n_P}^3+\mathsf{n_P}^2\right) \\ & advsacce\mathscr{A}_1P+\mathsf{n_P}^3\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_3 \\ & +\mathsf{n_P}^2\mathsf{Adv}^{\mathsf{SC-SACCE}}_{P}\mathscr{A}_4+\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{npPRF}}\mathscr{A}_5.\end{aligned}$$

Proof. Our proof has the following hops:

Game $\mathbb{G}_0$: This game works as the SC-game recounted in Appendix C.

Games $\mathbb{G}_0$-$\mathbb{G}_3$: We make similar successive reductions as in the previous proof to obtain the game $\mathbb{G}_3$ which behaves as the original game but with the restriction that P_i is a client, and for the targeted instance π_i^m it holds that: $\pi_i^m.\mathsf{pid} = P_j.\mathsf{name}$ with P_j being a Service and such that there exists some Engine P_k and an instance π_k^p such that π_k^p and π_i^m are 2-partnered (they have the same session ID).

The loss through to game $\mathbb{G}_3$ is as follows:

$$\begin{aligned}\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_2}\text{ wins}\right] \;\leq\; & \mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_3}\text{ wins}\right]+\mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1 \\ & +2\mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2.\end{aligned}$$

Winning game 3: This proof goes similarly to the one before, except that in the simulation of adversaries $\mathscr{A}_1$ and $\mathscr{A}_2$ we use a simulation akin to that of the SC-game, in particular with respect to simulating the encryption and decryption queries. The total success probability of the adversary is given by:

– For DHE : $\Pr\left[\mathscr{A}_{\mathbb{G}_3}\text{ wins}\right]$

$$\begin{aligned}\leq\; & \frac{1}{2}+\mathsf{n_P}^3\left(\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_3+\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1+\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2\right) \\ & +\mathsf{n_P}\cdot\mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}\mathscr{A}_4.\end{aligned}$$

– For RSA : $\Pr\left[\mathscr{A}_{\mathbb{G}_3}\text{ wins}\right]$

$$\begin{aligned}\leq\; & \frac{1}{2}+\mathsf{n_P}^3\left(\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_3+\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1+\mathsf{Adv}^{\text{2-ACCE}}_{P'}\mathscr{A}_2\right) \\ & +\mathsf{n_P}^2\cdot\mathsf{Adv}^{\mathsf{AKE}}\mathscr{A}_3 \\ & \mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{npPRF}}\mathscr{A}_5.\end{aligned}$$

In the last probability, the $\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{npPRF}}\mathscr{A}_5$ factor comes from the attacker in game 3 looking to break the PRF assumption and produce an adaptive N_E across several session to learn msk or ck for a new session. □

D.4 Accountability Proofs

Theorem 3. *Let P be the unilaterally-authenticated TLS 1.2 handshake (as seen by the Client), and P' be the AKE protocol between Engine and Crypto Service which, at each session, exports a key k indistinguishable from random.*

Consider a (t,q)-adversary $\mathscr{A}$ against the Acc*-security of Variant 2 of LURK1.2 in RSA Mode, running at most t queries and creating at most q party instances per party. We denote by $\mathsf{n_P}$ the number of parties in the system, and denote $\mathscr{A}$'s advantage by $\mathsf{Adv}^{\mathsf{Acc}}_{v2-LURK1.2-RSA}\mathscr{A}$. If such an adversary exists, then there exists adversary $\mathscr{A}_1$ against the* SACCE *security of P running in time $t'\sim O(t)$ and instantiating at most $q'=q$ instances per party, such that: $\mathsf{Adv}^{\mathsf{Acc}}_{v2-LURK1.2-RSA}\mathscr{A}\leq 2\cdot\mathsf{n_P}^2\cdot\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1$*

Proof. Our proof has the following hops:

Game 0: This game works as the Acc- game recalled Appendix C. We say that an adversary $\mathscr{A}$ *breaks the accountability* for an instance π_i^m with $P_i\in\mathscr{C}$, if the following conditions are verified:

(a) the acceptance flag for π_i^m is set (i.e., $\pi_i^m.\alpha=1$) such that $\pi_i^m.\mathsf{pid}=P_j.\mathsf{name}$ for an *uncorrupted* $P_j\in\mathscr{S}$ and $\pi_i^m.\mathsf{ck}=\mathsf{ck}$;

(b) There exists no instance $\pi_j^n\in P_j.\mathsf{Instances}$ such that $\pi_j^n.\mathsf{ck}=\pi_i^m.\mathsf{ck}$;

(c) There exists no probabilistic polynomial algorithm Sim which given the view of P_j (namely all instances $\pi_j^n\in P_j.\mathsf{Instances}$ with all their attributes), outputs ck.

We wish to show that, whenever condition (a) holds, then either the reverse of (b) or the reverse of condition (c) holds (except with negligible probability). We also need a simulator that fulfils condition (c). We first rule out a few exceptions.

Game $\mathbb{G}_1$: The adversary begins by guessing the identities of the targeted client P_i and of the server P_j such that π_i^m is the instance for which accountability is broken, and for which it holds $\pi_i^m.\mathsf{pid}=P_j.\mathsf{name}$. As a consequence, we have:

$$\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_0}\text{ wins}\right]\leq\mathsf{n_P}^2\cdot\mathbf{Pr}\left[\mathscr{A}_{\mathbb{G}_1}\text{ wins}\right].$$

We assume there exists an Engine party P_k such that there exists an instance π_k^p with $\pi_k^p.\mathsf{sid}=\pi_i^m.\mathsf{sid}$. There are two options.

First, the Engine could try to run the handshake on its own (there exist no instances π_j^n,π_x^ℓ such that $\pi_i^m.\mathsf{pck}$ is in fact $\pi_i^m.\mathsf{ck}$ as per the protocol description). Note that for this first option it does not necessarily have to hold that π_x^ℓ is an instance of P_k, *i.e.*, the Engine talking to the Client. In that case ,we can construct a reduction from this case to the server-impersonation security of the protocol P' (recall that the honest server P_j cannot be corrupted). For TLS-RSA, this is equivalent to decrypting the premaster secret; so, we lose a term $\mathsf{Adv}^{\text{2-SACCE}}_{P}\mathscr{A}_1$.

Or, there exist instances π_j^n,π_x^ℓ such that $\pi_i^m.\mathsf{pck}$ is in fact $\pi_i^m.\mathsf{ck}$ as per the protocol description. In this case, the simulator is trivial, namely, the simulator consists in simply seeking an instance π_j^n such that the record transcript of that instance contains the transcript of $\pi_i^m.\tau$, *i.e.*, the same tuple of nonces, key-exchange elements, and a verifying client finished message. Output the key ck sent to the Engine by that instance as the key of π_i^m. We also note that if the client finished message does not verify, then the Engine has to generate its own Finished message; if the adversary does that, we can construct a reduction from this game to the SACCE-security of P'

(*i.e.*, the standard TLS 1.2 handshake run between the client and the uncorrupted server), in which the adversary (possibly a collusion of all the malicious Engine) simulates all but parties P_i,P_j and will win by outputting the same instance and random sampling bit as the underlying adversary. So, we lose another term $\mathsf{Adv}_P^{\text{2-SACCE}}\mathscr{A}_1$.

This yields the given bound, *i.e.*,

$$\mathbf{Pr}\big[\mathscr{A}_{\mathbb{G}_0}\ \mathsf{wins}\big] \leq 2 \cdot {\mathsf{n_P}}^2 \cdot \mathsf{Adv}_P^{\text{2-SACCE}}\mathscr{A}_1.$$

□

E LURK based on TLS 1.3

The LURK extension designed for TLS 1.3 is ongoing, and its design has already been drafted in [11]. The design of the LURK TLS 1.3 reflects the differences between TLS 1.2 and TLS 1.3 as well as the usability of these respective protocols. On, the other hand the design of the extension for TLS 1.3 also leverages some designs of the LURK extension for TLS 1.2.

The primary difference between the two extensions is the scope of these extensions. The LURK extension for TLS 1.2 has been essentially scoped to protect TLS servers credentials during TLS handshake as well as to enable a delegation between a content owner and a CDN provider. In this latest case, the Cryptographic Service was responsible to generate the master secret or to simply sign the corresponding ECDHE public parameters. This overall represents a very narrow use case for the TLS usages. Typically, TLS mutual authentication and the usage of LURK by a TLS client has not been considered. One particular reason is that at the time the requirements for LURK were discussed at the IETF to define the scope of LURK, the use case of CDN was the only one the community wanted to address. By the time TLS 1.3 has been designed, a growing demand was to integrate all functionalities of TLS 1.3 into the cryptographic service. The particular motivation for it was a growing demand to use the cryptographic service for use cases that considered TL:S mutual authentication, and more especially the use of TLS to secure the deployment of containerized applications over an untrusted environment. As a result, the scope of the LURK extension for TLS 1.3 was definitively wider than the one considered for TLS 1.2, and aimed at designing a trusted TLS where all credentials could be secured and protected at any time.

As a result, the LURK extension for TLS 1.3 includes, session resumption that is supported by the key schedule of TLS 1.3, the new 0-RTT authentication as well as the authentication of the TLS client. Note that there is nothing that prevents the design of additional functionalities performed by the cryptographic service for TLS 1.2. All these functionalities can be defined in the future. These functionalities have not been provided because there were no need expressed by the industry for TLS 1.2. New usages will be implemented with TLS 1.3. LURK for TLS 1.2 is left for easing the transition from TLS 1.2 to TLS 1.3 while reducing the exposure of attack of TLS 1.2 where TLS 1.2 is necessary. The reason the transition is eased is that adaptation of the infrastructure for TLS 1.2 will not be necessary, and will be able to follow the same design as the infrastructures designed for TLS 1.3.

The common aspects of these two extensions can be summed up as follows: 1) Both extensions are using the framework of LURK.

2) the TLS 1.3 extension directly benefits from the mechanism of the freshness function that implements perfect forward secrecy and prevents the replay of an already played LURK exchange.

3) For the ECDHE authentication, the design of the LURK extension for TLS 1.3 greatly benefits from the findings of the LURK extension of TLS 1.3. While the signature used to generate the signature are different in TLS 1.2 and TLS 1.3 - and this at least to prevent cross protocol attacks. The exchange to generate the signature share a similar analysis and as such share some commonalities. As TLS 1.3 actively promotes the use of authentication methods that implements PEFS (ECHDE as well as PSK-ECHDE). In both cases, LURK for TLS 1.3 enables the re-use of Proof of ownership designed for TLS 1.2.

However, the LURK extension for TLS 1.3 differs greatly from the LURK extension for TLS 1.2.

1) Authentication methods differs. Typically RSA authentication has not been considered for TLS 1.3. On the other hand, PSK authentication - though supported by TLS 1.2 - has not been considered by the LURK extension for TLS 1.2, but has been considered by the LURK extension for TLS 1.3. As a result, the common scope regarding the authentication method is limited to ECDHE.

2) TLS 1.3 has a different key schedule as TLS 1.2. More especially, all different secrets or keys are derived from a key schedule that take the PSK and ECDHE share secret as inputs. Key sare then derived at various stages of the key schedule and take the available Handshake Context available at that stage. Such design requires also a different exchange for each secret - as different input and context are needed. As such the establishment of a TLS 1.3 session is likely to require multiple interactions with the cryptographic service and the definition of a LURK session. Note the delegation use case for which LURK TLS 1.2 ha been designed only requires a single exchange as well with TLS 1.3. Note also that the design of a trusted TLS requires more control of the secrets, and the cryptographic service of TLS 1.3 and thus provides more accountability which is missing in the LURK extension for TLS 1.2.

References

[1] LURK Extension version 1 for (D)TLS 1.2 Authentication. https://github.com/anon-data/anon_src/blob/master/drafts/tls12.txt, July 2018. Work in Progress.

[2] Barnes, R., Hoffman-Andrews, J., McCarney, D., and Kasten, J. Automatic Certificate Management Environment (ACME). RFC 8555, Mar. 2019.

[3] Barnes, R., Iyengar, S., Sullivan, N., and Rescorla, E. Delegated Credentials for TLS. Internet-Draft draft-draft-ietf-tls-subcerts, Internet Engineering Task Force, July 2019. Work in Progress.

[4] Bhargavan, K., Boureanu, I., Delignat-Lavaud, A., Fouque, P.-A., and Onete, C. A Formal Treatment of Accountable Proxying over TLS. In *Proceedings of IEEE S&P* (2018), IEEE.

[5] Bhargavan, K., Boureanu, I., Fouque, P., Onete, C., and Richard, B. Content delivery over TLS: a cryptographic analysis of Keyless SSL. In *Proceedings of Euro S&P* (2017).

[6] Blanchet, B. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *IEEE Computer Security Foundations Workshop* (Novia Scotia, Canada, 2001), IEEE Computer Society Press, pp. 82–96.

[7] Boeyen, S., Santesson, S., Polk, T., Housley, R., Farrell, S., and Cooper, D. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[8] Brzuska, C., kon Jacobsen, H., and Stebila, D. Safely exporting keys from secure channels: on the security of EAP-TLS and TLS key exporters. In *EuroCrypt* (2016).

[9] IETF Content Delivery Networks Interconnection Working Group (CDNI). https://datatracker.ietf.org/wg/cdni/about/, 2019.

[10] PyCryptodome: a self-contained Python package of low-level cryptographic primitives. https://pycryptodome.readthedocs.io, 2019.

[11] LURK Extension version 1 for (D)TLS 1.3 Authentication. https://github.com/anon-data/anon_src/blob/master/drafts/tls13.txt, July 2019. Work in Progress.

[12] ETSI. CYBER; Middlebox Security Protocol; Part 3: Profile for enterprise network and data centre access control. In *ETSI TS 103 523-3 V1.1.1* (2018). .

[13] Fiat, A., and Shamir, A. How to prove yourself: Practical solutions to identification and signature problems. Springer-Verlag, pp. 186–194.

[14] Gero, C., Shapiro, J., and Burd, D. Terminating ssl connections without locally-accessible private keys, June 20 2013. WO Patent App. PCT/US2012/070075.

[15] Gilad, Y., Herzberg, A., Sudkovitch, M., and Goberman, M. CDN-on-Demand: An affordable DDoS Defense via Untrusted Clouds. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016).

[16] Hoffman, P. E., and McManus, P. DNS Queries over HTTPS (DoH). RFC 8484, Oct. 2018.

[17] Hoffman, P. E., and Schlyter, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698, Aug. 2012.

[18] http.server: HTTP Servers. https://docs.python.org/3.4/library/http.server.html, 2018.

[19] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and Hoffman, P. E. Specification for DNS over Transport Layer Security (TLS). RFC 7858, May 2016.

[20] Istio. An Open Platform to Connect, Manage, and Secure Microservices. https://github.com/istio/istio, 2019.

[21] Jager, T., Kohlar, F., Schäge, S., and Schwenk, J. On the security of TLS-DHE in the standard model. In *Proceedings of CRYPTO 2012* (2012), vol. 7417 of *LNCS*, pp. 273–293.

[22] Lan, C., Sherry, J., Popa, R. A., Ratnasamy, S., and Liu, Z. Embark: Securely Outsourcing Middleboxes to the Cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016* (2016), K. J. Argyraki and R. Isaacs, Eds., USENIX Association, pp. 255–273.

[23] Lee, H., Smith, Z., Lim, J., Choi, G., Chun, S., Chung, T., and Kwon, T. T. matls: How to make TLS middlebox-aware? In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019* (2019).

[24] Levy, A., Corrigan-Gibbs, H., and Boneh, D. Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser. *IEEE Security Privacy 14*, 2 (2016), 22–28.

[25] Liang, J., Jiang, J., Duan, H., Li, K., Wan, T., and Wu, J. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *2014 IEEE Symposium on Security and Privacy* (May 2014), pp. 67–82.

[26] Loreto, S., Mattsson, J., Skog, R., Spaak, H., Druta, D., and Hafeez, M. Explicit Trusted Proxy in HTTP/2.0. Internet-Draft draft-draft-loreto-httpbis-trusted-proxy20, Internet Engineering Task Force, Feb. 2014. Work in Progress.

[27] Lurk: Practical and secure server-controlled tls delegation (long version). https://github.com/anon-data/anon_src/blob/master/long_version.pdf, 2019.

[28] McGrew, D., Wing, D., and Gladstone, P. TLS Proxy Server Extension. Internet-Draft draft-draft-mcgrew-tls-proxy-server, Internet Engineering Task Force, July 2012. Work in Progress.

[29] Narayanan, V. Explicit Proxying in HTTP - Problem Statement And Goals. Internet-Draft draft-draft-vidya-httpbis-explicit-proxy-ps, Internet Engineering Task Force, Oct. 2013. Work in Progress.

[30] Naylor, D., Schomp, K., Varvello, M., Leontiadis, I., Blackburn, J., Diego R. López, Papagiannaki, K., Pablo Rodriguez Rodriguez, and Steenkiste, P. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of SIGCOMM 2015* (2015), ACM, pp. 199–212.

[31] Nir, Y. A Method for Sharing Record Protocol Keys with a Middlebox in TLS. Internet-Draft draft-draft-nir-tls-keyshare, Internet Engineering Task Force, Mar. 2012. Work in Progress.

[32] Nottingham, M. Problems with Proxies in HTTP. Internet-Draft draft-draft-nottingham-http-proxy-problem, Internet Engineering Task Force, July 2014. Work in Progress.

[33] SVA Open Caching Working Group. https://www.streamingvideoalliance.org/technical-work/working-groups/open-caching/, 2019.

[34] Symbolic analysis of LURK1.2 with ProVerif. https://github.com/anon-data/anon_src/tree/master/pv, 2019. anonymised for submission.

[35] pylurk – a Python implementation of LURK. https://github.com/anon-data/anon_src/tree/master/pylurk, 2019. anonymised for submission.

[36] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018.

[37] Sheffer, Y., Lopez, D., de Dios, O. G., Pastor, A., and Fossati, T. Generating Certificate Requests for Short-Term, Automatically-Renewed (STAR) Certificates. Internet-Draft draft-draft-sheffer-acme-star-request, Internet Engineering Task Force, June 2018. Work in Progress.

[38] Sheffer, Y., Lopez, D., de Dios, O. G., Pastor, A., and Fossati, T. Support for Short-Term, Automatically-Renewed (STAR) Certificates in Automated Certificate Management Environment (ACME). Internet-Draft draft-draft-ietf-acme-star, Internet Engineering Task Force, Oct. 2019. Work in Progress.

[39] Sherry, J., Lan, C., Popa, R. A., and Ratnasamy, S. Blindbox: Deep packet inspection over encrypted traffic. *Computer Communication Review 45*, 5 (2015), 213–226.
[40] socketserver: A framework for network servers. https://docs.python.org/3.4/library/socketserver.html, 2018.
[41] SplitTLS. https://github.com/indutny/splittls.
[42] SSL: TLS/SSL wrapper for socket objects. https://docs.python.org/3.4/library/ssl.html, 2018.
[43] Stebila, D., and Sullivan, N. An analysis of tls handshake proxying. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (Aug 2015), vol. 1, pp. 279–286.
[44] Sullivan, N. Keyless ssl: The nitty gritty technical details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/, Sept. 2014.
[45] tinyec. https://github.com/alexmgr/tinyec, 2018.