LURK: Server-Controlled TLS Delegation

Ioana Boureanu[1], Daniel Migault[2] Stere Preda[2], Hyame Assem Alamedine[3], Sanjay Mishra[4], Frederic Fieau[5], and Mohammad Mannan[6]

[1] University of Surrey, UK, i.boureanu@surrey.ac.uk,
[2] Ericsson Security, CA, firstname.lastname@ericsson.com
[3] Ericsson Security, CA, hyame.a.alameddine@ericsson.com
[4] Verizon, USA, sanjay.mishra@verizon.com
[5] Orange, FR, frederic.fieau@orange.com
[6] Concordia University, m.mannan@concordia.ca

Abstract. By design, TLS (Transport Layer Security) is a 2-party, end-to-end protocol. Yet, in practice, *TLS delegation* is often deployed: that is, *middlebox proxies* inspect and even modify TLS traffic between the endpoints. Recently, industry-leaders (e.g., Akamai), standardization bodies (e.g., IETF, ETSI), and academic researchers have proposed numerous ways of achieving *safer* TLS delegation. We present LURK (Limited Use of Remote Keys), a suite of designs for TLS delegation, where the TLS-server is aware of the middlebox. We implement and test LURK. Also, we cryptographically prove and formally verify, in Proverif, the security of LURK.. Finally, we comprehensively analyze how our designs balance (provable) security and competitive performance.

1 Introduction

Decades ago, Internet protocols were designed such that the application logic operated only at the endpoints. However, today, this end-to-end paradigm is impeded primarily by the fact that traffic is now processed by a series of middleboxes before it is presented to the end user, normally in a personalised form (e.g., via user-customised web-acceleration and compression). Content delivery networks (CDN) have traditionally been at the core of this, but now they are just one of the many players in the field, alongside massive IoT (Internet of Things) and the rising 5G-fulled edge computing. We shall refer to this internet-traffic mediation by middleboxes as "collaborative content-delivery"

Over *unencrypted traffic*, collaborative content-delivery fits the following architecture. A third party found "on path" between the end-client and the end-server simply processes packets, and In this way, using a principle called implicit signalling [**?**], the endpoints are largely unaware of the mediated content-delivery by the third party. Since this design implies a high level of trust placed on the mediating point, we will refer to it as a *TTP (trusted third party)*. On the other hand, encryption of application-data, e.g., via *TLS* has become commonplace. As such, collaborative delivery over TLS-encrypted traffic is already largely adopted, e.g., by CDNs. This is most often referred to as *SSL inspection* or ***TLS delegation***. Their *edge servers* hold a valid X.509 certificate for the domain(s) and an associated private key on behalf the web-servers for which they deliver content. In this way, the CDN-owned TTP is still invisible to the end-user, as it impersonates the end web-server in a manner akin to that of implicit signaling.

At the same time as collaborative delivery with implicit signaling becomes ubiquitous, the Internet Architecture Board (IAB) calls for making all proxies collaborating in traffic

delivery visible to the endpoints. Meanwhile, there exist architectures that meet somewhere in the middle: they are more practical than fully visible proxying, and side with the IAB on reducing the TTP's invisibility. Concretely, in these cases, the mediating party is still invisible to the client, but not to the web-server, at least during the secure-channel establishment, e.g., the TLS handshake. We refer to these as ***server-controlled TLS delegations***. Such designs appeared first in a patent by the CDN-giant Akamai [11]. In this modus operandi, the CDN provisions the public key and associated X.509 certificate for the domain it delivers, but the associated private key remains on the web-servers' side. The CDN queries the web-server via an API for operations where designated private key is needed. In 2015, Cloudflare commercialized a version of this, in a product called *Keyless SSL* [28]. However, these designs obviously require modifications to the TLS server and sometimes even 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. [4] used a provable-security approach to show several vulnerabilities on Keyless SSL; they also advanced an alternative design, called *3(S)ACCE-K-SSL*, that provably achieves stronger security goals than Keyless SSL, albeit with reduced design-efficiency.

Contributions

1. We propose 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. We call our designs – *LURK (Limited Usage of Remote Key)*. LURK meets in the middle between the (insecure) Keyless SSL [28] and the (provably secure but inefficient) 3(S)ACCE-K-SSL [4]. In fact, LURK has different variants which offer a balance between security and practicality. For instance, we remove the content-soundness requirement in 3(S)ACCE-K-SSL [4], as it needs an expensive PKI. Similarly, for efficiency reasons, only certain LURK variants attain a new TLS-delegation property called accountability [4]. Meanwhile, all LURK designs require channel security and entity authentication in the collaborative, 3-party setting.

2. LURK is a generic design to accommodate most authenticated key-exchange (AKE) protocols. In this paper, we instantiated it with TSL 1.2, which we call $LURK1.2$.

3. Using the recent 3(S)ACCE formal security model for proxied AKE [4], we provide cryptographic proofs that $LURK1.2$ achieves its security goals.

4. In $LURK1.2$, we include a "freshness mechanism" to counter replay attacks.[7] As such, we encode $LURK1.2$ in RSA mode in the automatic protocol-verifier ProVerif and we formally check that perfect forward secrecy holds in $LURK1.2$ in RSA mode. With a further formalisation in ProVerif, we show that the elusion of our "freshness mechanism" would in fact lead to the same type of attacks as found in Keyless SSL. Thus, we formally prove that our "freshness mechanism" does indeed aid to ensure perfect forward secrecy in $LURK1.2$ in RSA mode.

5. We implement the more efficient variants of LURK and test them in practice.

Why $LURK1.2$? The purpose of LURK based on TLS 1.2 is to provide the necessary agility required during the transition from TLS 1.2 to TLS 1.3, all the while preventing that bespoke TLS 1.2 communications operate insecurely. While TLS 1.3

[7] 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 for Keyless SSL TLS 1.2 in RSA mode, and a repair was proposed via the 3(S)ACCE-K-SSL design [4]; our replay-prevention mechanism differs from this design.

has seen a remarkably fast adoption from large companies (Facebook, Google, Microsoft, Akamai) as well as standardization bodies such as 3GPP, TLS 1.2 is the de-facto version of TLS used worldwide. In fact, many services rely on so-called legacy devices, such as video on demand being provided by Customer Premises Equipment (CPE); for these, the move to TLS 1.3 is expected to take significantly longer. One of $LURK1.2$'s aim is to bridge this gap.

2 LURK: Delegated Secure Delivery with Server Control

LURK is a suite of designs for delegated credentials [2], based on TLS. To this end, LURK does not require any change to the TLS Client, whilst it does split the standard TLS Server into two services: 1). a *Cryptographic Service* ("Crypto Service" for short), denoted by S, which performs cryptographic operations associated to the private key of the TLS Server; 2). a *LURK Engine* ("Engine" for short), denoted by E, which performs the remainder of the TLS server-side process. The Crypto service and the LURK Engine can be collocated[8] services or not. These two services communicate using the LURK protocol. In other words, LURK facilitates "oracle"-like calls that the Engine E makes to Crypto Service S, needed for the signing or decryption operations that a TLS-server normally does. The queries from the LURK Engine to the Cryptographic Service are performed over a mutually-authenticated and secure channel with exported keys (e.g., EAP-TLS or other "TLS-like" protocol), which –as shown by [8]– can be transformed into a provably secure authenticated key exchange protocol where the exported keys are indistinguishable from random. Whilst the limited and restricted use of the Crypto Service is akin to that of an HSM (Hardware Security Module), the enforcement mechanisms in place to achieve such restricted usage in LURK differ from those in an HSM.

2.1 The *LURK*1.2 Designs

LURK instantiated with TLS 1.2, called $LURK1.2$, is parametric in a security parameter s, as well as on a function[9] called the *"freshness function"* and denoted φ. This is a pseudorandom function (PRF). There is a fresh key k, indistinguishable from random, exported from the AKE protocol run between the Engine and the Crypto Service, at each run of $LURK1.2$. In each such run, the key k is also used to key an instance of the PRF φ. In that sense, when we sometimes write "$\varphi(\cdot)$" we mean $\varphi_k(\cdot)$, where both φ and k are re-chosen/re-established at every new session.

$LURK1.2$ **in RSA Mode.** Figure 1 presents the $LURK1.2$ protocol[10] based on TLS1.2 in RSA-mode. We propose two slightly different variants of $LURK1.2$ in RSA mode. As per Fig. 1, the handshake starts as expected on the client side.

Thereafter, there are some differences. First, the server-nonce, here denoted N_E, is computed by the LURK Engine in a different manner than in TLS 1.2. RSA mode. The LURK Engine generates a nonce N_i at random; the length of N_i is parametric

[8] In CDN, the LURK Engine is hosted by the CDN provider at the edge node, while the Cryptographic Service is hosted by the content owner.

[9] We do not hard code this function in the design as per the guidelines of [14]. In this way, if concrete implementations have already allocated the space for different possibilities, then deprecation of specific choices and replacements are more easily made.

[10] Please see Appendix A for details on TLS 1.2.

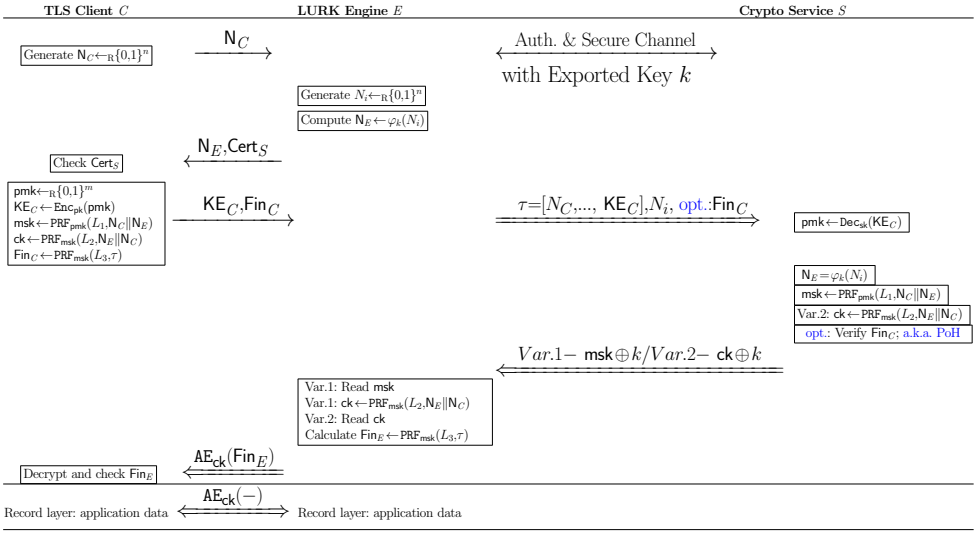


Fig. 1: $LURK1.2$ based on TLS1.2 in RSA mode: Two Variants

in a security parameter. In practice, in line with TLS parameters, this length can be chosen to be, e.g., 28×8 bits. Second, the LURK Engine applies the φ function to N_i. It is the result of $\varphi(N_i)$, i.e., $N_E = \varphi(N_i)$, that stands in for the "TLS server random" and is sent back by the LURK Engine to the Client. 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 forwards these (with or without the Fin_C), 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, verifies the Fin_C message (if it was sent) and then computes the master secret msk. Note that the sending by the Engine of the Fin_C message to be verified by the Crypto Service is optional and we also refer to it as the *Proof of Handshake (PoH)*.

Henceforth, $LURK1.2$ branches out in two variants. In Variant 1, the Crypto Service sends back the master-secret msk to the LURK Engine, whereas in Variant 2 – the channel key ck is sent back to the LURK Engine. Either message, msk or ck, is sent encrypted with the exported key. Then, the protocol between the Engine and the TLS 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 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. The first is presented in Figure 2, and the second in Figure 7 – found in Appendix C. In the first variant of $LURK1.2$ in DHE mode (Fig. 2), the LURK Engine generates the DHE keypair $(v, g^v \mod p)$ and KE_E. It sends the key share KE_E to the Crypto Service together with N_C and N_i, as well as –optionally- a *Proof of Ownership of* v, denoted $PoO(v)$; the latter can be seen as a non-interactive proof of knowledge of the secret exponent v.

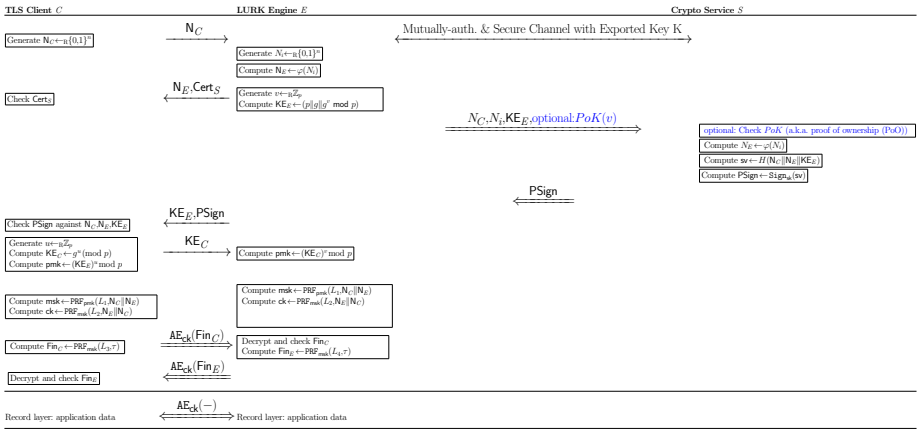


Fig. 2: $LURK1.2$ based on TLS1.2 in DHE mode: Variant1

The Crypto Service would only accept a specific data-structure for the messages received at this step and it will decline the communication otherwise. Then, the Crypto Service verifies the PoO (if it was sent), it then computes the hash sv, and it signs this hash. Then, the Crypto Service returns this signature to the LURK Engine. From here on, the Crypto Service 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 of $LURK1.2$ in DHE mode (Fig. 7 in Appendix C), 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, it therefore generates the pmk value and it only returns to the LURK Engine the channel key. In fact, our Variant 2 of $LURK1.2$ in DHE mode is an efficiency-driven variation of the 3(S)ACCE-K-SSL design proposed in [4]. Concretely, our Variant 2 of $LURK1.2$ in DHE mode does not require the heavy PKI that 3(S)ACCE-K-SSL needs for the context-soundness property (i.e., one certificate per each fragment delivered), as we do not aim to achieve this property.

Note: A detailed specification of $LURK1.2$ to the level of the network layer, packet formats, inner options, etc. is available at [22]. Section 4 will also detail on some of this.

2.2 *LURK*1.2'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 [17]. Meanwhile, [4] put forward *3(S)ACCE*, an ACCE-based model with formalisms and security requirements for "server-side delegated authenticated key-exchanges". So, for assessing LURK's security, we use the 3(S)ACCE model. We describe below the threat model and security requirements at the high-level; for the formal version, please refer to Appendix B, where we recall both the ACCE and the 3(S)ACCE models.

Threat Model. To recall, 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 "data 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). In the 3(S)ACCE model (recalled in Appendix B), these adversarial actions are formalised via oracle calls to a challenger simulating the protocol-execution.

Security Requirements for $LURK1.2$**.** The 3(S)ACCE formalism introduces four security requirements for proxied AKE protocols as described below (given formally, in Appendix B).

Entity Authentication (EA) [4]. An *EA attacker* can corrupt parties (i.e., making them do arbitrary actions), can open new sessions, can probe the results of sessions and can send its own messages. We say that there is an *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.

In the 3(S)ACCE formal model [4], the notion of "mixed-2-ACCE entity authentication" also appears; it is called *mixed* because "to the left" of the Engine – there is an unilateral authentication protocol, and "to the right" of the Engine – there is a mutually authenticated protocol, and the attacker needs to play the EA game both to the left and to the right at the same time.

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 can 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 Keyless SSL in RSA mode [4].

Accountability [4]. 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.

So, the $LURK1.2$ designs are expected to achieve channel security, entity authentication and accountability. Our position is that the first two requirements are essential (and should be demanded of all $LURK1.2$ designs); meanwhile, we view accountability as an optional security requirement, which one can consider trading off for the sake of efficiency. We will develop on these matters later in this section.

2.3 $LURK1.2$'s Design Choices: Different Levels of Security & Efficiency

$LURK1.2$'s Freshness Function φ. We included this mechanism in $LURK1.2$ in RSA mode to aid the enforcing of (channel security with) perfect forward secrecy. 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 nonce sent on the secure channel between the Engine and the Crypto Service. If later on, at time t, the adversary $\mathscr{A}$ corrupts the Engine E, said adversary will not find the old the N_i nonce in E's memory; we specifically require that N_i be deleted from E's memory at the end of its use. As such, since we require φ to be a non-programmable[11] PRF [**?**], the value N_E produced as $\varphi(N_i)$ cannot be guessed by the attacker. Since φ is a PRF, φ is also non-invertible and the adversary $\mathscr{A}$ cannot produce N_i out of N_E neither. So, $\mathscr{A}$ cannot present an old query to the Crypto Service. We will further discuss the PFS guarantees of $LURK1.2$ in RSA mode in Section 3.

We note that –whilst some cryptographic assumptions are needed– $LURK1.2$ in RSA mode builds in a PFS-enforcing mechanism that is more communication-effective than the repairs made to Keyless SSL via 3(S)ACCE-K-SSL in [4] (i.e., in $LURK1.2$, the Crypto Service does not need to send N_E to the Engine at the beginning of the handshake).

$LURK1.2$: Session Resumption vs. Accountability. *Session resumption* is a mechanism whereby the middle party in a proxied TLS connection can produce a new channel key ck without contacting the end-server, just by exchanging new nonces with the client and using an old msk. Variant 1 of $LURK1.2$ in RSA mode, the msk is returned to the TLS Engine E, and so the latter could perform session resumption. That said, LURK does *not* provide a mechanism for session resumption in its specification. And, in Variant 2 of $LURK1.2$ in RSA mode, the channel key ck is returned to the TLS Engine E, and so the latter can simply not perform session resumption.

So, if we modified LURK to explicitly open for session resumption, then –in practice– Variant 1 of $LURK1.2$ in RSA mode could become more communication-efficient than Variant 2 of $LURK1.2$ in RSA mode. Yet, Variant 2 of $LURK1.2$ in RSA mode achieves the aforementioned security requirement called accountability[12]. Thus, Variant 2 of $LURK1.2$ in RSA mode is closer to 3(S)ACCE-K-SSL in RSA mode in [4] (which also attains accountability); however, recall that $LURK1.2$ in RSA mode is more efficient that 3(S)ACCE-K-SSL in obtaining channel security with PFS.

Protection Against Signing Oracles. In LURK based on TLS 1.2 in RSA mode, the role of φ is to enforce freshness and as such prevent replay attacks.

Protection Against Malicious 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.

Protection Against Cross-Protocol Attacks. In the design-description, we mentioned the fact the Crypto Service would only accept specific data-structures for incoming messages and it will decline the communication otherwise. This is detailed further in our low-level specification found at [22]. These elements protect against cross-protocol attacks or injection attacks by a malicious Engine who would send illicit data to the Service.

[11] Non-programmability is just a detail pertaining to formal proofs: the great majority PRFs are non-programmable; see Appendix B for details.

[12] Following [4], it is known that session-resumption and accountability are mutually exclusive.

3 Formal Security Proofs & Analyses

We now discuss our formal security analysis of $LURK1.2$ in two parts. First, in Subsection 3.1, we provide the computational-security results [13] for Variants1 and 2 of $LURK1.2$ in RSA mode and for Variant 1 of $LURK1.2$ in DHE mode. This is done w.r.t. all security requirements mentioned in Section 2, incl. accountability.

Secondly, in Subsection 3.2, we use symbolic verification to show that $LURK1.2$ in RSA mode achieves perfect forward secrecy within its channel-security property.

3.1 Cryptographic-Analysis of $\boldsymbol{LURK1.2}$

In what follows, we state our provable-security results w.r.t. $LURK1.2$.

Using the 3(S)ACCE model in [4], we present the formal theorems and proofs of these statements in Appendix D.

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 [8], if the two protocols ensure 3(S)ACCE mixed entity authentication [4], if the signature and hash in TLS1.2 DHE mode are secure in their respective threat models, if the encryption in TLS1.2 RSA mode is secure, then Variant 1 of $LURK1.2$ in DHE mode and 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 Appendix D .

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 [8], if the two protocols ensure 3(S)ACCE mixed entity authentication [4], 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 non-programmable PRF [7], 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 Appendix D .

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 [8], if the two protocols ensure 3(S)ACCE mixed entity authentication, and the freshness function is a non-programmable PRF [7], 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 Appendix D .

[13] Computational or provable-security formalisms for security analysis consider messages as bitstrings, attackers to be probabilistic polynomial-time algorithms who will attempt to subvert cryptographic primitives, and attacks to have a probabilistic dimension the security parameters; e.g., [4] is a provable-security model for server-side delegated authenticated key exchange. Contrarily, symbolic models for security analysis abstract messages to algebraic terms, cryptographic primitives to be ideal and not subject to subversion by the adversary, and the attacks be possibilistic flaws mounted via a set of Dolev-Yao rules [10] applied over interleaved protocol executions.

3.2 Symbolic Verification of *LURK*1.2 in RSA Mode.

In Appendix D and Subsection 3.1, we prove and respectively recount that $LURK1.2$ in RSA mode attains channel security. Now, we aim to focus on the perfect forward secrecy (PFS) side of the channel security property. Namely, we use computer-assisted analysis to show that the bespoke way in which $LURK1.2$ in RSA mode introduces and uses the freshness function φ –which henceforth we call the "freshness mechanism"– does indeed attain channel security *with PFS*.

We use the ProVerif [5] symbolic verifier. ProVerif is based on applied pi-calculus [1]. As such, the protocol entities in our protocol (i.e., the Client, the LURK Engine, the Crypto Service) are modelled as applied-pi processes executing in parallel. The attacker is a separate process modelling a Dolev-Yao adversary [10].

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 simple terms, 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 3.

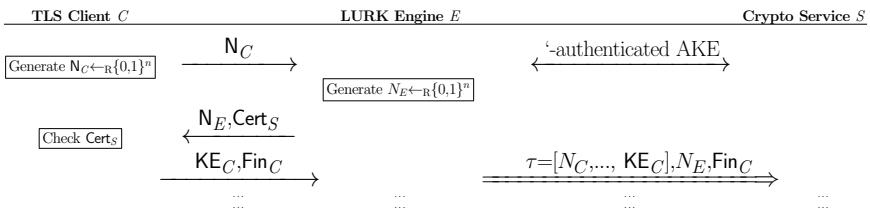


Fig. 3: "Weak LURK": $LURK1.2$ in RSA mode stripped of the freshness mechanism

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. Second, we need to formalise the property of channel security (with perfect forward secrecy) in ProVerif.

In the verification process, part of the aspects above would be abstracted into property over execution-traces, 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. But, 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: the attacker is able to query the Crypto Service and retrieve old master secrets. This is also confirmed by violating the secrecy property over the client master secret. ProVerif is able to show an attack trace with the attacker acting as an Engine and retrieving an old master secret as a result of querying the Crypto Service with captured data on the public channel.

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 (i.e., we rely on the "private" attribute in ProVerif). The results of the analysis show that, in $LURK1.2$, clear server random values can be accessed only by legitimate parties, i.e., received by the Crypto Service. In addition, the correspondence property holds for $LURK1.2$, guaranteeing that retrieval of past master secrets is no longer possible by querying the Crypto Service with cryptographic material inferred from the public channel. This formally proves that $LURK1.2$ employing the freshness mechanism is resilient against replay attacks from corrupted Engines.

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 Keyless SSL in RSA mode has a replay attack and does not attain PFS; this was only shown with "pen and paper" before, in [4]. To this end, the two analyses above also prove that our freshness mechanism represents a viable alternative to the solution proposed in [4] to patch the Keyless SSL 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 [23] .

4 System Implementation

pylurk [24], a Python implementation of $LURK1.2$ follows a modular Python implementation depicting a client/server setup between a `LURK` client (i.e., engine) and a `LURK` server (i.e., crypto service). We implemented Variants 1 of $LURK1.2$ in RSA and DHE mode. Our implementation supports UDP, TCP, TCP+TLS, HTTP and HTTPS for the interaction between the engine and the crypto service. UDP + DTLS has not been implemented because we were not able to find a suitable DTLS library in Python. Furthermore, UDP+DTLS would end up in a stateful protocol which removes the main characteristics of UDP. As a result, we followed the similar design as DoT [16] and DoH [12] 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 containerised environments where exchanges are performed on a given platform without exposure to the network.

We leveraged the socketserver module [26] for TCP and UDP implementation, the `http.server` [15] module for the HTTP implementation and `SSL` [27] as a TLS/SSL

"Weak LURK", without the freshness mechanism		
Security with PFS	**Code Excerpt**	**Comments**
Secrecy	**query secret** *mastersecretclient*. **query secret** *srv_rnd*.	**Result: False**. Both properties fail. Attack: an attacker assuming the role of the Engine obtains an old master secret with a replay attack.
Antireplay/ Corresp.	**query** *encpremaster*:bitstring, *srvrand*:bitstring; **inj-event**(keyserver_received_encpremaster (*encpremaster*,*srvrand*)) $\Longrightarrow$ **inj-event**(edge_resent_encpremaster (*encpremaster*,*srvrand*)).	**Result: False**. The query asserts that reception by the Crypto Service of a distinct pair of encrypted premaster secret with a server random N_E occurring only once in the same protocol run. The property fails, Attack: trace showing a replay of the same encrypted premaster Enc(pmk) and same server random N_E in two distinct instances of the Crypto Service process.
*LURK*1.2 **in RSA mode (Variant 1 encoded), with freshness mechanism**		
Security with PFS	**Code Excerpt**	**Comment**
Secrecy	**query secret** *mastersecretclient*. **query secret** *clearSrvRnd*.	**Result: True**. Both properties hold. Server random values may now be accessed only by legitimate parties and old master secrets cannot be obtained/inferred by the attacker.
Antireplay/ Corresp.	**query** *srvRand*:bitstring; **inj-event**(keyserver_recvd_srvrnd_clear (*srvRand*)) $\Longrightarrow$ **inj-event**(edge_sent_srvrnd_clear(*srvRand*)).	**Result: True**. The query asserts that reception by the Crypto Service of one given server random N_E occurs only once, as a result of an Engine transmitting this value. The property holds, guaranteeing that retrieval of old master secrets is no longer possible by re-querying Crypto Service with cryptographic material inferred from the public channel.

Table 1: ProVerif
Analysis of Channel Security of "Weak LURK" and *LURK*1.2 in RSA Mode

wrapper for socket objects to enable packets protection. We modified the TCPServer module implementation to allow multiple requests exchange per established session, eventually protected by TLS, with the client, which improves the performance. 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 results in a non optimal case.

As `LURK` server provides cryptographic services, we use the `Cryptodome` [9] package to allow the server to enforce cryptography primitives, while specific elliptic curve operations, such as the proof of ownership (PoO) of the DHE exponent, are enabled through `tinyec` [29]. Our implementation allows the use of SHA256, SHA384 and

SHA512 for the generation of the master secret. The freshness function φ is specified as SHA256. Other options can easily be added.

In RSA mode, the TLS Handshake is provided to the crypto service which also enforces the usage of specific cipher suites. In our case, we enforced the following cipher suites: TLS_RSA_WITH_AES_128_GCM_SHA256 and TLS_RSA_WITH_AES_256_GCM _SHA384. Encryption of the premaster secret was performed using a 2048-bit public key.

In DHE mode, namely ECDHE (Elliptic Curve Diffie Hellman), we enforced the use of secure hash functions in the signature scheme (SHA256 and SHA512) for both RSA and ECDSA. Similarly, secure elliptic curves (secp256r1, secp384r1, secp521r1) have been implemented for the generation of ECDHE, as well as for ECDSA signature. Experiments have limited the test to an RSA signature with SHA256 using a 2048-bit public key. ECDHE was performed using secp256r1. Again, other options can easily be added to the implementation.

Lastly, in RSA mode, the last message between the crypto service and the engine is sent (e.g., msk) instead of its encryption under the exported key k. This is because the channel is already secure and the said encryption is simply needed for strong 3(S)ACCE provable-security results but adds nothing to practical security. Note that more details on the system implementation can be found in Section 3 of our long version of this manuscript [20].

5 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 in terms of latency and CPU overhead. Further, we provide a comprehensive comparison of $LURK1.2$ with other works in the literature in Section 6 of our long version of this manuscript [20], given the space limitation herein. For all experiments, we use the Variants 1 of $LURK1.2$ in the `pylurk` [24] implementation (Section 4). 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.

5.1 Latency

For a given configuration, $l_{LURK} = p_{req} + RTT + p_{resp}$, is the measured latency 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 and can be estimated to: $\delta = \frac{l_{LURK}}{l_{TLS}}$. Note that such overhead is negligible if a UDP exchange is performed on a local host between the `LURK` engine and the Crypto Service.

Figure 4a 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 4b 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 4c – 4f 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 respective lack of such choices. The measurements shown in Figures 4c – 4f are performed over UDP.

On transport protocols The increased latency overhead introduced by TCP over UDP (i.e., a factor of 1.02 in RSA mode, 1.16 in RSA-Extended mode, and 1.02 in ECHDE mode) is a result of the TCP session establishment between an engine and the Crypto Service for all the requests (Figure 4a). In contrast, the additional latency overhead observed by HTTP over TCP (i.e., a factor of 1.46 in RSA mode, 1.25 in RSA-Extended mode and 1.50 for $LURK1.2$ in ECDHE mode) and by HTTPS over TCP+TLS depicted in Figure 4a and Figure 4b respectively, is due to the TCP session establishment for each new request between an engine and the Crypto Service.

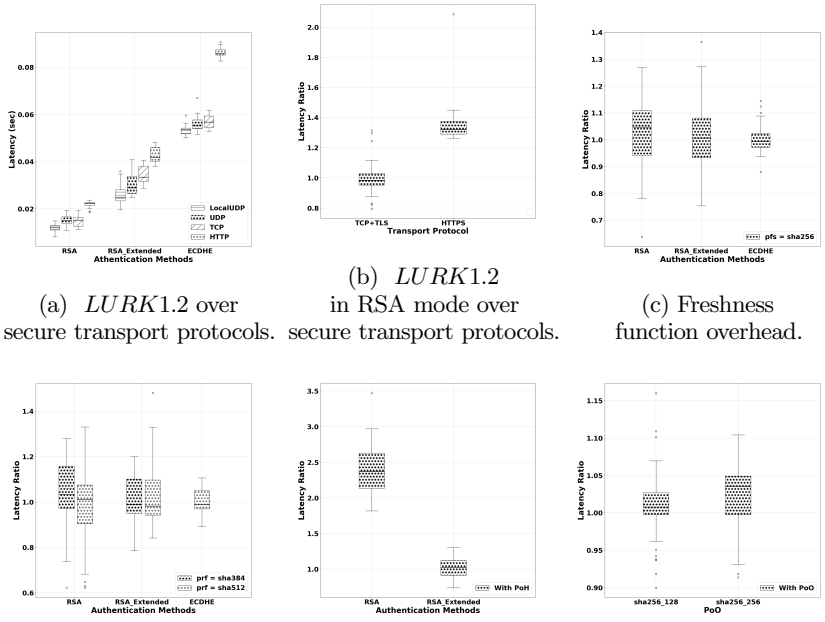
While UDP provides optimal performance, the lack of delivery control makes it a poor candidate for $LURK1.2$. Further, we identify no clear benefit from using HTTP instead of TCP, as for instance, the use of HTTP generates larger payloads. TLS does not impose measurable latency. As a result, we recommend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

Further, we note that the latency overhead introduced by $LURK1.2$ over TLS is limited in ECDHE mode but not in RSA mode, given that $LURK1.2$ implied more changes to TLS1.2 in RSA mode (e.g., use of freshness function, more interaction between the engine and the Crypto Service) than that in DHE mode. Figure 4b shows that 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: $\delta_{RSA}=1.27$, $\delta_{RSAExt.}=1.24$, $\delta_{ECDHE}=1.05$.

On TLS modes Figure 4a depicts that the latency of $LURK1.2$ varies with the underlying TLS mode. In fact, increased latency overhead is observed when using RSA extended and ECDHE modes in comparison to RSA mode. For example, $LURK1.2$ increases the latency by a factor of 2.2 and 3.73, in RSA Extended and ECDHE modes respectively, in comparison to RSA mode for TCP connections. The difference between RSA and RSA Extended is due to the additional processing and communication of the full TLS handshake. Whereas, the difference between ECDHE and RSA is mostly due to the cryptographic operations involved (e.g., more costly mathematical computations in ECHDE).

Other choices Figure 4c 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 which implies that the impact of φ on the overall latency is negligible. Figure 4d 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 4e 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



(a) $LURK1.2$ over secure transport protocols.

(b) $LURK1.2$ in RSA mode over secure transport protocols.

(c) Freshness function overhead.

(d) TLS' PRF overhead.

(e) PoH overhead.

(f) PoO overhead.

Fig. 4: Latency Measurements

RSA-Extended mode. Figure 4f depicts the impact of our added PoO of the DHE exponent over the average ECDHE latency. The impact observed is relatively negligible.

5.2 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. 5, 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. This is due to the efficient input/output processing managed by the HTTP libraries used, on one hand. On the other hand, the TCP implementation requires additional processing given the increased interactions between the user and the kernel (i.e., reading the `LURK` Header and the remaining bytes of the `LURK` request)

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 yield additional CPU overhead in the case of using $LURK1.2$ in RSA and RSA-Extended modes. 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.

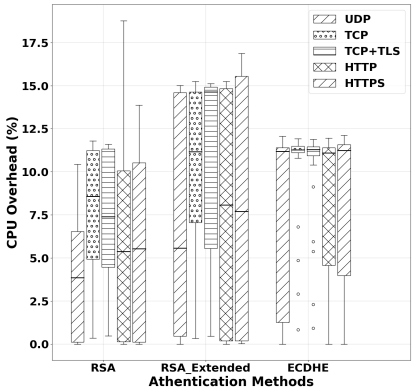


Fig. 5: CPU Overheads of the Crypto Service in different *LURK*1.2 modes.

6 Related Work

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.

We now detail Table 3. Liang et al. [19] 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. [19] 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 [6] 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* [2]. 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 [25], [**?**] 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 [**?**]. 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 [13] 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. [**?**] and Levy et al. [18] 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 [4] 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. [4] 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.

We also carried out the same type of comparison with TLS-delegation mechanisms, where the client is aware of the middlebox, i.e, Client-Invisible TLS Delegation.

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 [**?**] 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 [**?**], [**?**]. We recount these initiatives below.

First, *SplitTLS* [**?**] 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* [**?**] 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* [**?**] and *TLS Keyshare Extension* [**?**] 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)* [**?**] 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, [3] 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)* [**?**] 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.

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

Eighth, *BlindBox* [**?**] and *Embark* [**?**] 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	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

6.3 Keyless SSL [28] and 3(S)ACCE-K-SSL [28]

As we can see from Table 3, LURK aligns itself most with KeylessSSL [28] and 3(S)ACCE-K-SSL [4]. In fact, these CDN-driven architectures appeared first in a patent by Akamai [11]: 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* [28]. 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. [4] used a provable-security

Mechanism	Impact on TLS Client	Content-Owner control capabilities	Content-Owner audit capabilities
Shared long-term private key [19]	–	–	–
X.509 Name Constraints [6]	X.509 parsing	long-term private key, name	–
Delegated Credentials [2]	TLS ext.	name	–
STAR [25]	–	name	–
DANE [13]	DNSSEC	name	–
Stickler [18]	browser plugin	long-term private key, name, content	–
KeylesSSL [28]	–	long-term private key, name	–
3(S)ACCE-K-SSL [4]	–	long-term private key, name	yes
LURK	–	long-term private key, name	yes

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

approach to show several vulnerabilities on KeylessSSL: e.g., forward-secrecy attacks, signing oracle attacks or cross-protocol attacks, etc. So, Bhargavan et al. [4] 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[14] and by some different design choices[15], 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,

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

[15] 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).

the decryption is performed by the Crypto Service, while the computation of the master secret is performed by the Engine[16]. 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 [4].

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 *LURK*1.2 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 5, 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 [**?**] 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 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

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

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 [4].

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

	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
KeylessSSL	4	–	no	–	no	no

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

7 Discussions, Future Work & Conclusions

Our suite of designs, called *LURK*1.2, 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 [28], in a spirit similar to that of the recent 3(S)ACCE-KSL protocol in [3]. 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 [20], 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 [21]. In the long version of this paper [20], 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.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL'01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 104–115. ACM Press (2001)
2. Barnes, R., Iyengar, S., Sullivan, N., Rescorla, E.: Delegated Credentials for TLS. Internet-Draft draft-draft-ietf-tls-subcerts, Internet Engineering Task Force (Feb 2019), https://datatracker.ietf.org/doc/html/draft-draft-ietf-tls-subcerts, work in Progress
3. Bhargavan, K., Boureanu, I., Delignat-Lavaud, A., Fouque, P.A., Onete, C.: A Formal Treatment of Accountable Proxying over TLS. In: Proceedings of IEEE S&P. IEEE (2018)
4. Bhargavan, K., Boureanu, I., Fouque, P., Onete, C., Richard, B.: Content delivery over TLS: a cryptographic analysis of Keyless SSL. In: Proceedings of Euro S&P (2017)
5. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: IEEE Computer Security Foundations Workshop. pp. 82–96. IEEE Computer Society Press, Novia Scotia, Canada (2001)
6. Boeyen, S., Santesson, S., Polk, T., Housley, R., Farrell, S., Cooper, D.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (May 2008). https://doi.org/10.17487/RFC5280, https://rfc-editor.org/rfc/RFC5280.txt
7. Boureanu, I., Mitrokotsa, A., Vaudenay, S.: On the pseudorandom function assumption in (secure) distance-bounding protocols. In: Progress in Cryptology – LATINCRYPT 2012. pp. 100–120. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
8. Brzuska, C., kon Jacobsen, H., Stebila, D.: Safely exporting keys from secure channels: on the security of EAP-TLS and TLS key exporters. In: EuroCrypt (2016)
9. PyCryptodome: a self-contained Python package of low-level cryptographic primitives. https://pycryptodome.readthedocs.io (2019)
10. Dolev, D., Yao, A.: On the Security of Public-Key Protocols. IEEE Transactionson Information Theory 29 **29**(2) (1983)
11. Gero, C., Shapiro, J., Burd, D.: Terminating ssl connections without locally-accessible private keys (Jun 20 2013), http://www.google.co.uk/patents/WO2013090894A1?cl=en, wO Patent App. PCT/US2012/070075
12. Hoffman, P.E., McManus, P.: DNS Queries over HTTPS (DoH). RFC 8484 (Oct 2018). https://doi.org/10.17487/RFC8484, https://rfc-editor.org/rfc/RFC8484.txt
13. Hoffman, P.E., Schlyter, J.: The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Aug 2012). https://doi.org/10.17487/RFC6698, https://rfc-editor.org/rfc/RFC6698.txt
14. Housley, R.: Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. RFC 7696 (Nov 2015). https://doi.org/10.17487/RFC7696, https://rfc-editor.org/rfc/RFC7696.txt
15. http.server: HTTP Servers. https://docs.python.org/3.4/library/http.server.html (2018)
16. Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., Hoffman, P.E.: Specification for DNS over Transport Layer Security (TLS). RFC 7858 (May 2016). https://doi.org/10.17487/RFC7858, https://rfc-editor.org/rfc/RFC7858.txt
17. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Proceedings of CRYPTO 2012. LNCS, vol. 7417, pp. 273–293 (2012)
18. Levy, A., Corrigan-Gibbs, H., Boneh, D.: Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser. IEEE Security Privacy **14**(2), 22–28 (2016)
19. Liang, J., Jiang, J., Duan, H., Li, K., Wan, T., Wu, J.: When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In: 2014 IEEE Symposium on Security and Privacy. pp. 67–82 (May 2014)
20. Lurk: Practical and secure server-controlled tls delegation (long version). https://github.com/anon-data/anon_src/blob/master/long_version.pdf (2019)

21. Migault, D.: LURK Extension version 1 for (D)TLS 1.3 Authentication. Internet-Draft draft-mglt-lurk-tls13-02, Internet Engineering Task Force (Apr 2020), https://datatracker.ietf.org/doc/html/draft-mglt-lurk-tls13-02, work in Progress
22. Migault, D., Boureanu, I.: LURK Extension version 1 for (D)TLS 1.2 Authentication. Internet-Draft draft-mglt-lurk-tls12-02, Internet Engineering Task Force (Jan 2020), https://datatracker.ietf.org/doc/html/draft-mglt-lurk-tls12-02, work in Progress
23. Symbolic analysis of LURK1.2 with ProVerif. https://github.com/anon-data/anon_src/tree/master/pv (2019), anonymised for submission
24. pylurk – a Python implementation of LURK. https://github.com/mglt/pylurk (2019)
25. Sheffer, Y., Lopez, D., de Dios, O.G., Pastor, A., 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 2018), https://datatracker.ietf.org/doc/html/draft-draft-ietf-acme-star, work in Progress
26. socketserver: A framework for network servers. https://docs.python.org/3.4/library/socketserver.html (2018)
27. SSL: TLS/SSL wrapper for socket objects. https://docs.python.org/3.4/library/ssl.html (2018)
28. Stebila, D., Sullivan, N.: An analysis of tls handshake proxying. In: Trustcom/BigDataSE/ISPA, 2015 IEEE. vol. 1, pp. 279–286 (Aug 2015). https://doi.org/10.1109/Trustcom.2015.385
29. tinyec. https://github.com/alexmgr/tinyec (2018)

A TLS 1.2

We show the TLS1.2 handshake (i.e., the secure-channel establishment part of TLS 1.2) in Figure 6.

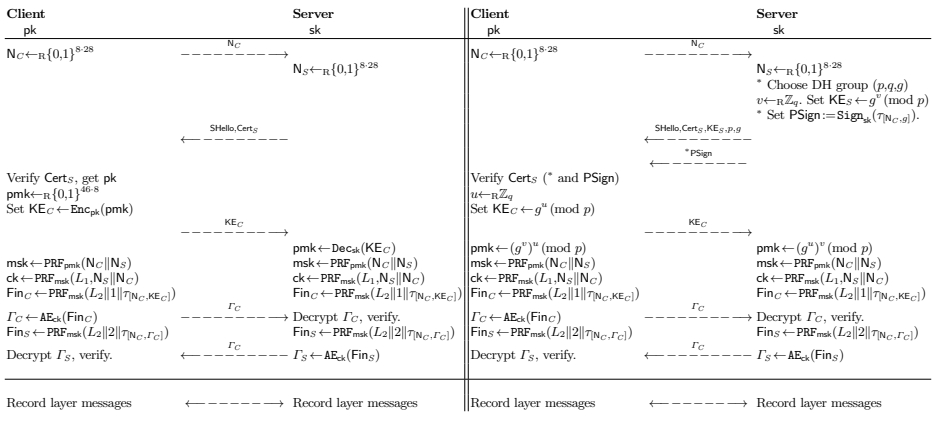


Fig. 6: TLS1.2 handshake. Left: in RSA mode; Right: in DHE mode.

TLS 1.2 Handshake in RSA Mode. The client C sends a nonce N_C to the server S, which responds with its own 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 $(v,g^v (\mathrm{mod}\ p))$. 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 $(u,g^u (\mathrm{mod}\ p))$. Both client and server then compute the pre-master secret pmk as $g^{uv} \pmod p$. The rest of the protocol and computations (msk,ck,Fin_C,Fin_S) proceed as in the RSA mode.

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

B.1 The (S)ACCE Models [17]

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) [4]. 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.

B.2 The 3(S)ACCE Model [4]

In [4], 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 [4] to re-use 2-party security definitions for authentication and channel security.

Accountability is a new security notion introduced in [4] 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 [4].

Entity Authentication (EA) [4]. 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) [4]. 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* 3(S)ACCE *protocol, if it terminates the channel security game with a tuple (π_i^j, b') 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:

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

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

Accountability (Acc) [4]. 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* $\pi_j^n \in P_j$.Instances *with all their attributes), outputs* π_i^m.ck.

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

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

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

B.3 Programmable PRFs

"Programmable PRFs" [**?**] capture PRFs that behave randomly to someone who does not know the key of its instances, but not to someone who knows said keys. In other words, there exist functions called "programmable PRFs" that are PRFs, but that contain trapdoors, *i.e.*, there exist chosen input values related to the key of the PRF instances, and for these inputs the PRF output is not random to those having provided the input. The notion of non-programmable PRFs comes to fill in the gap of security proofs that would need the PRF assumption at their bases, yet the adversary knows the keys of said PRF and/or the key of the PRF instance is used somewhere else in the protocol (and thus the "classical" PRF assumption does not apply).

Dishonest Engines do know their keys of PRF instances used in our construction, so they can exploit programmable PRFs. Also the key exported from the AKE protocol run between the Engine and the Service is used to key said PRF as well as at the end of the LURK handshake between the Engine and the Service. As such, we will need to assume that, e.g., the freshness function ϕ, is a non-programmable PRF. Note that most PRFs are non-programmable PRFs.

C *LURK*1.2 in DHE mode –Variant2

In this section, we present Variant2 of *LURK*1.2 in DHE mode. This is in fact the same as the 3(S)ACCE-K-SSL design [4], 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 *LURK*1.2 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 [4].

D LURK's Proofs

We now present different security proofs for LURK, using the 3(S)ACCE model in [4] and recalled in Section B.

D.1 Entity Authentication Proofs

We now prove the entity-authentication security of *LURK*1.2 in all variants.

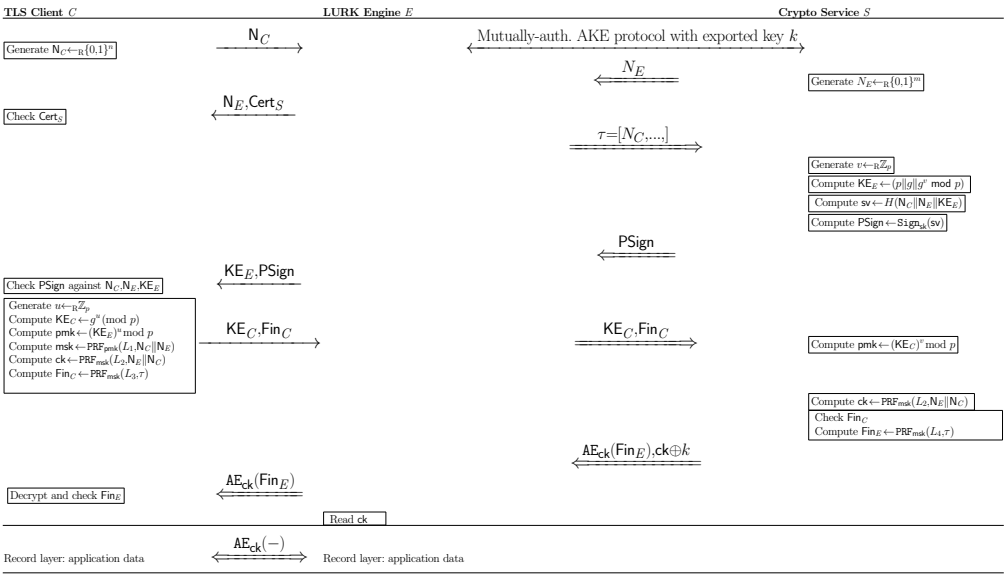


Fig. 7: *LURK*1.2 based
on TLS1.2 in DHE mode: Variant 2; in line with the 3(S)ACCE-K-SSL design [4]

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 LURK*1.2*, 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 LURK*1.2 *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 LURK*1.2 *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 B.

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

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

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,

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

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

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

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 premaster secret pmk of π_j^n and the transcript of π_i^m. In this case, we recall that the partnering in 3(S)ACCE 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 $\mathbf{Pr}[\mathscr{A}_{\mathbb{G}_3}\ \mathsf{wins}]$ 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 $\mathbf{Pr}[\mathscr{A}_{\mathbb{G}_3}\ \mathsf{wins}]$ is increased by the factor $\mathsf{n_P}^3 \cdot \mathsf{Adv}^{mEA}_{P,P'}(\mathscr{A}_3)$[17].

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 π_i^m.InstSet partnered with π_i^m. W.r.t. the our current EA game and the mEA-game, the simulation for RegParty, NewSession,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 $\mathsf{n_P}^3 \cdot \mathsf{Adv}^{\text{2-ACCE}}_{P'}(\mathscr{A}_2) + \mathsf{n_P}^3 \cdot \mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_4)$. 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 $\mathsf{n_P}^3 \cdot \mathsf{Adv}^{\text{2-ACCE}}_{P'}(\mathscr{A}_2)$

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

[17] 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.

P_j holding pmk (in RSA) or $(p,g,KE_S;Cert_E;PSign)$ (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 $(p,g,KE_S;Cert_E;PSign)$, 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.2 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 non-programmable PRF φ, such that

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

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

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

$$\begin{aligned}\mathsf{Adv}^{\mathsf{SC}}_{\Pi}(\mathscr{A}) \leq\ & (2\mathsf{n_P}^2+2\mathsf{n_P}^3)\cdot\mathsf{Adv}^{\text{2-ACCE}}_{P'}(\mathscr{A}_2)+(\mathsf{n_P}^3+\\ &+\mathsf{n_P}^2)\mathsf{Adv}^{\text{2-SACCE}}_{P}(\mathscr{A}_1)+\mathsf{n_P}^3\mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_3)\\ &+\mathsf{n_P}^2\mathsf{Adv}^{\mathsf{SC-SACCE}}_{P}(\mathscr{A}_4)+\mathsf{n_P}^3\cdot\mathsf{Adv}^{\mathsf{npPRF}}(\mathscr{A}_5).\end{aligned}$$

Proof. Our proof has the following hops:

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

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:

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

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 :

$$\Pr[\mathscr{A}_{\mathbb{G}_3} \text{ wins}] \leq \frac{1}{2} + \mathsf{n_P}^3(\mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_3) + \mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1) + \mathsf{Adv}_{P'}^{\text{2-ACCE}}(\mathscr{A}_2)) + \mathsf{n_P} \cdot \mathsf{Adv}_{\mathsf{Sign}}^{\mathsf{Unforg}}(\mathscr{A}_4).$$

– For RSA :

$$\Pr[\mathscr{A}_{\mathbb{G}_3} \text{ wins}] \leq \frac{1}{2} + \mathsf{n_P}^3(\mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_3) + \mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1) + \mathsf{Adv}_{P'}^{\text{2-ACCE}}(\mathscr{A}_2)) + \mathsf{n_P}^2 \cdot \mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_3) \; \mathsf{n_P}^3 \cdot \mathsf{Adv}^{\mathsf{npPRF}}(\mathscr{A}_5).$$

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 non-programmable PRF assumption and produce an adaptive N_E across several session to learn msk or ck for a new session.

D.3 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:*
$\mathsf{Adv}^{\mathsf{Acc}}_{v2-\ LURK1.2-RSA}(\mathscr{A}) \leq 2 \cdot \mathsf{n_P}^2 \cdot \mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1)$

Proof. Our proof has the following hops:

Game 0: This game works as the Acc- game recalled Appendix B. 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:

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

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 $\mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1)$.

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.*,

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