LURK: Server-Controlled TLS Delegation

Ioana Boureanu*, Daniel Migault†, Stere Preda†, Hyame Assem Alamedine†, Sanjay Mishra‡, Frederic Fieau§, and Mohammad Mannan¶
*University of Surrey, †Ericsson Security, ‡Verizon, §Orange, ¶Concordia University

***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, Cloudflare, Telefonica, Ericcson), standardization bodies (e.g., IETF, ETSI), and academic researchers have proposed numerous ways of achieving *safer* TLS delegation. We present *LURK* the `LURK` (Limited Use of Remote Keys) extension for TLS 1.2, a suite of designs for TLS delegation, where the TLS-server is aware of the middlebox. We implement and test *LURK*. We also cryptographically prove and formally verify, in Proverif, the security of *LURK*. Finally, we comprehensively analyze how our designs balance (provable) security and competitive performance.**

***Index Terms*—Internet security, Middleboxes, Cryptographic protocols, Transport protocols, TLS, Proverif**

I. 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 networks enabled by edge computing. We shall refer to this internet-traffic mediation by middleboxes as "collaborative content-delivery". The latter is forecast [39] to increase even further the number of players collaborating in the delivery of content and services over the Internet. To this end, tech giants (Akamai, Cloudflare, Telefonica, Ericcson) and standardisation bodies (ETSI, IETF) alike have recently devoted considerable attention to third-party-driven security [38].

For *unencrypted traffic*, collaborative content-delivery fits the following architecture. A third party which is "on path" between the end-client and the end-server simply processes packets, and in this way, using a principle called implicit signalling [15]. 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 much more common. Normally, the record layer of TLS follows a design of end-to-end encryption between two purported interlocutors, e.g., a client and a server. Yet, as we already mentioned, collaborative content-delivery and/or traffic servicing is both a chronic and an acute need. 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 of 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 has become ubiquitous, the Internet Architecture Board (IAB) calls for making all proxies collaborating in traffic delivery visible to the endpoints [15]. 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 [14]. 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* [34]. 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 3(S)ACCE-K-SSL, an alternative design of a 2 party Authenticated and Confidential Channel Establishment (ACCE) where the handshake is run in the presence of middleboxes such as CDN edge servers. 3(S)ACCE-K-SSL 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* that meets in the middle between the (insecure) Keyless SSL [34] 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. Also, *LURK* does takes into account recommendations made for KeylessSSL w.r.t. its replay-driven insecurities based on corruption of content-delivery party [4], by building in new mechanisms to avoid replay attacks.

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

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

4. In *LURK*, we include a "freshness mechanism" to counter replay attacks [1]. As such, we encode *LURK* in RSA mode in the automatic protocol-verifier ProVerif and we formally check that perfect forward secrecy holds in *LURK* 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 *LURK* in RSA mode.

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

Why *LURK*? 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 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. It is likely that the transition from TLS 1.2 to TLS 1.3 "in the wild" will take some time. 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 *LURK*'s aim is to bridge this gap and improve the security of existing TLS 1.2 deployment.

II. *LURK*'s Use Cases

Our proposed architecture, *LURK*, practically splits a TLS server into two micro-services: the *LURK* Engine and the Cryptographic Service (Crypto Service). This enables that the ownership of long-term cryptographic credentials and the execution of the bulk of the TLS handshake be operated by independent parties. The following use-cases could grasp the benefits of such an architecture.

[1]Since TLS 1.2 RSA mode does not ensure forward secrecy, placing a mediating party in between the client and the server can lead to replay attacks. This was shown 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.

A. Complex CDN-ing

Firstly, the Streaming Video Alliance (SVA) [35] brings together content providers, commercial CDN operators and network operators to collaborate on a partnership that allows to seamlessly provide abilities to offload video-based content to caches deep into the network-operator's edge. Secondly, leveraging work of the IETF CDN Interconnection Working Group (CDNI) [11], the Open Caching Working Group (OCWG) [28] has specified an architecture as well as an API to enable the delegation of content from between CDNs. An *upstream CDN (uCDN)* is a CDN that is willing to delegate content to a *downstream CDN (dCDN)*. CDNI and OCWG look at making this type of uCDN-to-dCDN delegation more workable. However, in this case of complex CDN-ing, placing trust on possibly unknown dCDNs to handle private keys on behalf of the content-owner is a big ask. So, this type of CDN-ing can benefit from the *LURK* architecture to enable delegation of encrypted content without sharing the private keys. Concretely, the *LURK* architecture allows uCDNs to delegate across the different administrative domains to dCDNs, without sharing long-term security credentials.

Current implementations of this architecture, in the absence of *LURK*, are forced to implement inferior solutions. For instance, they use short expiry of keys which cause operations overheads. Or, they create dedicated sub-domains for the dCDN's use, which cause security risks. Lastly, sometimes the usage of the dCDN's domain is enforced, simply to avoid sharing long-term security credentials.

The *LURK* framework allows for encrypted TLS handshake without sharing the private key to the delegated CDN. This makes *LURK* a natural fit for delegation of video streaming sessions across different administrative domains. The latency overhead could be solved by leveraging TEE and extending the trusted domain of the delegating CDN into the delegated CDN's infrastructure.

B. Service-to-Service Platforms

In service-based architectures, there are often service-mesh technologies called upon to create representation of a service as the interconnection of micro-services. In middle parties in these interconnections are called mesh proxies or sidecars. A recurrent feature is the interconnection of sidecars with TLS based on short lived private keys. An example is Istio [22], running on the Kubernetes Engine. In Istio, a particular component in the service-mesh control-plane is in charge of frequently triggering private-key rotation (e.g., Istio Citadel). It is a fact that such component plays a critical role and its corruption may lead to exposure of the managed secrets.

To solve the problem of the large amount of trust placed on the service-mesh control-plane, we could look at the *LURK* Crypto Service be deployed in virtual multi-tenant environments. To solve the issue, in fact, we would need an in-extremis solution where we colocate the Crypto Service with the TLS Engine. Bearing in mind that the latter can be deployed on untrusted platforms, a requirement is the availability of a root-of-trust, such as TEEs (Trusted Execution

Environments, e.g., Intel SGX [21], in order to instantiate the Crypto Service inside it.

At an initial assessment, we expect that a *LURK* deployment with the Crypto Service inside a TEE and the Engine implemented by sidecars: (1) enables a tighten control over the private keys alongside the necessary cryptographic operations on it; and (2) reduces the critical role of entities in charge of frequent key rotation and certificate renewal. Nonetheless, a challenge remains provisioning of TEE and managing the lifecycle of the Crypto Service inside it. IETF created the Trusted Execution Environment Provisioning Working Group (TEEP) [36] which we follow closely.

III. Related Work

A. 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 I, 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 I, that *LURK* is a competitive solution on this desirable space of secure, backwards-aware TLS delegation. More details on every mechanism listed in Table I are provided in Appendix D of the long version of the paper [8].

Mechanism	Impact on TLS Client	CO* control capabilities	CO* audit capabilities
Shared long-term private key [25]	–	–	–
X.509 Name Constraints [7]	X.509 parsing	long-term private key, name	–
Delegated Credentials [2]	TLS ext.	name	–
STAR [31]	–	name	–
DANE [17]	DNSSEC	name	–
Stickler [24]	browser plugin	long-term private key, name, content	–
KeylesSSL [34]	–	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 I: Client-Invisible, Server-Controlled TLS Delegation for CDNs —(*) CO: Content Owner

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

B. Keyless SSL and 3(S)ACCE-K-SSL

As we can see from Table I, *LURK* aligns itself most with KeylessSSL [34] and 3(S)ACCE-K-SSL [4]. In fact, these CDN-driven architectures appeared first in a patent by Akamai [14]: i.e., TLS-delegation systems where the TLS long-term private key stays on the server-side and the associated certificate goes with the middlebox, who can therefore impersonate the server in a way invisible to the client. In 2015, Cloudflare commercialised a version of this, in a product called *KeylessSSL* [34]. 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 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[2] and by some different design choices[3], *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 it requires 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., *LURK* Variant 1 can support session ID resumption if needed, whereas *LURK* 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.

Some further comparisons with KeylessSSL and 3-(S)ACCE-K-SSL are given in the long version of this paper [8].

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

[3]To add Perfect Forward Secrecy (PFS) to *LURK* 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).

IV. *LURK*: Delegated Secure Delivery with Server Control

LURK is a suite of designs to delegate TLS 1.2 credentials without any changes on 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[4] 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 Crypto 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 [10]—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.

A. The *LURK* Designs

LURK instantiated with TLS 1.2, called LURK, is parametric in a security parameter s, as well as on a function[5] 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 *LURK*. 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.

***LURK* in RSA Mode.** Figure 1 presents the *LURK* protocol[6]. based on TLS 1.2 in RSA-mode. We propose two slightly different variants of *LURK* 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 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, *LURK* 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.

***LURK* in RSA Extended Mode.** *LURK* in RSA Extended mode only differs from *LURK* 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.

***LURK* in DHE Mode.** W.r.t. *LURK* in DHE mode, we also propose two variants. The first is presented in Figure 2, and the second in Figure 9 —found in Appendix C of the long version of the paper [8]. In the first variant of *LURK* 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.

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 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 *LURK* in DHE mode (Fig. 9 in Appendix C of the long version of the paper [8]), 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 *LURK* in DHE mode is an efficiency-driven variation of the 3(S)ACCE-K-SSL design proposed in [4]. Concretely, our Variant 2 of *LURK* in DHE mode does not require the heavy PKI that 3(S)ACCE-K-SSL needs for the content-soundness property (i.e., one certificate per each fragment delivered), as we do not aim to achieve this property —see section III-B.

Note: A detailed specification of *LURK* , to the level of the network layer, packet formats, inner options, etc. is available at [27]. Section VI will also provide detail in this regard.

[4]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.

[5]We do not hard code this function in the design as per the guidelines of [18]. 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.

[6]Please see Appendix A of the long version of the paper [8] for details on TLS 1.2

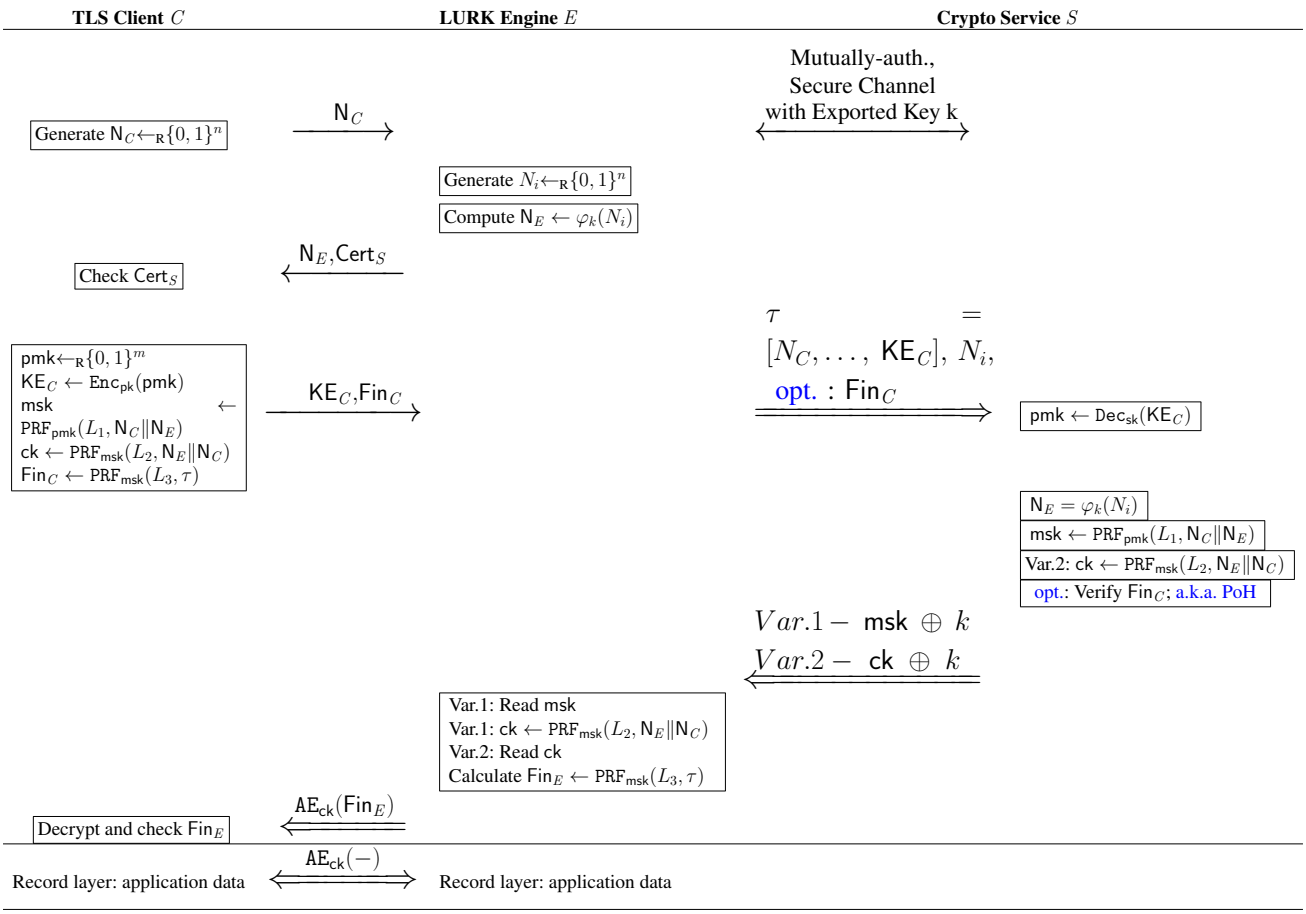


Fig. 1: *LURK* based on TLS 1.2 in RSA mode: Two Variants

B. *LURK*'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 [23]. 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 C in the long version of the paper [8], 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 C in the long version of the paper [8]), these adversarial actions are formalised via oracle calls to a challenger simulating the protocol-execution.

Security Requirements for *LURK*. The 3(S)ACCE formalism introduces four security requirements for proxied AKE protocols as described below (given formally in the long version of the paper [8]).

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. Above, X, Y can either be 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

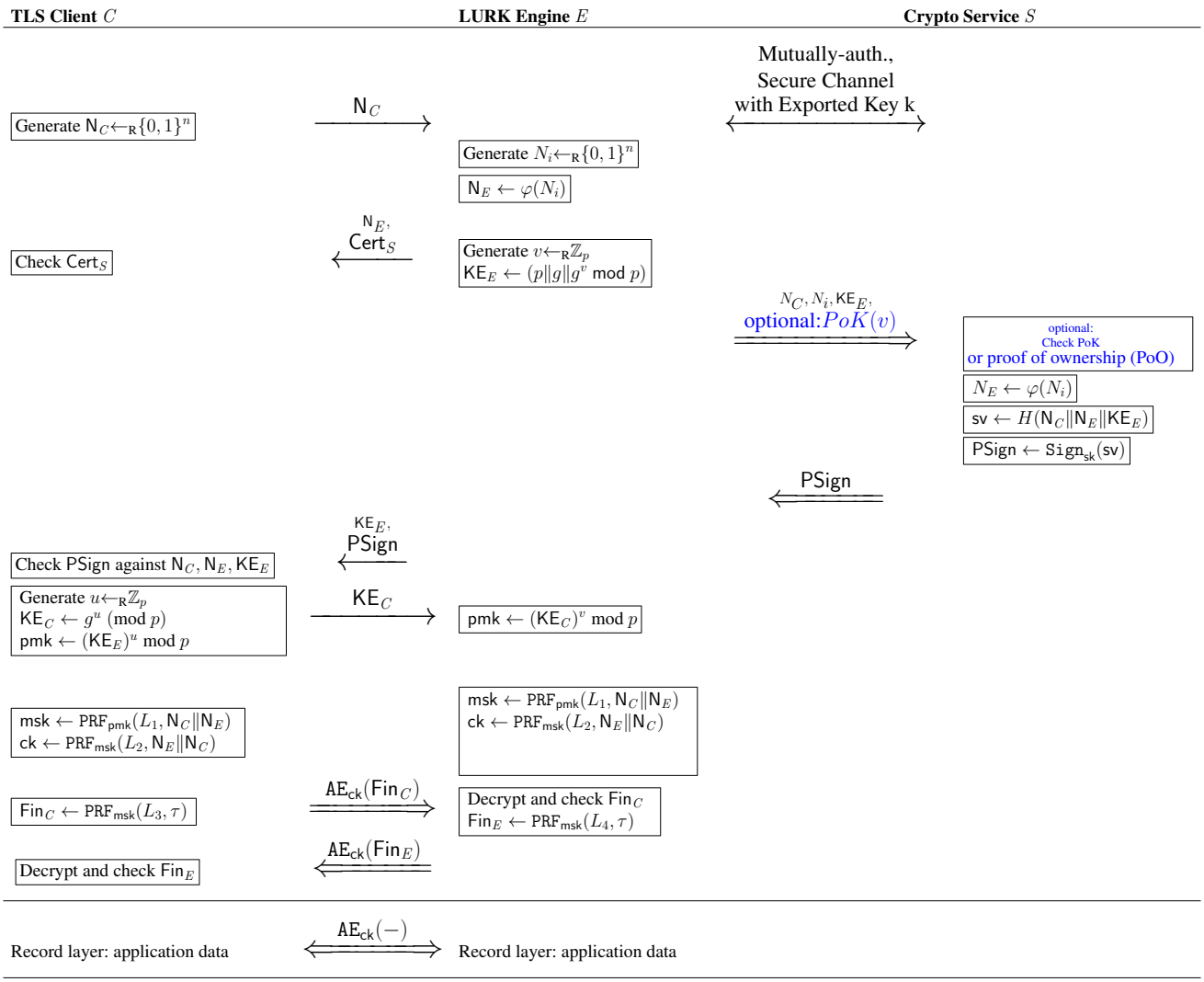


Fig. 2: *LURK* based on TLS 1.2 in DHE mode: Variant 1

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 TLS 1.2 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 empowers the Crypto Service to audit the activity of the *LURK* Service at the record layer, should this be required.

So, the *LURK* 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 *LURK* designs); meanwhile, we view accountability as an optional security requirement, which one can consider trading off for the sake of efficiency.

Note: In the long version of the paper found at [8], at this point, we discuss the design choices of *LURK* in relation to the above requirements and efficiency.

V. Formal Security Proofs & Analyses

We now discuss our formal security analysis of *LURK* in two parts. First, in Subsection V-A, we provide the computational-security results [7] for Variants 1 and 2 of *LURK* in RSA mode and for Variant 1 of *LURK* in DHE mode. This is

[7]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 [13] applied over interleaved protocol executions.

done w.r.t. all security requirements mentioned in Section IV, including accountability.

Secondly, in Subsection V-B, we use symbolic verification to show that *LURK* in RSA mode achieves PFS within its channel-security property.

A. Cryptographic-Analysis of *LURK*

In what follows, we state our provable-security results w.r.t. *LURK*. Using the 3(S)ACCE model in [4], we present the formal theorems and proofs of these statements in Appendix E of the long version of the paper [8].

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 [10], if the two protocols ensure 3(S)ACCE mixed entity authentication [4], if the signature and hash in TLS 1.2 DHE mode are secure in their respective threat models, if the encryption in TLS 1.2 RSA mode is secure, then Variant 1 of LURK *in DHE mode and Variants 1 and 2 of* LURK *in RSA mode are entity-authentication secure in the 3(S)ACCE model.*

This is formalised and proven in Theorem 1 of Appendix E of the long version of the paper [8].

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 [10], if the two protocols ensure 3(S)ACCE mixed entity authentication [4], if the signature in TLS 1.2 DHE mode is secure in its threat models plus, respectively, if the encryption in TLS 1.2 in RSA mode is secure and the freshness function is a non-programmable PRF [9], then Variant 1 of LURK *in DHE mode and, respectively, Variants 1 and 2 of* LURK *in RSA mode attain channel security in the 3(S)ACCE model.*

This is formalised and proven in Theorem 2 of Appendix E of the long version of the paper [8].

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 [10], if the two protocols ensure 3(S)ACCE mixed entity authentication, and the freshness function is a non-programmable PRF [9], then Variant 2 of LURK *in RSA mode attains accountability in the 3(S)ACCE model.*

This is formalised and proven in Theorem 3 in Appendix E of the long version of the paper [8].

B. Symbolic Verification of *LURK* in RSA Mode

In Appendix E of the long version of the paper [8] and Subsection V-A, we prove and respectively recount that *LURK* in RSA mode attains channel security. Now, we aim to focus on the PFS side of the channel security property. Namely, we use computer-assisted analysis to show that the bespoke way in which *LURK* 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 given that it is fully automated, supports an unlimitted number of protocol sessions and can prove various security properties such as secrecy and correspondence [6]. 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 [13].

Weak LURK. To reach our goal, we also model and check a modified version of *LURK* 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.

Symbolic Formalisation of *LURK*'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 PFS. Second, we need to formalise the property of channel security (with PFS) 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 is 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 PFS) 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) *LURK* 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 PFS 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 *LURK*. As opposed to "weak *LURK*", *LURK* introduces the

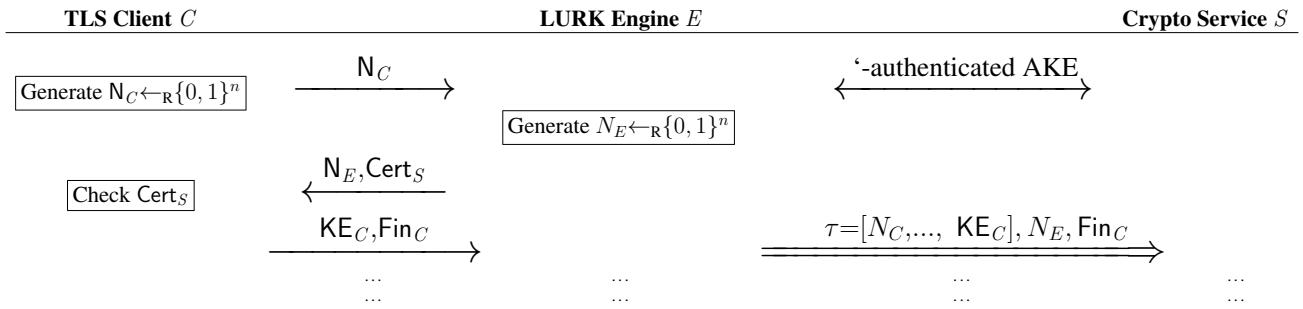


Fig. 3: "Weak *LURK*": *LURK* in RSA mode stripped of the freshness mechanism

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 *LURK*, 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 *LURK*, 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 *LURK* 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 PFS (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 II gives details on the property-encoding and summarises the results of the verification. Our ProVerif code can be found at [29].

VI. System Implementation

`pylurk` [30], a Python implementation of *LURK* , 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 *LURK* 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 [20] and DoH [16] and limited our scope for the CDN use case to TCP+TLS and HTTPS. UDP is left to the usage of *LURK* 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 [32] for TCP and UDP implementation, the `http.server` [19] module for the HTTP implementation and `SSL` [33] as a TLS/SSL 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` [12] 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` [37]. 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 [8].

Security with PFS	Code Excerpt	Comments
"Weak *LURK*", without the freshness mechanism		
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 $Enc(\mathsf{pmk})$ and same server random N_E in two distinct instances of the Crypto Service process.
***LURK* in RSA mode (Variant 1 encoded), with freshness mechanism**		
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 II: ProVerif Analysis of Channel Security of "Weak *LURK*" and *LURK* in RSA Mode.

VII. Performance Evaluation

We now investigate the performance of *LURK* vs. that of a classical TLS 1.2 handshake, and study how different design and implementation choices in *LURK* impact its overall performance in terms of latency and CPU overhead. Further, we provide a comprehensive comparison of *LURK* with other works in the literature in Section 6 of our long version of this manuscript [8], given the space limitation herein. For all experiments, we use the Variants 1 of *LURK* in the `pylurk` [30] implementation (Section VI). 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.

A. 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 Engine and the Crypto Service. We measure RTT on a local network, approximating the latency within a data center. The overhead of *LURK* compared to a standard TLS handshake can roughly be approximated as the latency between the 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.

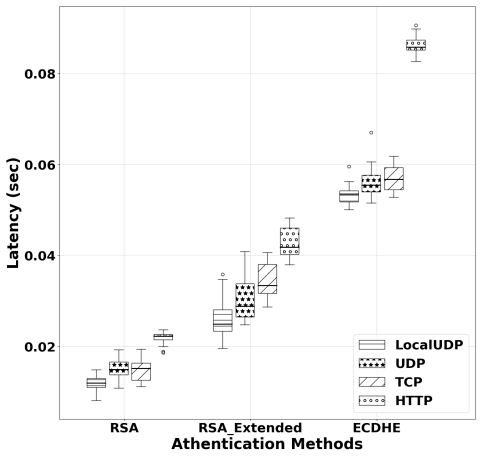
Figure 4a shows the latency in seconds for different *LURK* modes (i.e., RSA, RSA-extended and ECDHE) for different transport configurations (i.e., local UDP, UDP, TCP, HTTP). Figure 4b depicts the latency ratio of having *LURK* in RSA mode over TCP+TLS and HTTPS, compared to *LURK* in RSA mode over TCP, HTTP. In these cases, the PRF function in TLS and the φ freshness function we introduced in *LURK* are set to SHA256. Figures 5a – 6b show the latency ratio of *LURK* 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 5a – 6b 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 *LURK* 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.

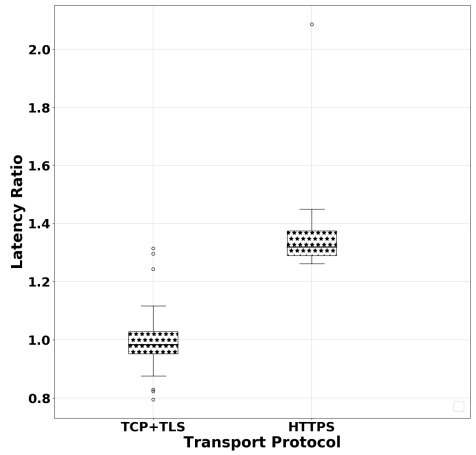
While UDP provides optimal performance, the lack of delivery control makes it a poor candidate for *LURK*. Further, we identify no clear benefit from using HTTP instead of TCP, as for instance, the use of HTTP generates larger payloads. TLS does not impose measurable latency. As a result, we recommend 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 *LURK* over TLS is limited in ECDHE mode but not in RSA mode, given that *LURK* implied more changes to TLS 1.2 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,

LURK (vs. TLS) increases the latency by a factor of 1.3. With TCP+TLS, the overhead of using *LURK* over the standard TLS 1.2 is estimated to be: $\delta_{RSA} = 1.27$, $\delta_{RSAExt.} = 1.24$, $\delta_{ECDHE} = 1.05$.

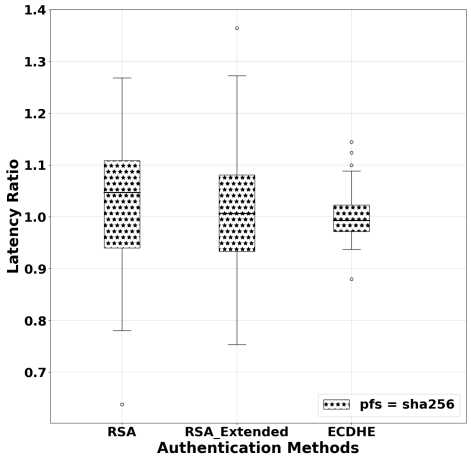


(a) *LURK* over secure transport protocols.

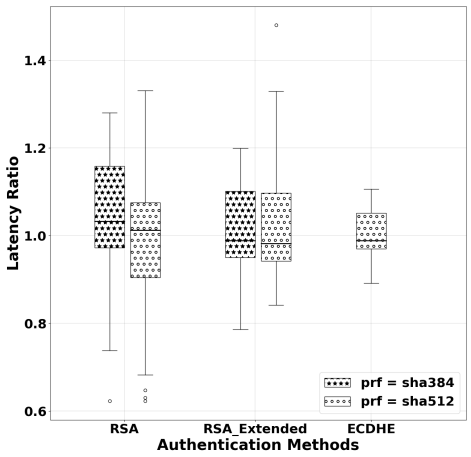


(b) *LURK* in RSA mode over secure transport.

Fig. 4: Latency Measurements



(a) Freshness function overhead.



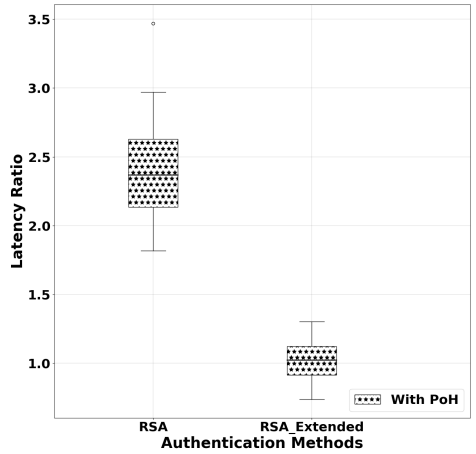
(b) TLS PRF overhead.

Fig. 5: Latency Measurements (cont'd)

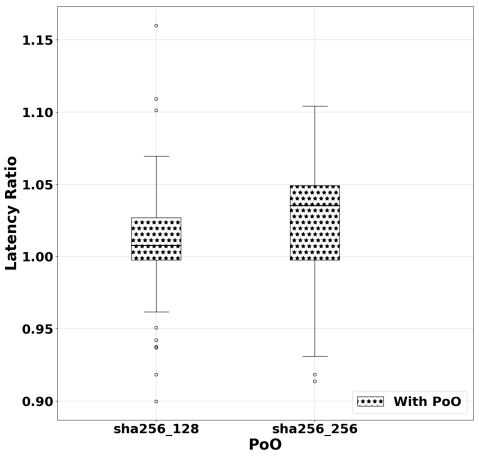
On TLS modes Figure 4a depicts that the latency of *LURK* varies with the underlying TLS mode. In fact, increased latency overhead is observed when using RSA extended and ECDHE modes in comparison to RSA mode. For example, *LURK* increases the latency by a factor of 2.2 and 3.73, in RSA Extended and 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 5a 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 5b depicts the RTT-degradation when TLS 1.2'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 6a 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 *LURK* in RSA mode with added PoH is comparable to that of TLS 1.2 in RSA-Extended mode. Figure 6b depicts the impact of our added PoO of the DHE exponent over the average ECDHE latency. The impact observed is relatively negligible.



(a) PoH overhead.



(b) PoO overhead.

Fig. 6: Latency Measurements (cont'd)

B. 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. 7, 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 *LURK* 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 *LURK* 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

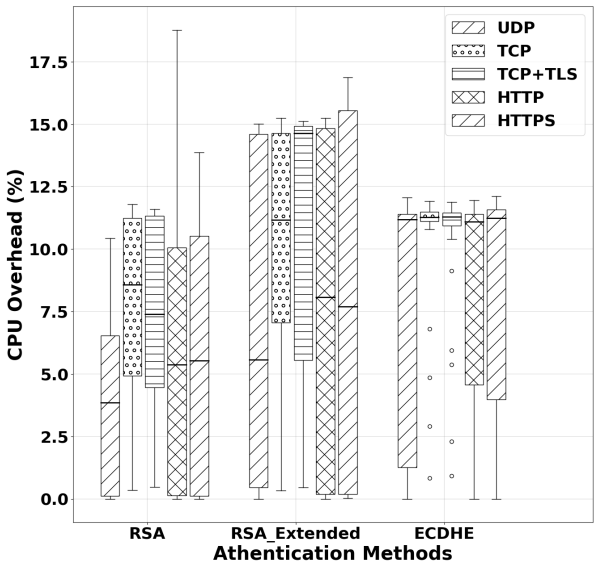


Fig. 7: CPU Overheads of the Crypto Service in different *LURK* modes.

CPU overhead in the case of using *LURK* in RSA and RSA-Extended modes. Concretely, the Crypto Service in RSA-Extended mode requires 1.39 times more resources than for *LURK* in RSA mode. In ECDHE mode, it requires 2.08 times more than for *LURK* in RSA mode.

VIII. Discussions, Future Work & Conclusions

Our suite of designs, called *LURK*, 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 improve the security of KeylessSSL [34], 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 [8], 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 some foreseeable future, 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. Overall, in this paper, our *LURK* constructs 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 [26]. In the long version of this paper [8], 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] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *POPL'01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2001, pp. 104–115.

[2] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla, "Delegated Credentials for TLS," Internet Engineering Task Force, Internet-Draft draft-draft-ietf-tls-subcerts, Jun. 2020, work in Progress.

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

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

[5] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *IEEE Computer Security Foundations Workshop*. Nova Scotia, Canada: IEEE Computer Society Press, 2001, pp. 82–96.

[6] B. Blanchet, *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*, 2016.

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

[8] I. Boureanu, D. Migault, S. Preda, H. A. Alamedine, S. Mishra, F. Fieau, and M. Mannan, "LURK: Server-Controlled TLS Delegation," Cryptology ePrint Archive, Report 2020/1366, 2020, https://eprint.iacr.org/2020/1366.

[9] I. Boureanu, A. Mitrokotsa, and S. Vaudenay, "On the pseudorandom function assumption in (secure) distance-bounding protocols," in *Progress in Cryptology – LATINCRYPT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 100–120.

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

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

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

[13] D. Dolev and A. Yao, "On the Security of Public-Key Protocols," *IEEE Transactionson Information Theory 29*, vol. 29, no. 2, 1983.

[14] C. Gero, J. Shapiro, and D. Burd, "Terminating ssl connections without locally-accessible private keys," Jun. 20 2013, wO Patent App. PCT/US2012/070075. [Online]. Available: http://www.google.co.uk/patents/WO2013090894A1?cl=en

[15] T. Hardie, "Transport Protocol Path Signals," RFC 8558, Apr. 2019.

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

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

[18] R. Housley, "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms," RFC 7696, Nov. 2015.

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

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

[21] Intel, "SGX: Software Guard Extensions," https://software.intel.com/en-us/sgx, 2019.

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

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

[24] A. Levy, H. Corrigan-Gibbs, and D. Boneh, "Stickler: Defending against Malicious Content Distribution Networks in an Unmodified Browser," *IEEE Security Privacy*, vol. 14, no. 2, pp. 22–28, 2016.

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

[26] D. Migault, "LURK Extension version 1 for (D)TLS 1.3 Authentication," Internet Engineering Task Force, Internet-Draft draft-draft-mglt-lurk-tls13, Apr. 2020, work in Progress.

[27] D. Migault and I. Boureanu, "LURK Extension version 1 for (D)TLS 1.2 Authentication," Internet Engineering Task Force, Internet-Draft draft-draft-mglt-lurk-tls12, Jul. 2020, work in Progress.

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

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

[30] "pylurk – a Python implementation of LURK," https://github.com/mglt/pylurk, 2019.

[31] Y. Sheffer, D. Lopez, O. G. de Dios, A. Pastor, and T. Fossati, "Support for Short-Term, Automatically Renewed (STAR) Certificates in the Automated Certificate Management Environment (ACME)," RFC 8739, Mar. 2020.

[32] "socketserver: A framework for network servers," https://docs.python.org/3.4/library/socketserver.html, 2018.

[33] "SSL: TLS/SSL wrapper for socket objects," https://docs.python.org/3.4/library/ssl.html, 2018.

[34] D. Stebila and N. Sullivan, "An Analysis of TLS Handshake Proxying," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1, Aug 2015, pp. 279–286.

[35] "Streaming Video Alliance (SVA)," https://www.streamingvideoalliance.org/technical-work/working-groups/open-caching/, 2019.

[36] "IETF Trusted Execution Environment Provisioning Working Group (TEEP)," https://datatracker.ietf.org/wg/teep/about/, 2019.

[37] "tinyec," https://github.com/alexmgr/tinyec, 2018.

[38] "CYBER; Middlebox Security Protocol; Part 2: Transport layer MSP, profile for fine grained access control," in *DTS/CYBER-0027-2*, no. TS 103 523-2, October 2020. [Online]. Available: https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=52930

[39] R. van der Meulen, "What Edge Computing Means for Infrastructure and Operations Leaders," October 2018. [Online]. Available: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders