LURK-T: Limited Use of Remote Keys with Added Trust in TLS 1.3

, Behnam Shobiri[a], Sajjad Pourali[a], Daniel Migault[b], Ioana Boureanu[c], Stere Preda[1], Mohammad Mannan[a,*], Amr Youssef[a]

[a] *Concordia University, Montreal, Quebec, Canada*
[b] *Ericsson, Montreal, Quebec, Canada*
[c] *University of Surrey, Guildfordl, UK*

Abstract

We propose *Limited Use of Remote Keys with Added Trust (LURK-T)*, a provably secure and efficiently decoupling of the server side of TLS 1.3 into a *LURK-T Crypto Service (CS)* and a *LURK-T Engine (E)*. *CS* executes all cryptographic operations in a Trusted Execution Environment (TEE), upon *E*'s requests, and together provide the whole TLS-server functionality.

One major benefit of our construction that is that it is **application agnostic.** Indeed, in our construction, the LURK-T Crypto Service could be collocated with the LURK-T Engine , or they could run on different machines or environments, as long as the two parts are connected via a mutually authenticated and secure channel (which is standard). Thus, our design allows for in-situ attestation and protection of the cryptographic side of the TLS server, as well as for all setups of Content Delivery Networks over TLS, and more.

To support such a generic decoupling, we **provide a full API (Application Programming Interface) for LURK-T**. To this end, we **implement our LURK-T Crypto Service using Intel SGX and integrate it with OpenSSL**.

It is non-trivial to produce a meaningful, generic, *secure* and *efficient* separation of the kind of LURK-T Crypto Service and LURK-T Engine, in such a way as to apply the least privilege principle onto the Crypto Service inside a TEE whilst limiting the performance overhead associated to the TEE. Thus, we **test LURK-T's**

*Corresponding author

Email addresses: `behnam.shobiri@concordia.ca` (Behnam Shobiri), `sajjad.pourali@concordia.ca` (Sajjad Pourali), `daniel.migault@ericsson.com` (Daniel Migault), `i.boureanu@surrey.ac.uk` (Ioana Boureanu), `stere.preda@ericsson.com` (Stere Preda), `m.mannan@concordia.ca` (Mohammad Mannan), `youssef@ciise.concordia.ca` (Amr Youssef)

efficiency extensively and show that from a TLS-client's perspective, HTTPS servers using LURK-T instead a traditional TLS-server have no noticeable overhead compared to OpenSSL when serving files greater than 1Mb.

We provide manual **cryptographic proofs, as well as carry out formal security verification** in ProVerif, showing that LURK-T does not degrade security compared to TLS 1.3, while adding new multi-party security guarantees leveraging the TEE, thus conferring a trustworthy TLS 1.3.

Keywords: Internet security, Middleboxes, TLS, Trusted Execution Environment

1. Introduction

Transport Layer Security (TLS) is the de-facto protocol for securing communication over the Internet. It is an authenticated key-establishment (AKE) protocol, whereby TLS client C (e.g., browser) always authenticates a TLS server S, and they derive *channel keys* to communicate securely thereafter. In TLS, the server S is authenticated by proving the possession of its private key or a so-called "pre-shared key (PSK)". So, these authentication credentials should not be accessible by other parties. However, real-world attacks such as Heartbleed [41], or even regular operations [93], can expose or leak these credentials, sometimes at a large-scale. As such, the private credentials of a TLS server clearly need to be provided with the highest and most reliable possible protection, even "in situ", that is if all server operations are local to said server.

For websites with the content cached and served by Content Delivery Networks (CDNs), a more common scenario is that of sharing of the TLS credentials between the website's TLS server (i.e., the "origin") and the CDN's so called "edge servers" [22], which can be distributed around the globe. Such sharing of long-term TLS credentials poses a grave risk, as the origin actually loses full ownership and control of their long-term private key [60]. Other CDN setups, in fact the original CDN architectures, where the private credentials of the TLS origin server are not shared with the CDN also exist (e.g., [36]); in this case, the CDN's edge servers query the origin servers on cryptographic material, and making only the legitimate queries as well as making sure they leak no private information. As 73% of the Internet traffic today is served by CDNs [25, 4], it is clear that protecting the cryptographic material "in situ" on the TLS-server, needs to be extended to the distributed-networks' interaction with mechanisms such as CDNs is vital and requires a careful treatment

Calling for a Generic Design of Isolation & Protection of Long-term Credentials in TLS

Above, possible "in-situ" leaks or breaches via querying by third-parties such as CDNs are just two scenarios of threats to long-term credentials in TLS, yet each with essentially different operational settings. We cannot exhaust all such cases, especially since – as we alluded to above – the setups for CDN-ing over TLS alone vary vastly [92, 42, 82, 91, 6, 83]. Thus, there is a call for a unitary and generic treatment of securing and protecting of long-term credentials of the TLS server, catering for as many distinct types of interactions with it as possible. To this end, we propose **LURK-T: a generic, provably secure and efficient decoupling of the TLS1.3 server**, into a cryptographic core called *LURK-T Crypto Service CS)*, and the component called *LURK-T Engine (E)* which securely queries this core from anywhere it may reside, and communicates further with classical TLS Client (C).

Current Solutions' Gaps in Generality, Security and/or Efficiency

We are not, by any means, the first to consider the decoupling of a TLS server and/or securing a modified version thereof. There are two lines of endeavour that primary pursued this: (a) the TEE-driven research focusing on isolating and securing the server; (b) the CDN-driven research focusing of modifying the TLS server to fit different CDN setups. Each line has its merits and its shortcomings, and with LURK-T we aim to produce a new solution, inspired by both, improving on both and augmenting its generic nature and its security, whilst carefully designing the TLS-server decoupling to obtain as good an efficiency as possible. Now, we discuss the main aspects of these lines, with further details given in Section 2.

(a) TLS Servers and CDNs. CDNs operate over TLS in a mechanism often broadly refereed to as "TLS delegation". To this, in a provably secure way, e.g., [15] or to cater for setup-specific scenarios [7, 81], major operational changes in TLS are required, which either break security, e.g.,[83] (as [11] showed) or render them completely incompatible with legacy clients , e.g.,[69]. Aside, neither of these solutions is generic enough to suit all or a subset of them, and the efficiency of delegation is not discussed at length or is foregone all together for added security, e.g., [10].

(b) TLS Servers with/in TEEs. For applications like TLS, NIST [8] recommends hardware-based Trusted Executing Environments (TEEs) such as Trusted Platform Modules (TPMs) or Hardware Security Modules (HSMs), for storing and using private keys. Yet, due to significant cost and performance issues of large-scale HSM deployment [63, 71, 28], many turned to using software-based TEEs. Still, not fully considering efficiency or deployability, several somewhat drastic proposals arose: the full application being placed in a TEE [92, 42, 82, 91], or the full TLS placed in a TEE [6] – both of which are explicitly mentioned as impractical by many standard bodies such as ETSI [35], 3GPP-SA3 [1] or ENISA [34] – or only the key [42, 18, 90]. Indeed, deciding what part of the (cryptographic side of) TLS-server to include in a TEE, such as to yield added security and keep efficiency, does appears non-trivial.

Our Contributions

1. To enhance the security and trust for TLS 1.3, we propose *Limited Use of Remote Keys with Added Trust (LURK-T)*. LURK-T splits a TLS 1.3 server into two parts: a *LURK-T Engine* (E) and *LURK-T Crypto Service* (CS). CS resides inside a TEE, and is only involved during the TLS handshake. CS handles and ensures the confidentiality of TLS-server credentials intrinsically needed for TLS key-security: private keys, PSK for so-called "session resumption", Elliptic Curve Ephemeral Diffie Hellman (ECDHE) keys to ensure so-called "Perfect Forward Secrecy (PFS)", whilst E handles the rest of server-side TLS. Moreover, our design is such that E's queries to CS cannot be made outside the scope of a fresh TLS 1.3 Key EXchange (KEX).

2. We give cryptographic proofs for LURK-T, in a cryptographic model for multi-party TLS [11], showing that LURK-T provides three-party TLS security (E, CS, and C). We also formally verify LURK-T's security using ProVerif, by first lifting the existing ProVerif specifications [75, 9] of a pre-standard TLS 1.3 to a ProVerif model for the standard TLS 1.3 [77], and then proving TLS 1.3 security for LURK-T; thus, we show LURK-T provably provides no degradation in security compared to TLS 1.3, including attaining perfect forward secrecy. We attain strong security guarantees (e.g., the accountability of [10]), as well as add a new strong propriety of trust which we call "trusted key-binding"; the latter hinges on the attestation of our TEE-based CS.

3. We implement CS using Intel SGX and integrate it with OpenSSL, both for Ubuntu and Windows. Our design's modularity entails only localized changes to OpenSSL and we carry out a seamless integration with it, using just well-known, widely-reviewed cryptographic libraries. To show the compatibility and portability of our implementation, we also develop a Rust HTTPS server and link it to our modified OpenSSL.

4. We test LURK-T's efficiency extensively, measuring different overheads compared to a standard TLS 1.3 handshake—for all the TLS 1.3 cipher suites and various CS configurations. We measure the maximum number of files served per second with HTTPS and show that in a worst case configuration, from the client's perspective, the overhead associated to LURK-T is negligible for files equal or greater than 1Mb.

2. Related Work

LURK-T partitions TLS 1.3 into two independent micro services (E and CS) with CS hosted by a TEE. This section positions LURK-T toward the work on partitioning application as well as as protocol extensions for TLS delegation and TLS multi parties.

2.1. TLS and TEE

Multiple frameworks [62] are able to host unmodified binary code into a TEE enclave. These rely on libOS (e.g. Graphene [88] – now Gramine [37] – SGX-LKL [74]) or musl-libc (e.g., SCONE [5]). However, this results in large TCB [87] with a huge amount of Line Of Code (LoC) prone to bugs [27] (and Iago attacks [23]), and with large overhead due to multiple ECALL/OCALLs [84].

Partitioning applications is expected to address these drawbacks. Specific manual approaches have been proposed for TLS 1.2 [6], [19], VC3 [79], Plinius [97] or Opaque [99]. A more generic approach, based on marking sensitive data in the source code for C/C++ applications has been proposed in *Glamdring* [61] and the execution of the resulting trusted part can be instrumented by sgx-perf [91] and analyzed. Other proposals such as Civet [89], Uranus [55], Montsalvat [98] partitioned Java code based on its byte-code. However, the coexistence of the trusted and untrusted part is handled via RPC-like mechanisms, exposing the interface to Iago-like attacks, while providing little assurance that states or data is not leaked. LURK-T defines standard interfaces, thus protecting CS against Iago-style attacks while enabling remote execution of the CS. The combination of CS and E is also formally proven to not alter TLS 1.3 security which has not been considered before and follows [11] which showed that lack of formal verification in new proposals can create vulnerability (e.g., in Keyless SSL [83] and mcTLS [70]).

Various efforts [6, 90, 91, 42, 18, 82, 87], were made to leverage TEE and port TLS applications into SGX enclaves. All these proposals were focused on TLS 1.2, and generally (e.g., TaLoS [6] and sgx-perf [91]) they place the full TLS stack into the TEE. STYX [90] provides a trusted way for the content owner to provision the hardware cryptographic accelerator provided by the CPU of an untrusted cloud provider and thus benefit from Quick Assist Technology (QAT [85]). As detailed in [11] w.r.t. Keyless [83], the use of generic cryptographic operations may be exploited.

Conclave [42] takes a higher level approach, by defining an architecture for securing a full service, an NGINX server in their case, which runs on an untrusted infrastructure. Conclave presents two configurations for TLS 1.2 alone: 1) only the private key is protected by the TEE, or 2) the entire TLS (including the session keys) is protected by the TEE. In addition, just executing the TLS in a TEE as per Conclave is not viable both on performance and operational perspectives. Security-wise, first, Conclave uses Graphene [38] which is a large library (more than 77000 LOC) and has a high probability for vulnerabilities as shown in [27]). In contrast, LURK-T has 3800 LOC and extends the private key protection to any authentication credentials used by TLS (including session resumption) without the need to deploy Graphene. Also, unlike LURK-T, Conclave does not provide anti-replay protection.

2.2. TLS protocol extensions

Similar to KeylessSSL, most previous works on TLS delegation [60, 14, 80, 44, 59, 83, 11, 90] are not designed for TLS 1.3 and suffer from the TLS 1.2 limitations [60, 15]. Bhargavan et al. [11] provide delegation for Authenticated and Confidential Channel Establishment (ACCE) with TLS 1.3, but two aspects are essentially different from us: ACCE is controlled by both ends (i.e., the client and the server), and it modified the TLS-record layer to achieve fined-grained access-rights for CDNs. To the best of our knowledge, LURK-T is the first design that provides a *server-controlled* delegation *specific to TLS 1.3*, without any modification to TLS 1.3, as well as leveraging TEEs for added trust.

Delegated credentials (DCs) [7] is a TLS 1.3 extension which eases the issuance of the authentication credential by a CDN provider. However, the content owner delegates the authentication to the CDN, and the deployed credentials by the CDN remain exposed. LURK-T can be used to avoid sharing the long-term key with the CDN. In addition, DCs also require support/control from both the client and server sides.

Boureanu et al. [15] used a similar design to LURK-T but for TLS 1.2 [67, 65]. Most differences between [15] and us stem from TLS 1.3 being different from TLS 1.2, but there are others: In contrast, LURK-T offers several variants for different TLS 1.3, in order to balance reasonable security vs. efficiency, which also address Boureanu et al.'s latency issues. In addition, LURK-T leverages TEEs and provable security for further trust.

Various services provided by CDNs can only function when they have access to plaintext data (referred to as middlebox), such as IDS, IPS, WAF, and L7 load balancing [31]. In some previous proposals [42, 6, 18], these services cannot operate well (within the CDN) since they do not have access to plaintext data. This problem was solved in TLS 1.2 by mcTLS [70], but with significant overhead and with heavy modifications to TLS 1.2 handshake and record-layer. It was solved generically for any ACCE protocol but even more costly [11]. However, LURK-T solves this for TLS 1.3 without any modification to TLS 1.3 and much more efficiently than in the aforementioned cases. More specifically, with LURK-T the IDS, WAF, etc. can each be engines accesses the plaintext data.

3. Background

In this section, we provide a brief background on TEE attestation, and formal analysis models/tools used in our security analysis. We assume the reader is familiar with TLS 1.3 and provide an overview of TLS 1.3 in Section 10.

3.1. TEE Attestation

TEEs can transfer the trust level to the hardware, i.e., the software loaded on a device can be trusted, even if an attacker were to control all the machine apart from its CPU. To achieve this, the TEE model, attested software uses "private" region of the memory that executes in isolation from the rest of the software on the machine/*host*. There are a variety of TEE designs (for mobile devices, desktops, and servers) that accomplish TEE-model security goals [30]. We focus on Intel SGX (Software Guard Extensions) [48].

SGX protects the confidentiality and integrity of the code and data loaded/running inside an SGX enclave. More details on porting application as well as on runtime attestation is provided in Section 11.

3.2. Our Security Analyses

There are two schools of thought w.r.t. provable security: *computational analysis* (a.k.a. *cryptographic proofs*) and *symbolic analysis*. Computational or provable-security formalisms for security analysis consider messages as bitstrings, attackers to be probabilistic polynomial-time algorithms who attempt to subvert cryptographic primitives, and attacks to have a probabilistic dimension of the security parameters. On the other hand, symbolic models abstract messages to algebraic terms, assume cryptographic primitives to be ideal and not subject to subversion by the adversary, and the attacks be possibilistic (i.e., not probabilistic) flaws mounted via a set of Dolev-Yao rules [33] applied over interleaved protocol executions. A more detail description and comparison can be found in Appendix 13.

We use both the computational analysis: namely, we extend the (S)ACCE model [11] for multi-party AKEs, and we extend the symbolic analysis of a TLS 1.3 draft ProVerif [9]. We use both because they provide different guarantees. Firstly, both analysis use different threat models. For example, in the computational analysis, the adversary interacts with parties by calling oracles, and can in this way even "break" cryptographic primitives. Instead, in the symbolic analysis, the adversary is a rule-based Dolev-Yao attacker [33] which cannot "break" cryptography. In addition, the arbitrary session-interleaving modeled in the symbolic analysis is generally not checked/modeled in the computational analysis. Secondly, the properties verified are different: e.g., PFS can be easily checked in the symbolic analysis, but all the arbitrary corruption and cryptographic AKE (authentication key exchange) can only be verified via computational analysis.

We extended method the computational analysis to work for the actual current TLS 1.3 and TEEs, as well as applied it to LURK-T. We also extended the symbolic verification to work for TLS 1.3 draft 18 in [9] (which e.g., did not consider some AEAD encrypted payloads during the handshake); then, we applied it to LURK-T.

Importantly, we could forego the computational analysis if there was a computational-soundness result [29], but this does not exist for TLS, let alone for multi-party TLS.

4. Threat Model

Our protocol, like TLS, is an authenticated key-exchange protocol. Unlike the standard TLS, it operates over 3 parties: C, E and CS, with E and CS

implementing S. We undertake two security analyses: one computational and one symbolic, and as such, we have two respective attacker models.

Our Cryptographic Attacker. This is the threat model for the analysis in Section 8.2. Therein, our cryptographic attacker is a 3(S)ACCE adversary, with the following capabilities. (1) The attacker can compromise E, as well as the different end-points, i.e., C and CS. Clearly, as per the usual, not all 3 parties can be compromised in the same LURK-T execution, as in that case no security can be attained. Further, the TEE works as per its threat model [48, 30], onboard the CS and over its interfaces; this means that the TEE and its interfaces cannot be corrupted. (2) The attacker can control 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 13), these adversarial actions are formalised via oracle calls to a challenger simulating the protocol execution.

So, since we work in the 3(S)ACCE formalism, the LURK-T designs are expected to achieve –in the above threat model– the requirements of channel security, entity authentication and accountability. Our position is that the first two requirements are essential (and should be demanded of all LURK-T designs); meanwhile, we view accountability as an optional security requirement, which one can consider trading off for the sake of efficiency. On top, the TEE usage, allows us to prove a new security property which we call *trusted key-binding* – which we detail in Section 8.2.

Our Symbolic Attacker. This threat model is relevant for the analysis in Section 8.3. Therein, as always in symbolic verification, there is an underlying Dolev-Yao attacker. Also, along with other properties, we specifically verify perfect forward secrecy, which entails that we encode key-leakage in our models and check for new keys' secrecy after this old keys' leakage.

5. LURK-T Description

In this section, we describe our LURK-T design, instantiated with TLS 1.3, including the protocol and example use cases for E and CS based on their deployment.

5.1. LURK-T – Design

LURK-T involves the following parties: a TLS client C (e.g., a user's browser), a LURK-T TLS Engine E (e.g., a content-proxying server), and a LURK-T TLS Crypto Service CS (e.g., a content-owning server). The last two are either collocated, or there is a pre-established, mutually authenticated and encrypted channel between them. The TLS execution between C and CS, is actively proxied by E; i.e., E acts as the TLS server (S) to C, but E does not have (direct access to) the private key of the origin (i.e., of the CS) in order to generate the PSign TLS message.

We have two modular variants of LURK-T: *"LURK-T with DHE-passive CS"* and *"LURK-T with DHE-active CS"* providing higher assurance on PFS and resumed TLS sessions. The CS generates the (EC)DHE keyshare, PSign, and all secrets – including handshake/application/resumption h, a and r (see Figure 1a). This ensures the freshness policy of the (EC)DHE keyshare is enforced by CS, and the PSK used during the session resumption remains protected by the CS. In "LURK-T with DHE-passive CS", CS only generates the either PSign, h, a, or only PSign(see Figure 1b), designated as "normal" or "keyless", respectively.

With LURK-T being based on TLS, each such modular LURK-T variant can be instantiated in various TLS modes. We mainly consider the ECDHE mode and its associated PSK-ECDHE mode for resumed session. TLS has a third PSK mode which does not provide PFS is hardly used in a web context but works similarly to PSK-ECDHE and has been omitted. Both Figures 1a and 1b depict LURK-T instantiated with TLS 1.3 in ECDHE mode (for PSK-ECDHE, the only difference stem from the key-derivation in TLS). We now describe in more detail the two main variants of LURK-T each in two TLS modes: ECDHE and PSK-ECDHE.

"LURK-T with DHE-active CS" ECDHE Mode. Following Figure 1a, first, as per the TLS 1.3 $\mathsf{ClientHello}$ and KeyShare extension steps, C produces a nonce N_C, and (EC)DHE secret-exponent u and its g^u key-share KE_C; C then sends N_C and KE_C to E.

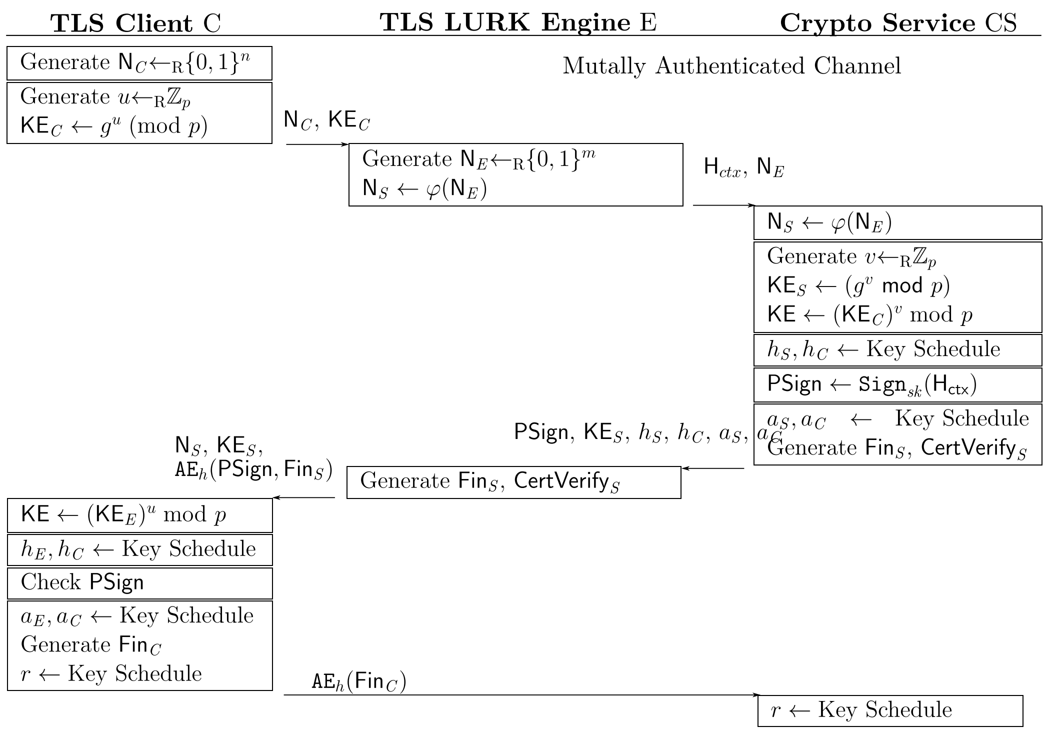
Second, E generates a nonce N_E and applies a pseudorandom function ϕ to produce a bitstring denoted N_S. In all variants and modes of LURK-T, N_E is deleted from memory at the end of the handshake. The function φ is added in LURK-T to improve post-compromise security.

Third, E sends to CS the whole of its view of the transcript H_{ctx} (incl. therefore N_C and KE_C), the bitstring N_S. The key-schedule algorithm on the server side takes H_{ctx} and KE as inputs, and produces the encryption secrets (e.g., h_S, h_C, a_S, a_C) used to complete the handshake. CS uses its private key sk to sign the transcript into the signature PSign, and then sends all the server-side ingredients back to E: $\mathsf{KE}, \mathsf{PSign}, h_S, h_C, a_S, a_C$.

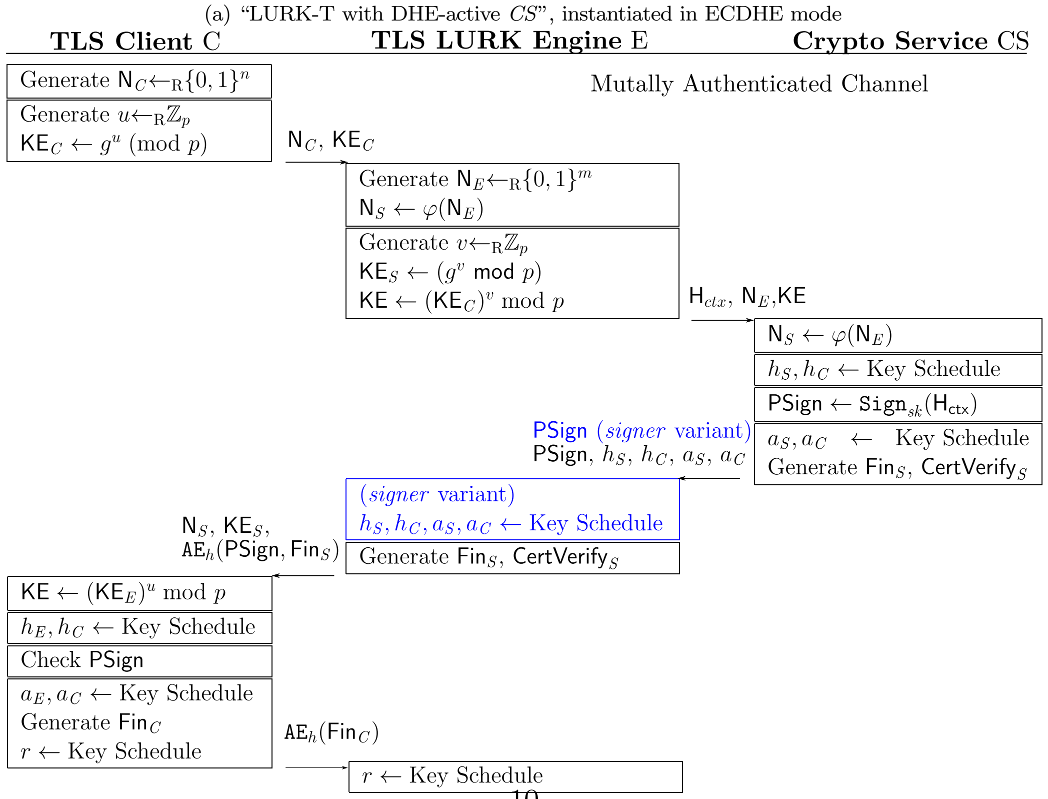
Finally, E receives all these, and, as per TLS, produces the encryption of PSign and Fin_E under h, adds the nonce N_S used by CS to produce PSign and Fin_E, and sends it all to C. Then C performs the normal TLS handshake execution, oblivious of S being split into E and CS.

Note that CS, generates "CertVerif" and Fin_E similarly to E to have the necessary handshake context to generate the application secrets a as well as when needed the resumption secret r from which is derived the PSK used in the future resumed session.

"LURK-T with DHE-passive CS" ECDHE Mode. In this variant, CS does not generate the (EC)DHE key-share, which is instead generated by E (see Fig-


TLS Client C
TLS LURK Engine E
Crypto Service CS
Generate $\mathsf{N}_C \leftarrow_{\mathrm{R}} \{0,1\}^n$
Mutally Authenticated Channel
Generate $u \leftarrow_{\mathrm{R}} \mathbb{Z}_p$
$\mathsf{KE}_C \leftarrow g^u \pmod p$
N_C, KE_C
Generate $\mathsf{N}_E \leftarrow_{\mathrm{R}} \{0,1\}^m$
$\mathsf{N}_S \leftarrow \varphi(\mathsf{N}_E)$
H_{ctx}, N_E
$\mathsf{N}_S \leftarrow \varphi(\mathsf{N}_E)$
Generate $v \leftarrow_{\mathrm{R}} \mathbb{Z}_p$
$\mathsf{KE}_S \leftarrow (g^v \bmod p)$
$\mathsf{KE} \leftarrow (\mathsf{KE}_C)^v \bmod p$
$h_S, h_C \leftarrow$ Key Schedule
$\mathsf{PSign} \leftarrow \mathtt{Sign}_{sk}(\mathsf{H}_{\mathsf{ctx}})$
PSign, KE_S, h_S, h_C, a_S, a_C
$a_S, a_C \leftarrow$ Key Schedule
Generate Fin_S, $\mathsf{CertVerify}_S$
N_S, KE_S,
$\mathtt{AE}_h(\mathsf{PSign}, \mathsf{Fin}_S)$
Generate Fin_S, $\mathsf{CertVerify}_S$
$\mathsf{KE} \leftarrow (\mathsf{KE}_E)^u \bmod p$
$h_E, h_C \leftarrow$ Key Schedule
Check PSign
$a_E, a_C \leftarrow$ Key Schedule
Generate Fin_C
$r \leftarrow$ Key Schedule
$\mathtt{AE}_h(\mathsf{Fin}_C)$
$r \leftarrow$ Key Schedule

(a) "LURK-T with DHE-active *CS*", instantiated in ECDHE mode


TLS Client C
TLS LURK Engine E
Crypto Service CS
Generate $\mathsf{N}_C \leftarrow_{\mathrm{R}} \{0,1\}^n$
Mutally Authenticated Channel
Generate $u \leftarrow_{\mathrm{R}} \mathbb{Z}_p$
$\mathsf{KE}_C \leftarrow g^u \pmod p$
N_C, KE_C
Generate $\mathsf{N}_E \leftarrow_{\mathrm{R}} \{0,1\}^m$
$\mathsf{N}_S \leftarrow \varphi(\mathsf{N}_E)$
Generate $v \leftarrow_{\mathrm{R}} \mathbb{Z}_p$
$\mathsf{KE}_S \leftarrow (g^v \bmod p)$
$\mathsf{KE} \leftarrow (\mathsf{KE}_C)^v \bmod p$
H_{ctx}, N_E,KE
$\mathsf{N}_S \leftarrow \varphi(\mathsf{N}_E)$
$h_S, h_C \leftarrow$ Key Schedule
$\mathsf{PSign} \leftarrow \mathtt{Sign}_{sk}(\mathsf{H}_{\mathsf{ctx}})$
PSign (*signer* variant)
PSign, h_S, h_C, a_S, a_C
$a_S, a_C \leftarrow$ Key Schedule
Generate Fin_S, $\mathsf{CertVerify}_S$
(*signer* variant)
$h_S, h_C, a_S, a_C \leftarrow$ Key Schedule
N_S, KE_S,
$\mathtt{AE}_h(\mathsf{PSign}, \mathsf{Fin}_S)$
Generate Fin_S, $\mathsf{CertVerify}_S$
$\mathsf{KE} \leftarrow (\mathsf{KE}_E)^u \bmod p$
$h_E, h_C \leftarrow$ Key Schedule
Check PSign
$a_E, a_C \leftarrow$ Key Schedule
Generate Fin_C
$r \leftarrow$ Key Schedule
$\mathtt{AE}_h(\mathsf{Fin}_C)$
$r \leftarrow$ Key Schedule

(b) "LURK-T with DHE-passive *CS*", instantiated in ECDHE mode

Figure 1: LURK-T's Two Variants: Instantiation with TLS 1.3 in ECDHE Mode

ure 1b). E generates N_E exactly as in "LURK-T with DHE-active CS"variant. Then, E generates the private DHE exponent v and its key-share KE_S g^v and computes KE g^{uv} which is provided to CS, alongside the H_{ctx} and N_E. CS then computes N_S as in "LURK-T with DHE-active CS", initiates the key-schedule algorithm on the server side with H_{ctx} and KE as inputs, computes PSign and optionally h_S, h_C, a_S, a_C – as these later secrets may also be generated by E. The rest continues as in the "LURK-T with DHE-active CS" variant.

LURK-T PSK-ECDHE Mode. The main difference between the PSK-ECDHE mode and the ECDHE mode is that the former is used for session resumption. LURK-T in PSK-ECDHE vs. ECDHE mode varies in as much as TLS 1.3 varies across these two modes. LURK-T in PSK-ECDHE mode requires more exchanges between E and CS. Typically, upon the reception of the ClientHello, E needs to check the PSK proposed by C by performing a HMAC with a binder key derived from the PSK; this binder key can be generated only by CS, and E needs to request it from CS. Once the PSK binders have been checked, E interacts with CS to generate the various secrets as in ECDHE mode, but without PSign being generated.

LURK-T's Freshness Function φ. We derive the "server-nonce" N_S by applying a non invertible PRF φ instance to a nonce N_E generated by E to prevent replay attacks. If an adversary $\mathscr{A}$ collects plaintext information from a handshake, then $\mathscr{A}$ will gather N_C, KE_C and N_S (from the channel in between E and C). However, $\mathscr{A}$ will not be able to collect nor to derive N_E due to the non invertible property of φ. If later on, $\mathscr{A}$ corrupts E, she will not find the old N_E nonce in E's memory; we require that N_E be deleted from E's memory at the end of its use. Exhaustive search of the right N_E would also be exponential in the size of the domain of ϕ, so it will be impossible for our polynomial attackers, and thus preventing replay attacks – as N_E is necessary for the exchange.[1]

5.2. LURK-T - Use Cases and Deployments

We consider different deployment scenarios for LURK-T as discussed below. The management of TLS is impacted by the management of TEE (with attestation) as well as the management of the long term private key, other aspects of TLS are not impacted. Similarly as only the TLS library is impacted LURK-T enables the CDN to continue provide added services and as such keep TLS a multi-party TLS. A more complete description can be found in the appendix (Section 12).

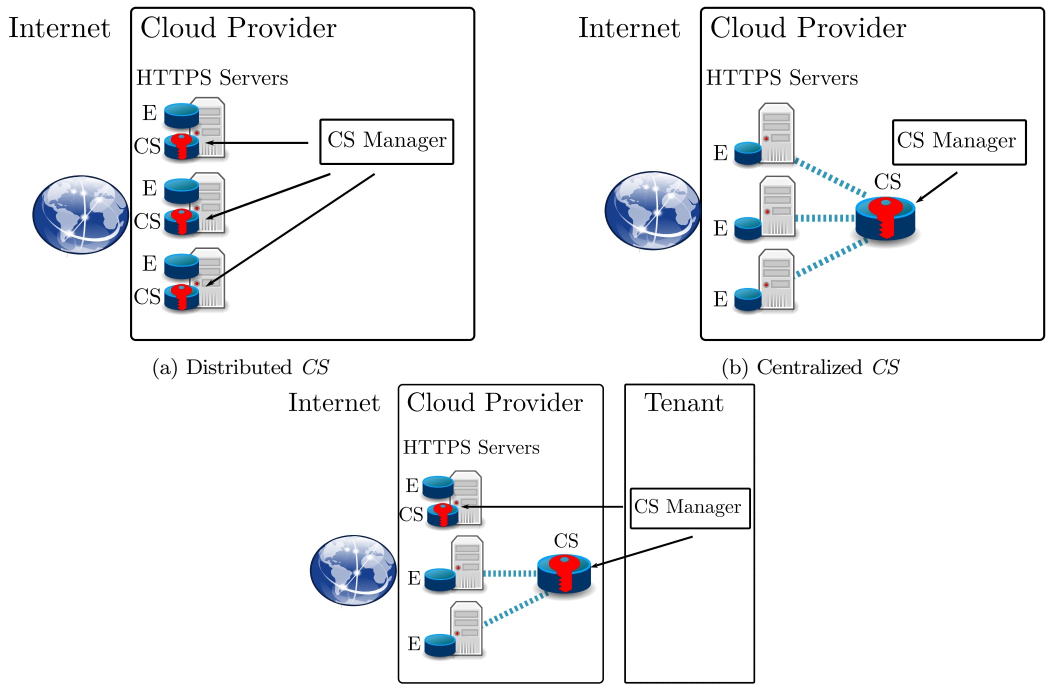
Deployments Driven by CDN providers. Figure 2a shows the case where LURK-T is deployed as a substitute of TLS 1.3 libraries. In this case, server-side

[1]We provide LURK-T design details in [66] as well as a python python implementation pylurk at `https://github.com/pylurk`

TLS libraries are replaced both by *E* and *CS*. The main challenge associated with this case is that CDN providers will need to manage (and provision) multiple instances of *CS*. Figure 2b shows the case of a more centralised infrastructure, with just one *CS* with an SGX enclave communicating securely with multiple *E*s. In both cases, it remains crucial to implement an attestation-ready provision of the *CS*. As the attestation is to be performed by the CDN provider within its own network, DCAP [50, 78] seems appropriate in combination with TLS-RA [58].

Deployments Driven by Content Owners. Figure 2c shows the case where *CS* is provisioned by a CDN tenant, such as a content owner. Therein, *CS* is likely to be implemented by a third party (*CS* developer), trusted by the content owner and the cloud provider – e.g., with open source code. The tenant will need to perform an attestation of the *CS*, e.g., using Intel IAS [56, 64]; this should use a group signature in order for the tenant not to find out the identifier of the exact CPU running the *CS*. This is also likely to be combined with TLS-RA [58].

Finally, note that from a tenant's perspective there is little difference between instantiating a centralised *CS*, or multiple *CS* instances; the difference is mostly in the way *CS* is implemented, which can also be checked by the tenant via attestation (detailed in Section 12).



(a) Distributed *CS* (b) Centralized *CS* (c) Tenant-controlled *CS*

Figure 2: CS deployment use cases

6. System Implementation

In this section, we describe our implementation of *CS* and *E* based on OpenSSL [72] published as open source.[2] CS centralizes the cryptographic operations; however, OpenSSL has not been developed with such a centralized cryptographic architecture and instead performs TLS operations sequentially. Thus, following the OpenSSL design would lead to numerous interactions between *E* and *CS*, and degrade performance, especially when interactions are between the Rich Execution Environment (REE) and TEE. In particular, for SGX enclaves, the interaction between TEE and REE results in 8,200 - 17,000 cycles overhead as opposed to 150 cycles for a standard system call [92]. We also balance the compliance to the LURK-T specification [66] and changes to OpenSSL to ease the maintainability of our code. As a result, we implement *E* by updating 184 lines of the OpenSSL code and introduce a maximum of 2 additional ECALLs compared to the LURK-T specification [66]. Based on [26] *CS* contains 33 files which represents 3867 LOC.

6.1. Crypto Service (CS)

We implement *CS* in an SGX enclave based on Intel SDK version 2.13. We had several options regarding the cryptographic library. While some cryptographic libraries ([6, 95]) support terminating the TLS connection inside SGX, we did not use them since they are either not maintained [6], or not fully compatible with OpenSSL [95]. We chose the actively maintained Intel SGX-SSL [52] that compiles OpenSSL source code as-is to create SGX compatible APIs (ensuring compatibility and easy upgrades with future versions of OpenSSL). However, SGX-SSL has limited functionalities. For example, it does not support terminating TLS inside SGX and lacks all the TLS and network related structures. Therefore, part of the *CS* implementation mimics the TLS specific functions implemented by OpenSSL using lower-level APIs and structures supported by SGX-SSL (we use OpenSSL 1.1.1g for SGX-SSL).

6.1.1. CS in TLS ECDHE mode

CS is responsible to genrate the different parts of the handshake such as the signature and optionally —depending if *CS* operates in "LURK-T with DHE-active *CS*" or "LURK-T with DHE-passive *CS*" variant —(EC)DHE keys and secrets as detailed Section 5.1. Our implementation supports all these variants as depicted by Figure 3 which details the exchanges between *E* and *CS*. While our design defines a single SInitCertificateVerify exchange [66], our implementation, when necessary (depending on the *CS* configuration), repeats up to 3 times that exchange in order to retrieve

[2] https://github.com/lurk-t/openssl

different pieces of information (depending on the CS configuration, see Figure 3). Table 1 depicts the supported CS configurations and for each one, which of the E or by CS generates the (EC)DHE or the secrets h, a and r. Binder keys and signature are always generated by CS in their respective ECDHE or PSK-ECDHE modes.

CS config (Cert)	$CS_{cert}^{dhe,r}$	CS_{cert}^{dhe}	CS_{cert}	$CS_{cert}^{keyless}$
(EC)DHE	CS	CS	E	E
handshake	CS	CS	CS	E
application	CS	CS	CS	E
resumption	CS	–	CS	E
#ECALLs	4	3	2	1
CS config (PSK)	$CS_{psk}^{dhe,r}$	CS_{psk}^{dhe}	CS_{psk}^{r}	CS_{psk}
(EC)DHE	CS	CS	E	E
handshake	CS	CS	CS	CS
application	CS	CS	CS	CS
resumption	CS	–	CS	–
#ECALLs	5	4	4	3

Table 1: CS configurations indicating where (EC)DHE or secrets are generated (when generated) and associated number of ECALLs by our implementation

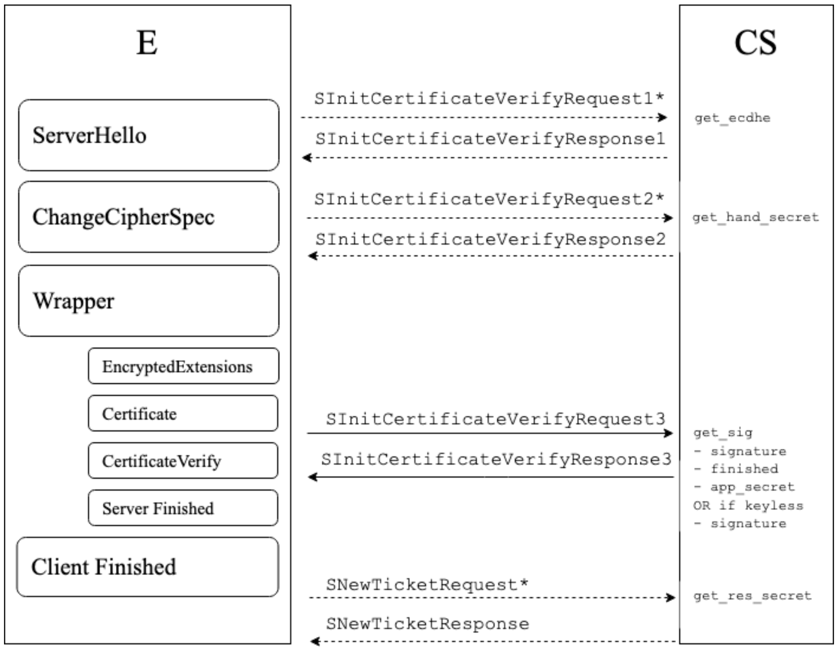


Figure 3: Messages between E and CS for the ECDHE mode. * designates optional exchange depending on the CS config.

Generating (EC)DHE. In the "LURK-T with DHE-active *CS*" variant, *CS* generates the (EC)DHE keys for *S*. *E* retrieves the *S*'s (EC)DHE public key with an additional SInitCertificateVerify1 exchange (see Figure 3). *CS* generates the (EC)DHE shared secret using *C*'s (EC)DHE public key and the *S*'s (EC)DHE private key – that is kept secret by *CS*.

Generating h and a secrets. When *CS* is configured to generate h, *E* performs an additional SInitCertificateVerify2 exchange to retrieve them (see Figure 3). `get_hand_secret` takes the ClientHello to ServerHello as inputs and returns h.

When *CS* is configured to generate a, these are generated together with the signature in the `get_sig` function (SInitCertificateVerify3). `get_sig` takes the ClientHello to EncryptedExtension messages, generates the signature, completes the TLS handshake by generating the CertificateVerify and the server Finished messages to compute a. In contrast, in the *keyless* configuration, `get_sig` only generates the signature; therefore, in this case, our implementation fully matches the LURK-T specification with a single ECALL.

Note that the SInitCertificateVerify3 is only necessary to provide EncryptedExtension message. Since this message is usually empty in the case of X25519 or X448, we could have easily reduced the number of ECALLs. However, we chose to provide a generic implementation of *CS* and instead have this extra exchange.

Session resumption. When session resumption is enabled, the ticket is retrieved via a SNewTicket exchange. This exchange provides the full TLS handshake (from ClientHello to client Finished) and a nonce to the *CS*. Our implementation generates a stateful ticket in which *CS* generates the resumption master secret r and, subsequently, uses it for generating the PSK. *CS* stores the PSK and the LURK-T session ID (that is used as a PSK ID) in the TEE. Therefore, *E* caches the LURK-T session ID as a PSK ID to further identify the PSK. OpenSSL handles the generation of the NewSessionTickets messages as well as the ability to bind a ticket in a resumed session to the PSK generated in a previous TLS session. To fully re-use OpenSSL ticket management functions the PSK ID is stored where OpenSSL used to store the clear text PSK.

6.1.2. CS in TLS PSK-ECDHE mode

During a session resumption, our implementation intercepts the access of OpenSSL to the PSK, and instead *E* sends a SInitEarlySecretRequest to *CS*. This exchange provides the PSK ID so *CS* can restore the PSK and initiate a Key Schedule and return the binder key.

Similar to Section 6.1.1, the specified SHanshakeAndApp is implemented in 3 ECALLs when *CS* generates the (EC)DHE (`get_ecdhe`, `get_hand_secret` and `get_app_secret`), or 2 ECALLs when *E* generates the (EC)DHE (`get_hand_-`

`secret` and `get_app_secret`). Generation of the resumption secrets r by *CS* requires an additional ECALL (`get_res_secret`).

6.2. TLS Engine (E)

E is based on OpenSSL 1.1.1g is implemented by updating 9 C files out of the 44 files in the SSL directory. Upon configuration, E executes the native OpenSSL function or initiates an exchange with *CS*. OpenSSL defines two core structures: `SSL` and `SSL_CTX`. `SSL` is created for each new TLS connection and contains all TLS sessions' context (e.g., cipher suite, session, secrets, etc) [43]. Communication between E and *CS*, are handled via the `LURKRequest` and the `LURKResponse` structures added to the `SSL`.

`SSL_CTX` contains the information common to all `SSL` structures (e.g., session resumption and the number of new TLS connections). Typically, C and S create one `SSL_CTX` structure and reuse it for all their TLS connections. Since *CS* is shared across all TLS connections, it is instantiated at the creation of `SSL_CTX`. Thus, initiating the enclave – which is a time-consuming – only happens once for S.

To apply the freshness function, we need both the full TLS messages as well as the ServerHello.random (before applying the freshness function – N_E see Section 5.1). However, by default, OpenSSL prevents the access to the TLS messages as it continuously hashes them TLS messages to avoid storing large handshake data. To overcome this, our implementation stores the value of ServerHello.random (N_E generated by OpenSSL) as well as handshake data. When OpenSSL generates N_E, it is intercepted by the freshness function, stored in the LURKRequest, and replaced by N_S so OpenSSL proceeds to the generation of the ServerHello using N_S. Later on, *CS* checks $\mathsf{N}_S = \varphi(\mathsf{N}_E)$, with N_S being the ServerHello.random (N_S) in the TLS message and N_E the stored value.

Finally, *CS* is integrated into E as an external library. We successfully linked (by updating OpenSSL Makefile) and tested our library for dynamic and static versions of OpenSSL.

7. Performance Evaluation

7.1. Methodology for Measuring LURK-T TLS Overhead over OpenSSL

In this section, we report the performance overhead of our TLS library. The performance is measured in terms of TLS Key EXchange per second (KEX/s), following the methodology used in RUST TLS performance evaluation [12]. Δ_{KEX} expresses the relative difference [94] in term of KEX/s between LURK-T TLS and the native OpenSSL TLS. Δ_{KEX} is expressed in percentage for a given configuration $conf$ which represents the TLS cipher suites (see Table 2) and the tasks performed by *CS* (see Table 1).

$$\Delta_{KEX} = \frac{|KEX_{LURK-T} - KEX_{OpenSSL}|_{conf}}{KEX_{OpenSSL}}$$

Our measurements are performed on an Intel i9-9900K CPU @3.60GHz over Ubuntu 18.04 LTS and we took the average time after performing 10,240 handshakes.[3]

Notation	Description	KEX/s
RSA-2048	(prime256v1, RSA-2048)	1715
RSA-3078	(prime256v1, RSA-3078)	316
RSA-4096	(secp384, RSA-4096)	243
P-256	(prime256v1, P-256)	5251
P-384	(secp384, P-384)	496
Ed25519	(X25519, Ed25519)	6113
Ed448	(X448, Ed448)	1251

Table 2: Native OpenSSL TLS key exchange (KEX) performance for different cipher suites

TLS cipher suites configuration. We base our selection of cipher suites in Table 2, on Mozilla's *modern compatibility* configuration which recommends ECDSA (P-256) or RSA-2048 combined with X25519, prime256v1, secp384r1 [68]. To which we added ECDSA (P-384) and Ed25519 projecting the measurement toward long-term deployments.

CS configurations. Besides TLS cipher suites, we measured various configurations for CS. The primary purpose of CS is to protect authentication credentials (private key *cert* or *psk*). In the ECDHE mode (expressed as *cert*), session resumption may be enabled (expressed as r), so future handshakes may use the PSK-ECDHE mode. Notably, as mentioned in Section 10, to remain coherent across sessions in term of PFS, we only considered the PSK-ECDHE mode (expressed as *psk*). As mentioned in Section 5.1 the PSK is derived from the generated (EC)DHE shared secret. Thus, the PSK used for the session resumption can only remain confidential in a "LURK-T with DHE-active CS" variant (e.g., generates the (EC)DHE private key). This is expressed with the following configuration $CS_{cert}^{dhe,r}$. Of course, without session resumption, "LURK-T with DHE-active CS" or "LURK-T with DHE-passive CS" variants are valid configuration (though with different level of PFS) expressed as CS_{cert}^{dhe}, CS_{cert} or $CS_{cert}^{keyless}$ (when only the signature PSign is generated, see Section 5.1). In the PSK-ECDHE mode, and unlike the ECDHE mode, session resumption may be enabled with both "LURK-T with DHE-active

[3]which corresponds to the smallest *multiplier* over 10,000 interactions of our performance evaluation tool.

CS" or "LURK-T with DHE-passive *CS*" variants and Table 1 summarizes the meaningful *CS* configurations with the associated number of ECALLs.

7.2. Experimental Measurements of LURK-T TLS Overhead over OpenSSL

ECDHE mode. Figure 4 depicts Δ_{KEX} as a function of the number of ECALLs which characterizes *CS* configuration (see Table 1). Note that the number of ECALLS and the linear regression used to link the measured performances are only used to better visualize the performances. As shown in Figure 4, ECALLs do not equally affect all cipher suites and Δ_{KEX} does not linearly increase with the number of ECALLs. However, as per Table 2, cipher suites that require more resources (RSA-3078, RSA-4096, P-384, Ed448) seem less impacted by LURK-T TLS and their overhead depends more linearly on the number of ECALLs. A possible explanation is a low ratio of allocated slots by the scheduler which results in either an interruption or an exitless process wasting the remaining allocated cycles. With our current configurations, the measured overhead for Ed448, RSA-3078, P-384 and RSA-4096 is low (between 1.2% and 10%) and the number of ECALLs have very little impact. Other cipher suites (including RSA-2048) are more impacted by the number of ECALLs. Nonetheless, our implementation presents a higher overhead for the P-256 and Ed25519 cipher suites. P-256 has up to 39.7% overhead when (EC)DHE is performed by *CS* and 14.7% in the *keyless* configuration. Ed25519 is less affected in the "LURK-T with DHE-passive *CS*" variant (less than 23%) compared to the "LURK-T with DHE-active *CS*" variant (up to 33%). Finally, the *keyless* configuration provides an acceptable overhead (17% for P-384, 7.6% for RSA-2048, less than 4.3% for the others).

PSK-ECDHE mode. Similarly to the ECDHE mode, Figure 5 shows the most efficient ciphers (P-256, X25519) are more impacted by the number of ECALLs than the others – such as P-384 and X448. Overall, the preliminary measurements of our implementation show encouraging results with a limited and acceptable overhead.

The observed overhead might be further improved (both for ECDHE and PSK-ECDHE modes) by reducing the number of ECALLs per handshake in two ways. Firstly, we can reduce the number of ECALLs. However, we do not apply them to avoid major modifications to the OpenSSL architecture or to keep the implementation generic in the case of Encrypted Extension (see Section 6). Secondly, we can aggregate multiple LURK-T requests in each ECALL. The optimal number of LURK-T requests that need to be aggregated is expected to depend on the CPU, the cipher suite, and the *CS* configuration. The optimum performance will be reached when the multiple operations can be completed within the allocated number of cycles, minimizing the number of unused cycles. This is likely to benefit the most to Ed25519 or P-256.

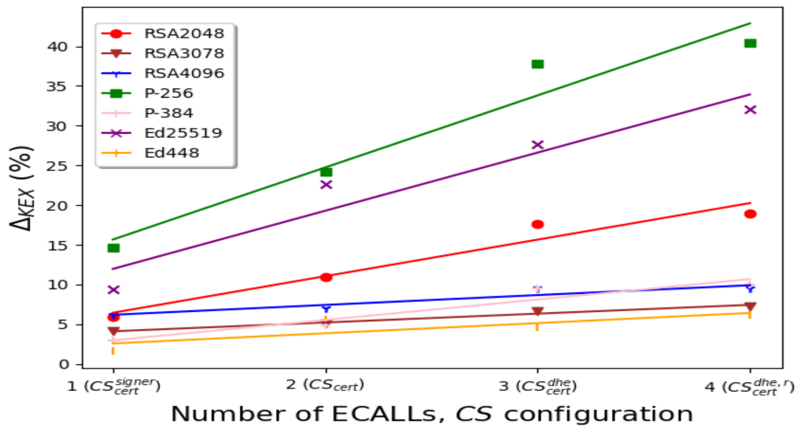


Figure 4: KEX LURK-T TLS relative overhead over OpenSSL (Δ_{KEX}) in ECDHE mode. Measured values are linked using a linear regression.

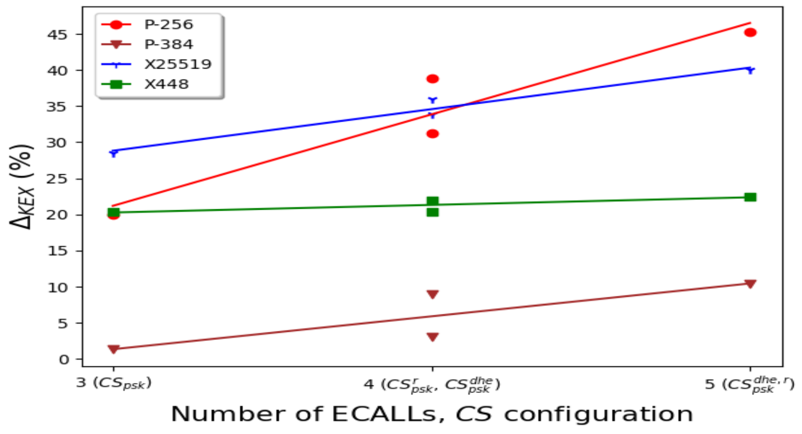


Figure 5: KEX LURK-T TLS relative overhead over OpenSSL (Δ_{KEX}) in PSK-ECDHE mode.

7.3. SGX Vulnerabilities Mitigation Overhead

This section measures the overhead associated to the mitigations – micro code or SDK [51]– of SGX vulnerabilities for our CPU. Section 7.2, considered the default SGX configuration; that is without vulnerabilities. While we expect future CPUs to address the currently known vulnerabilities –leading to the performances of Section 7.2– we also anticipate new vulnerabilities to be disclosed and their mitigation will come with an additional overhead for the cloud provider. Previous works usually omit such section as the penalty provided by the default is sufficiently prohibitive.

	$SGX^{SRDBS,cf}$	$SGX^{SRDBS,ld}$	$SGX^{default}$
RSA-2048	1162	132	1715
RSA-3078	282	35	316
RSA-4096	181	17	243
P-256	2353	971	5251
P-384	277	48	496
Ed25519	3312	909	6113
Ed448	1081	78	1251

Table 3: SGX performances (KEX/s) with SRDBS and LVI mitigation enabled versus default SGX for *CS* configured with $CS^{dhe,r}_{cert}$.

According to Intel [51], our CPU provides hardware mitigation against the following vulnerabilities [49, 46, 47]. However, it is vulnerable to Special Register Buffer Data Sampling (SRBDS) [57] and CrossTalk attack [76] for which Intel provides a microcode update. Similarly, our CPU is vulnerable to Load Value Injection (LVI) [21] which we respectively mitigate both via the SDK or via the SGX-SSL `cve_2020_0551_load` (ld) or `cve_2020_0551_cf` (cf) [52, 53].

Similar to Section 7.1, performance is measured in terms of the number of KEX/s. In our case, the overhead of the microcodes – SRBDS – for is negligible while the one of the SDK and SGX-SSL – for LVI – is not. While we measure the performance for all *CS* configurations, and all mitigations, all ciphers suites, Table 3 summarizes for each cipher suite, the measurements with SRDBS and LVI mitigation enabled for the most completed *CS* configuration ($CS^{dhe,r}_{cert}$). Mitigations have different overheads across cipher suites. For a given SGX configuration, the overhead increases with the number of operations performed by *CS*. However, for a given cipher suite, it remains hard to correlate the number of operations to the expected overhead.

7.4. LURK-T TLS overhead for HTTPS

We developed a multithreaded HTTPS server in RUST (using OpenSSL). Subsequently, we modified it to use LURK-T TLS to confirm that migration to LURK-T TLS is easy and can be used in other programming languages that support OpenSSL (in this case RUST). Similarly to Δ_{KEX} in Section 7.1, we measure in Table 4 Δ_{HTTPS}, the overhead of LURK-T TLS over HTTPS with OpenSSL by measuring the relative difference in requests by second of various file sizes being served. To do so, we developed and modified the benchmark tool wrk [96] to force TLS 1.3 as well as to be able to specify a specific cipher suites. The HTTPS server and benchmark tools are published as open source [4].

We measure the number of HTTPS requests per second performed by wrk with 10 parallel connections to introduce some concurrency similarly to [90] – though in the measurements, we did not observe a significant difference between 10 and a single connection. To prevent underestimating the impact of LURK-T TLS, we considered our LAN with a 10 ms latency with 100 Mb bandwidth that reflects the interactions with a NIC while lowering the impact of the latency. Similarly, the download file is always cached in the memory of the HTTPS server; thus, reducing S's latency (avoid reading from a hard drive). We limit CS's configuration to the most secure configuration which has the highest overhead ($CS_{cert}^{dhe,r}$). Moreover, to consider the SGX vulnerabilities, we performed the same measurements on the fully mitigation-enabled SGX (enabling ld and $SRDBS$) which has the most overhead.

The measurements of Table 4 show that even with $CS_{cert}^{dhe,r}$, the overhead is always negligible when downloading a 1Mb file (and more). In other words, for such a file, the transfers overtake the overhead introduced by the LURK-T TLS handshake. For file sizes lower than 100Kb, LURK-T TLS seems to introduce a slight overhead, but we are not able to find a clear relation with the file size, and the overhead seems primarily determined by the cipher suite. Similar to the measurements of CS in Section 7.2, P-384 and Ed448 seem to provide better performances compared to other cipher suites. It is important to note that in this section, we report the time for each connection. Our numbers are valid from both client and server perspectives. However, resource wise S's LURK-T overhead is the one measured in Section 7.2 pondered by the handling of application data.

When the mitigations (ld and $SRDBS$) are enabled, the measurements show that P-256, Ed25519, RSA-2048 provide a negligible overhead for 1Mb files. For other cipher suites, such a pivot seems to occur for file sizes over 1 Mb.

7.5. SGX Memory Usage

The memory that our design needs depends on the configuration. Without session resumption, our implementation uses about 25 KB stack and 8 KB heap memory at most. Moreover, the memory requirement does not change with the number of connections. Similarly, when session resumption is enabled in the stateless mode, we do not need to store anything in SGX protected memory. However, in the stateful mode, session information needs to be stored in the SGX memory. For each session, we approximately need 104 bytes of SGX memory to store the PSK and session ID. Note that we report memory usage in the debug version (since the Enclave Memory Measurement Tool provided by Intel only works in the debug mode). Therefore,

[4]`https://github.com/lurk-t/https`

Δ_{HTTPS}	0Kb	1Kb	10Kb	100Kb	1Mb
RSA-2048	16.0	17.7	8.9	7.9	0
RSA-4096	7.4	8.7	7.2	8.6	0
P-256	5.0	4.0	4.2	10.8	0
P-384	5.7	8.1	0.8	-3.0	0
Ed25519	10.9	13.9	3.4	3.4	0.1
Ed448	0	0	0	2.9	0

(a) Default SGX: LURK-T TLS overhead in term of download/s is negligible for files larger than 1Mb.

Δ_{HTTPS}	0Kb	1Kb	10Kb	100Kb	1Mb
RSA-2048	88.1	88.0	86.3	83.7	0.6
RSA-3078	86.6	86.7	86.5	86.5	69.1
RSA-4096	90.5	90.7	90.6	90.8	85.2
P-256	76.6	76.4	78.2	78.8	41.7
P-384	78.3	78.9	79.5	79.8	60.4
Ed25519	63.2	63.4	58.1	38.5	0
Ed448	78.1	77.9	78.5	82.7	38.2

(b) Mitigation-enbaled SGX (ld and S)

Table 4: HTTPS download/s LURK-T TLS relative overhead over OpenSSL (Δ_{HTTPS}) in ECDHE mode and "LURK-T with DHE-active CS" ($CS_{cert}^{dhe,r}$).

the memory usage will possibly reduce in practice (e.g., due to the optimization in the release mode).

7.6. LURK-T TLS overhead for HTTPS over other proposals

In this section we briefly discusses the LURK-T TLS overhead with the one of other solutions of Section 2, reported in Table 5. The comparison is only indicative as the overheads have been made in very different contexts involving different HTTPS servers (NGINX, APACHE and WRK) and different TLS libraries (different version of OpenSSL and libreSSL - though derived from OpenSSL). Such differences are expected to be lowered by the measurement of a relative difference Δ.

LURK-T is the only design that applies for TLS 1.3 with the lowest overhead in term of KEX compared to TLS 1.2 and fewest LOC in TEE. As mentioned in Section 2 specific approaches (LURK-T and Talos) seems to provide a lower overhead ove generic frameworks (Graphene – see Conclave keyless). On the other hand, the larger overhead measured for Panoply, Graphene, Talos and Conclave-crypt can largely be explained by the protection of TLS session.

In term of LOC LURK-T and Talos limits the potential vulnerability with fewer lines of code and make these solution more likely to be deployed by cloud providers

	LURK-T	Talos	C-k	C-c	Panoply	Graphene
LOC (K)	3.8	5.4	$>$ 770	1300	20	1300
Δ_{HTTPS}	0 - 17.7	22	57	81	49	40

Table 5: HTTPS measurement comparison between Panoply, Graphene, Conclave-keyless (C-k), Conclave-crypt (C-c), Talos. Δ_{HTTPS} represents the relative overhead (%) associated to the number of HTTPS download per second for a 1 Kb file.

as less TCB is required. This results both in limiting the necessary (limited to 90Mb) memory resource as well as increases the ability to share the other part of the untrusted library between containers and other applications.

8. Formal Security Proofs and Analyses

As per Section 5, one can have several modes and several variants of our LURK-T protocol. In what follows, we will show security-analyses for all these variants. We start by stating LURK-T's requirements semi-formally, in 8.1. On top of the existing 3(S)ACCE [11] properties, we add a requirement and a proof for a new property stemming from our use of TEEs; we call this property *trusted key-binding*. All requirements and their proofs are given formally in Appendix 13.

In Section 8.2, we provide the computational-security results for both "LURK-T with DHE-active *CS*" and "LURK-T with DHE-passive *CS*" in EC-DHE mode, and discuss in this framework why "LURK-T with DHE-active *CS*" offers more provable-security guarantees. For "LURK-T with DHE-active *CS*" in EC-DHE mode, if executed in what we call the runtime-attested handshake-context mode, the property we call *trusted key-binding* holds (see Section 8.1); this is a stronger form of accountability, hinging on TEEs.

In Section 8.3, we use symbolic verification to show that "LURK-T with DHE-active *CS*" in EC-DHE mode attains all the same requirements that TLS1.3 does, and a new, 3-party security property that shows that C, E, CS have matching views of the handshake even in the presence of a Dolev-Yao attacker.

8.1. Cryptographic Requirements

In order to give our cryptographic proofs that LURK-T achieves its security goals, we use the recent 3(S)ACCE formal security model for proxied AKE [11], presented in Section 3 and in Appendix 13. In essence, we will use this 3(S)ACCE model, extended with an additional 4th party, namely the *attester* AS, who interacts with the CS and (the AS may be called upon via E), but this interaction AS-CS is outside of the ACCE computation. Because of this, we continue to call the model

3(S)ACCE (as in, with 3 parties); we just make a note that a 4th party – the attester– is present, "out of band".

Security Requirements for LURK-T. For LURK-T, we prove the 3(S)ACCE requirements: i.e., entity authentication, channel security and accountability. Below, we add a new one, linked to the fact that the attester party. We call this requirement *trusted key-binding*, linked to the aforementioned attester.

LURK-T with Runtime-attested Handshake-context. This paragraph describes a sub-variant of LURK-T. In this sub-variant, a *runtime* TEE system is called to yield a separate "quote" on/over the whole handshake done inside the CS during a TLS session. So, we request the quote from the remote enclave (found on the CS) and verify this using the Attestation Service. Namely, we request the quote as soon as the CS prepares the PSign and before it does so[5]. Then, we encrypt the buffer containing the operations on the CS and its arguments (i.e., it will just contain H_{ctx}), encrypt it with the shared key established via remote attestation (e.g., *seal_key*). In this optional sub-variant, this step is done and the Attestation Service therefore will receive a "binding"/"context" to the channel-key calculated during the handshake. We call this type of LURK-T – *LURK-T with runtime-attested handshake-context.*

Trusted Key-binding. We now state our new, attestation-relevant property more widely than for LURK-T, for a server-controlled delegated TLS achieves trusted key-binding with uses runtime attestation on the CS. We say a *server-controlled delegated TLS achieves trusted key-binding* if CS is able to compute the channel keys ck used by C and E and the handshake context/transcript corresponding to these keys ck is asynchronously attested. I.e., if presented with this handshake context by the attester again, then CS can recompute these keys ck and produce the same $\mathsf{PSign}, h, a, etc$ sent to E in the handshake where ck was used.

Note that the attester only gets a minimal amount on the handshake; e.g., see the footnote 9. Notably, it does not get PSign, h, a, so it cannot impersonate the CS or resume a session as a CS. Further, in some TEE systems (e.g., if we use a TPM – trusted platform module), we could open a "TEE session" for the whole part of the handshake run on the CS and sign that as a proof of computation for the attester, yet we deliberately go against that. Such a design would give the attester all the information of the computation on the CS side which we believe will place too much trust on the attester, allowing it to see *long-term* secrets of the CS pertaining to another, specific protocol, i.e., TLS.

Finally, trusted key-binding is a type of enhanced 3(S)ACCE accountability which is based on the LURK-T CS executing its part of the TLS-server on an

[5]To get the timing right, this can be triggered by the E, soon after N_E was sent by the E to the CS. The report on the quote includes just H_{ctx}.

enclave. The trusted key-binding property is formalised in Section 14.4.

8.2. Cryptogtaphic Proofs

W.r.t. the properties recalled/given above, we now state our cryptographic guarantees semi-formally. The formalisation for each such statement/guarantee comes in Appendix 14.

Entity-Authentication Result. *If TLS 1.3 is secure w.r.t. unilateral entity authentication, if the protocol between E and CS is a secure ACCE protocol or they are collocated, if the two protocols ensure 3(S)ACCE mixed entity authentication [11] in the case where E and CS are not collocated, if the signature and hash in TLS 1.3 server-side are secure in their respective threat models, if the authentication encryption used in TLS 1.3 is secure in its model, then "LURK-T with DHE-active CS" and "LURK-T with DHE-passive CS" in EC-DHE mode are entity-authentication secure in the 3(S)ACCE model.* This is formalised and proven in Theorem 1 in Appendix 14.

Channel Security Result. *If TLS 1.3 is secure w.r.t. unilateral entity authentication, if the protocol between E and CS is a secure ACCE protocol or they are collocated, if the two protocols ensure 3(S)ACCE mixed entity authentication [11] in the case where E and CS are not collocated, if the signature in TLS 1.3 server-side is secure in its threat model, if the authentication encryption used in TLS 1.3 is secure in its model, and the freshness function is a non-programmable PRF [17], then "LURK-T with DHE-passive CS" in EC-DHE mode are entity-authentication secure in the 3(S)ACCE model attain channel security in the 3(S)ACCE model.* This is formalised and proven Theorem 2 in Appendix 14.

Note that the two security results above apply to all variants and sub-variants of LURK-T. These two requirements are the main requirements for any AKE protocol, now cast and proven here not over two but over three parties, in the 3(S)ACCE model. This alone makes LURK-T a secure TLS decoupling between the Crypto Service to the Engine. So, the next two statements can be viewed as "bonus" security, attained only by the variants of LURK-T which are computationally more expensive.

Accountability Result. *If TLS 1.3 is secure w.r.t. unilateral entity authentication, if the protocol between E and CS is a secure ACCE protocol or they are collocated, if the two protocols ensure 3(S)ACCE mixed entity authentication in the case where E and CS are not collocated, and the freshness function is a non-programmable PRF [17], then "LURK-T with DHE-active CS" attains accountability in the 3(S)ACCE model.* This is formalised and proven in Theorem 3 in Appendix 14.

Accountability requires that CS always be able to compute all keys and subkeys of the session established between the client and E. So, accountability is incompatible with session-resumption possible by the E alone (i.e., "LURK-T with DHE-passive CS"). That is the above security statement w.r.t. accountability only holds for "LURK-T with DHE-active *CS*". Note that this is not critical in practice. Also, it comes at a cost (i.e., "LURK-T with DHE-active *CS*" is more computationally expensive expensive than "LURK-T with DHE-passive *CS*"). So, with LURK-T, we provide a series of variants, allowing the deployment-stage to choose between security and efficiency.

Trusted Key-binding Result. *If TLS 1.3 is secure w.r.t. unilateral entity authentication, if the protocol between E and CS is a secure ACCE protocol or they are collocated, if the two protocols ensure 3(S)ACCE mixed entity authentication in the case where E and CS are not collocated, and the freshness function is a non-programmable PRF [17] and if the TEE allows for runtime remote attestation, then "LURK-T with DHE-active CS" attains trusted key-binding.* This is formalised and proven in Theorem 4 in Appendix 14.

Trusted key-binding can be seen as an attested form of accountability. So, like accountability, it will only hold for variants of LURK-T where there is no session resumption by the E alone. Again, this is not critical in practice – since trusted key-binding is an arguably very strong requirement of security and trust.

8.3. Symbolic Verification

We perform a symbolic verification using ProVerif [13] to show that the LURK-T protocol, from a symbolic verification perspective, attains the same security properties as TLS 1.3, and more. In this section, we focus on the verification of "LURK-T with DHE-active *CS*"[6] Concretely, we firstly show that LURK-T does not impact TLS security (from a symbolic-verification perspective). Then, we show that the addition of the 3rd party still attains security w.r.t. symbolic verification. All this is complementary to the results in Section 8.2).

This section is structured as follows. First, we report on a ProVerif-verification of TLS 1.3 which we lifted from TLS 1.3 pre-standardisation (i.e., draft 20) to the current standard. Then, we show that all the 2-party, TLS 1.3-centred properties are preserved on LURK-T. We also add a new 3-party agreement property for LURK-T, which ProVerif proves to hold, thus showing LURK-T to be a secure proxied TLS.

• All our ProVerif files and results are published at `https://github.com/lurk-t/proverif`.

8.3.1. Verifying Standardised TLS 1.3 in ProVerif

Our approach was to reuse a ProVerif specification of a draft of TLS 1.3, given in [75, 9]. The latest available version of this specification encoded draft 20 of

[6]From a security perspective, "LURK-T with DHE-active *CS*" is shown in Section 8.2 to be the version of choice for LURK-T. So, in this section, we analyse further. That said, the previous sections show "LURK-T with DHE-passive *CS*" as the choice for efficiency.

TLS 1.3 pre-standardisation (no newer version as confirmed by the authors). So, first, we updated this existing ProVerif specification of TLS with the RFC 8446. In short, the ProVerif model did not specify the handshake to include AEAD encryption for the *Certificate*, *CertificateVerify* and *Finished* messages. We applied the necessary updates to ProVerif models and verified that the original properties still held. The only difference observed is that, in our newly updated models for standard-TLS 1.3, the automatic proofs take longer, as we detail below.

Specification in ProVerif. The ProVerif specification from [75, 9] as well as ours is structured in two files:

- a library (.pvl) file – this contains the protocol description with the client and server as separated processes communicating over a public network channel. As expected, the file also comprises a series of reusable ProVerif constructs allowing the attacker to infer sensitive data from the public channel if weak DH groups, weak authenticated encryption, or weak hash functions are negotiated between the client and the server. Other aspects, such as the definitions of various events necessary to write the ProVerif "queries" corresponding to the targeted security goals, as expected, appear within this file. The pvl library also includes the ProVerif code reflecting the commonplace threat model for TLS w.r.t. symbolic verification, which is described in [9]. For instance, compromising a long-term key at the end of a protocol session is modelled and such post-compromise security is verified as expected, i.e., by using the ProVerif concept of "phases".
- a separate (.pv) file – this contains just the set of queries to capture the protocol security goals. These queries, as described in [9], fall into two big categories: a) secrecy and forward secrecy queries to verify under what precise conditions (events) secret data can be inferred by the attacker; b) authentication and anti-replay queries to verify, for example, that data reception by an honest party is preceded by the other honest party sending this data over a matching session.

From Draft-20 to the RFC for TLS 1.3. After we migrated the ProVerif specification of draft20 of TLS 1.3 to the current standard, we verified the original 24 queries from [75] against our updated ProVerif specification of TLS 1.3. We saw the same results as before i.e., we ascertain that TLS 1.3 as standardised, achieves the TLS security goals. During this phase, we also analysed the impact of a particular line of code in both the client and the server that explicitly leaks the handshake secret on the public channel; we concluded this artefact was left for sanity checks in the original code, and we disabled it in our LURK-T modelling.

The measured time to prove the newly updated model is significantly increased compared to the original version: running on a Windows 10 with Intel(R) Core(TM) i7-8650U CPU @1.90GHz and 32 GB RAM, without the option to generate the attack graphs as separate files for the proofs supposed to fail, ProVerif could

```
query cr:random, sr:random, cr':random, sr':random,
      psk:preSharedKey,p:pubkey, e:element,
      o:params, m:params,
      ck:ae_key,sk:ae_key,ms:bitstring,cb:bitstring, log:bitstring;

      inj-event(ClientFinished(TLS13,cr,sr,psk,p,o,m,ck,sk,cb,ms)) ==>
      (inj-event(PreServerFinished(TLS13,cr,sr,psk,p,o,m,ck,sk,cb))  ==>
(inj-event (TLS13_recvd_CV (cr, sr, p, log))  ==>
(inj-event (CS_sent_CV(cr, sr, p, log) ) ==>
inj-event (TLS13_sent_cr_sr_to_CS (cr, sr, p, log))
)
)
  )
      || (event(WeakOrCompromisedKey(p)) &&  (psk = NoPSK
|| event(CompromisedPreSharedKey(psk)))) ||
      event(ServerChoosesKEX(cr,sr,p,TLS13,DHE_13(WeakDH,e))) ||
      event(ServerChoosesHash(cr',sr',p,TLS13,WeakHash)).
```

Figure 6: Agreement query between C, E, CS

verify the original TLS 1.3 model in about 20 minutes but took 16 hours for the updated version. This is mainly due to an inclusion of certain AEAD (authenticated encryption with associated data) ciphers, as their equational theories cause some exponential increase in the search done by ProVerif w.r.t. proving certain queries.

8.3.2. Verifying LURK-T in ProVerif

We modelled LURK-T in ProVerif. We therefore split the ProVerif S-process in two: a CS process and a E process. In each, we encoded “LURK-T with DHE-active CS” and, specifically, also the case in which the CS and the E are not collocated. In this case, we modelled a secure channel between the CS and the E, as per the LURK-T specifications; in ProVerif, this is what is called a private channel, not accessible to the underlying Dolev-Yao attacker. We inherited all the Diffie-Hellman exponentiation aspects (incl. modelling weak subgroups, etc.) from the TLS implementation. Note that we do not model the TEE specifically, but since CS cannot be adaptively corrupted in the model at hand (which is the case symbolic verification), that equates to the TEE being modelled “by default”.

The query we added to the ones inherited from TLS 1.3 expresses that there is always a correct/secure session interleaving and execution between the C, E and CS, even with the Dolev-Yao attacker in the middle. In practice, this means that a Dolev-Yao attacker cannot find a way to mis-align the execution of the three parties by doing a man-in-the-middle-type attack. The query in encoded in Fig. 6.

Our 3-party, added query in Fig. 6 denotes that either there has been a compromise (of CS, or of the PSK, or by working in a weak DH subgroup), or security/agreement holds: i.e., in every LURK-T session/execution in which C finished a handshake (i.e., ClientFinished(...)), prior to that E acted as a TLS server and reached the point of sending out a DH share to the client (i.e., PreServerFinished(...)), and before that this E got from the CryptoService said share signed (i.e., TLS13_recvd_CV(...)), and before that the CryptoService sent this share (i.e., CS_sent_CV(...)), and before

E contacted the with C's handshake details (i.e., TLS13_sent_cr_sr_to_CS $(\ldots)$). Moreover, as one can see in the query in Fig. 6, the data is bound (i.e., coincides) between all such events on any given execution and the events are required to be injective, meaning that there is a 1-to-1 mapping in occurrences between them. In turn, the latter means that not only does this sequentiality and data-agreement hold for every execution of LURK-T, but also every single execution of CS will be mapped with one and only one single execution of E, and one and only one execution of C, by a unique set of matching handshake data, in the parameters specified in the query.

8.3.3. Experimental Setup

We conducted our ProVerif verification in two settings: (1) using an Ubuntu 20.04 Focal VM with a V100 GPU and 128 GB RAM on KVM; (2) using a laptop with Windows 10 and Intel(R) Core(TM) i7-8650U CPU @1.90GHz and 32 GB RAM and the latest version 2.02 for ProVerif. While the powerful setting (1) proved very suitable for the development phase of our ProVerif models and for fast verifications, we consider that setting (2) is more plausible to be used for reproducing our proofs. In setting (2), without the option to generate the attack graphs, analysing all 29 queries automatically (the 24 queries inherited from [75] plus the 5 queries we added specifically for LURK-T including the query detailed above), takes 17.25 hours. Verified separately, the query in Figure 6 requires 1.5 hours to be proved (true).

9. Conclusions

We introduce LURK-T – a provably secure and efficient extension of TLS 1.3, and of the generic LURK framework [65], to provide a trustworthy TLS with all cryptographic key material secured – not limited to those performed with the long-term private key alone. We design LURK-T with a TLS server S decoupled into a *LURK-T TLS Engine* (E) and a *LURK-T Crypto Service* (CS) and split the TLS handshake across the two modules; CS is executed inside a TEE and it accepts very specific and limited requests. We offer several modular variants of LURK-T, balancing security and efficiency. For all our LURK-T variants, we formally prove all standard AKE-security properties as well as TLS-delegation and trust-centred properties – in a cryptographic model for multi-party TLS. We also formally analysed LURK-T in the symbolic model, using ProVerif. We implemented CS using Intel SGX, and we integrated our implementation to OpenSSL with minimal changes. Finally, our experimental results looked at LURK-T's overheads compared to TLS 1.3 handshakes and demonstrated that LURK-T provides competitive efficiency.

References

[1] "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Security Aspects; Study on Security Impacts of Virtualisation (Release 18)," Mar. 2022. [Online]. Available: https://www.3gpp.org/ftp/Specs/archive/33_series/33.848/33848-0c0.zip

[2] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, C. Hankin and D. Schmidt, Eds. ACM, 2001, pp. 104–115. [Online]. Available: https://doi.org/10.1145/360204.360213

[3] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 743–754. [Online]. Available: https://doi.org/10.1145/2976749.2978358

[4] "2021 State of the Internet / Security Research Report: Gaming in a Pandemic," 2021. [Online]. Available: https://www.akamai.com/our-thinking/the-state-of-the-internet/global-state-of-the-internet-security-ddos-attack-reports

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[6] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves," 2017. [Online]. Available: https://www.doc.ic.ac.uk/research/technicalreports/2017/DTRS17-5.pdf

[7] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla, "Delegated Credentials for TLS," Internet Engineering Task Force, Internet-Draft draft-draft-ietf-tls-subcerts, Jan. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-ietf-tls-subcerts

[8] M. Bartock, M. Souppaya, R. Savino, T. Knoll, U. Shetty, M. Cherfaoui, R. Yeluri, A. Malhotra, and K. Scarfone, “Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases,” in *Draft NISTIR 8320*, may 2021. [Online]. Available: https://doi.org/10.6028/NIST.IR.8320-draft

[9] K. Bhargavan, B. Blanchet, and N. Kobeissi, “Verified models and reference implementations for the tls 1.3 standard candidate,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 483–502.

[10] 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.

[11] K. Bhargavan, I. Boureanu, P. Fouque, C. Onete, and B. Richard, “Content delivery over TLS: a cryptographic analysis of keyless SSL,” in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 1–16. [Online]. Available: https://doi.org/10.1109/EuroSP.2017.52

[12] J. Birr-Pixton, “rustls versus OpenSSL: handshake performance,” jun 2020. [Online]. Available: https://jbp.io/2019/07/02/rustls-vs-openssl-handshake-performance.html

[13] B. Blanchet, “Modeling and verifying security protocols with the applied pi calculus and proverif,” *Found. Trends Priv. Secur.*, vol. 1, no. 1-2, pp. 1–135, 2016. [Online]. Available: https://doi.org/10.1561/3300000004

[14] 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. [Online]. Available: https://rfc-editor.org/rfc/RFC5280.txt

[15] I. Boureanu, D. Migault, S. Preda, H. A. Alamedine, S. Mishra, F. Fieau, and M. Mannan, “LURK: server-controlled TLS delegation,” in *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020, Guangzhou, China, December 29, 2020 - January 1, 2021*, G. Wang, R. K. L. Ko, M. Z. A. Bhuiyan, and Y. Pan, Eds. IEEE, 2020, pp. 182–193. [Online]. Available: https://doi.org/10.1109/TrustCom50675.2020.00036

[16] ——, “LURK: server-controlled TLS delegation,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 1366, 2020. [Online]. Available: https://eprint.iacr.org/2020/1366

[17] I. Boureanu, A. Mitrokotsa, and S. Vaudenay, "On the pseudorandom function assumption in (secure) distance-bounding protocols - prf-ness alone does not stop the frauds!" in *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Hevia and G. Neven, Eds., vol. 7533. Springer, 2012, pp. 100–120. [Online]. Available: https://doi.org/10.1007/978-3-642-33481-8_6

[18] A. Brandao, J. Resende, and R. Martins, "Hardening of cryptographic operations through the use of secure enclaves," *Computers & Security*, p. 102327, 2021.

[19] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, "Securekeeper: Confidential zookeeper using intel SGX," in *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016.* ACM, 2016, p. 14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2988350

[20] C. Brzuska, H. Jacobsen, and D. Stebila, "Safely exporting keys from secure channels - on the security of EAP-TLS and TLS key exporters," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, M. Fischlin and J. Coron, Eds., vol. 9665. Springer, 2016, pp. 670–698. [Online]. Available: https://doi.org/10.1007/978-3-662-49890-3_26

[21] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 2020, pp. 54–72. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00089

[22] F. Cangialosi, T. Chung, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "Measurement and analysis of private key sharing in the HTTPS ecosystem," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 628–640. [Online]. Available: https://doi.org/10.1145/2976749.2978301

[23] S. Checkoway and H. Shacham, "Iago attacks: why the system call API is a bad untrusted RPC interface," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, V. Sarkar and R. Bodík, Eds. ACM, 2013, pp. 253–264. [Online]. Available: https://doi.org/10.1145/2451116.2451145

[24] G. Chen, Y. Zhang, and T. Lai, "OPERA: open remote attestation for intel's secure enclaves," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 2317–2331. [Online]. Available: https://doi.org/10.1145/3319535.3354220

[25] Cisco, "Cisco visual networking index: forecast and methodology, 2017-2022," 2017. [Online]. Available: https://s3.amazonaws.com/media.mediapost.com/uploads/CiscoForecast.pdf

[26] "cloc: Count Lines of Code." [Online]. Available: https://github.com/AlDanial/cloc

[27] T. Cloosters, M. Rodler, and L. Davi, "Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 841–858. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters

[28] "AWS CloudHSM: Managed hardware security module (HSM) on the AWS Cloud," 2022. [Online]. Available: https://aws.amazon.com/cloudhsm/

[29] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *J. Autom. Reason.*, vol. 46, no. 3-4, pp. 225–259, 2011. [Online]. Available: https://doi.org/10.1007/s10817-010-9187-9

[30] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[31] X. d. C. de Carnavalet and P. C. van Oorschot, "A survey and analysis of tls interception mechanisms and motivations," *arXiv preprint arXiv:2010.16388*, 2020.

[32] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A. Sadeghi, "LO-FAT: low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. ACM, 2017, pp. 24:1–24:6. [Online]. Available: https://doi.org/10.1145/3061639.3062276

[33] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983. [Online]. Available: https://doi.org/10.1109/TIT.1983.1056650

[34] "NFV SECURITY IN 5G: Challenges and Best Practices," in *European Union Agency for Cybersecurity (ENISA)*, Feb. 2022. [Online]. Available: https://www.enisa.europa.eu/publications/nfv-security-in-5g-challenges-and-best-practices/@@download/fullReport

[35] "Network Functions Virtualisation (NFV); NFV Security; Report on use cases and technical approaches for multi-layer host administration," in *ETSI GR NFV-SEC V1.2.1, European Telecommunications Standards Institute (ETSI)*, 2017. [Online]. Available: https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/009/01.02.01_60/gr_NFV-SEC009v010201p.pdf

[36] 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

[37] "Graphene Library OS with Intel SGX Support," 2020. [Online]. Available: https://github.com/gramineproject/gramine

[38] "Graphene Library OS with Intel SGX Support," 2020. [Online]. Available: https://github.com/oscarlab/graphene

[39] D. Gruss, "Transient-execution attacksand defenses," in *Transient-Execution Attacksand Defenses*. Graz University of Technology, 2020. [Online]. Available: https://gruss.cc/files/habil.pdf

[40] S. Gueron, "A memory encryption engine suitable for general purpose processors." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[41] "The Heartbleed Bug." [Online]. Available: https://heartbleed.com/

[42] S. Herwig, C. Garman, and D. Levin, "Achieving keyless cdns with conclaves," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 735–751. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/herwig

[43] L. Hewlett-Packard Development Company, "Hp open source security for openvms volume 2: Hp ssl for openvms," Hewlett-Packard Development Company, L.P. [Online]. Available: https://vmssoftware.com/docs/HP_Open_Source_Security_Vol2.pdf

[44] P. E. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698, Aug. 2012. [Online]. Available: https://rfc-editor.org/rfc/RFC6698.txt

[45] K. Igoe, D. McGrew, and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms," RFC 6090, Feb. 2011. [Online]. Available: https://rfc-editor.org/rfc/RFC6090.txt

[46] "Resources and Response to Side Channel L1 Terminal Fault," *Intel White Paper*. [Online]. Available: https://www.intel.ca/content/www/ca/en/architecture-and-technology/l1tf.html

[47] "Side Channel Vulnerabilities: Microarchitectural Data Sampling and Transactional Asynchronous Abort," *Intel White Paper*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html

[48] "Intel Software Guard Extensions SDK for Linux OS," *Developer Reference*, 2016. [Online]. Available: https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.7_Open_Source.pdf

[49] "Intel Software Guard Extensions (SG) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits," *Intel White Paper*, feb 2018. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/180204-sgx-sdk-developer-guidance-v1-0.pdf

[50] "Intel SGX Data Center Attestation Primitives (Intel SGX DCAP)," *Product brief*, 2019. [Online]. Available: https://download.01.org/intel-sgx/dcap-1.1/linux/docs/Intel_SGX_DCAP_ECDSA_Orientation.pdf

[51] "Affected Processors: Transient Execution Attacks & Related Security Issues by CPU," *Intel Security Center*, apr 2021. [Online]. Available: https://software.intel.com/security-software-guidance/processors-affected-transient-execution~{}-attack-mitigation-product-cpu-model

[52] Intel Corporation, "intel/intel-sgx-ssl," 2021. [Online]. Available: https://github.com/intel/intel-sgx-ssl

[53] ——, "intel/linux-sgx: Intel SGX for Linux*," 2021. [Online]. Available: https://github.com/intel/linux-sgx

[54] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 273–293. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_17

[55] J. Jiang, X. Chen, T. O. Li, C. Wang, T. Shen, S. Zhao, H. Cui, C. Wang, and F. Zhang, "Uranus: Simple, efficient SGX programming and its applications," in *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, H. Sun, S. Shieh, G. Gu, and G. Ateniese, Eds. ACM, 2020, pp. 826–840. [Online]. Available: https://doi.org/10.1145/3320269.3384763

[56] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services ," *Intel White Paper*, 2016. [Online]. Available: https://software.intel.com/content/dam/develop/public/us/en/documents/ww10-2016-sgx-provisioning-and-attestation-final.pdf

[57] "SRBDS - Special Register Buffer Data Sampling," *The Linux kernel user's and administrator's guide, The kernel development community.* [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/special-register-buffer-data-sampling.html

[58] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xingand, and M. Vij, "Integrating Intel SGX Remote Attestation with Transport Layer Security," *Intel White Paper*, jul 2019. [Online]. Available: https://arxiv.org/pdf/1801.05863.pdf

[59] A. Levy, H. Corrigan-Gibbs, and D. Boneh, "Stickler: Defending against malicious content distribution networks in an unmodified browser,"

IEEE Secur. Priv., vol. 14, no. 2, pp. 22–28, 2016. [Online]. Available: https://doi.org/10.1109/MSP.2016.32

[60] 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, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 2014, pp. 67–82. [Online]. Available: https://doi.org/10.1109/SP.2014.12

[61] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[62] W. Liu, H. Chen, X. Wang, Z. Li, D. Zhang, W. Wang, and H. Tang, "Understanding TEE containers, easy to use? hard to trust," *CoRR*, vol. abs/2109.01923, 2021. [Online]. Available: https://arxiv.org/abs/2109.01923

[63] "Safenet Luna performance and upgrades." [Online]. Available: (https://thalesdocs.com/gphsm/luna/5.3/docs/network/007-011136-006_lunasa_5-3_webhelp/Content/administration/faq.htm

[64] J. P. Mechalas, "Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example," *Intel White Paper*, mar 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html

[65] D. Migault, "LURK Protocol version 1," Internet Engineering Task Force, Internet-Draft draft-draft-mglt-lurk-lurk, Feb. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-mglt-lurk-lurk

[66] ——, "LURK Extension version 1 for (D)TLS 1.3 Authentication," Internet Engineering Task Force, Internet-Draft draft-draft-mglt-lurk-tls13, Jan. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-mglt-lurk-tls13

[67] 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, Jan. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-mglt-lurk-tls12

[68] "Mozilla Wiki: Security/Server Side TLS," jun 2020. [Online]. Available: https://wiki.mozilla.org/Security/Server_Side_TLS

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

[70] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. R. Rodríguez, and P. Steenkiste, "Multi-context TLS (mctls): Enabling secure in-network functionality in TLS," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, S. Uhlig, O. Maennel, B. Karp, and J. Padhye, Eds. ACM, 2015, pp. 199–212. [Online]. Available: https://doi.org/10.1145/2785956.2787482

[71] "nShield Connects HAM." [Online]. Available: https://go.ncipher.com/rs/104-QOX-775/images/nShield-Connect-ds.pdf

[72] "OpenSSL: Cryptography and SSL/TLS Toolkit," 2019. [Online]. Available: https://www.openssl.org

[73] M. Pei, H. Tschofenig, D. Thaler, and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture," Internet Engineering Task Force, Internet-Draft draft-draft-ietf-teep-architecture, Jul. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-ietf-teep-architecture

[74] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," *CoRR*, vol. abs/1908.11143, 2019. [Online]. Available: http://arxiv.org/abs/1908.11143

[75] "Inria Prosecco - RefTLS," 2018. [Online]. Available: https://github.com/Inria-Prosecco/reftls

[76] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CROSSTALK: Speculative Data Leaks Across Cores Are Real," 2020. [Online]. Available: https://download.vusec.net/papers/crosstalk_sp21.pdf

[77] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://rfc-editor.org/rfc/RFC8446.txt

[78] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives," *Intel White Paper*, apr 2019. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/supporting-third-party-attestation-for-intel-sgx-data-center-attestation-primitives.html

[79] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: trustworthy data analytics in the cloud using SGX," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 2015, pp. 38–54. [Online]. Available: https://doi.org/10.1109/SP.2015.10

[80] 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. [Online]. Available: https://rfc-editor.org/rfc/RFC8739.txt

[81] Y. Sheffer, D. Lopez, A. Pastor, and T. Fossati, "An ACME Profile for Generating Delegated Certificates," Internet Engineering Task Force, Internet-Draft draft-draft-ietf-acme-star-delegation, Jun. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-draft-ietf-acme-star-delegation

[82] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with SGX enclaves," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/

[83] D. Stebila and N. Sullivan, "An analysis of TLS handshake proxying," in *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1.* IEEE, 2015, pp. 279–286. [Online]. Available: https://doi.org/10.1109/Trustcom.2015.385

[84] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "Ts-perf: General performance measurement of trusted execution environment and rich execution environment on intel sgx, arm trustzone, and RISC-V keystone,"

IEEE Access, vol. 9, pp. 133 520–133 530, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3112202

[85] H. Tadepalli, "Intel® QuickAssist Technology with Intel Key Protection Technology in Intel Server Platforms Based on Intel Xeon Processor Scalable Family," in *White Paper*. Intel Corporation, 2017. [Online]. Available: https://www.aspsys.com/images/solutions/hpc-processors/intel-xeon/Intel-Key-Protection-Technology.pdf

[86] "IETF Trusted Execution Environment Provisioning (teep) Working Group," 2020. [Online]. Available: https://datatracker.ietf.org/wg/teep/documents/

[87] D. J. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. B. Butler, "A practical intel SGX setting for linux containers in the cloud," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, G. Ahn, B. M. Thuraisingham, M. Kantarcioglu, and R. Krishnan, Eds. ACM, 2019, pp. 255–266. [Online]. Available: https://doi.org/10.1145/3292006.3300030

[88] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[89] C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An efficient java partitioning framework for hardware enclaves," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 505–522. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/tsai

[90] C. Wei, J. Li, W. Li, P. Yu, and H. Guan, "STYX: a trusted and accelerated hierarchical SSL key management and distribution system for cloud based CDN application," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 2017, pp. 201–213. [Online]. Available: https://doi.org/10.1145/3127479.3127482

[91] N. Weichbrodt, P. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel SGX enclaves," in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14,*

2018, P. Ferreira and L. Shrira, Eds. ACM, 2018, pp. 201–213. [Online]. Available: https://doi.org/10.1145/3274808.3274824

[92] O. Weisse, V. Bertacco, and T. M. Austin, "Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 81–93. [Online]. Available: https://doi.org/10.1145/3079856.3080208

[93] D. Wessels, "Disclosure of root zone TSIG keys." [Online]. Available: https://lists.dns-oarc.net/pipermail/dns-operations/2020-May/020198.html

[94] "Relative change and difference," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Relative_change_and_difference

[95] wolfSSL Inc., "wolfSSL with Intel® SGX," 2018. [Online]. Available: https://www.wolfssl.com/wolfssl-with-intel-sgx

[96] "wrk - a HTTP benchmarking tool," 2020. [Online]. Available: https://github.com/wg/wrk

[97] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana, "Plinius: Secure and persistent machine learning model training," in *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 2021, pp. 52–62. [Online]. Available: https://doi.org/10.1109/DSN48987.2021.00022

[98] P. Yuhala, J. Ménétrey, P. Felber, V. Schiavoni, A. Tchana, G. Thomas, H. Guiroux, and J. Lozi, "Montsalvat: Intel SGX shielding for graalvm native images," in *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds. ACM, 2021, pp. 352–364. [Online]. Available: https://doi.org/10.1145/3464298.3493406

[99] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 283–298. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

Appendices

10. TLS 1.3

We assume some familiarity with TLS 1.3 [77] and give just a high-level overview of it.

TLS 1.3 Modes. The part of the TLS 1.3 protocol [77] which establishes the *session-key* or *channel-key* is called a *handshake*; there are other parts, e.g., the record layer, etc. We are herein concerned mainly with the handshake.

The most important exchanges in a TLS handshake are: the ClientHello and ServerHello messages (containing a random nonce by C and S, respectively), the protocol-extensions' negotiation, a message authenticating S to C, and a "Server Finished" message (Fin_E) which is a HMAC of the S's view of handshake. TLS1.3 has (operation) *modes*, which equate to different families of actual authenticated-key exchange (AKE) protocol run within these handshake exchanges. Two mainstream modes we use herein are: (1) Elliptic Curve Ephemeral Diffie-Hellman (ECDHE), and (2) Pre-shared Secret Key with ECDHE (PSK-ECDHE). Please refer[7] to Figure 7.

The essential difference between these two modes is w.r.t. two notions called: *session resumption* and *perfect forward secrecy (PFS)*. PFS is a security property of interest to TLS whereby if the current session-key leaks, the security of prior session-keys is not affected. Session resumption is an extension negotiated between C and S whereby the future session-key can be computed in a "shortcut" handshake from a piece of data called *pre-shared secret key (PSK)* calculated during the current handshake. Thus, a session-key generated via resumption is related to the session-key in the previous session/handshake. In this vein, ECDHE mode entails an AKE which can give PFS, whereas PSK-ECDHE is used in resumed sessions and generally does not attain PFS.

TLS 1.3 Key Schedule. On Figure 7, the handshake messages are included using the standard TLS terminology. We now focus on the computation of the channel key.

A TLS sub-process called "Key Schedule", which runs on both C and S (see Figure 7), is the one that implements the key-derivation to obtain the *channel/session key*. After the ClientHello and ServerHello, the Key Schedule computations yield the C and the S' handshake secrets h_C and h_S, respectively. Using these secrets, Key Schedule generates keys to encrypt and decrypt the handshake payloads (see $\mathtt{AE}_h(\ldots)$ on Figure 7); "$\mathtt{AE}$" stands for authenticated encryption. In the CertificateVerify message, S provides the signature (PSign) on the transcript, and then S' side

[7]We note that on Figure 7, the "key_share" extension of the ClientHello and ServerHello messages carry the respective (EC)DHE key-shares computed by C and S, respectively.

of Key Schedule generates the Fin_E message. Thereafter, the key derivation continues (from h), and Key Schedule generates the application secrets a_C and a_S, which are combined to derive the actual channel keys a, for encrypting/decrypting ApplicationData (see $\mathtt{AE}_a(\ldots)$ on Figure 7).

What we described thus far (i.e., key derivation from h to a) corresponds to the ECDHE mode. In this mode, when session resumption is enabled, S's side Key Schedule also generates the resumption secret (r) and a nonce to produce the PSK, which will be used in future resumed runs. After this generation, in the so-called "stateful" mode, S sends a reference to that PSK, or –in "stateless" mode– S sends the PSK itself encrypted; both in the NewSessionTicket message from S to C.

Session Resumption. In PSK-ECDHE mode, when C wants to open a new connection to S by resuming an old one they had, then the Key Schedule does not start from h as per the above. Instead, it is initiated with the PSK indicated in the NewSessionTicket of the previous exchange (the lower half of Figure 7). Notably, C includes the PSK and a binder — a proof of possession of that PSK — in the ClientHello, all the while using the key_shareextension.

When initialized with the PSK, Key Schedule first derives (from the PSK) the binder key that enables validating the possession (typically a HMAC) of the indicated PSK by C. Upon the binder validation, Key Schedule derives h_C, h_S, a_C and a_S, as in the ECDHE mode.

Finally, the PSK-ECDHE mode may also enable session resumption itself, in which case, similarly to the ECDHE, a NewSessionTicket message is sent by S.

Above, we assumed and described only the common case of where just S authenticates to C.

11. TEE Application and Runtime Attestation

11.1. Trusted Execution Environments (TEEs)-based Attestation

A developer needs to divide the target program into two separate parts: trusted and untrusted. The trusted part (runs in an SGX enclave) uses a special part of the DRAM called Enclave Page Cache (EPC). To protect against memory attacks, EPC is encrypted (and integrity-protected) using transparent memory encryption (the keys are only available to the CPU [40]). Applications can only call the trusted APIs (with well-defined entry points) in the SGX (via ECALLs and OCALLs). Therefore, this part of the memory cannot be accessed by other applications or OS/hypervisors [39].

The CPU measures the software (e.g., code) and hardware configuration of the CPU's SGX. Subsequently, the user can determine the integrity of the configuration and code running on SGX before using it. This process is called *attestation* and can be done using different methods (local or remote attestation). For instance,

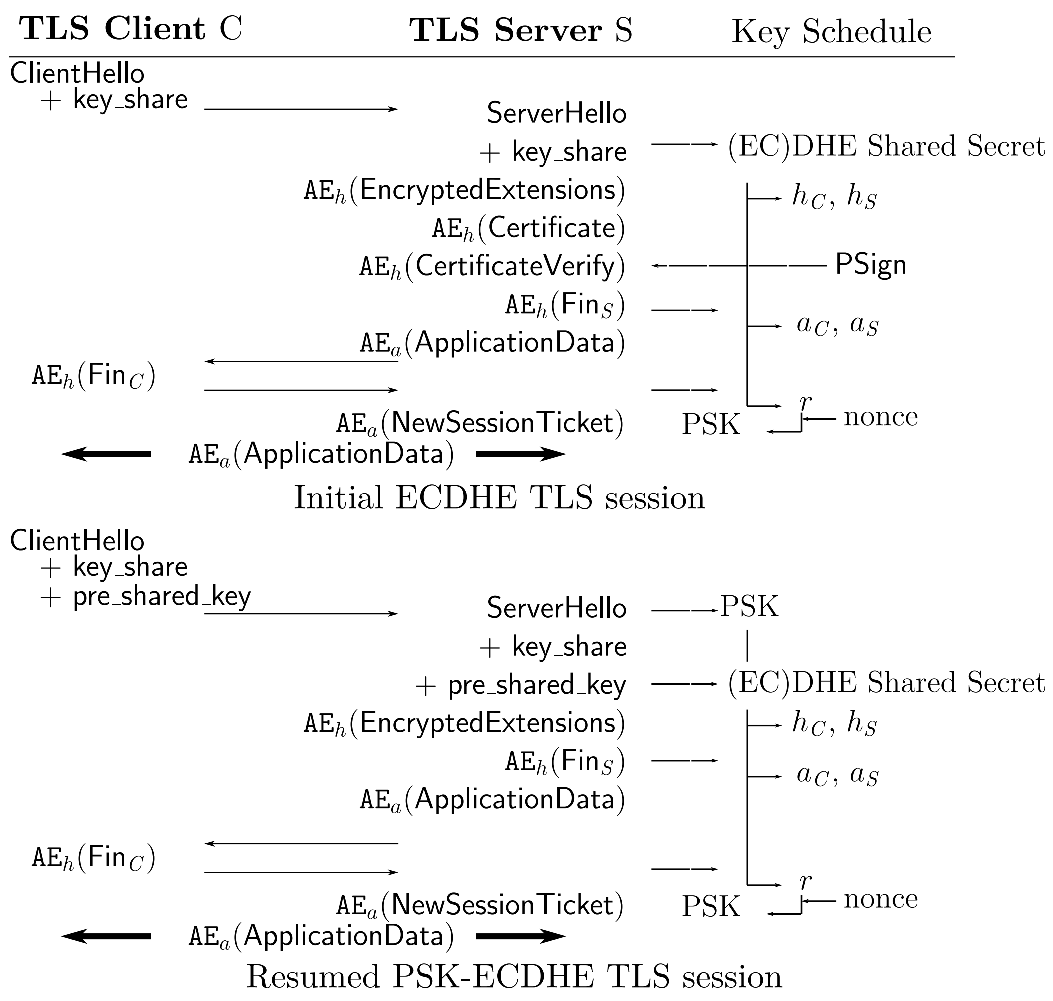


Figure 7: ECDHE and PSK-ECDHE TLS 1.3 Handshakes [77]

in remote attestation, the CPU provides a signed *quote* from the enclave (the hardware/software configuration as well as the enclave public key). The user can check the signed quote using various ways such as Intel Attestation Service (IAS [56, 64]), Intel Data Center Attestation Primitives (DCAP [50, 78]), or alternatives like OPERA [24].

11.2. Runtime Attestation

Traditional attestation measures in software (e.g., SGX) or even hardware (e.g., TPM - Trusted Platform Module) generally attest binaries at load time, and do not (try to) catch run-time attacks. By contrast, *runtime attestation* schemes aim to measure and attest the flow (i.e., the control-flow) of an executing code. We will describe the idea behind runtime attestation generically, though variations to this exist. For runtime attested code, instructions of runtime-attested code are smartly tagged prior to deployment such that they can be picked up by a piece of code called *runtime tracer (RTT)*. The RTT recovers the source and destination addresses of executions, which are passed to a *measurement engine (ME)*. The ME will hash/MAC these code and addresses together with some measurements, and this will be a passed to an attester which checks them against the supposed/pre-specified control-flow graph of the code. The ME runs itself in a secure enclave. Examples of well-known runtime attestation schemes are C-FLAT [3] and LO-FLAT [32]. We do not use runtime attestation in the variants of LURK-T we test in practice. Yet, in Section 8.1, we introduce a sub-variant of LURK-T that does use runtime attestation in order to attain a stronger security property w.r.t. CS accounting for the handshake it signs: i.e., indirectly, this is further protection/reassurance of/for the client against malicious middleboxes.

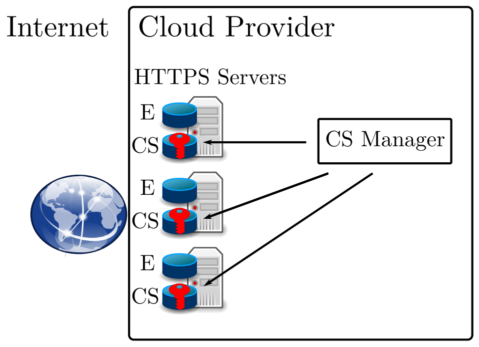
12. Extended Use Cases Description

As detailed in section 4, the owner of TLS credentials uses CS implemented in a TEE in order to prevents the credentials to be disclosed, and to protect the confidentiality of the TLS sessions when served in an untrusted environement such as a cloud provider.

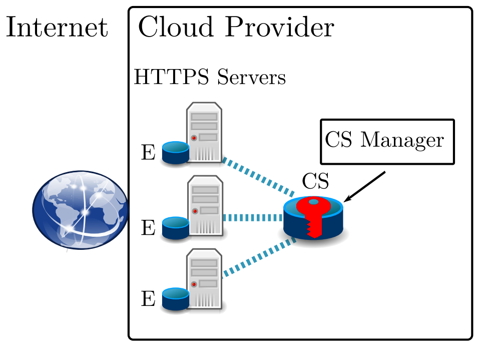
This section depicts in Figure 8 three use cases that we believe are representative to the motivations for deploying CS. Figure 8a and section 12.1 describe the case where a cloud provider manages multiple TLS front ends, each instantiated by E and CS. Figure 8b and section 12.2 describe the case where a centralized CS is shared by multiple Es. Both use cases can be seen as the cloud provider improving the security of the TLS deployment while Figure 8c and section 12.3 extends the previous scenarios where CS is managed by a entity that is not the cloud provider —typically the content owner. It is worth clarifying that CS does not offer additional protection for the content, but ensures the confidentiality of the private key (or identity) of the content owner. One motivation for considering the content and the credentials as two different pieces of data is that a cloud provider may need to access the data to operate and provide DDoS protection, content caching, application firewall, while it does not need to access the identity of the content owner.

The formal verification in this paper covers all three scenarios but the implementation described in section 6 only covers the cases of CS co-located with the TLS engine described in section 12.1 and 12.3. However this section describes in sufficient details how the implementation described in section 6 can be extended to the other use cases.

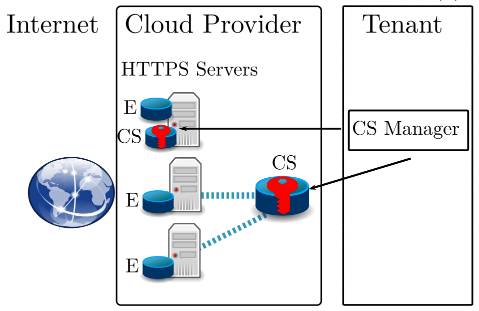
This paper limiting its scope to TEE implemented though SGX [30] in which case, an SGX enclave is initialized from code load from the untrusted environment also designated as Rich Execution Environment (REE). This prevents an enclave



(a) Distributed *CS*



(b) Centralized *CS*



(c) Tenant's *CS*

Figure 8: *CS* deployment use cases

to be provisioned at the instantiation. Instead *CS* manager provision *CS* after its instantiation via a remote attestation process that provides some guarantees the enclaves is a legitimate *CS* instantiated into a legitimate SGX enclave.

For conciseness, this sections limits the remote attestation procedures to those provided by Intel, i.e the Intel Attestation Service (IAS) [56], [64] and [48] as well as Intel Data Center Attestation Primitives (DCAP) [50], [78]. Alternatives like OPERA [24] are likely to extend attestation procedure documented by Intel.

Note that the IETF Trusted Execution Environment Provisioning (TEEP) WG [86] is designing a protocol for managing trusted application within TEE [73] it remains unclear to the authors how it applies to SGX.

12.1. Cloud Provider with CS co-located to E

In Figure 8a each HTTPS server that terminated the TLS session implements *E* and a co-located *CS*. This architecture minimizes the changes over the traditional use of a TLS library as it can be transparently be replaced by a bundle composed of *E* and *CS*. This architecture also minimizes the latency of the communication between *CS* and *E*. The use of multiple instances of *CS*, avoids *CS* to present a bottle neck issue (as it could be the case the centralized CS described in section 12.2), but comes with the drawback of having multiple CS to manage which may also become an issue when the number is increasing.

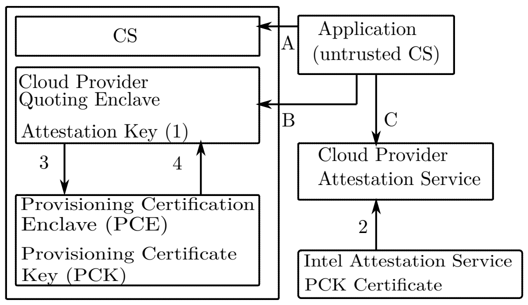
CS is hosted where the TLS session is terminated, and as such may remain relatively exposed to an attacker. An attacker that get the control of E would be able to interact with CS. The protection of the secrets is ensured by the TEE as well as the design of the possible interactions with CS.

In this example the Cloud Providers manages both CS the TEE - that is the hosts of CS - and the provisioning of CS. Because the TEE is owned by the cloud provider, there is no need for the cloud provider to provide privacy around the identity of the TEE. There is no concerns on tracking the TEEs by the cloud provider. In addition, the cloud provider is likely to be willing to limit the necessary interactions with attestation infrastructures outside the cloud provider. With these constraints attestation may be provided using Intel Data Center Attestation Primitives (DCAP) [50], [78] based on ECDSA and represented in Figure 9a.

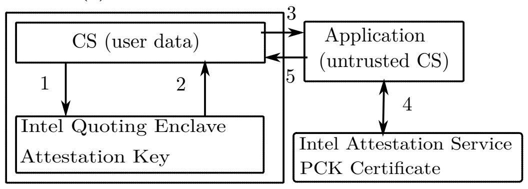
The Cloud Provider Quoting Enclave (CP-QE) is provided by Intel and generates a hard coded CDSA P-256 Attestation Key (AK) (1) via EGETKEY. The exact value of AK is not known by Intel as it includes a seal secret and is used to sign quotes. The CP-QE requests the Provisioning Certificate Enclave (PCE) a certificate that contains the CP-QE identity (MRENCLAVE) and the public part of the AK (3, 4). A Provisioning Certificate Enclave (PCE) provided by Intel acts as an intermediary certificate authority for the Cloud Providers Quoting Enclaves (CP-QE). The PCE provides Platform Provisioning ID (PPID) that characterizes the platform and signs certificate requests with a hard coded Provisioning Certification Key (PCK) retrieved via EGETKEY. The PCK is specific to the platform and the version of the SGX (CPUSVN) and generated on ECDSA P-256 curve [45]. For each CPUs, the Cloud Provider Attestation Service (CP-AS) retrieves via the Provisioning Certification KeyCertificate API Intel signed certificates for the PCK as well as revocation list (2). The certificates are regular X509 certificates with custom fields that includes PPID, CPUSVN, PCE's ISVSVN, Family-Model-Stepping-Platform Type-SKU of the CPU, and additional product type specific fields.

When CS is initialized, CS is signed (MRSIGNER) and contains the Intel public key used to check the signature. CS is instantiated and the signature is checked by SGX that owns the Intel secret key. CS generates (A ,B) REPORT which is attested and signed by the CP-QE into a Quote. Among other information the measurements of the enclave (MRENCLAVE) and TCB information such as (ISV, Product ID, and report data that can be any 512-bit values).

Upon being provided a Quote (C), the CP-AS requests the quote of CS and proceeds as follows:


CS
Cloud Provider Quoting Enclave
Attestation Key (1)
Provisioning Certification Enclave (PCE)
Provisioning Certificate Key (PCK)
Application (untrusted CS)
Cloud Provider Attestation Service
Intel Attestation Service PCK Certificate
A
B
C
2
3
4

(a) Data Center Attestation Primitives


CS (user data)
Intel Quoting Enclave
Attestation Key
Application (untrusted CS)
Intel Attestation Service PCK Certificate
1
2
3
4
5

(b) Attestation Service Utilizing Enhanced Privacy ID (EPID)

Figure 9: *CS* Remote Attestation

1. Verify the integrity of the signature chain from the Quote to the Intel-issued (PCK) certificate.
2. Verify no keys in the chain have been revoked.
3. Verify CP-QE is from a suitable source and is up to date.
4. Verify the status of the Intel SGX TCB described in the chain.
5. Verify the MRENCLAVE reflects *CS* identity.

Attestation alone is not sufficient to enable a secure communication from *CS* Manager to *CS*. Session keys needs to be agreed so *CS* Manager can provision *CS* securely. There are different ways to generates this key material and some mechanisms uses the attestation to bind a public (EC)DHE value generated by the enclave to further proceed to a SIGMA protocol [64] and [48]. Such designs presents some significant drawbacks as attestation procedures are required for any exchange, and that the agreement on a shared secret is not itself sufficient to establish a session. As Intel Attestation Infrastructure is remote, this increases the latencies when a TLS session is established and makes the IAS a point of failure. Note that this inconvenient is even stronger in the the centralized where multiple E establish a TLS session with *CS*. TLS is the defacto protocol to set a security session and TLS Remote Attestation (TLS-RA) [58] describes how attestation can be combined with a TLS exchange. Note that the changes are not affecting TLS and all changes are implemented via X509 extensions. As a result, while the paper considers TLS 1.2, the approach is can be applied to TLS 1.3 and this is the setting considered in the remaining of the section.

To *CS* needs to be extended with a TLS server, as well as a mechanisms that ensures a pair of keys are generated in the enclave, right after the instantiation. These keys will be used to authenticate *CS* in a TLS ECDHE authentication. It is important these keys are generated in the enclave at its creation to guarantee the keys have not leaked. The insurance that keys are fresh is typically provided by reading the source code of the enclave. The key is bound to the Quote by inserting their hash as user data. All these information are bundled into a X509 certificate via newly defined extensions.

It is recommended that only *CS* Manager is authorized to access *CS*, in which case *CS* may be configured to perform TLS mutually authenticated session with only *CS* Manager. Once the TLS session is set, *CS* Manager is able to configure *CS* securely. Note that in this case, the TLS session is only being used for management purpose and the frequent ECALL/OCALL are not expected to impact the operation of *CS*. It also worth noticing that only the TLS and above layers are expected to terminate inside the enclave. The TCP session for example is expected to terminate outside the enclave.

12.2. Cloud Provider with a centralized CS

In Figure 8a multiple *E*s interacts with a common centralized *CS*. When TEE was not involved, such architecture prevented the dissemination of security credentials across various hosts and reduces their exposition. Firstly, the keys are better controlled and stored by design into a single location (the centralized *CS*) as opposed to duplicated in every backups stored at various locations in data center. Secondly, a centralized *CS* is likely to be stored in the in back end network as opposed *E* that terminates the TLS session and are likely to be exposed to the Internet. If *CS* is not running in a TEE, the private key remains visible to the CDN provider and anyone accessing the server hosting *CS* as described in [16] and [67]. The purpose of *CS* is to prevent interactions with *CS* to leak the credentials but assumes that the attacker does not gain control of *CS*. TEE prevents direct access by any attacker to *CS*. As a result, *CS* does not necessarily need to be located in less exposed location. Typically, TEE make it as secure to co-locate *CS* in *E* as described in section 12.1, it also obsoletes the need of delegation mechanisms such as [7] to be able to provision untrusted clouds with credentials.

When *CS* is running in a TEE, it is important the TEE security domain is extended to the remote *E*s via, for example, a secure communication between each *E* and *CS*. In our case, the secure communication is enforced by TLS. *CS* is running a TLS server that is configured to establish mutually authenticated TLS sessions with the authorized *E*s and *CS* Manager. Note that in this case, the TLS session goes inside the enclave for every exchange between *E* and *CS* as opposed to section 12.1 where the TLS session was only involved for the configuration. However this does not represent a huge change as it does not provides additional ECALL/OCALL exchanges as in section 12.1. The only difference is that an encrypted packet will be exchange.

The main advantage of this architecture is that *CS* is located at one point which eases the management but also makes it a single point of failure.

Attestation and provisioning is performed as described in section 12.1. It is recommended that *E* uses similar certificate validation as *CS* Manager. The certificate message of *CS* is expected to be relatively larger than in most TLS session. While this may be improved over time with compression mechanisms or by reducing the chain itself, this negotiation is expected to happen only at start up. In order to enable fast re-connection, it is however recommended to enable session resumption.

12.3. CS in untrusted Cloud

In Figure 8c *CS* is hosted by a cloud provider but is provisioned by an external tenant —such as a content owner for example. This scenario is designated as untrusted cloud as the tenant does not share the credentials with the cloud provider. *CS* may be provided or be chosen by the cloud provider but the tenant must agree and trust *CS* implementation. It is likely that *CS* will be provided by a third party (*CS* developer) that is neither the cloud provider nor the tenant but trusted by both of them.

The cloud provider is not willing to have a tenant being able to track the CPUs used by the Cloud Provider. Typically the use of a ECDSA key for the CP-QE enables the tenant to know whether the same or a different CPU is used. Note that on a tenant perspective there are little difference between instantiating a centralized *CS* or a specific *CS*. It is likely that the centralized architecture will be preferred as it enables the tenant to configure a single *CS* and leave the management of the TLS termination to the cloud provider.

It is also worth noticing that while the scenario is designated a untrusted cloud, the content is not necessarily protected. In some cases, the content may be encrypted, but in some other cases the cloud provider will have access to the content. As a result, it is likely the tenant needs to have some (limited) trust in the cloud provider.

The main advantage is that the tenant keep the control of its identity by avoiding sharing it with the cloud provider. Without using TEE, it was required that *CS* being hosted in tenant's administrative domain which required the cloud provider to establish a TLS session with the tenant. This obviously introduced some latency as well as an operational risk that the session between the cloud provider becomes the weak point to a DDoS attack for example.

As the cloud provider is not willing to disclose the identities of its CPUs, an anonymous attestation scheme is required. In this case Intel provides Attestation Service Utilizing Enhanced Privacy ID (EPID) [56], [48]. The CPU belongs to a group and is able to sign as a member of the group. Intel is able to verify the signature and validates the signature has been generated by a genuine CPU but is not able to determine which CPU has signed. Groups are managed by Intel and the use of group signature protects the cloud provider from having its CPUs being tracked individually. More specifically, the TEE has a private group key that is used as the attestation key and Intel owns the public group key which makes it able to verify any signature generated by the private group key.

For the same reasons described in section 12.1 a secure session between the tenant and *CS* as well as, in the case of a centralized *CS*, between *E*s and *CS*. TLS is the defacto protocol to establish secure session, and is combined with the Intel attestation [58] as illustrated in Figure 9b.

Once instantiated, *CS* generates a new private public key pair that will be used to authenticate the enclave. Trust in *CS* code or in *CS* developer ensures the freshness of the key. *CS* inserts the hash of the public key as user-data in the REPORT which is signed by the Quoting Enclave (1-2). The Untrusted *CS* is then responsible to proceed to the attestation from IAS (4) and returns to *CS* (5) the attestation verification generated by Intel with its associated signature and the chain certificate for the attestation verification signing key. These pieces of information are included in the certificate via X509 extensions. The report generated by *CS* is included in the attestation verification report and so contains the identity of *CS* (MRENCLAVE and MRSIGNER) as well as information related to the TEE such as the CPUSVN.

13. Formal Security Notions (for TLS)

In this appendix, we give the formal notions for ACCE and 3(S)ACCE in Section 13.1 and Section 13.2, respectively. In Section 13.3, we recall a notion on pseudorandom functions useful in two of our formal proofs.

13.1. The (S)ACCE Models [54]

We briefly describe the authenticated and confidential channel establishment (ACCE) security model. We use the notations Brzuska et al. [20].

Parties and instances. The ACCE model considers a set $\mathscr{P}$ of parties, which can be either *clients* $C \in \mathscr{C}$ or *servers* $S \in \mathscr{CS}$. 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) [11]. 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.

13.2. Details on the 3(S)ACCE Model [11]

We first recall the 3(S)ACCE [11] security requirements.

3(S)ACCE Entity Authentication (EA). An *EA attacker* can corrupt parties (i.e., making them perform 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* 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 C or CS, and X is different from Y. We are mostly interested in the case where X is "C" and Y is "CS", i.e., the EA views the authentication of CS being forged to a given C. We say a *trusted server-controlled delegated TLS achieves entity-authentication*, if there is no EA attack onto the protocol.

In the 3(S)ACCE formal model [11], the notion of "mixed-2-ACCE entity authentication" also appears. It is called *mixed* because (see e.g. Figure 1a) "to the left" of E —there is an unilateral authentication protocol, and "to the right" of E —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.

3(S)ACCE Channel Security. We say a *trusted server-controlled delegated TLS 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 E 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 (PFS).

3(S)ACCE Accountability. We say a *trusted server-controlled delegated TLS achieves accountability* if CS is able to compute the channel keys used by C and the middle party, which in our case is E. This empowers CS to audit the activity of the TLS service at the record layer, should this be required.

For a start, the 3(S)ACCE model adds new aspects linked to the Engine party: the set $\mathscr{E}$ of such parties, new notions and attributes to align the 3 types of parties. For instance, there is a registration oracle setting up Engines and CSs, etc.

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-T there are 4 instances of parties that form one partnering: a C instance, one E instance (for the left-side communication), another E instance (for the right-side communication), a Service instance. This type of partnering, allows [11] to re-use 2-party security definitions for authentication and channel security. W.r.t. partnering, we stipulate that "$\pi_i^m.\mathsf{PSet}$" – denotes all the parties partner with this instance/session m of party i, and $\pi_i^m.\mathsf{InstSet}$ – denotes all the instances/sessions partnered with the said instance/session m of party i.

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 [11].

Entity Authentication (EA) [11]. 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 $P_j \in \{\mathscr{CS}, \mathscr{E}\}$, 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 $P_j \in \{\mathscr{CS}, \mathscr{E}\}$ 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 $\mathsf{Adv}^{\mathsf{EA}}_{\mathsf{LURKT}}(\mathscr{A})$, is defined as its winning probability *i.e.*:

$$\mathsf{Adv}^{\mathsf{EA}}_{\mathsf{LURKT}}(\mathscr{A}) := \mathbb{P}[\mathscr{A} \text{ wins the EA game}],$$

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

Channel Security (CS) [11]. 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:

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

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

Accountability (Acc) [11]. 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 $P_j \in \mathscr{CS}$*;*
(b) There exists no instance $\pi_j^n \in P_j.\mathsf{Instances}$ *such that* $\pi_j^n.\mathsf{ck} = \pi_i^m.\mathsf{ck}$*;*
(c) There exists no probabilistic algorithm Sim *(polynomial in the security parameter) which given the view of P_j (namely all instances* $\pi_j^n \in P_j.\mathsf{Instances}$ *with all their attributes), outputs* $\pi_i^m.\mathsf{ck}$*.*

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

$$\mathsf{Adv}^{\mathsf{Acc}}_{DHE-active CS}(\mathscr{A}) := \mathbb{P}[\mathscr{A} \text{ wins the Acc game}],$$

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

13.3. Programmable PRFs

"Programmable PRFs" [17] 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 Es do know their keys of PRF instances used in our construction, so they can exploit programmable PRFs. 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.

13.3.1. ProVerif

ProVerif is a symbolic verification tool, based on applied pi-calculus [2] – which is a formalism to model and analyze security protocols via a process-algebra semantics. As in most symbolic methods, the various protocol messages are modelled as *algebraic terms* exchanged over *private/public channels* between *processes* (C, E, CS), and their instances. As in all symbolic formalisms, a Dolev-Yao adversary [33] is present in the formalism, referred to as an *attacker*, who can access public channels by reading, modifying or replaying messages. Cryptographic primitives applicable on terms in the various processes (e.g., encryption, signing) are publicly known to honest parties and, unless specified private, also accessible by the protocol adversary. The attacker does not, by default, know secret/symmetric keys.

The protocol in ProVerif is encoded as a transition system (defined via processes reading and writing messages onto communicating channels); the execution of the said system generates *traces*, i.e., interleaving of protocol runs with the attacker "in the middle". Security properties in ProVerif are most often encoded as trace properties, i.e., check some condition on protocol variables across processes is met on a trace. In this shape, as trace properties, one can easily express notions of authentication (via data-agreement) as well as secrecy. Once we encode a property in ProVerif, the tool would try to prove it holding. Whenever a property is not holding (i.e., the tool returning *false* on it), ProVerif is also capable of providing a protocol trace example, which illustrates how the attacker can infringe the property. In essence, in ProVerif –as in all symbolic tool– it is often about using the tool to prove that there is an interleaving of protocol execution/session by the attacker leading to a security property failing, or proving that all its traces respect the property (e.g., authentication).

14. Proofs

We now present different security proofs for LURK-T using the 3(S)ACCE model in [11] and recalled in Section 13.

14.1. Entity Authentication Proof

We now prove 3(S)ACCE entity-authentication security for LURK-T in both its versions. The security game for 3(S)ACCE entity-authentication is recalled in Section 13.

Theorem 1. *Let P be the unilaterally-authenticated TLS 1.3 handshake (as seen by C) and if E and CS are not collocated, let P' be the ACCE protocol between between E and CS. Assume that P and P' are together mEA-secure, if E and CS are not collocated.*

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-T with DHE-active CS" or "LURK-T with DHE-passive CS", running at most t queries and creating at most q party instances per party, where $\mathscr{A}$'s advantage is written $\mathsf{Adv}^{\mathsf{EA}}_{\mathsf{LURKT}}(\mathscr{A})$.

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' (when E and CS are not collocated), $\mathscr{A}_4$ against the ACCE security of P', $\mathscr{A}_5$ against the existential unforgeability (EUF-CMA) of the signature used to generate* PSign *and $\mathscr{A}_6$ against the hash function H, or $\mathscr{A}_7$ 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:*

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

Proof. We prove the case where E and CS are not collocated, as that is harder – with an extra channel, thus extra steps. The other case reduces to this case trivially.

Our proof has the following hops:

Game $\mathbb{G}_0$: This game works as the EA-game recalled in Section 13.

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 CS.

In the EA definition, the only way the adversary can win if the party P_i is a CS 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 E. Since such an attacker must guess the identity of the CS that will maliciously accept, and E 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 E. If that is the case, then its partner party P_j can only be a CS. 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 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 CS and such that there exists some E 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

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

where Δ is obtained as per the below.

Let us first describe what it would mean to win this game. **Winning game $\mathbb{G}_3$:**

In this game, we will rely on the fact that the CS P_j, which is uncorrupted, provides distinct certificates per E party.

We first prove that, not only is E the real partner and a CS 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 sub-keys of π_j^n and the transcript of π_i^m.

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

To begin with, we focus on the client transcript of π_i^m. Assume that this transcript is linked to a certificate (and public key) Cert_{P_k,P_j}.

We first rule out the possibility that another party P_x as above is able to maliciously authenticate to P_i using that certificate. For this, we rely on the fact that the CS P_j, which is uncorrupted, provides distinct certificates per E. Then, we 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: $\mathsf{n_P} \cdot \mathsf{Adv}_{\mathsf{Sign}}^{\mathsf{Unforg}}(\mathscr{A}_5)$, accounting for getting which party the signature is generated for. Now, we 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. against the entity authentication of P and respectively $\mathscr{A}_2'$ against the ACCE-security of P'. Whenever $\mathscr{A}_{\mathbb{G}_3}$ queries information for C-E sessions, the queries made via $\mathscr{A}_3$ are with $\mathsf{flag}_i^m = 0$. Whenever $\mathscr{A}_{\mathbb{G}_3}$ queries E-CS information, the queries made via $\mathscr{A}_3$ are with $\mathsf{flag}_k^l = 1$. So, the probability $\mathbf{Pr}[\mathscr{A}_{\mathbb{G}_3} \text{ wins}]$ is increased by the factor $\mathsf{n_P}^3 \cdot \mathsf{Adv}_{P,P'}^{mEA}(\mathscr{A}_3)$. The two adversaries have a common database of responses to $\mathsf{RegParty}$ and $\mathsf{NewSession}$.

W.r.t. our current EA game, we also note that the EA definition further stipulates that no Reveal query can be made on the instances $\pi_i^m.\mathsf{InstSet}$ partnered with π_i^m.

W.r.t. the our current EA game and the mEA-game, the simulation for $\mathsf{RegParty}$, $\mathsf{NewSession}$, $\mathsf{Corrupt}$, Reveal clearly work with no issue, as in the 2(S)ACCE, TLS cases.

The difference (between the our 3-party EA setting and the 2-party mEA setting) occurs for the Send oracle, since in order to simulate correctly the record-layer transcript of E-CS session between π_k^ℓ and π_j^n. Here, on this E-S 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).

There is nothing elso to simulate, and the probability that $\mathscr{A}_{\mathbb{G}_3}$ win is increased is augmented by $\mathsf{n_P}^3 \cdot \mathsf{Adv}_{P'}^{\mathsf{2\text{-}ACCE}}(\mathscr{A}_2)$

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

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

□

14.2. Channel Security Proof

We now prove the 3(S)ACCE channel security for LURK-T in both its versions. The security game for 3(S)ACCE channel security is recalled in Section 13.

Theorem 2. *Let P be the unilaterally-authenticated TLS 1.3 handshake (as seen by C), and when E and CS are not collocated, let P' be the ACCE protocol between E and CS.*

Consider a (t,q)-adversary $\mathscr{A}$ against the SC*-security of the protocol "LURK-T with DHE-active CS" or in "LURK-T with DHE-passive CS" 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* $\mathsf{Adv}_{\mathsf{LURKT}}^{\mathsf{SC}}(\mathscr{A})$.

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 ACCE security of P' (considered below only when E and CS are not collocated) and $\mathscr{A}_4$ against the existential unforgeability (EUF-CMA) of the signature algorithm used to generate* PSign*, $\mathscr{A}_5$ against the channel security of P, each adversary running in time $t' \sim O(t)$ and instantiating at most $q' = q$ instances per party, $\mathscr{A}_6$ against the non-programmable PRF φ, such that*

And then,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{SC}}_{\mathsf{LURKT}}(\mathscr{A}) \leq\ & (2\mathsf{n_P}^2 + 2\mathsf{n_P}^3) \cdot \mathsf{Adv}^{\text{2-ACCE}}_{P'}(\mathscr{A}_2) \\ & +(\mathsf{n_P}^3 + \mathsf{n_P}^2)\mathsf{Adv}^{\text{2-SACCE}}_{P}(\mathscr{A}_1) \\ & +\mathsf{n_P}^3\mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_3) \\ & +\mathsf{n_P}^2\mathsf{Adv}^{\mathsf{SC-SACCE}}_{P}(\mathscr{A}_4) \\ & +\mathsf{n_P}^2\mathsf{Adv}^{\mathsf{AKE}}(\mathscr{A}_5) + \mathsf{n_P}^3 \cdot \mathsf{Adv}^{\mathsf{npPRF}}(\mathscr{A}_6).\end{aligned}$$

Proof. We prove the case where E and CS are not collocated, as that is harder with an extra channel, thus extra steps. The other case reduces to this case trivially.

Our proof has the following hops:

Game $\mathbb{G}_0$: This game works as the SC-game recounted in Appendix 13

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 CS and such that there exists some E 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:

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

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

14.3. Accountability Proof

We now prove the 3(S)ACCE accountability for "LURK-T with DHE-active CS". The security game for 3(S)ACCE accountability is recalled in Section 13.

Theorem 3. *Let P be the unilaterally-authenticated TLS 1.3 handshake (as seen by C), and when E and CS are not collocated, let P' be the ACCE protocol between E and CS.*

Consider a (t,q)-adversary $\mathscr{A}$ against the Acc*-security of "LURK-T with DHE-active CS" 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* $\mathsf{Adv}^{\mathsf{Acc}}_{DHE-active CS}(\mathscr{A})$. *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, and an adversary $\mathscr{A}_4$ against the existential unforgeability (EUF-CMA) of the signature algorithm used to generate* PSign*, such that:*
$\mathsf{Adv}^{\mathsf{Acc}}_{DHE-active CS}(\mathscr{A}) \leq \mathsf{n_P} \cdot \mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}(\mathscr{A}_4) + 2 \cdot \mathsf{n_P}^2 \cdot \mathsf{Adv}^{\mathsf{2\text{-}SACCE}}_{P}(\mathscr{A}_1)$

Proof. We prove the case where E and CS are not collocated, as that is harder with an extra channel, thus extra steps. The other case reduces to this case trivially.

Our proof has the following hops:

Game 0: This game works as the Acc- game recalled Appendix 13. Recall, 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* $P_j \in \mathscr{CS}$ 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 crypto-service 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 E 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, E 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.*, E talking to C. 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). This is equivalent to forging the signature PSign. Hence, a quantity of $\mathsf{n_P} \cdot \mathsf{Adv}^{\mathsf{Unforg}}_{\mathsf{Sign}}(\mathscr{A}_4)$ is added to the bound.

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 keys sent to E by that instance as the key of π_i^m. We also note that if the client finished message does not verify, then E 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 1.3 handshake run between the client and the uncorrupted crypto-service), in which the adversary (possibly a collusion of all the malicious E) 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 adds $\mathsf{n_P}^2 \cdot \mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1)$ to the final bound.

In total, the two case add to the final bound, i.e., $\mathsf{Adv}_{DHE-activeCS}^{\mathsf{Acc}}(\mathscr{A}) \leq \mathsf{n_P} \cdot \mathsf{Adv}_{\mathsf{Sign}}^{\mathsf{Unforg}}(\mathscr{A}_4) + 2 \cdot \mathsf{n_P}^2 \cdot \mathsf{Adv}_P^{\text{2-SACCE}}(\mathscr{A}_1)$.

□

14.4. Trusted Key-binding Proof

We now state and the prove trusted key-binding for "LURK-T with DHE-active *CS*".

Trusted Key-binding. In our *trusted key-binding* game the adversary may arbitrarily use all the 3(S)ACCE oracles as well as calls to *runtime attester* $\mathscr{B}$ to attest handshakes, in the case that it corrupted the Engine E. The full model contains a partnering that links the session of S to sessions of $\mathscr{B}$. Also, $\mathscr{B}$ issues a statement called **proof'** on attributes of parties S, attributes that we call **transcript'** (i.e., a minimal part transcript that the runtime attester attests of S's sessions).

The adversary finally halts 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 4 (Winning Conditions – Tkb)**.** *An adversary $\mathscr{A}$* trusted key-binding *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 $P_j \in \mathscr{C}\mathscr{S}$*;*

(b) There exists no instance $\pi_j^n \in P_j.\mathsf{Instances}$ such that $\pi_j^n.\mathsf{ck} = \pi_i^m.\mathsf{ck}$ and $\pi_j^n.\mathsf{ck}.transcript' \in \pi_l^h.proof$, for all instances π_l^h of $P_l \in \mathscr{B}s$

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

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

$$\mathsf{Adv}^{\mathsf{Tkb}}_{DHE-active CS}(\mathscr{A}) := \mathbb{P}[\mathscr{A} \text{ wins the Tkb game}],$$

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

In other words this is a tighter accountability game, in which condition (b) is stronger, and says that not only is there no CryptoService matching the view of the client but no instance of the CryptoService that would match that had a runtime-attester check its handshake.

Trusted Key-binding Proof.

We now prove our notion above of trusted key-binding for "LURK-T with DHE-active *CS*".

Theorem 4. *Let P be the unilaterally-authenticated TLS 1.3 handshake (as seen by C), and when E and CS are not collocated, let P' be the ACCE protocol between E and CS. Let the CryptoService $\mathscr{CS}$ parties run augmented with runtime attesters, identified as parties in $\mathscr{B}s$*

Consider a (t,q)-adversary $\mathscr{A}$ against the Tkb*-security of "LURK-T with DHE-active CS" 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* $\mathsf{Adv}^{\mathsf{Tkb}}_{DHE-active CS}(\mathscr{A})$*. 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, and an adversary $\mathscr{A}_8$ against the runtime-TEE security model, such that:*

$$\mathsf{Adv}^{\mathsf{Tkb}}_{DHE-active CS}(\mathscr{A}) \leq 2 \cdot \mathsf{n_P}^2 \cdot (\mathsf{Adv}^{\mathsf{2\text{-}SACCE}}_{P}(\mathscr{A}_1) + \mathsf{Adv}\ (\mathscr{A}_8))$$

The proof is, unsurprisingly, very similar to the accountability one above, with Game1 being finally reduced to triviality, due to the presence of the attester.

Proof. We prove the case where E and CS are not collocated, as that is harder with an extra channel, thus extra steps. The other case reduces to this case trivially.

Our proof has the following hops:

Game 0: This game works as the Tkb- game above.

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 crypto-service P_j such that π_i^m is the instance for which trusted key-binding 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 E 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, E 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^q is an instance of P_k, *i.e.*, E talking to C. In that case, we can construct a reduction from this case to the server-impersonation security of the protocol P' (recall that the honest crypto-service P_j cannot be corrupted). This is equivalent to forging the signature PSign. Now, this case reduces to the E also needing to provide the right handshake-part $transcript'$ to an attester-instance $\mathscr{B}$. And, this need does not hinge on forging hashes or colliding nonces, etc., it hinges[8] on uniqueness of runtime measurements in TEE and E being able to forge that, thus breaking the TEE-security model.

Or, there exist instances π_j^n, π_l^h, π_x^q 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 keys sent to E by that instance as the key of π_i^m. We also note that if the client finished message does not verify, then E 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 1.3 handshake run between the client and the uncorrupted crypto-service), in which the adversary (possibly a collusion of all the malicious E) 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 $\mathsf{Adv}_P^{\mathsf{2\text{-}SACCE}}(\mathscr{A}_1)$. There is an extra condition to be satisfied namely that the simulator would be able to simulate the CS run to an attester session $\mathscr{B}$; yet this is only possible if one defeats the TEE security model, as we explained in the case above.

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

$$\mathbf{Pr}[\mathscr{A}_{\mathbb{G}_0} \text{ wins}] \le 2 \cdot \mathsf{n_P}^2 \cdot (\mathsf{Adv}_P^{\mathsf{2\text{-}SACCE}}(\mathscr{A}_1) + \mathsf{Adv}\ (\mathscr{A}_8)).$$

□

15. Performance Comparison with HSM

Our prototype LURK-T on a standard desktop can perform better than mainstream HSMs like Luna SA 7000 (except for RSA-2048), and nShield Connect 6000+ for P-256 (as well as Luna SA 1700 and nShield Connect 500+, XC Base, 1500+ models).

[8] In practice, this is impossible because the TEE model assumes no corruption of the TEE, and no other execution of E would ever produce the measurements that instance of crypto-service P_j would produce.

For Ed448, RSA, P-384 the LURK-T overhead is around 15% of that of standard OpenSSl, meaning we achieve 85% of OpenSSL's maximum rate - only performing KEX. HSM comes with a number of disadvantages including cost, proprietary interface and performance issues. [63, 71] provide the following performance for different HSMs for a single signature operation. The comparison with LURK-T is indicative and needs to consider LURK-T considers a full TLS KEX, with 1) multiple interactions with the CS, 2) additional operations such as SHA256 for the nonce, ECDHE generation and computation of share secret, generations of secrets. To ease the comparison, though we considered the keyless configuration of the CS which is the closest one to an HSM – and does not involve the ECDHE part. Given the hardware involved, LURK-T is very competitive in terms of performance compared to an HSM. LURK-T is also much more secure - as it prevents signing oracle attacks.

While costs of these HSM are not public, the cost of the CPU we used is around 600 USD, making it hard for any HSM to compete. AWS Cloud HSM [28] also proposes the use of HSM around 1.45 USD an hour, which means the price of the CPU is reached after roughly 17 days.

		nShieldConnect						Luna	
	LURK-T	500+	XC Base	1500+	6000+	XC Mid	XC High	SA 1700	SA 7000
RSA-2048	1613	150	430	450	3000	3500	8600	1700	7000
RSA-4096	228	80	100	190	500	850	2025	50	160
P-256	4479	540	1210	1260	2400	7515	14400	490	1000

Table 6: LURK-T performance comparison versus nShield Connect and Luna HSM