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

Behnam Shobiri, Sajjad Pourali, Daniel Migault, Ioana Boureanu, Stere Preda, Mohammad Mannan and Amr Youssef, *Senior Member, IEEE*

***Abstract*—In many web applications, such as Content Delivery Networks (CDNs), TLS credentials are shared, e.g., between the websites TLS origin server and the CDN's edge servers, which can be distributed around the globe. To enhance the security and trust for TLS 1.3 in such scenarios, we propose LURK-T, a provably secure framework which allows for limited use of remote keys with added trust in TLS 1.3. We efficiently decouple 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. CS and E together provide the whole TLS-server functionality. A major benefit of our construction is that it is application agnostic; the LURK-T Crypto Service could be collocated with the LURK-T Engine, or it could run on different machines. 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 CDNs over TLS. To support such a generic decoupling, we provide a full Application Programming Interface (API) for LURK-T. To this end, we implement our LURK-T Crypto Service using Intel SGX and integrate it with OpenSSL. We also test LURK-T's efficiency and show that, from a TLS-client's perspective, HTTPS servers using LURK-T instead a traditional TLS-server have no noticeable overhead when serving files greater than 1MB. In addition, we provide cryptographic proofs and formal security verification using ProVerif.**

***Index Terms*—Internet security, Middleboxes, TLS**

I. 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 and require special attention.

For TLS servers managed "in situ", e.g., when the owner of the TLS server also owns the infrastructure and entirely manage the TLS servers, the authentication credentials must be protected for example against operational mistakes[1] as well as web server compromise such as Heartbleed.[2]

With 73% of the Internet traffic today being served by Content Delivery Networks (CDNs) [16], a common scenario is sharing the TLS credentials between the website's TLS server (i.e., the "origin") and the CDN's "edge servers", which can be distributed around the globe. Such sharing of long-term TLS credentials poses a grave risk, as the origin loses full ownership and control of their long-term private key [29].

The proposed setups for CDN over TLS alone vary vastly from splitting the TLS implementation [41], to leveraging Trusted Execution Environment (TEE) and either improving the performances of the enclaves for network applications [49], [22], [40] or improving a specific application running inside an enclave [3] – which ends into splitting the application between components running inside the TEE and outside the TEE. Thus, a generic treatment of securing and protecting the long-term credentials of the TLS server is essential, catering for as many distinct types of interactions 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 a component called *LURK-T Engine (E)*which securely queries this core from anywhere it may reside, and communicates with a classical TLS Client (C).

We are not the first to consider the decoupling of a TLS server and/or securing a modified version thereof. Current efforts can be divided into two types: (a) TEE-driven approach focusing on isolating and securing the server; (b) CDN-driven approach focusing on modifying the TLS server to fit different CDN setups. Each approach has its merits and shortcomings. Inspired by both these approaches, we propose a new solution, by decoupling the TLS-server in a way that results into acceptable, deployment-friendly performance. Now, we discuss the two main aspects of our design compared to existing work (details in Section II).

(a) TLS servers and CDNs. CDNs operate over TLS in a mechanism often broadly referred to as "TLS delegation". To enable such delegation in a provably secure way (as in e.g., [11]), or to support specific scenarios [4], major operational changes in TLS are required. Such changes either break security (see e.g., [41]), or render them completely incompatible with legacy clients (see e.g., [34]). Besides, the efficiency of delegation is usually not considered/discussed at length or is sacrificed in favor of enhanced security (see e.g., [7]).

(b) TLS servers and TEEs. To protect TLS credentials, NIST [5] recommends hardware-based 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, such TEE integration is not common for CDN scenarios. TEE-based academic proposals vary significantly where the full application is placed in a TEE [49], [22], [40], [48], or the full TLS is placed in a TEE [3] – both of which are explicitly mentioned as impractical by several standard bodies such as ETSI,[3] 3GPP-SA3,[4] and ENISA.[5] Some other proposals protect only the

Behnam Shobiri, Sajjad Pourali, Mohammad Mannan and Amr Youssef are with the Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, QC, Canada e-mail: behnam.shobiri@concordia.ca, s_poural@ciise.concordia.ca, m.mannan@concordia.ca, and youssef@ciise.concordia.ca}.Ioana Boureanu is with the Surrey Centre for Cyber Security and the Department of Computer Science, University of Surrey, Guildfordl, UK e-mail: i.boureanu@surrey.ac.uk. Daniel Migault and Stere Preda are with Ericsson Canada, e-mail:{daniel.migault,stere.preda}@ericsson.com

[1] https://lists.dns-oarc.net/pipermail/dns-operations/2020-May/020198.html

[2] https://heartbleed.com/

[3] https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/009/01.02.01_60/gr_NFV-SEC009v010201p.pdf

[4] https://www.3gpp.org/ftp/Specs/archive/33_series/33.848/33848-0c0.zip

[5] https://www.enisa.europa.eu/publications/nfv-security-in-5g-challenges-and-best-practices/@@download/fullReport

keys [22], [13], [47]. Indeed, deciding which part of the cryptographic side of TLS-server to include in a TEE, such as to yield added security without high performance penalty, appears to be non-trivial. **Our contributions** can be summarized as follows:

1. To enhance the security and trust for TLS 1.3 in applications where the TLS credentials are shared (e.g., in CDN applications), we propose *Limited Use of Remote Keys with Added Trust (LURK-T)*. To balance security and efficiency, LURK-T splits the 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 session resumption, Elliptic Curve Ephemeral Diffie Hellman ((EC)DHE) keys to ensure Perfect Forward Secrecy (PFS). 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). See Figure 1 in Section IV for an overview of LURK-T components.

2. We implement CS using Intel SGX and integrate it with OpenSSL, both for Ubuntu and Windows. The modularity of our design entails only localized changes to OpenSSL. To show the compatibility and portability of our implementation, we develop a Rust HTTPS server and link it to our modified OpenSSL.

3. 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 the worst case configuration, the client's overhead associated to LURK-T is negligible for files equal or greater than 1MB. The server's overhead is limited to the TLS handshake and we measured it between 1.2% and 33% which is far less than similar solutions (see Section II and Table V).

4. We present cryptographic proofs for LURK-T, in a cryptographic model for multi-party TLS [8], 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 [36], [6] of a pre-standard TLS 1.3 to a ProVerif model for the standard TLS 1.3 [38], and then proving TLS 1.3 security for LURK-T; thus, we show that LURK-T suffers no degradation in security compared to TLS 1.3, including attaining perfect forward secrecy. We achieve strong security guarantees (e.g., the accountability of [7]), as well as add a new property of trust which we call "trusted key-binding", achieved through the attestation of our TEE-based CS.

II. Related Work

LURK-T partitions TLS 1.3 into two independent micro services (E and CS) with CS hosted by a TEE. In this section, we summarize related work on partitioning applications, as well as as protocol extensions that support TLS delegation, and multi-party TLS.

A. TLS and TEE

Multiple frameworks are able to host unmodified binary code into a TEE enclave (see e.g., [31]). These frameworks rely on libOS (e.g., Graphene [45], SGX-LKL [35]), or musl-libc (e.g., SCONE [2]). However, this results in a large trusted code base (TCB) [44] with a vast number of Line of Code (LoC) prone to bugs [17] (and Iago attacks [15]), and with large overhead due to multiple ECALLs/OCALLs [42].

Partitioning applications is expected to address these drawbacks. Specific manual approaches have been proposed for TLS 1.2 as in [3]. A more generic approach, based on marking sensitive data in the source code for C/C++ applications has been proposed in *Glamdring* [30] and the execution of the resulting trusted part can be instrumented by sgx-perf [48]. Other proposals such as Montsalvat [50] partition Java code based on its byte-code. However, the coexistence of the trusted and untrusted part is handled via remote procedure call (RPC)-like mechanisms, exposing the interface to Iago-like attacks, while providing little assurance that data or states are not leaked. LURK-T defines standard interfaces in [32], thus protecting CS against Iago-like 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 following [8], which showed that the lack of such formal verification can hide the existence of vulnerabilities (e.g., in Keyless SSL [41] and mcTLS [34]).

Various efforts (e.g., [3], [47], [48], [22], [13], [40], [44]) were made to leverage TEE, and port TLS applications into SGX enclaves. All these proposals were focused on TLS 1.2, and generally they place the full TLS stack into the TEE (e.g., TaLoS [3] and sgx-perf [48]). STYX [47] 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 Intel Quick Assist Technology (QAT [43]). Also, in STYX, an SGX enclave attested by the content owner is used to provision the TLS private key to the QAT engine, which is natively interfaced with OpenSSL [24]. This design suffers from the fact that interactions between the QAT engine and untrusted application are not limited to TLS 1.3 specific operations. As detailed in [8] w.r.t. Keyless SSL [41], the use of such generic cryptographic operations may be exploited.

Conclave [22] takes a higher level approach by defining an architecture for securing a full service NGINX server, 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 from performance and operational perspectives. Security-wise, Conclave uses Graphene which is a large library (more than 77000 LoC) and has a high probability for vulnerabilities as shown in [17]. 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 Conclave, LURK-T provides anti-replay protection.

B. TLS Protocol Extensions

Similar to Keyless SSL, most previous works on TLS delegation (e.g., see [29], [41], [8], [47]) are not designed for TLS 1.3 and suffer from the TLS 1.2 limitations [29], [11]. Bhargavan et al. [8] provide delegation for Authenticated and Confidential Channel Establishment (ACCE) with TLS 1.3, yet there are two essential differences compared to our approach: ACCE is controlled by both ends (i.e., the client and the server), and it requires modifications to the TLS-record layer to achieve fine-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) [4] 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. In a DC deployment, LURK-T can enable the CDN to protect the CDN authentication credentials (or the CDN can use any other TEE-based alternatives to protect the

credentials). On the other hand, from the content owner perspective, LURK-T makes DC unnecessary as the content owner's authentication credentials can be used without being shared to the CDN. This could be useful to ensure that existing/legacy TLS clients can authenticate the server; note that DC deployments require support/control from both the client and server sides (supported by the Firefox browser since 2019, and used by Cloudflare and Facebook services).

Boureanu et al. [11] used a similar design to LURK-T but for TLS 1.2. Most differences between Boureanu et al. [11] and us stem from TLS 1.3 being different from TLS 1.2. LURK-T also offers several variants to interact with the CS in the TEE to balance reasonable security vs. efficiency, which also addresses Boureanu et al.'s latency issues. In addition, LURK-T leverages TEEs and provable security for further trust.

Various services (referred to as middleboxes) provided by CDNs can only function when they have access to plaintext data, such as IDS, IPS, WAF, and L7 load balancing [20]. In some previous proposals [22], [3], [13], 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 [34], but with significant overhead and heavy modifications to TLS 1.2 handshake and record-layer. It was also solved generically for any ACCE protocol, but again with significant overhead [8]. However, LURK-T solves this for TLS 1.3 without any modification to TLS 1.3 with an acceptable level of overhead (see Table V).

III. Design Goals and Threat Model

The main difference between LURK-T and the standard TLS is that LURK-T operates over 3 parties: C, E and CS. The E and CS implement the server S. CS handles the authentication credentials and derives the necessary TLS secrets for E which interacts with the C. Standard TLS instead operates over 2 parties: C and S.

A. Goals

The purpose of LURK-T is to ensure security properties of a TLS communication between C and E: providing authentication ensured by trustworthy credentials (private keys as well as PSK), and enabling PFS. In particular, LURK-T ensures that a TLS communication between C and E remains trustworthy even if E becomes compromised in the future as well as even if other C or E on another edge server is compromised. These properties are also provided when E and CS are operated "in situ" or by a CDN. To meet these properties the following goals are derived:

- The CS must provide read protection of the authentication credentials from a compromised E to prevent them from being used in future TLS sessions. With LURK-T, a compromised E in the "in situ" scenario or a network admin in the CDN scenario is not able to access the credentials via root access – including dumping the memory. This differs from the standard TLS 1.3 threat model where S must not be compromised, and anyone with privilege access to S can access the credentials. A direct consequence is that with LURK-T, once an attacker has compromised E, and later when E recovers from this compromise, that attacker is not be able to interfere with any future TLS session – including resumed sessions.
- The compromise of E, or CS being administered by a CDN, does not provide any advantage to an attacker (although CS is interfaced via LURK [32]) compared to the use of a regular C. This is achieved by strongly binding the interactions between E and CS to the TLS 1.3 exchange between C and E via the freshness mechanism (see IV-A) as well as enforcing CS to operate over full TLS exchanges as opposed to hash of such exchanges. This also provides a very efficient anti-replay protection for example when PFS is not properly enforced to meet a performance criterion and limit the generation of (EC)DHE keys (see [39] Section 7.4).
- CS must be able to impose PFS by enforcing CS to generate new and unique (EC)DHE keys for each TLS sessions. This differs from the standard TLS 1.3 model, where in the case of a CDN, PFS is expected to be enforced by the CDN (we avoid this trust in CDNs).

B. Adversary Capabilities

We assume CS is trustworthy. In LURK-T, CS runs inside a TEE whose threat model assumes that TEE and its interfaces cannot be corrupted [19]. CS is expected to be developed with formal verification. The current 3800 LOC makes such assumptions realistic.

The private key must be securely provisioned to CS. This may involve the key being generated and distributed from a TEE [21] or the enclave being provisioned securely [28]. The latter is expected to be achieved with TLS 1.3 being implemented in the CS. The attacker can control all Cs and Es, and interact either using TLS 1.3 or LURK. Note that, in a TLS session where the C or E is under the control of the attacker, all session secrets are exposed to the attacker and the TLS session is trivially decrypted (but the private keys remain protected under CS).

Regarding the network capabilities of the attacker, we also assume as per the Dolev-Yao's threat model, that all public channels are accessible to the attacker to read, replay, block and inject messages.

IV. LURK-T Design and Deployment Scenarios

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

A. LURK-T – Design

Components and the protocol. LURK-T involves the following entities: a TLS client C, a LURK-T TLS Engine E, and a LURK-T TLS Crypto Service CS; see Figure 1. The last two are either collocated, or there is a pre-established, mutually authenticated and encrypted channel between them. Such channel is expected to be implemented via a TLS library embedded into CS that terminates into the TEE to prevent the communication between E and CS being compromised by the node hosting CS. The key provisioning service is responsible for ensuring that the correct key is securely conveyed to the CS using a secure channel that terminates into the enclave. This can be achieved using solutions such as Blindfold [21].

The purpose of TLS is to authenticate and agree on sessions keys so that C and E can encrypt and exchange application data. The Key Schedule is responsible to generate the various secrets between C and S and includes, among others, the client/server handshake secrets (h_C, h_S) used to derive the keys that protect the TLS key exchange, the client/server application secrets (a_C, a_S) used to derive the keys that protect the application data, the session resumption secret (r) used to generate the PSK for authenticating C and S. The Key Schedule generates these secrets thanks to shared secrets such as PSK or (EC)DHE shared secret KE as well as the TLS handshake context H_{ctx}. The ClientHello.random N_C and ServerHello.random N_S provide some randomness to generate these secrets.

TLS supports three basic key exchange modes: (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves), PSK-only and PSK with (EC)DHE. The TLS (EC)DHE mode corresponds

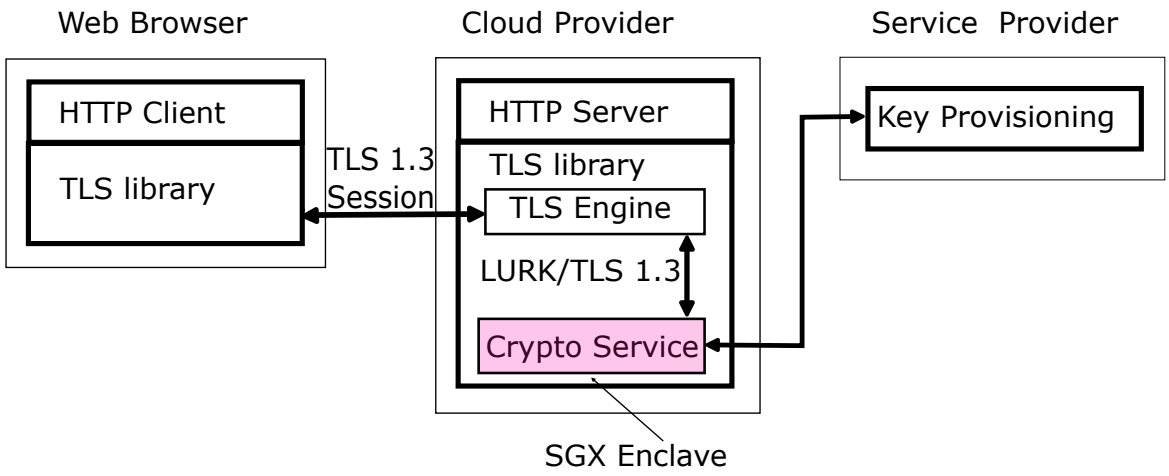


Fig. 1: LURK-T entities: a TLS client C (a regular web browser), a LURK-T TLS Engine E, and a LURK-T TLS Crypto Service CS (both part of a third-party hosting provider), and a key provisioning server (under the content owner's control).

to certificate-based authentication with S being authenticated by C by proving the ownership of a private key sk via a signature over the TLS exchange context H_{ctx} designated as $\mathsf{PSign} = \mathsf{Sign}_{sk}(\mathsf{H}_{ctx})$. sk constitutes the authentication credential. (EC)DHE ensures perfect forward secrecy, with C and S generating their respective private keys u and v, exchanging their respective corresponding public key KE_C=g^u and KE_S=g^v to generate a common (EC)DHE shared secret $\mathsf{KE} = g^{uv}$. The TLS PSK with (EC)DHE mode is based on a PSK shared between C and S as well as (EC)DHE, while the TLS PSK-only mode does not provide perfect forward secrecy. In both cases, PSK is the authentication credential. The TLS (EC)DHE mode is commonly used on the web together with the TLS PSK in the (EC)DHE mode to resume TLS sessions.

LURK-T always assumes that the authentication credentials sk or PSK are handled and hosted in CS. As a result, in the TLS (EC)DHE mode, PSign is generated by CS and in the TLS PSK modes the Key Schedule is performed by CS. On the other hand, LURK-T enables the private (EC)DHE key to be generated either by CS or E which leads to the respective variants *"LURK-T with DHE-active CS"* and *"LURK-T with DHE-passive CS"* illustrated in Figure 2a and Figure 2b. The difference is highlighted in red. "LURK-T with DHE-active CS" provides higher (compared to "LURK-T with DHE-passive CS") assurance on perfect forward secrecy (where TEE both protects the (EC)DHE key and attests the key is not reused), and on resumed TLS sessions (where TEE protects the PSK). 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 CS) in order to generate the PSign TLS message.

Note that Figure 2a and Figure 2b depict LURK-T instantiated with TLS 1.3 in (EC)DHE mode. For PSK with (EC)DHE, the only difference stems from the key-derivation in TLS. We now describe in more detail the two main variants of LURK-T, each of these two modes. Description of the PSK-only mode is omitted as PSK-only can be easily derived from the PSK with (EC)DHE mode and this mode is rarely used in the web context, with even discussions at the IETF to deprecate that mode [1].

"LURK-T with DHE-active CS" – TLS (EC)DHE mode. Following Figure 2a, C initiates the TLS key exchange with E by sending a ClientHello message which contains the random N_C as well as the (EC)DHE public key.

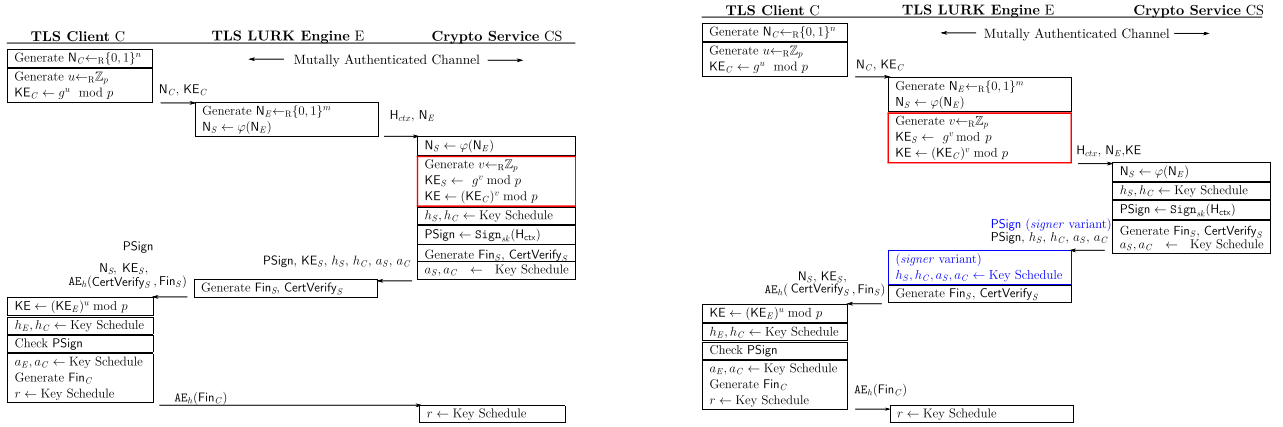
Upon receiving the ClientHello, E applies the freshness mechanism detailed in Section IV-A to protect against replay and signing oracle attacks and provides the necessary handshake context to CS to perform the Key Schedule and generate the signature. E generates a nonce N_E and applies a pseudorandom 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. Then, E sends to CS the whole of its view of the handshake H_{ctx} (including N_C and KE_C), the bitstring N_E.

CS generates N_S from N_E similarly to E. As in "LURK-T with DHE-active CS", CS generates the private (EC)DHE secret key v and KE_S and KE. KE is used together with H_{ctx} by the Key Schedule to generate the handshake secrets (h_C, h_S). CS generates the signature PSign. CS then generates the remaining handshake messages to update H_{ctx} and have sufficient context to generate the application secrets (a_C, a_S). The formed messages are CertificateVerify (CertificateVerify), which contains the signature, as well as the server Finished message (Fin_E) which is a hash MAC of H_{ctx}. Generating these message avoid an additional round trip between E and CS. CS then provides E the signature PSign, the (EC)DHE public key KE_S and handshake and application secrets.

Upon receiving KE_S, E generates and sends the ServerHello message with KE_S as well as the previously computed random N_S to the TLS client C. E generates the CertificateVerify (CertificateVerify) and server Finished message (Fin_E) and encrypts them with the session keys generated with the handshake secrets ($\mathtt{AE}_h$).

C performs its Key Scheduler (similarly to S), checks the signature, generates the client Finished message (Fin_C) and encrypts it ($\mathtt{AE}_h$) with session keys derived from the handshake secrets (h_C, h_S) before finally deriving the session resumption secret. Upon receiving Fin_C, E forwards it to CS so that CS can generate the session resumption secret r and the PSK, which will be used later during the session resumption.

"LURK-T with DHE-passive CS" – TLS (EC)DHE mode. In this variant, CS does not generate the (EC)DHE private key, which is instead generated by E (see Figure 2b). E generates N_E exactly as in the "LURK-T with DHE-active CS" variant. Then, E generates the (EC)DHE private key v and associated public key KE_S, g^v, and computes KE, g^{uv}, which is provided to CS, alongside the H_{ctx} and



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

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

Fig. 2: The two variants of LURK-T instantiated with TLS 1.3 in (EC)DHE Mode

N_E. *CS* then computes N_S as in "LURK-T with DHE-active *CS*", initiates the Key Scheduler with H_{ctx} and KE as inputs, computes PSign and optionally the handshake and application secrets h_S, h_C, a_S, a_C – as these later secrets may also be generated by E. These two sub-variants are represented in blue in Figure 2b, and are designated respectively as "keyless" or "normal". The rest continues as in the "LURK-T with DHE-active *CS*" variant. Note that with "LURK-T with DHE-passive *CS*", when session resumption is enabled, the resumption secret r is generated by E (not by *CS*, and thus *CS* is unable to guarantee its confidentiality).

LURK-T variants with TLS PSK with (EC)DHE mode. The main difference between the PSK with (EC)DHE mode and the (EC)DHE mode is that the former is used for session resumption. LURK-T in PSK with (EC)DHE vs. (EC)DHE mode varies in as much as TLS 1.3 varies across these two modes. LURK-T in PSK with (EC)DHE 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 (EC)DHE mode, but without PSign being generated.

Notes on 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 derive N_E due to the non-invertible property of φ. If later on, $\mathscr{A}$ corrupts E, $\mathscr{A}$ 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.

B. LURK-T - Use Cases and Deployment Scenarios

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. The CS Manager is the entity responsible to administrate and provision *CS*. Unless the private keys are generated inside the enclave, the CS is responsible to provision the *CS* with the secret key. Securely provisioning the enclave can be achieved by combining attestation and terminating the communication within the enclave. The enclave implementation must be verified by the CS manager (requiring *CS* code to be open-sourced). It also likely requires a TLS library being embedded into the enclave. Similarly, as only the TLS library is impacted, LURK-T enables the CDN to continue providing added services, and as such, keeps TLS a multi-party TLS.

Deployments driven by CDN providers. Figure 3a shows the case where LURK-T is deployed as a substitute of TLS 1.3 libraries. In this case, the server-side 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 3b shows the case of a more centralized infrastructure, with just one *CS* with an SGX enclave communicating securely with multiple Es. 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 seems appropriate in combination with TLS-RA [28].

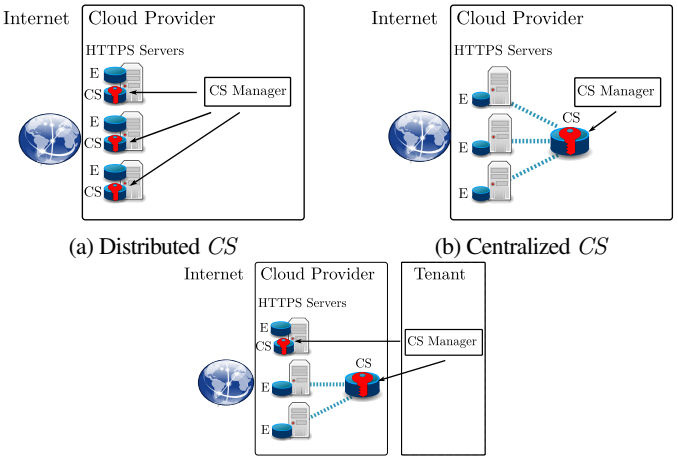
Deployments driven by content owners. Figure 3c shows the case where *CS* is provisioned by a CDN tenant, such as a content owner. Therein, *CS* is likely to be implemented by a third party (*CS* developer), trusted by the content owner and the cloud provider – e.g., with open source code. The tenant will need to perform an attestation of the *CS*, e.g., using Intel IAS [26]; 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 RA-TLS [28].

Finally, note that from a tenant's perspective there is a little difference between instantiating a centralized *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.

V. System Implementation

In this section, we describe our implementation[6] of *CS* and E based on OpenSSL. *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 [49].

[6] Available from https://github.com/lurk-t/



(a) Distributed *CS* (b) Centralized *CS*

(c) Tenant-controlled *CS*

Fig. 3: CS deployment use cases

We also balance the compliance to the specification of the LURK extension for TLS 1.3 [32] 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 introducing a maximum of 2 additional ECALLs compared to the LURK specification [32]. Our *CS* implementation contains 33 files with 3867 LoC.

A. Crypto Service (CS)

We implemented *CS* in an SGX enclave based on Intel SDK version 2.13. We had several options regarding the cryptographic library. While some cryptographic libraries support terminating the TLS connection inside SGX, we did not use them since they are either not maintained [3], or not fully compatible with OpenSSL.[7] We chose the actively maintained Intel SGX-SSL [25] 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).

1) CS in TLS (EC)DHE mode: *CS* is responsible for generating the different parts of the handshake such as the signature, and optionally — depending whether *CS* operates in the "LURK-T with DHE-active *CS*" or "LURK-T with DHE-passive *CS*" variant — (EC)DHE keys and secrets as detailed in Section IV-A. Our implementation supports all these variants as depicted in Figure 4 which details the exchanges between E and *CS*. While our design defines a single SInitCertificateVerify exchange [32], our implementation, when necessary (depending on the *CS* configuration), repeats up to 3 times that exchange in order to retrieve different pieces of information (depending on the *CS* configuration, see Figure 4). Table I shows the supported *CS* configurations, and for each one, which entity (E or *CS*) generates the (EC)DHE or the secrets h, a and r. Binder keys and signature are always generated by *CS* in their respective (EC)DHE or PSK with (EC)DHE modes.

Generating (EC)DHE. In the "LURK-T with DHE-active *CS*" variant, *CS* generates the (EC)DHE private key for S. E retrieves S's (EC)DHE public key with an additional SInitCertificateVerify1 exchange (see Figure 4). *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*. This is implemented with our `get_ecdhe` function which represents an additional ECALL compared to the LURK specification.

Generating h and a. When *CS* is configured to generate h, E performs an additional SInitCertificateVerify2 exchange to retrieve handshake secrets h_C and h_S (see Figure 4): our function `get_hand_secret` takes the ClientHello to ServerHello messages as inputs and returns h. This represents an additional ECALL compared to the LURK specification. When *CS* is configured to generate a, both a_C and a_S are generated together with the signature in our `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 specification with a single ECALL.

Session resumption. When session resumption is enabled, a new session 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

[7] https://www.wolfssl.com/wolfssl-with-intel-sgx

CS config (Cert)	$CS^{dhe,r}_{cert}$	CS^{dhe}_{cert}	CS_{cert}	$CS^{keyless}_{cert}$
(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)	$CS^{dhe,r}_{psk}$	CS^{dhe}_{psk}	CS^{r}_{psk}	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 I: *CS* configurations indicating where (EC)DHE or secrets are generated (when generated) and associated number of ECALLs by our implementation

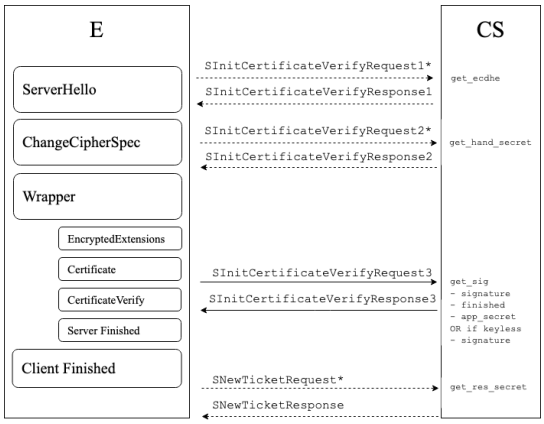


Fig. 4: Messages between E and *CS* for the (EC)DHE mode. * designates an optional exchange depending on the *CS* configuration

in a resumed session to the PSK generated in a previous TLS session. To fully reuse OpenSSL ticket management functions, the PSK ID is stored where OpenSSL used to store the clear text PSK.

2) CS in TLS PSK with (EC)DHE mode: During a session resumption, our implementation blocks OpenSSL from accessing 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 V-A1, 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`).

B. TLS Engine (E)

E, which 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). The communication between E and CS, is 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 N_E (before applying the freshness function) – see Section IV-A. However, by default, OpenSSL prevents the access to the TLS messages as it continuously hashes the 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.

VI. Performance Evaluation

A. 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.[8] Δ_{KEX} expresses the relative difference in terms of KEX/s between LURK-T TLS and the native OpenSSL TLS. In particular, Δ_{KEX} is expressed as a percentage for a given configuration $conf$ which represents the TLS cipher suites (see Table II) and the tasks performed by CS (see Table I).

[8]https://jbp.io/2019/07/02/rustls-vs-openssl-handshake-performance.html

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

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 10,000 handshakes.

Notation	Description	KEX/s
RSA-2048	(prime256v1, RSA-2048)	1715
RSA-3072	(prime256v1, RSA-3072)	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 II: Native OpenSSL TLS key exchange (KEX) performance for different cipher suites

TLS cipher suites configuration. We base our selection of cipher suites in Table II, on Mozilla's *modern compatibility* configuration which recommends ECDSA (P-256) or RSA-2048 combined with X25519, prime256v1, secp384r1.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 (EC)DHE mode (expressed as $cert$), session resumption may be enabled (expressed as r), so future handshakes may use the PSK with (EC)DHE mode. To remain coherent across sessions in terms of PFS, we only considered the PSK with (EC)DHE mode (expressed as psk). As mentioned in Section IV-A, 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., the CS generates the (EC)DHE private key). This is expressed with the following configuration $CS^{dhe,r}_{cert}$. Of course, without session resumption, "LURK-T with DHE-active CS" or "LURK-T with DHE-passive CS" variants are valid configuration expressed as CS^{dhe}_{cert}, CS_{cert} or $CS^{keyless}_{cert}$ (when only the signature PSign is generated, see Section IV-A). In the PSK with (EC)DHE mode, and unlike the (EC)DHE mode, session resumption may be enabled with both "LURK-T with DHE-active CS" or "LURK-T with DHE-passive CS" variants, and Table I summarizes the meaningful CS configurations with the associated number of ECALLs.

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

(EC)DHE mode. Figure 5 depicts Δ_{KEX} as a function of the number of ECALLs which characterizes CS configuration (see Table I). As shown in Figure 5, ECALLs do not equally affect all cipher suites and Δ_{KEX} does not linearly increase with the number of ECALLs. However, as per Table II, cipher suites that require more resources (RSA-3072, 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-3072, 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%

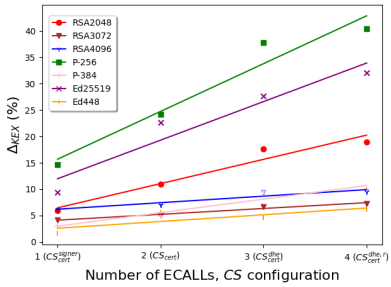


Fig. 5: KEX LURK-T TLS relative overhead over OpenSSL (Δ_{KEX}) in (EC)DHE mode. Measured values are linked using a linear regression.

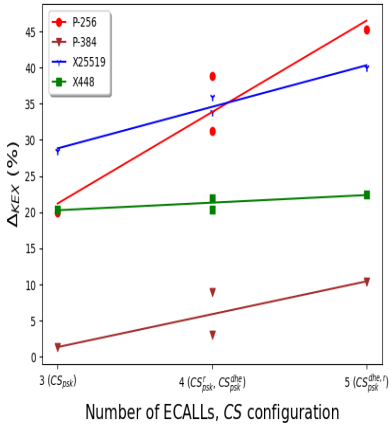


Fig. 6: KEX LURK-T TLS relative overhead over OpenSSL (Δ_{KEX}) in PSK with (EC)DHE mode.

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 apparently acceptable overhead (17% for P-384, 7.6% for RSA-2048, less than 4.3% for the others).

PSK with (EC)DHE mode. Similar to the (EC)DHE mode, Figure 6 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 (EC)DHE and PSK with (EC)DHE modes). Firstly, we can reduce the number of ECALLs, which may incur major modifications to the OpenSSL architecture, and may affect the case of Encrypted Extension (see Section V). 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 multiple operations can be completed within the allocated number of cycles, minimizing the number of unused cycles. This is likely to benefit Ed25519 or P-256.

C. SGX Vulnerabilities Mitigation Overhead

In this section, we discuss the overhead associated to the available mitigations – micro code or SDK [23] – of SGX vulnerabilities for our CPU. The discussion in Section VI-B considers 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 VI-B– we also anticipate new vulnerabilities to be disclosed and their mitigation will come with an additional overhead for the cloud provider. Note that previous proposals did not measure performance with these added security measures (which incur significant performance overhead).

	$SGX^{SRDBS,cf}$	$SGX^{SRDBS,ld}$	$SGX^{default}$
RSA-2048	1162	132	1715
RSA-3072	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 III: SGX performances (KEX/s) with SRDBS and LVI mitigation enabled versus default SGX for CS configured with $CS^{dhe,r}_{cert}$.

According to Intel [23], our CPU remains vulnerable to Special Register Buffer Data Sampling (SRBDS) [27] and CrossTalk attack [37] for which Intel provides a microcode update. Similarly, our CPU is vulnerable to Load Value Injection (LVI) [14] which we respectively mitigate both via the SDK or via the SGX-SSL `cve_2020_0551_load` (*ld*) or `cve_2020_0551_cf` (*cf*) [25].

Similar to Section VI-A, the performance is measured in terms of the number of KEX/s. In our case, the overhead of the microcodes – SRBDS – is negligible while the one of the SDK and SGX-SSL – for LVI – is not. Table III summarizes our measurements for each cipher suite. From the table, it is clear that for a given SGX configuration, the overhead increases with the number of operations performed by CS. However, for a given cipher suite, we could not correlate the number of operations to the expected overhead.

D. 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). Similar to Δ_{KEX} in Section VI-A, we measure Δ_{HTTPS} (see Table IV), 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 modified the benchmark tool wrk[9] to force select TLS 1.3 as well as to be able to specify a specific cipher suite. The HTTPS server and benchmark tools are published as open source.[10]

We measure the number of HTTPS requests per second performed by wrk with 10 parallel connections to introduce some concurrency similarly to [47] – 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, and thus, reducing S's latency (by avoiding reading from the hard drive). We limit CS's configuration to the most secure configuration which has the highest overhead ($CS^{dhe,r}_{cert}$).

[9] https://github.com/wg/wrk
[10] 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 HTTP request/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-3072	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 IV: HTTPS download/s LURK-T TLS relative overhead over OpenSSL (Δ_{HTTPS}) in (EC)DHE mode and "LURK-T with DHE-active CS" ($CS^{dhe,r}_{cert}$).

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 in Table IV show that even with $CS^{dhe,r}_{cert}$, the overhead is always negligible when downloading 1MB (or larger) files. In other words, for such files, 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 VI-B, P-384 and Ed448 seem to provide better performances compared to other cipher suites. Our reported measurements are valid from both the client and server perspectives. Note that, resource wise, S's LURK-T overhead is the one reported in Section VI-B.

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.

E. SGX Memory Usage

The memory that our design needs depends on the chosen configuration. Without session resumption, our implementation uses at most about 25 KB stack and 8 KB heap memory. 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 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, the memory usage will possibly reduce in practice (e.g., due to the optimization in the release mode).

F. LURK-T TLS Overhead for HTTPS Over Other Proposals

In this section we briefly discuss the LURK-T TLS overhead with the one of other solutions of Section II, reported in Table V. 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 versions of OpenSSL and libreSSL - though derived from OpenSSL).

	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 V: 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.

LURK-T is the only design applicable to TLS 1.3 with the lowest overhead in terms of KEX compared to TLS 1.2 and fewest LOC in TEE. As mentioned in Section II, specific approaches (LURK-T and Talos) seem to provide a lower overhead over generic frameworks (Graphene – see Conclave keyless). On the other hand, the large overhead measured for Panoply, Graphene, Talos and Conclave-crypt can be attributed to the resources they require for the protection of the TLS application data.

In terms of LOC, LURK-T and Talos limit the potential vulnerability with fewer lines of code and make these solutions more likely to be deployed by cloud providers as less TCB is required. This results both in limiting the necessary memory resources (limited to 90MB) as well as increasing the ability to share the other part of the untrusted library between containers and other applications.

VII. Formal Security Proofs and 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 bit strings, attackers to be probabilistic polynomial-time algorithms who attempt to subvert cryptographic primitives, and attacks to have a probabilistic dimension of the security parameters. Computational analysis is proved generally "by hand" in a game-based cryptographic model and is appropriated to verify arbitrary corruption and cryptographic AKE (authentication key exchange). 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 applied over interleaved protocol executions - assuming cryptography cannot be broken. Symbolic analysis is tool-assisted, automated, in a protocol-semantics and is appropriated to prove some properties such as PFS for example.

We use both the computational analysis: namely, we extend the (S)ACCE model [8] for multi-party AKEs, and we extend the symbolic analysis of a TLS 1.3 draft in ProVerif [6]. We extend the computational analysis to work for the actual current TLS 1.3 and TEEs, as well as applied it to LURK-T. We also extend the symbolic verification to work for TLS 1.3 draft 20 in [6] (which does not consider some AEAD encrypted payloads during the handshake); then, we apply it to LURK-T. Importantly, we could forego the computational analysis if there was a computational-soundness result [18], but this does not exist for TLS, let alone for multi-party TLS.

As per Section IV, 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 VII-A1. On top of the existing 3(S)ACCE [8] 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*.

In Section VII-A2, 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 VII-A1); this is a stronger form of accountability (than without the runtime-attested handshake-context mode), hinging on TEEs.

In Section VII-B, 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.

A. Computational Analysis

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 [8].

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 attester party, and call this requirement *trusted key-binding*.

LURK-T with runtime-attested handshake-context. In this sub-variant, a *runtime* TEE system is called to yield a separate "quote" 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. Then, we encrypt the buffer containing the operations on the CS and its arguments (it will just contain H_{ctx}), 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 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. That is, if presented with this handshake context by the attester again, then CS can recompute these keys ck and produce the same PSign, h, a, etc sent to E in the handshake where the keys (ck) were used.

Note that the attester gets only necessary parameters from the handshake. 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 in an enclave.

2) Cryptogtaphic proofs: W.r.t. the properties recalled/given above, we now state our cryptographic guarantees.

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 (the one between C and E, and the one between E and CS) ensure 3(S)ACCE mixed entity authentication [8] 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.*

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 [8] 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 [12], 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.*

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 [12], then "LURK-T with DHE-active CS" attains accountability in the 3(S)ACCE model.*

Accountability requires that CS always be able to compute all keys and sub-keys of the session established between the client and E. So, accountability is incompatible when session-resumption is done 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 [12] and if the TEE allows for runtime remote attestation, then "LURK-T with DHE-active CS" attains trusted key-binding.*

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.

B. Symbolic Verification

We perform a symbolic verification using ProVerif [9] to show that the LURK-T protocol, from a symbolic verification perspective, attains the same security properties as TLS 1.3, along with additional properties as described below. In this section, we focus on the verification of "LURK-T with DHE-active CS". We first 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 VII-A2.

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 18) 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 available at: https://github.com/lurk-t/proverif.

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 [36], [6]. The latest available version of this specification encoded draft 20 of 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.

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 an E process. In each, we encoded "LURK-T with DHE-active CS" and, specifically, also the case in which the CS and E are not collocated. In this case, we modelled a secure channel between the CS and 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 (including modelling weak subgroups) 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.

As shown in Fig. 7, our added query captures the execution of the following sequence of events: 1) TLS13_sent_cr_sr_to_CS denoting that E contacted the CS with clinets handshake details; 2) CS_sent_CV denoting that the CryptoService sent a signed share to E; 3) TLS13_recvd_CV denoting that E got from the said share signed from the CryptoService; 4) PreServerFinished denoting that E acted as a TLS server and reached the point of sending out a DH share to the client; 5) ClientFinished denoting that C finished a handshake.

```
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)).
```

Fig. 7: Agreement query between C, E, CS

By considering the introduced parameters in these events, one can observe that the data is bound among all such events during any given execution. These events are required to be injective, implying a one-to-one mapping in occurrences between them. Therefore, not only must this sequential events and data agreement hold for every LURK-T execution, but each CS execution will also uniquely correspond to a single E execution and a single C execution, through a distinct set of matching handshake data. This security agreement goal is demonstrated to persist, except in the cases of compromised CS, or compromised PSK, or due to the use of a weak DH subgroup, or a weak hash function. Such exceptions are comprehensively addressed by the list of disjunct terms appended at the end of the query.

3) Experimental setup: We conducted our ProVerif verification in two settings: (a) using an Ubuntu 20.04 Focal VM with a V100 GPU and 128 GB RAM on KVM; (b) 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 (a) is evidently very suitable for the development phase of our ProVerif models and for fast verification, we consider that setting (b) is more plausible to be used for reproducing our proofs. In setting (b), without the option to generate the attack graphs, analyzing all 29 queries automatically (the 24 queries inherited from [36] 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 7 requires 1.5 hours to be proved (true).

VIII. Potential Vulnerabilities

Although LURK-T protocol's security goals are formally verified, a LURK-T deployment may still face practical security challenges. For example, several new SGX attacks have been reported almost every year for the past several years—see e.g., the recent Downfall [33] and ÆPIC attacks [10]; a survey of attacks, CPU update delays, and exploits for commercial SGX applications are provided by SGX.fail [46]; and a comprehensive list[11] is also maintained by Intel. A LURK-T deployment must remain up-to-date with all pertinent fixes (SDK and microcode, if available). Missing any such updates, or attacks with no available mitigations may make any SGX deployment vulnerable, including LURK-T. Possible vulnerabilities may also be introduced in LURK-T's implementation. Our code is open-sourced, and relatively small in size (under 4K LOC), and thus the possibility of such vulnerabilities is relatively low. The private keys that LURK-T aim to protect, can still be leaked through content owner's negligence; e.g., if keys are provisioned without attesting a CS implementation (solutions such as Blindfold [21] can be used for secure key provisioning).

[11] https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

IX. Conclusions

We introduced LURK-T – a provably secure and efficient extension of TLS 1.3, and of the generic LURK framework.We designed LURK-T with a TLS server decoupled into a *LURK-T TLS Engine* and a *LURK-T Crypto Service* and split the TLS handshake across the two modules; the Crypto Service is executed inside a TEE and it accepts very specific and limited requests. We offered several modular variants of LURK-T, balancing security and efficiency. In addition, we implemented the Crypto Service using Intel SGX, and integrated our implementation to OpenSSL with minor changes. Finally, our experimental results looked at LURK-T's overheads compared to TLS 1.3 handshakes and demonstrated that it provides competitive efficiency.

References

[1] RFC 9257. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9257.txt

[2] 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 *USENIX OSDI*, 2016, pp. 689–703.

[3] P.-L. Aublin, F. Kelbert, D. OKeeffe, 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.

[4] 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

[5] 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

[6] 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.

[7] 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.

[8] K. Bhargavan, I. Boureanu, P. Fouque, C. Onete, and B. Richard, "Content delivery over TLS: a cryptographic analysis of keyless SSL," in *IEEE EuroS&P*, 2017, pp. 1–16.

[9] 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.

[10] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture," in *Usenix Security Symposium*, Boston, MA, USA, Aug. 2022.

[11] I. Boureanu, D. Migault, S. Preda, H. A. Alamedine, S. Mishra, F. Fieau, and M. Mannan, "LURK: server-controlled TLS delegation," in *IEEE TrustCom*, 2020, pp. 182–193.

[12] 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 *LATINCRYPT*, vol. 7533, 2012, pp. 100–120.

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

[14] 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*. IEEE, 2020, pp. 54–72.

[15] S. Checkoway and H. Shacham, "Iago attacks: why the system call API is a bad untrusted RPC interface," in *ASPLOS*. ACM, 2013, pp. 253–264.

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

[17] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *USENIX Security Symposium*, 2020, pp. 841–858.

[18] 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

[19] 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

[20] 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.

[21] H. Galal, M. Mannan, and A. Youssef, "Blindfold: Keeping private keys in PKIs and CDNs out of sight," *Computers & Security*, vol. 118, Jul. 2022.

[22] S. Herwig, C. Garman, and D. Levin, "Achieving keyless cdns with conclaves," in *USENIX Security Symposium*, 2020, pp. 735–751.

[23] "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

[24] Intel Corporation, "Intel QuickAssist Technology (Intel QAT) and OpenSSL-1.1.0: Performance," 2018. [Online]. Available: https://01.org/sites/default/files/downloads/intelr-quickassist-technology/337003-001-intelquickassisttechnologyandopenssl-110.pdf

[25] ——, "intel/intel-sgx-ssl," 2021. [Online]. Available: https://github.com/intel/intel-sgx-ssl

[26] 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

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

[28] 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

[29] 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*, 2014, pp. 67–82.

[30] 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*, 2017, pp. 285–298.

[31] 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.

[32] D. Migault, "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

[33] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *Usenix Security Symposium*, Anaheim, CA, USA, Aug. 2023.

[34] 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 *ACM SIGCOMM*, 2015, pp. 199–212.

[35] 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.

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

[37] 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

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

[39] Y. Sheffer, P. Saint-Andre, and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 9325, Nov. 2022. [Online]. Available: https://www.rfc-editor.org/info/RFC9325

[40] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves," in *NDSS*, 2017.

[41] D. Stebila and N. Sullivan, "An analysis of TLS handshake proxying," in *IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland*, 2015, pp. 279–286.

[42] 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.

[43] 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

[44] 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 *ACM CODASPY*, 2019, pp. 255–266.

[45] 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*, 2017, pp. 645–658.

[46] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom, "SoK: SGX.Fail: How stuff get eXposed," https://sgx.fail, 2022.

[47] 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 *ACM SoCC*, 2017, pp. 201–213.

[48] N. Weichbrodt, P. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel SGX enclaves," in *ACM/IFIP Middleware*, 2018, pp. 201–213.
[49] 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*, 2017, pp. 81–93.
[50] 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 *ACM/IFIP Middleware*, 2021, pp. 352–364.

Behnam Shobiri is a security researcher at Tigera wher he is researching Cloud and Kubernetes security. Prior to that, he was a master's student at Concordia University and worked on TLS and CDN security. He got his bachelor's degree from the Ferdowsi University of Mashhad in the field of computer engineering.

Sajjad Pourali is currently pursuing his Ph.D. degree in Information and Systems Engineering at Concordia University. His research interests include internet, application and system security and privacy.

Daniel Migault is an expert in the Ericsson cybersecurity team and is actively involved in standardizing security protocols at the IETF.

Ioana Boureanu is a Professor in Secure Systems at University of Surrey. She is the deputy director of Surrey Centre for Cyber Security, the co-director of University of Surrey Gold-level ACE-CSE, as well as Director of our GCHQ-accredited Information Security MSc.

Stere Preda received his PhD in Computer Science from TELECOM Bretagne, France. He is currently a senior researcher with expertise in cybersecurity at Ericsson. He has been an an active contributor to ETSI NFV security standardization.

Mohammad Mannan in an associate professor at the Concordia Institute for Information Systems Engineering, Concordia University. His research interests lie in the area of Internet and systems security. Dr. Mannan is involved with several well-known conferences (e.g., USENIX Security, ACM CCS), and journals (e.g., IEEE TIFS and TDSC).

Amr Youssef received the B.Sc. and M.Sc. degrees from Cairo University, Cairo, Egypt, in 1990 and 1993 respectively, and the Ph.D. degree from Queens University, Kingston, ON., Canada, in 1997. Dr. Youssef is currently a professor at the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Canada. His research interests include cryptology, security and privacy.