TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves

Pierre-Louis Aublin†, Florian Kelbert†, Dan O'Keeffe†, Divya Muthukumaran†, Christian Priebe†,
Joshua Lind†, Robert Krahn‡, Christof Fetzer‡, David Eyers¶, Peter Pietzuch†
†Imperial College London ‡TU Dresden ¶University of Otago

ABSTRACT

We introduce *TaLoS*[1], a drop-in replacement for existing transport layer security (TLS) libraries that protects itself from a malicious environment by running inside an Intel SGX trusted execution environment. By minimising the amount of enclave transitions and reducing the overhead of the remaining enclave transitions, TaLoS imposes an overhead of no more than 31% in our evaluation with the Apache web server and the Squid proxy.

1. INTRODUCTION

TaLoS is a TLS [4] library that securely terminates TLS connections by maintaining security sensitive code and data inside a hardware-protected Intel SGX execution environment [5]. In particular, TaLoS protects private keys and session keys from a malicious environment.

TaLoS compiles to a shared library and exposes the same API as OpenSSL [10] and LibreSSL [13]. As a result, TaLoS acts as a drop-in replacement for existing TLS libraries and can be used transparently by a wide range of existing applications, including the Apache [1] and Nginx [11] web servers, the Squid [12] proxy, and the JabberD [8] XMPP server. Applications merely require linking against TaLoS rather than OpenSSL or LibreSSL.

TaLoS meets the following objectives:

Security and privacy. TaLoS is resilient to different threats, including a compromised operating system, hypervisor, BIOS, and malicious administrators. In addition, TaLoS does not affect the confidentiality or integrity of data handled by the application.

Ease-of-deployment. TaLoS is easy to deploy with existing applications, requiring no changes to the client implementation and no or only minor modifications of the server. In addition, TaLoS does not affect the scalability or availability of applications.

Performance overhead. TaLoS imposes a low performance overhead with respect to native application execution.

2. INTEL SGX

Intel's *Software Guard Extensions* (SGX) [5] is a set of CPU instructions that enables applications to maintain data confidentiality and integrity, even when the hardware and all privileged software, including the OS, hypervisor and BIOS, are controlled by an untrusted entity.

Enclaves. With Intel SGX, *enclaves* provide an isolated execution environment that is protected by the CPU. Enclave code and data reside in a region of protected physical memory called the *enclave page cache* (EPC). When enclave code and data are cache-resident, they are guarded by CPU access controls; when flushed to DRAM or disk, they are transparently encrypted and integrity protected by an on-chip memory encryption engine. Non-enclave code cannot access enclave memory, but only invoke enclave execution through a predefined enclave interface; enclave code is permitted to access enclave and non-enclave memory. Since enclaves execute in user mode, privileged operations such as system calls must be executed outside the enclave.

Enclaves are created by untrusted application code. Memory pages can then be assigned to the EPC, which records page permissions and enforces security restrictions. When the enclave is initialised, a cryptographic measurement of it is created, which allows a third-party to attest the enclave is genuine. For a thread to execute enclave code, the CPU switches to enclave mode and control jumps to a predefined enclave entrypoint.

The use of enclaves incurs a performance overhead: (i) thread transitions between enclave code and the outside incur additional CPU operations for checks and updates, such as a TLB flush; (ii) enclave code pays a higher penalty for cache misses because the hardware must encrypt and decrypt cache lines; and (iii) in current implementations, enclaves that use memory beyond the EPC size limit (typically less than 128 MB) must swap pages between the EPC and unprotected DRAM, which incurs a high overhead.

SGX SDK. Intel provides an SDK for programmers to use SGX [6]. Developers can create *enclave libraries* that are loaded into an enclave. A developer defines the interface between the enclave code and other, untrusted application code: (i) a call into the enclave is referred to as an *enclave entry call* (*ecall*). For each defined *ecall*, the SDK adds instructions to marshal parameters outside, unmarshal the parameters inside the enclave and execute the function; conversely (ii) *outside calls* (*ocalls*) allow enclave functions to call untrusted functions outside of the enclave.

3. TaLoS TLS TERMINATION

TaLoS is a port of LibreSSL [13] that securely terminates TLS connections by executing and maintaining security-sensitive code and data inside an SGX enclave. Fig. 1 shows which parts of TaLoS are placed inside the enclave.

Among others, TaLoS places the following parts inside the enclave: private keys, session keys used to communicate with clients, and code related to the TLS protocol implementation, e.g., function `SSL_read()`, which reads encrypted data from the network, decrypts it and returns the plaintext to the application, and function `SSL_write()`, which takes plaintext as input, encrypts it and sends the result along an existing TLS network connection.

Non-sensitive code and data, such as the `BIO` data structure that abstracts an I/O stream, as well as API wrapper functions, are placed outside of the enclave for performance reasons. Function calls that

[1]The source code of TaLoS is publicly available at https://github.com/lsds/TaLoS.

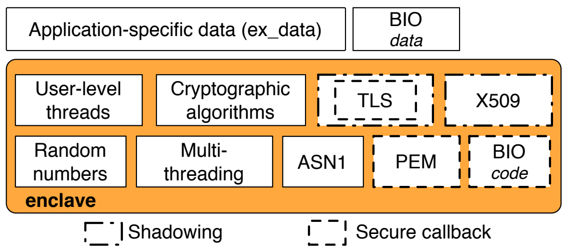


Figure 1: TaLoS TLS implementation

cross the enclave boundary are converted into *ecalls* and *ocalls*, as supported by the SGX SDK (see §2).

3.1 Enclave TLS implementation

We face two challenges when implementing TLS inside the enclave: (i) function callbacks are part of the LibreSSL API, but are untrusted and must be invoked outside the enclave, which could leak sensitive data. We address this issue by implementing *secure callbacks*; and (ii) applications may try to access internal TLS data structures that are security-sensitive and thus placed inside the enclave. We support this by *shadowing* such data structures as explained below.

Secure callbacks. Some API functions permit applications to submit function pointers. For example, `SSL_CTX_set_info_callback()` registers a callback to obtain information about the current TLS context. To execute such callback functions referring to outside code from within the enclave, TaLoS must execute corresponding *ocalls* rather than regular function calls. TaLoS proceeds in four steps as shown in the following listing (with error checks, shadow structures and SDK details omitted for simplicity):[2]

```
1  /* LibSEAL API */
2  void SSL_CTX_set_info_callback(SSL_CTX *ctx, void (*cb)(const
         SSL *ssl, int type, int val)) {
3    ecall_SSL_CTX_set_info_callback(ctx, (void*)cb);
4  }
5
6  int ocall_SSL_CTX_info_callback(const SSL* ssl, int type, int
         val, void* cb) {
7    void (*callback)(const SSL*, int, int) = (void (*)(const
           SSL*, int, int))cb;
8    return callback(ssl, type, val);
9  }
10
11 /* inside the enclave */
12 void* callback_SSL_CTX_info_address = NULL;
13
14 static int callback_SSL_CTX_info_trampoline(const SSL* ssl, int
         type, int val) {
15   return ocall_SSL_CTX_info_callback(ssl, type, val,
           callback_SSL_CTX_info_address);
16 }
17
18 void ecall_SSL_set_info_callback(SSL_CTX *ctx, void* cb) {
19   callback_SSL_CTX_info_address = cb;
20   SSL_CTX_set_info_callback(ctx,
           &callback_SSL_CTX_info_trampoline);
21 }
```

(1) The TaLoS API function executes an *ecall* into the enclave (line 3); (2) the enclave code saves the address of the outside callback (line 19) and passes the address of a newly-defined callback trampoline function (line 14) to the original API function (line 20); (3) when the callback function is invoked, the trampoline function is called instead (line 14); and (4) the trampoline function retrieves the callback address and performs an *ocall* into the outside application code (lines 6 and 15).

[2]Note that while there are two functions `SSL_CTX_set_info_callback()`, there is no name clash as only one is inside the enclave.

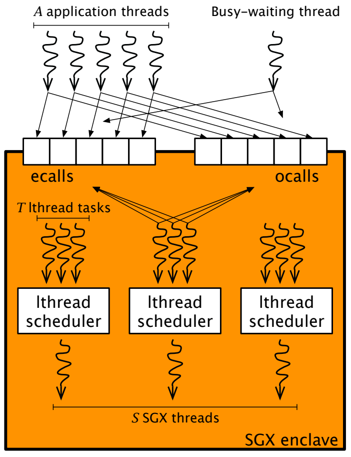


Figure 2: Asynchronous enclave transitions in TaLoS

For applications that register multiple callback functions, TaLoS uses a hashmap to store and retrieve callback associations.

We manually inspect 17 callbacks for LibreSSL to ensure that TaLoS does not leak sensitive data. In the worst case, TaLoS can pass a pointer to trusted memory outside of the enclave. SGX ensures that these pointers can not dereferenced by untrusted code. Further manual checks and the shadowing mechanism presented below mitigate pointer swapping attacks.

Shadowing. Applications may access fields of TLS data structures directly. For example, Apache and Squid access the `SSL` structure, which stores the secure session context. TaLoS supports such accesses in a secure manner by employing *shadow structures*. In addition to the security-sensitive structure inside the enclave, TaLoS maintains a sanitised copy of the `SSL` structure outside the enclave, with all sensitive data removed. TaLoS synchronises the two `SSL` structures at *ecalls* and *ocalls* as follows:

```
BIO * ecall_SSL_get_wbio(const SSL *s) {
  SSL* out_s = (SSL*) s;
  SSL* in_s = (SSL*) hashmapGet(ssl_shadow_map, out_s);

  SSL_copy_fields_to_in_s(in_s, out_s);
  BIO* ret = SSL_get_wbio((const SSL*)in_s);
  SSL_copy_sanitized_fields_to_out_s(in_s, out_s);
  return ret;
}
```

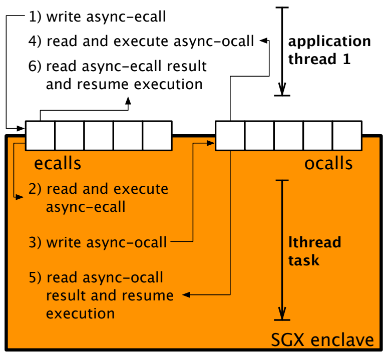
The association between the enclave structure and the shadow structure is stored in a in-enclave thread-safe hashmap.

3.2 Reducing enclave transitions

Implementing the TLS API requires enclave transitions. However, each enclave transition imposes a cost of 8,400 CPU cycles—6× more costly than a typical system call. We therefore apply three techniques to reduce the number of *ecalls* and *ocalls* in TaLoS:

(1) Instead of performing *ocalls* to allocate non-sensitive objects from within the enclave, TaLoS uses a pre-allocated memory pool to manage small objects frequently allocated from inside the enclave. This avoid *ocalls* to `malloc()` and `free()` by replacing them with less costly enclave-internal calls to the memory pool.

(2) Instead of using the `pthread` library [2] for synchronisation, TaLoS avoids *ocalls* to `pthread` by using the thread locks implementation provided by the SGX SDK. TaLoS further uses the SGX random number generator inside the enclave to avoid *ocalls* to the `random` system call.



1) write async−ecall
4) read and execute async−ocall
6) read async−ecall result and resume execution
application thread 1
ecalls
ocalls
2) read and execute async−ecall
3) write async−ocall
5) read async−ocall result and resume execution
lthread task
SGX enclave

Figure 3: Asynchronous ecall invoking an asynchronous ocall

(3) TaLoS reduces the number of *ecalls* by storing application-specific data written to TLS data structures outside the enclave. For example, Apache references a structure representing the current request in the TLS object. As the TLS object is stored inside the enclave, this would require an *ecall* for access. To avoid such enclave transitions, TaLoS stores application-specific data and their association with the TLS object outside of the enclave.

Together, these optimisations reduce the number of *ecalls* and *ocalls* for Apache by up to 31% and 49%, respectively, improving request throughput by up to 70%.

3.3 Reducing transition overhead

TaLoS reduces the overhead of the remaining enclave transitions by executing them asynchronously. Instead of threads entering and exiting the enclave, user-level tasks, implemented by the lthread library [9] inside the enclave, perform call executions.

Fig. 2 shows how these *asynchronous enclave transitions* are executed. Inside the enclave, *S enclave threads* execute *T* lthread *tasks*, each handling the *async-ecalls* from *A application threads*[3]. Application threads are further responsible for processing *async-ocalls* made by enclave-internal lthread tasks.

TaLoS uses an array of *ecall* request slots that is shared between the enclave and outside code. The array provides one slot for each application thread. While an lthread task can execute an *async-ecall* for any application thread, the opposite is not true: application threads have their own context (e.g. a client network connection). TaLoS ensures that when application thread *a* executes an *async-ecall*, the necessary *async-ocalls* and the result are also handled by *a*. To that end, each application thread is bound to a slot in both the *async-ecalls* and *async-ocalls* arrays. Similarly, the lthread task resuming an *async-ecall* after an *async-ocall* is the same as the one starting the *async-ecall*.

When an application thread wants to invoke an *ecall*, it issues an *asynchronous ecall* (*async-ecall*) as follows (see Fig. 3): (1) the *ecall* type and its arguments are written into this application thread's request slot; (2) the lthread scheduler detects a pending *async-ecall*. It finds the first available lthread task inside the enclave and resumes its execution, passing it the *async-ecall* arguments. In the meantime, the application thread waits for the result of the *async-ecall* or an *async-ocall*; (3) if it is necessary to execute a function outside the enclave, the lthread task adds its request to the application thread's slot in the *ocalls* array; (4) the application thread then retrieves the *async-ocall* arguments, executes the call and returns the result; (5) once the result of the *async-ocall* is available, the lthread scheduler finds the lthread task that requested this *async-ocall* and schedules it; and (6) when the *async-ecall* result is available, the application thread retrieves it and resumes execution.

[3]Due to limitations of current SGX implementations, it is not possible to dynamically add threads to the enclave.

Webpage	0 B	1 KB	10 KB	64 KB
No async. calls	1,126	1,095	882	644
With async. calls	1,771	1,722	1,693	1,375
Improvement	57%	57%	92%	114%

Table 1: Throughput (in requests/sec) of Apache with TaLoS when using asynchronous enclave transitions

To avoid having all application threads busy waiting for asynchronous call results, TaLoS uses a dedicated busy waiting thread that polls both arrays and wakes up the corresponding application thread.

Using asynchronous *ecalls*/*ocalls*, the performance of Apache with TaLoS increases by more than 57%, from 1126 requests/sec to 1771 requests/sec (see §4.2).

4. EVALUATION

Implementation. TaLoS uses the Intel SGX SDK 1.7 for Linux and LibreSSL 2.4.1. It consists of 282,200 lines of code (LOC), 270,600 of which are LibreSSL. TaLoS exposes 205 *ecalls* and 55 *ocalls* (28 of which are to standard C library functions). Around 5,400 LOC of the API are auto-generated *ecall* and *ocall* wrappers.

Experimental set-up. We run experiments on an SGX-capable 4-core Intel Xeon E3-1280 v5 at 3.70 GHz (no hyper-threading) with 64 GB of RAM, Ubuntu 16.04 LTS, Linux kernel 4.4. Clients connect via a Gigabit network. We use Apache 2.4.23 and Squid 3.5.23.

4.1 Enclave TLS overhead

We evaluate the overhead of TaLoS by measuring the throughput and latency of Apache and Squid with a libcurl [3] client. For Squid, the clients request webpages from an HTTPS server on a third machine within the same cluster.

We compare the maximum throughput of LibreSSL to TaLoS. Since we are interested in worst case performance, we focus on non-persistent connections, i.e. the client initiates a TLS handshake for each request. Indeed, the TLS handshake operation becomes the performance bottleneck.

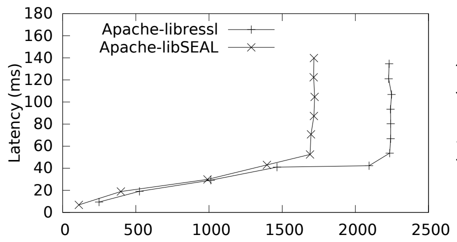
Fig. 4a and Fig. 4b report the latency and throughput for Apache and Squid for webpages of 1 KB. For Apache, TaLoS incurs a 23% performance overhead, decreasing the maximum throughput from 2,200 requests/sec to 1,700 requests/sec. The overhead for Squid is 31%, from 850 requests/sec to 590 requests/sec. The low throughput of Squid is due to the presence of two TLS connections: from the client to the proxy and from the proxy to the server.

In these experiments the CPU is the bottleneck. We observed similar results for webpages of different sizes until the network became the bottleneck, at which point TaLoS and LibreSSL offer the same performance.

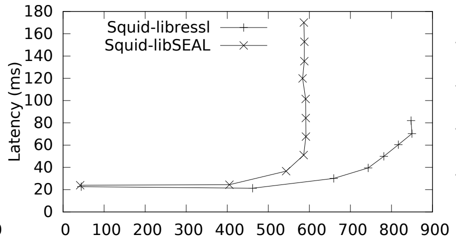
4.2 Impact of asynchronous calls

Asynchronous enclave transitions (§3.3) are motivated by the increasing cost of enclave transitions as more threads execute inside the enclave. For example, executing an empty *ecall* takes 20× more CPU cycles with 48 concurrent threads compared to one, from 8,500 cycles to 170,000 cycles.

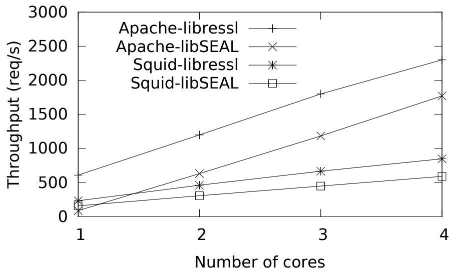
Tab. 1 shows that asynchronous enclave transitions increase performance by at least 57% when serving webpages of different sizes



(a) Apache throughput versus latency



(b) Squid throughput versus latency



(c) Scalability

Figure 4: Performance of Apache and Squid

using Apache. For pages larger than 10 KB, the performance benefit over synchronous enclave transitions is around 100%.

4.3 Scalability

We observe the maximum throughput for Apache and Squid as we increase the number of CPU cores. As Fig. 4c shows, performance improves linearly with the number of cores, demonstrating that TaLoS exploits multi-core CPUs. Due to the current unavailability of SGX-capable CPUs with more than 4 cores, we cannot evaluate further scaling behavior.

5. RELATED WORK

Intel's SgxSSL [7] and mBedTLS-SGX [14] are TLS libraries designed to execute inside Intel SGX enclaves. WolfSSL [15] is a traditional TLS library that can be executed within an SGX enclave. We highlight differences with TaLoS.

Transparency to applications. Unlike TaLoS, both SgxSSL and mBedTLS-SGX require the entire application to execute inside an SGX enclave. Depending on the application, this approach may require substantial changes to the application, in particular if the latter is written in a language different from C or C++. TaLoS, instead, only requires the TLS library to be executed inside the enclave, requiring no changes to existing applications.

Interface compatibility. Unlike TaLoS, both mBedTLS-SGX and WolfSSL are not compatible with the widely used OpenSSL/LibreSSL API. By providing the OpenSSL/LibreSSL API, TaLoS (i) supports a broad range of existing applications, and (ii) acts as a transparent drop-in replacement for existing applications, since no or only minor changes to the application code are required.

6. REFERENCES

[1] APACHE. HTTP server project. https://httpd.apache.org/, 2017.

[2] BARNEY, BLAISE. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/, 2017.

[3] CURL PROJECT. libcurl - the multiprotocol file transfer library. https://curl.haxx.se/libcurl/, 2017.

[4] DIERKS, T., AND RESCORLA, E. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[5] INTEL CORP. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[6] INTEL CORP. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk, 2016.

[7] Intel SgxSSL Library. https://software.intel.com/en-us/sgx-sdk/download, Nov. 2016.

[8] JabberD 2.x project. http://jabberd2.org/, 2017.

[9] LTHREAD. lthread, a multicore enabled coroutine library written in C. https://github.com/halayli/lthread, 2017.

[10] OPENSSL SOFTWARE FOUNDATION, INC. OpenSSL. https://www.openssl.org/, 2017.

[11] REESE, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[12] Squid project. http://www.squid-cache.org/, 2017.

[13] THE OPENBSD PROJECT. LibreSSL. https://www.libressl.org/, 2017.

[14] TLS for SGX: a port of mbedtls. https://github.com/bl4ck5un/mbedtls-SGX, Mar. 2017.

[15] WOLFSSL. wolfSSL with Intel SGX. https://www.wolfssl.com/wolfSSL/Blog/Entries/2017/1/17_wolfSSL_with_Intel_SGX.html, Jan. 2017.