LibSEAL: Revealing Service Integrity Violations Using Trusted Execution

Technical Report

Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind
Imperial College London

Robert Krahn, Christof Fetzer
TU Dresden

David Eyers
University of Otago

Peter Pietzuch
Imperial College London

ABSTRACT

Users of online services such as messaging, code hosting and collaborative document editing expect the services to uphold the integrity of their data. Despite providers' best efforts, data corruption still occurs, but at present service integrity violations are excluded from SLAs. For providers to include such violations as part of SLAs, the competing requirements of clients and providers must be satisfied. Clients need the ability to independently identify and prove service integrity violations to claim compensation. At the same time, providers must be able to refute spurious claims.

We describe *LibSEAL*, a SEcure Audit Library for Internet services that creates a non-repudiable audit log of service operations and checks invariants to discover violations of service integrity. LibSEAL is a drop-in replacement for TLS libraries used by services, and thus observes and logs all service requests and responses. It runs inside a *trusted execution environment*, such as Intel SGX, to protect the integrity of the audit log. Logs are stored using an embedded relational database, permitting service invariant violations to be discovered using simple SQL queries. We evaluate LibSEAL with three popular online services (Git, ownCloud and Dropbox) and demonstrate that it is effective in discovering integrity violations, while reducing throughput by at most 14%.

1 INTRODUCTION

Today, users rely on the correct operation of many Internet services: they expect Dropbox [32] and Google Drive [46] to store their files reliably; GitHub [40] to accurately record the commit histories of repositories; and Google Docs [45], Office 365 [69] and ownCloud [79] to preserve the integrity of shared documents. Service providers are not immune to data loss or corruption (e.g. both Dropbox and Gmail have lost data in the past [35, 44]) but the terms and conditions for best-effort services typically absolve them of any legal liability [33, 41, 47]. When services fail, it is challenging for users to uncover these failures and receive compensation.

Instead, when users require assurances of higher integrity from services, there is a business opportunity for providers to offer premium *integrity-assured* services with stronger service level agreements (SLAs), e.g. offering compensation if integrity violations occur. However, users must obtain independent indisputable proof of integrity violations, which is hard. At the same time, malicious users must be prevented from fabricating evidence to slander the provider's reputation.

Our goal is to help users discover service integrity violations, such as incorrect processing or data loss, for integrity-assured services and demonstrate unequivocally that a violation has taken place. We achieve this goal by creating a trusted, non-repudiable *audit log* of user operations and their service responses over time, which constitutes a ground truth for dispute resolution. In the case of Dropbox [32], for example, the audit log could include the hashes of all files uploaded by a user. Violations of service integrity can then be detected as *invariant violations* over the audit log. For Dropbox, this could be the failure to retrieve a file that was uploaded but not subsequently deleted.

A secure auditing solution must satisfy multiple requirements in order to be adopted in practice: it must (a) be *flexible* and *expressive* to support a wide range of Internet services and service-specific integrity invariants; (b) be *easily deployable* with existing Internet services and avoid third-party dependencies that affect its scalability or availability; (c) *protect* the audit log and *sensitive data* handled by the service; and (d) incur *low performance overhead*.

We describe **LibSEAL**, a **SEcure Audit Library** for Internet services that creates a non-repudiable log of information about service requests and responses. It then checks invariants over the log to discover service integrity violations. LibSEAL makes the following contributions:

(i) Auditing using trusted execution. LibSEAL executes as part of a provider's infrastructure to avoid third-party dependencies. It uses a *trusted execution environment* (TEE), such as Intel SGX [56], to protect the audit log integrity and shield itself from tampering. LibSEAL observes *all* service requests and responses, by acting as a *termination endpoint* for transport layer security (TLS) [29] connections to the service. Using the TEE's cryptographic capabilities, the audit log is stored securely on persistent storage. The TEE also protects invariant checks over the audit log issued by clients.

(ii) Efficient TLS support within the TEE. The performance of LibSEAL depends on how efficiently it handles TLS connections inside the TEE. LibSEAL uses an existing TLS implementation (LibreSSL [77]) and, for performance reasons, executes non-sensitive parts outside the TEE. It uses *shadow pointers* to permit untrusted

Imperial College London, March 2018, London
.

code to refer to protected TLS data structures without compromising security. It avoids expensive TEE code transitions by relying on *user-level threading* inside the TEE and implementing *asynchronous* calls to and from the TEE. The above techniques do not change the TLS API, allowing LibSEAL to provide a transparent replacement for existing TLS libraries.

(iii) Relational audit log and invariant queries. Even though the information logged and the invariants checked are service-specific, LibSEAL is easy to apply to new services. It uses a relational log format with a service-specific schema. Small service-specific modules parse requests and responses and log information. The log is stored using an embedded database [96] running inside the TEE. LibSEAL persists the log to avoid data loss and protect its integrity and regularly prunes old superfluous log entries.

Invariants are written as standard SQL queries and clients can trigger invariant checks by including a special header field as part of HTTP requests. A web browser plug-in receives the results and alerts users of invariant violations. For example, the following query is an invariant for the Git service to detect when the list of branches advertised to a client is incomplete:

```
SELECT time, repo FROM advertisements
NATURAL JOIN branchcnt
GROUP BY time, repo, cnt HAVING COUNT(branch) != cnt;
```

We evaluate a prototype implementation of LibSEAL, deployed with Intel SGX as a TEE, using three Internet services: (i) the web-based Git version control service [39]; (ii) the ownCloud collaborative document service [79]; and (iii) the Dropbox file storage service [32]. We demonstrate how LibSEAL can discover a range of integrity violations, including teleport, rollback and reference deletion attacks for Git [101], lost document edits for ownCloud, and inconsistent or lost files for Dropbox. The performance overhead is modest: LibSEAL reduces the throughput for ownCloud by 13%, for Git by 14%, and does not impact the latency of Dropbox.

This paper is organised as follows: §2 introduces the problem of service integrity violations, states our threat model and discusses existing solutions; §3 describes the design of LibSEAL; §4 explains the techniques that LibSEAL uses to efficiently terminate TLS connections; §5 details LibSEAL's log format and invariant checking; §6 presents our evaluation results; §7 compares against related work; and §8 concludes.

2 INTEGRITY OF INTERNET SERVICES

Next we motivate the problem of integrity violations in Internet services (§2.1) and describe different application scenarios (§2.2). We further introduce our threat model (§2.3), survey the space of existing solutions (§2.4) and give background on trusted execution environments (§2.5).

2.1 Violations of service integrity

Users of Internet services expect them to uphold service integrity. We define service integrity to be the correctness of the state of the service with respect to its public API, taking into account all user interactions. For example, GitLab [42] and Dropbox [32] are expected to maintain correct histories and versions of all files; collaborative document services such as Google Docs [45] and ownCloud [79] must offer consistent views across documents, even under concurrent edits by multiple users; and messaging services such as Slack [94] and XMPP [112] should deliver messages without modification and should not drop them.

Service providers cannot avoid data loss and corruption altogether: in February 2017, GitLab lost several hours' worth of user repository data, including merge requests and code snippets [91]; in October 2014, Dropbox admitted to a bug that caused thousands of files to be deleted [35]; in February 2011 and January 2014, Gmail [44] lost the emails of thousands of users [52, 111]; and, in August 2014, Microsoft's OneDrive service corrupted stored Excel spreadsheet files [71].

Since many Internet services offer a free best-effort service, their terms and conditions exclude liability under violations of service integrity. A survey reports that *'providers not only avoided giving undertakings in respect of data integrity but actually disclaimed liability for it'* [16]. Dropbox' terms state that *'to the fullest extent permitted by law [...] in no event will Dropbox [...] be liable for [...] any loss of use, data, business, or profits, regardless of legal theory'* [33]; Google's terms state that it *'[...] will not be responsible for lost profits, revenues, or data, financial losses, or indirect, special, consequential, exemplary, or punitive damages'* [47].

If users want assurances that go beyond a best-effort service, providers may offer premium versions of services with stronger SLAs that give compensation after integrity violations. However, users must then rely on providers disclosing integrity violations after they have occurred. Without independent means of detecting violations, it may be tempting for service providers to deny, downplay or hide violations. Often, data breaches come to light years after they occurred [25, 36, 54, 72], sometimes only when aggrieved users post incidences on social media or discussion forums [75, 86].

2.2 Application scenarios

Our goal is to establish an independent basis for providers and users to agree that an integrity violation for an Internet service has happened. This raises the challenge of how users can discover integrity violations as they occur and credibly demonstrate them during dispute resolution. Next we give examples of Internet services that may suffer from service integrity violations and would benefit from our approach.

Object sharing services such as Dropbox [32], GitLab [91] and GitHub [40] allow users to share files and track changes to them. Typically these services rely on a central provider that stores objects, manages the order of updates and provides data to authorised users. Since the service provider acts a central authority, failures or bugs in its software or hardware may lead to the corruption or loss of objects. Even though these services sometimes encrypt objects or maintain an object hash as metadata to ensure integrity, this is not sufficient: some services must process or transform object contents, and the metadata itself may become corrupted.

Collaborative document editing services—Google Docs [45], Office 365 [69] and ownCloud [79] for example—allow users to edit a range of documents types collaboratively and concurrently. For all of these services, a central cloud service instance coordinates the editing process between users, thus deciding, e.g. about the order of document updates and the access rights to documents. A software or hardware bug may lead to inconsistencies within the resulting document or users being unable to access documents [107].

Online payment systems such as PayPal [82] and Stripe [99] allow end users and businesses to hold and exchange funds via web interfaces and APIs. While such services use TLS encryption to protect the confidentiality and integrity of all messages exchanged with clients, internal software bugs or database failures may result in lost or even miscredited funds [24].

Communication and instant messaging services such as email, XMPP [112] and WhatsApp [109] allow end users to exchange different message types—both directly as well as within user groups. For most of these services, the communication between users is relayed via one or more service providers. Faults or bugs may compromise message integrity, e.g. causing messages to be dropped, modified or delivered to the wrong recipients [28].

2.3 Threat model

We assume that providers of integrity-assured services are not actively malicious: they take the necessary precautions to maintain service integrity, but misconfigurations, hardware failures, compromised or buggy software, malpractice, negligence on behalf of system administrators, and other human errors can all result in data loss and corruption [19, 24, 37, 62]. In such cases, the providers may act only to protect their reputation. Under this threat model, we assume the service provider to be "imperfect and selfish" [84], i.e. susceptible to integrity violations and selfish about revealing such incidents to users. For example, a 2012 study on healthcare data breaches found that, on average, breaches are identified after 85 days and customers are only notified after an additional 68 days [53].

We assume that clients have an inherent interest in service integrity. Our aim is not to prevent integrity violations from occurring but to enable clients to discover them after the fact, and have a non-repudiable proof of the violation. This also thwarts disingenuous clients seeking to slander the provider's reputation with false claims of integrity violations.

Note that we do not target data confidentiality, i.e. we assume that the service provider can read the content of client data stored on its machines. For some services, confidentiality can be ensured by encrypting data on the client side [26, 100]; for others, including collaborative services, this is not possible without modifications to the server, e.g. by adding cryptographic key management so that multiple clients can read and modify the same encrypted data.

We also do not consider availability—at any point, the service provider may decide to stop the processing of client requests. This is an orthogonal problem that can be addressed with other means: service replication [61, 76] can be used to ensure availability of the service; our approach can also be extended to detect service downtimes.

2.4 Existing approaches for integrity assurance

Next we survey existing approaches for avoiding or detecting integrity violations of Internet services:

Cryptographic protection can ensure the integrity (and confidentiality) of data given to service providers. EncFS [48] or GnuPG [43] may be used to encrypt and sign files before uploading them to a storage service such as Dropbox; the Git version control system uses hash chains and signed commits to ensure integrity. However, this limits the processing a service can carry out on behalf of clients. Collaboration services such as Google Docs or ownCloud require data to be modifiable on the server side to support features such as data sharing and content editors. Fully homomorphic encryption [38] allows for computation over encrypted data, but the overheads are impractical for production services. Cryptographic techniques also require mitigation for attack vectors by which integrity can be silently violated such as teleport, rollback, or reference deletion attacks [101].

Redundant services. Service integrity can be enhanced by relying on multiple redundant services for data storage or processing [3, 21]. Clients may maintain multiple data replicas with different storage services, and check data for consistency upon retrieval. Similarly, clients may use multiple data processing services in parallel, executing the same computation and comparing results. Such techniques, however, impose a burden on clients, which must integrate with different services, and increase the resource footprint of services.

Third-party integrity services. Another approach is to employ a third-party service that validates invariants over client requests and service responses [73, 83, 105]. CloudProof [83] executes using an existing cloud storage service such as Microsoft Azure [70], and permits clients to exchange file modifications via integrity-protected, authenticated messages; in DIaaS [73], clients exchange messages with both the cloud storage service and an integrity management service, requiring one additional network round-trip for each request. These third-party services require substantial changes to server- and/or client-side code, making their use non-transparent. They also introduce external dependencies, which may impair performance and availability.

PeerReview [50] maintains tamper-evident logs of messages exchanged between the participants of a distributed application. These logs are periodically shared with *witnesses*, which detect faulty behaviour by replaying messages to recreate the current application state. PeerReview, however, requires modifications to the application, a complex state machine specification, and additional resources for the replay.

2.5 Trusted execution environments

Flexible integrity checking requires a root of trust acceptable to both the service provider, who must remain in control, and the clients, who must obtain trustworthy proofs of integrity violations. *Trusted execution environments* (TEE), e.g. as supported by Intel CPUs through the *Software Guard Extensions* (SGX) [56] can provide this root of trust. TEE enables applications to maintain data confidentiality and integrity, even when the hardware and all privileged software (OS, hypervisor and BIOS), are controlled by an untrusted entity.

Enclaves. Intel SGX provides a TEE through *enclaves*, and enclave code and data reside in a region of protected physical memory called the *enclave page cache* (EPC), where they are protected by CPU access controls. When flushed to DRAM or disk, they are encrypted and integrity protected transparently by an on-chip memory encryption engine. Non-enclave code cannot access enclave memory, but only invoke enclave execution through a pre-defined 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.

Enclaves are created by untrusted application code, and during initialisation, a cryptographic measurement of it is created. To execute enclave code, the CPU switches to enclave mode and control jumps to a predefined enclave entrypoint. SGX supports multi-threaded execution. The use of enclaves incurs a performance overhead: (i) transitions between enclave and the outside incur additional CPU checks and 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 using 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.

Remote attestation and sealing. A remote party can verify the integrity of an enclave [4]. Based on the measurement during enclave initialisation, a dedicated *quoting enclave* signs the measurement using a secret CPU key. Intel provides an auxiliary attestation service to verify the validity of the signed measurements. Enclaves allow data to be written to persistent storage securely—a process known as *sealing*. Sealed data can be bound to a signing authority, which allows enclaves to persist state across reboots. Any enclave signed by the same authority can subsequently unseal it.

SGX SDK. Intel provides an SDK for programmers to use SGX [57]. 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.

3 LIBSEAL DESIGN

We describe LibSEAL,[1] a *SEcure Auditing Library* for detecting service integrity violations. The design of LibSEAL satisfies the following requirements:

R1: *Generality and flexibility.* LibSEAL is widely applicable, and supports different types of services with varying integrity requirements (see §2.1). LibSEAL intercepts client requests and service responses by terminating TLS network connections on behalf of services. It therefore observes all interactions between the clients and the service[2] and logs information from the requests and responses in an audit log. Integrity violations are expressed as violations of invariants over the audit log. The audit log has a relational schema, allowing invariant checks to be written as simple SQL queries.

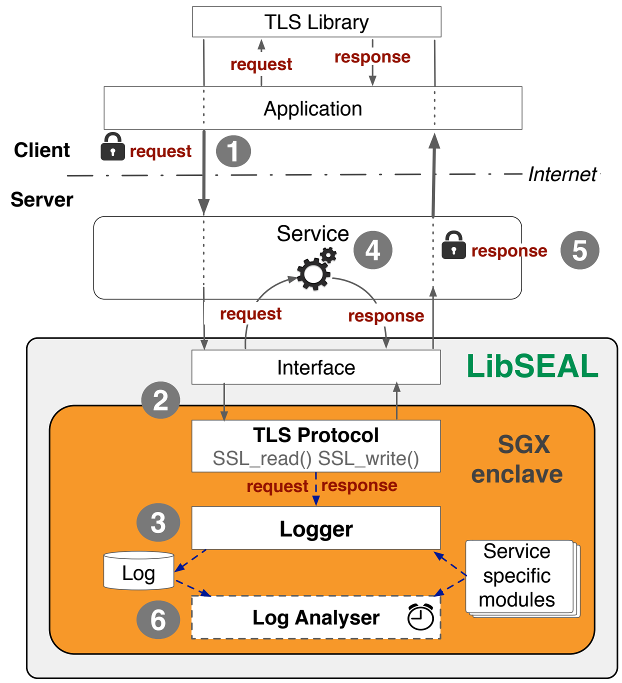


Figure 1: LibSEAL architecture

R2: *Ease-of-deployment.* LibSEAL is easy to deploy with existing services, requiring few, if any, changes to service and client implementations. In addition, LibSEAL does not significantly affect the scalability or availability of services. It acts as a drop-in replacement for existing TLS libraries. By using a TEE as the root-of-trust, it does not introduce third-party dependencies that may affect availability. Service providers have an incentive to deploy LibSEAL as it increases the perceived assurance of their services. It also creates opportunities for new premium integrity-assured services with stronger SLAs.

R3: *Security and privacy.* LibSEAL is secure according to our threat model (see §2.3). It does not affect the confidentiality or integrity of data handled by services and does not reveal internal details about service implementation to clients. LibSEAL protects itself and the information in the audit log using a TEE. When the audit log is stored to disk, it is cryptographically signed by the TEE to prevent tampering by the service provider.

R4: *Performance overhead.* LibSEAL imposes a small performance overhead with respect to native service execution. LibSEAL avoids costly TEE transitions by permanently associating threads with the enclave. It achieves fast and tamper-resistant persistent logging by leveraging a distributed monotonic counter protocol.

3.1 Architecture

Fig. 1 shows the architecture of LibSEAL and summarises its operation: a client issues a TLS-protected request (using e.g. HTTP or IMAP) to the service (step 1); the service passes the request to LibSEAL for decryption, which calls into the enclave and executes the TLS protocol (step 2); the decrypted request is passed to a *logger*,

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

[2] While the popularity and ubiquity of TLS means that LibSEAL can be applied to many existing Internet services, its design does not preclude it from working with other encrypted protocols such as SSH.

which invokes a *service-specific module* to parse the request and write pertinent information to the audit log (step 3); the decrypted request is then returned to the service outside the enclave, which processes it (step 4); the service response is also passed to LibSEAL, which logs it, encrypts it according to the TLS protocol and returns the encrypted response (step 5). Periodically, a client may invoke the *log analyser* to perform service-specific invariant checks over the audit log (step 6).

TLS connection termination (§4). LibSEAL provides a TLS API compatible with OpenSSL and LibreSSL. It can thus be used transparently by existing services, such as the Apache [11] and Nginx [87] web servers, the Squid [97] proxy, and the JabberD [1] XMPP server. To employ LibSEAL for auditing, services require linking against LibSEAL. LibSEAL executes LibreSSL in an enclave, allowing it to (i) terminate TLS connections securely; and (ii) protect session keys.

Audit logging (§5.1). Instead of logging all service request and response data, LibSEAL only stores the minimum amount of information required to check the integrity invariants. A service-specific module extracts the required information and passes it to the logger. For our use cases, these modules are between 250 and 400 lines of C++ code.

To maintain the audit log, LibSEAL uses the embedded SQLite relational database engine [96], which executes inside of the enclave. This allows invariant checking to be done using SQL queries. Each service-specific module stipulates the relational schema of the information to be logged. For example, for Git, the schema of the audit log is as follows:

```
updates(time, repo, branch, cid, type)
advertisements(time, repo, branch, cid)
```

The `updates` relation records all changes to branch and tag pointers that clients push to the server, while the `advertisements` relation records all branch and tag pointer advertisements sent to clients in response to client requests.

Invariant checking (§5.2). The invariants to check are service-specific SQL queries, and provided in addition to the service-specific module. For example, an invariant for Git is that "every advertisement must correspond to the most recent update for the corresponding (repo, branch, cid) triple". The following invariant query checks if there exists an `advertisement` such that the advertised commit ID does *not* correspond to the most recent update:

```
SELECT a.time,a.repo,a.branch FROM advertisements a
JOIN updates u ON u.time < a.time AND u.repo = a.repo
   AND u.branch = a.branch
WHERE a.cid != u.cid AND u.time = (SELECT MAX(time)
   FROM updates WHERE branch = u.branch
   AND repo = u.repo AND time < a.time);
```

The results of invariant checks on the audit log are returned to clients in-band, through the TLS connection.

3.2 Discussion

Invariants in LibSEAL must be robust against non-deterministic behaviour of services. For example, in the case of Facebook [34], it is not possible to specify an invariant over the exact order of posts in the newsfeed because this is non-deterministic from the point of view of the client requests and service responses (i.e. decided by an unknown algorithm). The content of the newsfeed and the validity of posts by different users, however, could be verified because these are observable by clients.

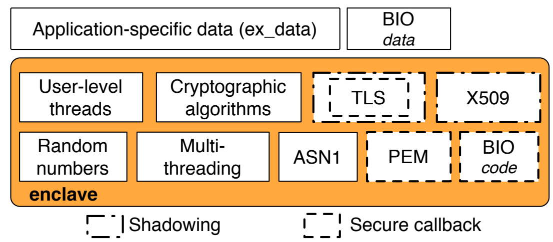


Figure 2: LibSEAL TLS implementation

LibSEAL checks integrity invariants against the audit log of a single service instance. For scalable services with many instances, LibSEAL can be deployed at the load-balancer or reverse proxy. This will log all requests and responses, even if they are served by different service instances. In §6.4, we evaluate this deployment scenario with Git and Apache as a reverse proxy.

When multiple LibSEAL instances are required, for example, when scaling out, the requests of a single client may be processed by different service instances. In this case, each instance would log a subset of the client interactions with the service. These partial logs must first be merged into a single log before invariant checking. The design of LibSEAL could be extended so that each LibSEAL instance manages a local log and periodically combines logs from other instances for invariant checking. This approach would be similar to distributed tracing systems, such as Dapper [93], that also collect remote logs.

4 TLS TERMINATION

LibSEAL ensures that all client requests and responses are part of the audit log by terminating the TLS network connection and executing the security-sensitive TLS protocol code inside the enclave.

For ease-of-deployment, LibSEAL compiles to a library that exposes the API of conventional TLS libraries (§4.1). To reduce the performance penalty imposed by SGX §2.5, LibSEAL implements several performance optimisations (§4.2). Finally, LibSEAL uses an asynchronous enclave transition mechanism with user-level threading to avoid costly enclave transitions (§4.3).

4.1 Enclave TLS implementation

LibSEAL provides the same API as OpenSSL and LibreSSL. It implements (i) the `SSL_read()` function, which reads encrypted network data, decrypts it and returns the plaintext to the caller (i.e. the application); and (ii) the `SSL_write()` function, which takes as input plaintext, encrypts it and sends the result along an existing TLS network connection.

As shown in Fig. 2, LibSEAL ports LibreSSL [77] to SGX, executing and maintaining security-sensitive code and data inside an enclave. This includes all code related to the TLS protocol implementation, as well as the private keys and session keys used to communicate with clients. The leakage of keys would allow an attacker to tamper with the messages exchanged between the client and server, thus defeating LibSEAL's audit log. 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 cross the enclave boundary are converted into *ecalls* and *ocalls*, as supported by the SGX SDK (see §2.5).

The implementation of TLS inside the enclave faces two challenges: (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. Several API functions permit the application to submit function pointers. This is for example the case of function `SSL_CTX_set_info_callback()`, which registers a callback used to obtain information about the current TLS context. To execute such callback functions referring to outside code from within the enclave, LibSEAL must execute corresponding *ocalls* rather than regular function calls. LibSEAL proceeds in four steps as shown in the following listing (with error checks, shadow structures and SDK details omitted for simplicity):[3]

```
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 LibSEAL 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 callback trampoline function (line 14) to the original API function (line 20); (3) upon invocation of the callback, 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). For applications that register multiple callback functions, LibSEAL uses a hashmap to store and retrieve the callback associations.

We manually inspect 19 callbacks for LibreSSL to ensure that LibSEAL does not leak sensitive data. In the worst case, LibSEAL can pass a pointer to trusted memory outside of the enclave. 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. To avoid modifications to the applications, we refrain from using *ecalls* to access such fields.

Instead, LibSEAL employs *shadow structures*. In addition to the security-sensitive structure inside the enclave, LibSEAL maintains a sanitised copy of the `SSL` structure outside the enclave, with all sensitive data removed. As shown in the listing below, LibSEAL synchronises the two `SSL` structures at *ecalls* and *ocalls*:

```
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 hashmap inside the enclave.

4.2 Reducing enclave transitions

Enclave transitions are necessary to implement the TLS API. Our micro-benchmarks show, however, that 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*:

(1) LibSEAL preallocates a memory pool outside of the enclave. This pool is used for frequent allocations of small objects by the enclave that do not require integrity or confidentiality guarantees. This approach avoids *ocalls* to `malloc()` and `free()` by replacing them with less costly enclave-internal calls to the memory pool. For example, LibSEAL uses this for `BIO` objects that are freed from inside the enclave when a TLS connection is closed.

(2) LibSEAL uses the thread locks and random number generator provided by the SGX SDK, therefore avoiding *ocalls* to the `pthread` library [14] and `random` system call.

(3) Finally, LibSEAL reduces the number of *ecalls* by storing application-specific data written to TLS data structures outside the enclave. For example, Apache stores the current request in the TLS object. As the TLS object is stored inside the enclave for LibSEAL, this would require an *ecall* for access. To avoid such enclave transitions, LibSEAL ensures that this data is stored outside of the enclave.

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

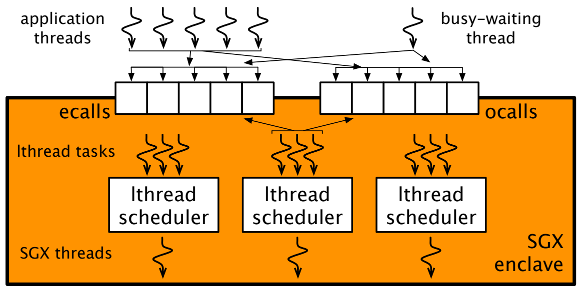


Figure 3: Asynchronous enclave calls in LibSEAL

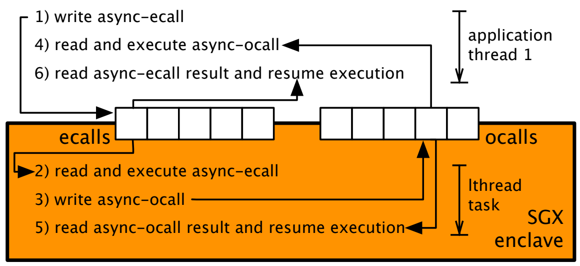


Figure 4: Asynchronous *ecall* that invokes an asynchronous *ocall*

4.3 Reducing transition overhead

LibSEAL reduces the overhead of the remaining calls by executing them asynchronously. Instead of threads entering and exiting the enclave, user-level tasks, implemented by the lthread library [65], execute the calls. This approach is similar to previous proposals [95]. While Eleos [78] and SCONE [12] execute asynchronous system calls when an entire application runs inside an enclave, LibSEAL executes arbitrary *ecalls* and *ocalls* asynchronously.

Fig. 3 shows how these *asynchronous enclave calls* (*async-ecalls*) are executed. Inside the enclave, *S enclave threads* each execute *T* lthread *tasks*, handling the *async-ecalls* from *A application threads*.[4] Application threads also process *async-ocalls* made by lthread tasks.

The number of enclave threads and lthread tasks impact the performance (see §6.8). To reduce the overhead of executing threads inside the enclave, *S* should be less than the number of physical CPU cores. A heuristic for the number of lthread tasks per enclave thread is $T \geq \frac{A}{S}$, which ensures that application threads wait a minimum amount of time when issuing an asynchronous *ecall*. LibSEAL does not affect the optimal number of application threads.

LibSEAL uses an array of *ecall* request slots, with one slot for each application thread, that is shared between the enclave and outside code. 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). LibSEAL 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 invokes an *ecall*, it issues an *async-ecall* as follows (see Fig. 4): (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 *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.

To avoid the overhead of all application threads busy-waiting for *async-ecall* results, LibSEAL could use two approaches: (i) only a single application thread that has invoked an *async-ecall* busy-waits, while all other threads sleep. The thread polls the slots in the *ecalls* and *ocalls* arrays and wakes up the corresponding application threads; or (ii) an additional dedicated thread busy-waits, polling both arrays, and waking up the corresponding application thread. The former approach requires synchronisation between the application threads whereas the latter approach adds the overhead of an extra thread. We find empirically that a dedicated polling thread results in better performance, and LibSEAL uses this solution.

Using *async-ecalls* and *async-ocalls*, the performance of LibSEAL with Apache increases by more than 57%, from 1,126 requests/sec to 1,771 requests/sec (see §6.8).

5 AUDIT LOGGING AND CHECKING

Here we describe how LibSEAL generates the audit log (§5.1) and how it checks integrity invariants (§5.2).

5.1 Logging

LibSEAL generates the audit log based on client requests and service responses. It observes all messages exchanged in a TLS connection by instrumenting the functions `SSL_read()` and `SSL_write()`. To prevent data loss under failure, LibSEAL writes the audit log to local persistent storage.

Service-specific logging. Rather than logging whole requests and responses, LibSEAL employs *service-specific modules* (SSMs) to log only the data required to verify the service invariants. Each SSM: (i) parses the requests and responses using a protocol parsing library (e.g. HTTP or IMAP); (ii) extracts the data required to verify the service invariants; and (iii) appends the data to a relational audit log. We envision the SSMs and service invariants being provided by service developers.

Each SSM defines a relational schema, the relations of which are created in the enclave during initialisation. For example, as mentioned in §3.1, the Git schema consists of two relations: (i) relation `updates` records all changes for all repositories (field `repo`) to branch and tag pointers (field `branch`) that clients *push* to the server. This includes the creation and deletion of branch/tag pointers, as well

[4] Due to a limitation of current SGX implementations, it is not possible to dynamically add threads to the enclave.

as their modification to point to a different commit ID (field `cid`); (ii) relation `advertisements` records all branch/tag pointer advertisements that the server returns to clients in response to a *git fetch* query. The schema uniquely identifies each tuple via the fields `time`, `branch`, and `repo`, with `time` being a logical timestamp maintained in the enclave.

LibSEAL provides a simple API for SSMs. For each request/response pair, it invokes the SSM using function

```
void libseal_log(char *req, char *rsp, size_t req_len,
        size_t rsp_len, void (*cb)(char *));
```

where `req` and `rsp` contain the request and response; and the callback function `cb` returns zero or more result tuples according to the SSM's log schema, which are inserted into the audit log.

Some services, such as ownCloud, use HTTP sessions or stateful protocols to maintain state across different request/response pairs. The SSM developer can use the LibSEAL log to maintain any part of this state that is relevant for auditing the service at a later point in time.

Log persistence and integrity. To prevent data loss under failure, LibSEAL synchronously flushes the log to persistent storage after each request/response pair. Since this storage is untrusted, a service provider may manipulate the log by forging new entries, deleting entries, or modifying them. To protect integrity, LibSEAL constructs a hash chain over all tuples, similarly to PeerReview [50]. A cryptographic signature ensures that only LibSEAL can add valid entries. LibSEAL verifies the log integrity by recomputing the hash of each entry, comparing it to the stored hashes, and verifying the signature. For the signatures, LibSEAL uses an ECDSA public/private key pair, as supported by the SGX SDK and created during enclave provisioning.

To prevent rollback attacks [81, 98] in which an attacker presents an older version of the log, LibSEAL requires secure persistent counters. The SGX hardware provides monotonic counters for this purpose but they have poor performance and limited lifespans [98]. LibSEAL therefore employs the distributed protocol of ROTE [67]: for each log entry, LibSEAL contacts n nodes, including itself, to retrieve and update a monotonic counter, where $n = 3f + 1$ with f being the tolerable number of malicious nodes. To be independent of third-parties, we envision these nodes being LibSEAL instances under the control of the service provider. If this is infeasible, dedicated instances of a ROTE service may be used. We evaluate the corresponding performance implications in §6.4.

Log trimming. To prevent the audit log from growing without bound, LibSEAL trims the log periodically using one or more service-specific *trimming queries*. Trimming queries remove log entries no longer needed for future invariant checks. Depending on the invariants, trimming queries may either truncate the log or, if required, selectively discard log entries to avoid missed or spurious violations.

For example, for Git, we define two trimming queries:

```
DELETE FROM advertisements;
DELETE FROM updates WHERE time NOT IN
      (SELECT MAX(time) FROM updates GROUP BY repo,
           branch);
```

The first query discards all advertisements, as they must be checked only once for the Git invariants. The second query selectively removes all but the most recent update for each branch: at least one update must be retained per branch for the completeness invariant and that update must be the most recent to enforce the soundness invariant (see §6.2).

Since trimming may lead to an inconsistent hash chain, LibSEAL recomputes the hashes of the remaining log entries. To prevent expensive updates to each entry of the log, LibSEAL stores the hashes separately and associates them with their corresponding entry via their primary key.

5.2 Checking

Invariants typically express *soundness* and *completeness* properties over the recorded tuples. Soundness invariants verify that any data returned from the service to clients corresponds to expected values; completeness invariants verify that the service does not fail to return data to clients.

Invariant specification. Invariants in LibSEAL are expressed as SQL queries over the audit logs' relations. SQL is well-known by developers, and database engines provide efficient means to store and query structured data. We find that SQL is sufficiently expressive to specify all desired invariants for our real-world use cases.

Concretely, SQL `SELECT` queries express invariants that must hold for all log entries. Since invariant violations are generally more interesting, queries express the negation of an invariant. For our use cases, we can detect relevant integrity violations using only 1–2 invariants, each consisting of around 10 lines of SQL (see §6.2).

Invariant checking. The default behaviour of LibSEAL is to trigger invariant checks after configurable time intervals. Clients may also trigger checks explicitly by setting a `Libseal-Check` request header for HTTP-based services. When a client triggers an invariant check, LibSEAL executes the corresponding query against all the clients' entries in the database. The result set contains all log entries that violate the invariant; if the set is non-empty, the client is notified.

Result notification. LibSEAL communicates the results of checks in-band. For HTTP-based services, clients retrieve the result of the most recent check in a `Libseal-Check-Result` HTTP response header. This header can be viewed using a web browser plugin [30, 88]. For other protocols, in-band communication may require a set of changes to the client.

The implementations of LibSEAL and the service-specific modules are publicly available. Anyone can audit the code, understand which information is being logged and verify the correctness of the invariants. This provides assurance to clients that LibSEAL correctly checks for integrity violations.

6 EVALUATION

We evaluate the security and performance of LibSEAL using three popular Internet services: (i) the *Git* version control service and (ii) the *ownCloud* collaborative document service, both using an *Apache* web server; and (iii) the *Dropbox* file storage service using a *Squid* proxy server.

Module	LOC		#ecalls	#ocalls
LibreSSL	269,400	(78.1%)	206	23
Enclave shim layer	9,400	(2.7%)	0	19
Async. transitions	3,400	(1.0%)	1	1
SQLite	61,000	(17.7%)	0	12
Audit logging	1,700	(0.5%)	2	0
LibSEAL total	344,900	(100%)	209	55

Table 1: Lines of code and enclave interface of LibSEAL

Our evaluation results show that: (i) invariants are simple to write yet expressive enough to detect service integrity violations (§6.2); (ii) LibSEAL is secure against interface attacks and prevents log bypassing (§6.3); (iii) LibSEAL has a low performance overhead of at most 14% (§6.4); and (iv) log sizes and log checking times are practical (§6.5).

Implementation. LibSEAL uses the Intel SGX SDK 1.9 for Linux and LibreSSL 2.4.1. It consists of 344,900 lines of code (LOC), 78.1% of which is LibreSSL (Tab. 1). The enclave shim layer includes: 5,400 lines of boiler-plate wrappers for *ecalls* and *ocalls*; 1,200 LOC for the shadowing and secure callbacks; and 2,800 LOC for supportive data structures. Out of the 1,700 LOC for audit logging, 600 LOC are glue code to parse HTTP messages and execute SQL queries. There are 61,000 LOC for the SQLite implementation. The SSMs account for 450 LOC (Git), 350 LOC (ownCloud), and 300 LOC (Dropbox), respectively. The enclave interface consists of 209 *ecalls* defining the LibSEAL API and 55 *ocalls* to access libc and support the secure callbacks.

6.1 Use cases

Git [39] is a distributed version control system used by the popular GitHub and GitLab services. It provides integrity guarantees using a hash chain: each commit ID is a result of hashing the committed files, the commit message, and the previous commit ID. Tags and branches are pointers to IDs.

We explore how LibSEAL discovers integrity violations that are not prevented by Git's hash chain [101]. Git's hash chain detects changes to committed files but fails to protect branches and tags: *teleport* violations allow a branch pointer to reference an arbitrary commit on a different branch; *rollback* violations allow a branch pointer to reference an old commit; and *reference deletion* violations enable the removal of entire branches or tags.

ownCloud [80] is a collaborative document editing service. Users can edit documents by adding, deleting and annotating text. The server and its clients synchronise changes to the document by exchanging JSON messages within sessions with one or more clients. When a client leaves a session, it creates a snapshot of the document and sends it to the server. Clients subsequently joining the session receive the latest snapshot and all subsequent updates.

Due to the collaborative nature of ownCloud, the service provider must read and modify the document content. Client-side protection of the document, such as encryption or digital signatures, is not supported. We evaluate if LibSEAL can verify the integrity of snapshopts and update messages and thus of the resulting document.

Dropbox [32] is a popular file sharing service. Internally, files are split into 4 MB blocks, each with a hash value. The list of hashes, called the *blocklist*, is part of the file metadata. To upload a file, a client sends `commit_batch` messages that specify the blocklist, the filename, and the size (−1 in case of deletion). Next, any missing blocks are sent to the server. Periodically, the clients send `list` requests, asking the server for their list of files. For each file that is different, the server returns the size of the file and its blocklist.

While Dropbox protects the integrity of the files' blocks, it does not safeguard the metadata. We explore if LibSEAL can detect violations in the blocklist integrity, and the completeness of the returned file list.

6.2 Integrity invariants

We describe the LibSEAL integrity invariants and their associated audit log schemas for the use cases.[5]

Git. LibSEAL can detect teleport, rollback and reference deletion attacks [101] by recording and verifying the metadata exchanged between the server and its clients.

Log schema. §5.1 described the log schema for Git.

Invariant. To verify integrity, we define both a soundness and a completeness invariant. The soundness invariant specifies that every advertisement must correspond to the most recent update for the corresponding `(repo, branch, cid)` triple. We specify a query that returns all `(time, repo, branch)` triples for which the advertised commit ID did not match:

```
SELECT * FROM advertisements a WHERE cid != (
  SELECT u.cid FROM updates u WHERE u.repo = a.repo AND
       u.branch = a.branch AND u.time < a.time ORDER BY
       u.time DESC LIMIT 1)
```

The completeness invariant states that when an advertisement happens, *all* `(repo, branch, cid)` triples must be advertised to the client. For this, we specify an auxiliary SQL view `branchcnt` returning the number of non-deleted branches for each repository at each point in time:

```
CREATE VIEW branchcnt AS
SELECT DISTINCT a.time,a.repo,COUNT(u.branch) AS cnt
FROM advertisements a
JOIN updates u ON u.time < a.time AND u.repo = a.repo
WHERE u.type != 'delete' AND u.time = (SELECT MAX(time)
    FROM updates WHERE branch = u.branch
    AND repo = u.repo AND time < a.time) GROUP BY
        a.time,a.repo,a.branch;
```

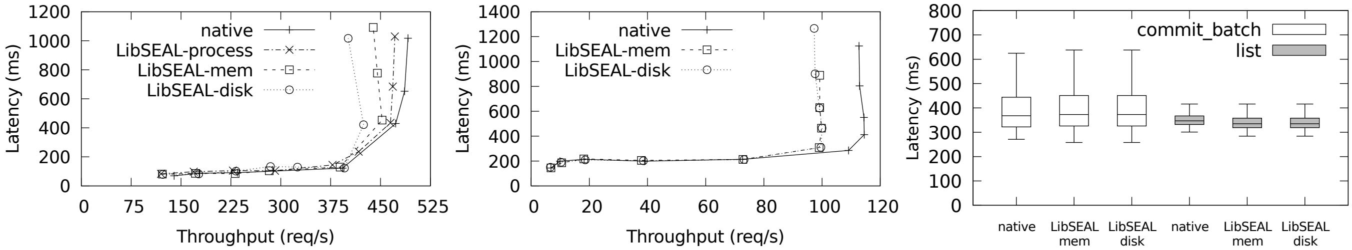
The actual completeness invariant leverages this view (see §1).

Trimming. For the trimming query, see §5.1.

ownCloud. LibSEAL detects document integrity violations related to text edits by recording document updates exchanged between the ownCloud service and its clients.

Log schema. The schema consists of a single relation recording the JSON document updates synchronised between the service and all of its clients. For details, see Appendix A.1.

[5] Due to space constraints, we defer the complete list of invariants, log schemas, and trimming queries to a technical report [13].



(a) Git throughput and latency **(b) ownCloud throughput and latency** **(c) Dropbox latency**

Figure 5: Performance of Git, ownCloud and Dropbox with and without LibSEAL

Invariants. We model a document as a snapshot and an ordered list of updates. Our invariants check the following: (i) snapshots sent to new clients match the latest snapshot, and (ii) at each point in time, the aggregate history of synchronised updates between the service and a client for a document corresponds to a *prefix* of the aggregate history of updates that the service received from *all* clients. We formalise the invariant in terms of a document prefix because the service may receive updates from multiple clients before redistributing them to other clients. For details, see Appendix A.1.

Trimming. For the trimming query, see Appendix A.1.

Dropbox. LibSEAL detects integrity violations concerning (i) the list of files sent to the client and (ii) the content of files. It checks the completeness of the list of files, as well as the soundness of all blocklists. Since the Dropbox client verifies the integrity of the blocks' contents, LibSEAL does not: if Dropbox serves wrong block contents, the Dropbox client will detect the incorrect hash of the received block and drop it. As LibSEAL stores the original block hash sent by the client, the client can later prove that either the metadata or the block contents have been modified.

Log schema. LibSEAL logs the `commit_batch` and `list` messages into two relations:

```
commit_batch(time,file,blocks,account,host,size)
list(time,file,blocks,account,host,size)
```

An entry in relation `commit_batch` indicates that at logical time `time`, the blocklist `blocks` of file `file` was updated by an account `account` from host `host` and sent to Dropbox; `list` refers to updates sent by Dropbox, following a client request to receive a list of all new or updated files.

Invariants. LibSEAL verifies whether for each client and point in time, the list of files sent by the server is consistent with the client's most recent file uploads and deletions. The corresponding invariant ensures that each file update or deletion is reported to clients when they request an updated file list. LibSEAL further verifies whether, for each file served to a client, the announced blocklist is correct. The associated invariant specifies that the blocklist returned by the server must correspond to the blocklist most recently uploaded by the client. For the SQL queries for both invariants, see Appendix A.2.

Trimming. For the trimming query, see Appendix A.2.

6.3 Security discussion

We discuss attacks according to our threat model (§2.3).

Bypassing logging. We assume that a provider has agreed to offer LibSEAL as part of a premium integrity-assured service. In some cases, the provider may still try to deactivate logging by linking to a traditional TLS library instead of LibSEAL. By using the SGX attestation and provisioning facilities, LibSEAL can obtain the TLS certificate and the corresponding private key in a secure manner, after the enclave has been successfully established and verified. Clients can thus verify whether the presented certificate indeed belongs to a genuine LibSEAL enclave. Since LibSEAL stores the private keys and session keys in protected enclave memory, it is not possible to forge the TLS certificate.

Impersonating clients. The cloud provider may impersonate a client and create fake actions that mimic user requests in order to hide an integrity violation. Clients can prevent such attacks by using TLS client authentication. For each request, LibSEAL then ensures the authenticity of the request by checking the certificate provided by the client.

Log privacy. The audit log may contain sensitive information about clients or services. For example, for ownCloud, LibSEAL records the entire—potentially sensitive—document history. To prevent privacy violations, LibSEAL can encrypt the log when written to persistent storage. Note that, as mentioned in §2.5, the sealing mechanism is not tied to a specific CPU but to a signing authority, allowing the sealed log to be shared across machines if necessary.

Log inconsistency. Crashes of LibSEAL instances may lead to inconsistencies between the log and the service state. This can happen if the server has executed a client request but has not yet sent a reply. In this case, we assume the client will retry the request until it receives a reply, at which point the log contains consistent information about both the request and response.

Interface attacks. A service provider may attempt to access enclave data by manipulating parameters and return values of *ecalls* and *ocalls*, respectively [22]. To tackle this problem (i) LibSEAL applies checks on the parameters passed to *ecalls* and *ocalls* (see §4.1). These checks reduce the attack surface by allowing only correct value ranges. If interface checks fail, LibSEAL aborts; and (ii) LibSEAL uses the shadowing mechanism to enforce the validity of pointers for sensitive data structures.

SGX-specific attacks. Exploiting security vulnerabilities in the LibSEAL implementation or side-channels in SGX may compromise the security of sensitive data held within the enclave or the integrity of the log [106, 108, 113]. While the current implementation of LibSEAL executes the entire LibreSSL codebase inside the enclave, only a few components actually require protection: (i) the private keys associated with the TLS certificate and the session keys (to ensure that an attacker cannot impersonate a client or bypass LibSEAL); and (ii) the code accessing the log (to ensure that an attacker cannot create or modify log entries). Other parts of LibreSSL are not security-sensitive, thus drastically reducing the attack surface. Well-known techniques such as CFI [2, 18], memory protection [17, 20] and cache protection [49] can be applied to LibSEAL to mitigate such attack vectors.

Denial-of-service attacks. As explained in §5.2, clients may trigger a log check at the server. A malicious client could use this to perform a denial-of-service attack. LibSEAL therefore imposes a limit on the rate at which clients can check the log.

6.4 Performance overhead

Experimental set-up. We deploy services on an SGX-capable 4-core Intel Xeon E3-1280 v5 at 3.70 GHz (no hyper-threading) with 64 GB of RAM that runs Ubuntu 16.04 LTS with Linux kernel 4.4. The clients are connected to the service via a 10 Gbps network. We use Apache 2.4.23, Squid 3.5.23, Git 2.10.1 and ownCloud 9.1.3.

In the case of Git and ownCloud, we measure the achieved throughput as the number of clients increases.

Git. We evaluate the performance impact of LibSEAL on the Git service by replaying the first few hundred commits from six real-world repositories [5–10]. We emulate a large-scale Git service by setting up Apache in reverse proxy mode and linking against LibSEAL. This instance of Apache logs all requests/responses and forwards the traffic to Git backend servers for request processing. The monotonic counter service (see §5.1) is configured to synchronise with three other nodes in the same cluster, therefore tolerating one malicious server. To understand which parts of LibSEAL impose performance overheads, we explore several configurations.

Fig. 5a shows the latency and throughput when replaying the commons-validator repository [8]; other repositories gave similar results. Native execution with LibreSSL serves as a baseline with a maximum throughput of 491 requests/sec (Fig. 5a, native). We measure the SGX enclave overhead alone by running Git behind Apache/LibSEAL without logging. This results in a maximum throughput of 472 requests/sec (Fig. 5a, LibSEAL-process), or 4% overhead. If, in addition, LibSEAL logs to an in-memory database, the maximum throughput is 452 requests/sec (Fig. 5a, LibSEAL-mem)—an 8% overhead compared to native execution. Finally, persisting the log to disk results in an overall throughput of 425 requests/sec (Fig. 5a, LibSEAL-disk), i.e. 14% overhead.

ownCloud. We set up an experiment in which multiple clients send document updates—consisting of both single characters as well as entire paragraphs—to the ownCloud service.

Fig. 5b shows the throughput and latency of ownCloud with: (i) LibreSSL (Fig. 5b, native); (ii) LibSEAL with in-memory logging (Fig. 5b, LibSEAL-mem); and (iii) LibSEAL with persistent logging (Fig. 5b, LibSEAL-disk). Overall, LibSEAL imposes a 13%

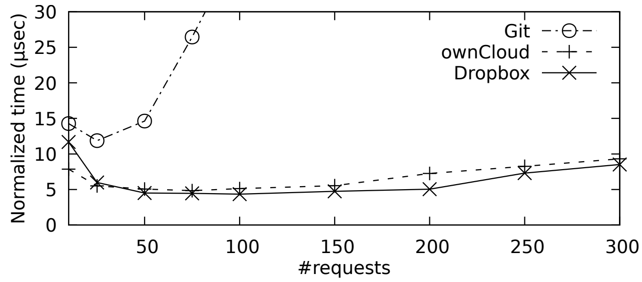


Figure 6: Normalized invariant checking and trimming time

overhead, from 115 requests/sec to 100 requests/sec. As the bottleneck is the underlying PHP engine, logging to disk does not impose additional overhead.

Dropbox. As we are unable to monitor the Dropbox servers, (i) we route all Dropbox traffic through a Squid proxy service, linking it against LibSEAL, and disable the clients' certificate verification [60]; and (ii) measure the request latency imposed by Squid/LibSEAL rather than maximum throughput, as we can not saturate the Dropbox service. The average network latency between Squid and Dropbox is 76 ms. We use the benchmark by Drago et al. [31] to create and delete text and binary files inside a Dropbox folder.

Fig. 5c shows the latency of `commit_batch` and `list` messages. Overall, the median and quartile values are close for both types of messages and for all configurations. For `commit_batch` messages, native execution (i.e. Squid with LibreSSL) achieves a median latency of 363 ms (Fig. 5c, native), LibSEAL logging in-memory 370 ms (Fig. 5c, LibSEAL-mem), and LibSEAL logging to disk 377 ms (Fig. 5c, LibSEAL-disk), respectively, which are all marginal increases. The results for `list` messages are similar.

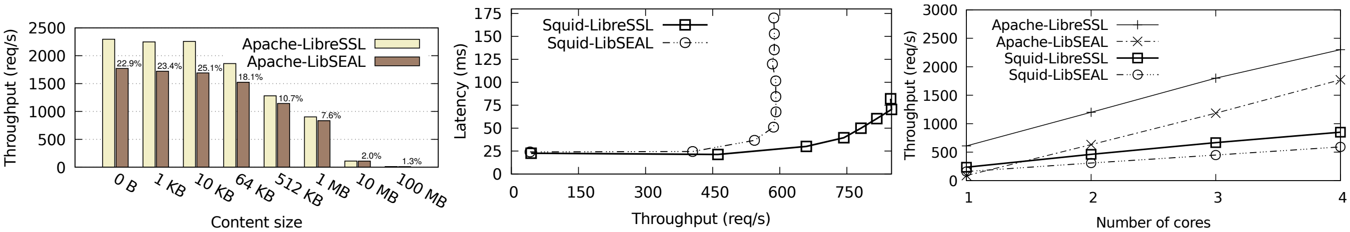
6.5 Checking/trimming overhead and log size

For each service, we evaluate: (i) the time needed for invariant checking and log trimming; and (ii) the growth in log size.

Checking and trimming. We explore the cost of invariant checking and trimming by considering their execution at various frequencies. For each use case, Fig. 6 shows the combined time for checking and trimming for different request intervals. Since the absolute checking and trimming times increase with larger intervals, i.e. when executed less often, the reported time is normalized by the interval size. We observe that there is an optimal interval length at which the normalised cost is lowest: 25 requests for Git, 75 requests for ownCloud, and 100 requests for Dropbox.

Configuring LibSEAL to use the above interval lengths, the execution of log checking and trimming takes a total of 0.3 ms for Git and 0.4 ms for both ownCloud and Dropbox. Since these times are orders of magnitude smaller than the observed latencies for these services (see §6.4), we consider checking and trimming at these intervals to be practical.

Log size. Trimming at the above regular frequency, the log sizes for our use cases are proportional to certain workload parameters: (i) for Git, the log size is proportional to the number of branch and tag pointers. Since logging one pointer takes 530 bytes, the resulting log size is #pointers × 530 bytes; (ii) for ownCloud, the log size



(a) Apache throughput and overhead **(b) Squid throughput versus latency** **(c) Multi-core scalability**

Figure 7: Performance of Apache and Squid

is proportional to the number of edited documents as well as the number of clients and updates in the last session. Since each document update causes a constant logging overhead of 124 bytes, small updates lead to a large log size. Assuming only single-character updates, this results in a maximum log size of #documents × #clients × #updates × 131 bytes—with 7 bytes carrying information about the actual update; and (iii) for Dropbox, the log size is proportional to the number of files. Since LibSEAL stores a 64 byte hash for each file blocklist, the resulting log size is #files × 64 bytes. Overall, these log sizes are reasonably small, especially when compared with the size of the service's actual payload data.

6.6 Enclave TLS overhead

To measure the overhead introduced by LibSEAL's TLS implementation inside an enclave, we evaluate the throughput and latency of Apache and Squid by retrieving contents of different sizes using the libcurl client [27]. For the Squid experiments, the clients request contents from an HTTP server on a third machine located in the same cluster.

We compare the maximum throughput of LibreSSL to LibSEAL without auditing. To evaluate worst case performance, we use non-persistent connections, i.e. a new TLS connection for each request. Indeed, the TLS handshake becomes the performance bottleneck. For all experiments, we show the maximum throughput when the CPU is saturated.

Fig. 7a shows the maximum throughput for Apache for different content sizes. The performance overhead ranges from 1% (with 100 MB of data) to 23%–25% (with 0 Byte to 10 KB of data). The high performance overhead for small content sizes is due to the significant cost of the TLS handshake. This cost is amortised when transferring more data, resulting in a throughput of 8.7 Gbps for 100 MB. For comparison, we execute Apache inside an SGX enclave using SCONE [12], which results in an overhead of 32% for non-persistent connections with 1 KB of data.

Fig. 7b reports the latency and throughput for Squid with a content size of 1 KB. With LibSEAL, the throughput decreases from 850 to 590 requests/sec, i.e. a 31% overhead. The lower throughput and higher overhead for Squid is due to the presence of two TLS connections: one from the client to the proxy and the other from the proxy to the server, resulting in additional TLS handshakes and data en-/decryption.

In conclusion, we observe similar results for different content sizes: LibSEAL and LibreSSL offer the same performance once the network becomes the bottleneck.

Content size	0 Byte	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 2: Throughput (in requests/sec) of Apache with LibSEAL when using asynchronous enclave calls

6.7 Multi-core scalability

We investigate the throughput for Apache and Squid as we increase the number of CPU cores. As Fig. 7c shows, performance improves linearly with the number of cores, demonstrating that the multi-threaded implementation of LibSEAL exploits multi-core CPUs. Due to the current unavailability of SGX-capable CPUs with more than 4 cores, we cannot evaluate further scaling behaviour.

6.8 Impact of asynchronous calls

LibSEAL's use of asynchronous enclave calls (see §4.3) is motivated by the increasing cost of enclave transitions as more threads execute inside the enclave: invocation of one *ecall* takes 8,500 cycles when one thread is executing, compared to 170,000 cycles with 48 threads—a 20× increase.

Tab. 2 reports the performance with and without asynchronous calls for Apache when retrieving content of different sizes. Asynchronous *ecalls*/*ocalls* improve the performance by at least 57%. For content sizes larger than 10 KB, the performance benefit is around 2×. This gain is due to the increasing number of *ocalls* when transferring more data, in which case the asynchronous calls mechanism decreases the relative enclave transitions overhead.

The asynchronous enclave call mechanism has multiple tuning parameters. We explore the impact of the number of SGX threads (Tab. 3) and lthread tasks (Tab. 4) on the performance of Apache-LibSEAL, for a 1 KB content size.

The number of SGX threads has a major impact on performance: adding SGX threads increases performance from 593 requests/sec (1 SGX thread) to 1,722 requests/sec (4 SGX threads). We also notice that executing more SGX threads increases the CPU utilisation. Once the CPU utilisation reaches the maximum (i.e. 400% with 3 SGX threads on a 4-core machine), increasing the number of SGX threads further decreases performance. This is due to increased contention between the SGX and Apache threads.

We do not observe a correlation between the number of lthread tasks and the throughput, which is around 1,700 requests/sec. However, the number of lthread tasks affects the latency observed by the client. Without enough lthread tasks, a request takes longer to be processed. Using more lthread tasks improves the probability of having one or more available user-level threads, which results in less waiting time for the Apache threads when they execute an *ecall*. Increasing the number of lthread tasks beyond the number of application threads does not have an effect on the system.

7 RELATED WORK

TrInc [63] and A2M [23] predate the commercial availability of TEEs and propose using custom trusted hardware components to enforce accountability—neither of these approaches support invariant checking. Nguyen et al. [74] propose a cloud-based secure logger using Intel SGX and a TPM for medical devices. Similar to our work, the communication between the medical device and the enclave is secured so that the system is resilient against replay and injection attacks. However, they neither implement nor evaluate their approach.

Wang et al. [102–105] verify the integrity of data stored in the cloud. They support the privacy-preserving auditing of cloud data by third parties. In contrast, LibSEAL verifies more general integrity invariants. Since the verifier runs inside a TEE, the privacy of the audited data is preserved.

Proof of storage [51, 92, 115] and proof of violation [55] permit clients of distributed applications to verify the integrity of their data stored at a server. These solutions are limited to data storage services and involve complex cryptographic operations that are not transparent to the client and server.

Auditing-as-a-service [85, 89, 90, 114] requires a trusted third party to maintain an audit logs for detecting integrity violations. LibSEAL does not require a trusted third party, but can instead rely on the TEE for privacy and integrity.

Depot [66] and CloudProof [83] provide secure storage on top of untrusted cloud storage services. Client data is augmented with metadata to ensure integrity: Depot builds a hash-chained modification log for the data; CloudProof uses cryptographic keys to implement access control policies. LibSEAL is more generic by supporting arbitrary services.

SUNDR [64] is a network file system that provides integrity guarantees to clients. Clients can detect attempts by a malicious server to tamper with files. Each client operation is signed and saved on the server side. SUNDR constructs a history of operations that the clients use to check for the integrity of the file system. While SUNDR specifically protects file system integrity, LibSEAL is more general and applicable to a larger variety of services.

mBedTLS-SGX [68] and WolfSSL [110] are TLS libraries that execute inside SGX enclaves. Unlike LibSEAL, their interface is not compatible with OpenSSL/LibreSSL, thus supporting fewer services. mBedTLS-SGX and the Intel SGX SSL library [58] target applications running entirely inside an enclave, whereas in LibSEAL only the TLS library executes inside an enclave. The auditing approach of LibSEAL, however, can also be applied to these libraries.

In contrast to SCONE [12] and Haven [15], we chose not to execute both the application and the TLS library inside the enclave. This way, LibSEAL reduces the TCB size and remains compatible with current applications.

Linux kernel 4.13 introduces in-kernel TLS encryption/decryption support [59], thus speeding up TLS applications. This feature, however, does not suit the threat model of LibSEAL: we consider the OS kernel to be untrusted, and SGX enclaves do not support the secure execution of kernel code.

#SGX threads	Throughput (req/sec)	Latency (ms)	%CPU
1	593	152	216
2	1,172	179	325
3	1,722	160	400
4	1,516	119	400

Table 3: Asynchronous enclave calls when varying the number of SGX threads (48 lthread tasks per thread)

#lthread tasks	Throughput (req/sec)	Latency (ms)	%CPU
12	1,710	184	400
24	1,701	161	400
36	1,711	166	400
48	1,722	160	400

Table 4: Asynchronous enclave calls when varying the number of lthread tasks per thread (3 SGX threads)

8 CONCLUSIONS

To enable service providers to differentiate their offerings in terms of integrity assurances, we have created LibSEAL, a new SEcure Audit Library that can identify and provide indisputable proof about integrity violations. LibSEAL acts as a drop-in replacement for a TLS library and creates an audit log of service operations. LibSEAL stores logs in a relational database, protected by a TEE in modern CPUs, and checks invariants expressed as SQL queries to discover violations. We implemented LibSEAL using Intel SGX and showed that it is effective at identifying integrity violations for three popular services with a low performance overhead.

Source code availability. LibSEAL is open source and can be downloaded from https://github.com/lsds/LibSEAL.

REFERENCES

[1] JabberD 2.x Project. 2017. JabberD 2.x. http://jabberd2.org/. (2017).

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS '05)*. ACM.

[3] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, 229–240.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*.

[5] Apache Foundation. 2017. Apache Commons IO. https://commons.apache.org/proper/commons-io/. (2017).

[6] Apache Foundation. 2017. Apache Commons Lang. https://commons.apache.org/proper/commons-lang/. (2017).

[7] Apache Foundation. 2017. Apache Commons Math. http://commons.apache.org/proper/commons-math/. (2017).

[8] Apache Foundation. 2017. Apache Commons Validator. https://commons.apache.org/proper/commons-validator/. (2017).
[9] Apache Foundation. 2017. Apache Groovy. http://groovy-lang.org/. (2017).
[10] Apache Foundation. 2017. Apache Sling. https://sling.apache.org/. (2017).
[11] Apache Foundation. 2017. HTTP server project. (2017).
[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI' 16, Savannah, GA, USA, November 2-4, 2016.*
[13] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Khran, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. *LibSEAL: Revealing Service Integrity Violations Using Trusted Execution.* Technical Report. Imperial College London. https://www.doc.ic.ac.uk/research/technicalreports/2018
[14] Barney, Blaise. 2016. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/. (2016).
[15] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 8:1–8:26 pages.
[16] Simon Bradshaw, Christopher Millard, and Ian Walden. 2011. Contracts for Clouds: Comparison and Analysis of the Terms and Conditions of Cloud Computing Services. *International Journal of Law and Information Technology (IJLIT '11)* 19, 3 (2011), 187.
[17] Ajay Brahmakshatriya, Piyus Kedia, Derrick Paul McKee, Pratik Bhatu, Deepak Garg, Akash Lal, and Aseem Rastogi. 2017. An Instrumenting Compiler for Enforcing Confidentiality in Low-Level Code. *arXiv* (2017).
[18] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (April 2017), 16:1–16:33 pages.
[19] Dell Cameron and Kate Conger. 2017. GOP Data Firm Accidentally Leaks Personal Details of Nearly 200 Million American Voters. https://gizmodo.com/gop-data-firm-accidentally-leaks-personal-details-of-ne-1796211612. (June 2017).
[20] Scott A Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS '17)*. ACM.
[21] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. 2010. How to Enhance Cloud Architectures to Enable Cross-Federation. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD '10)*. 337–345.
[22] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 253–264.
[23] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *ACM SIGOPS Operating Systems Review (OSR '07)*, Vol. 41. ACM, 189–204.
[24] CNN. 2013. PayPal Accidentally Credits Man $92 Quadrillion. https://edition.cnn.com/2013/07/17/tech/paypal-error/index.html. (July 2013).
[25] Cox, JosephTroy Hunt. 2016. Another Day, Another Hack: Tens of Millions of Neopets Accounts. https://motherboard.vice.com/en_us/article/neopets-hack-another-day-another-hack-tens-of-millions-of-neopets-accounts. (2016).
[26] Cryptomator. 2017. Cryptomator. https://cryptomator.org. (2017).
[27] cURL project. 2017. libcurl - the Multiprotocol File Transfer Library. https://curl.haxx.se/libcurl/. (2017).
[28] CVE 2005. Buffer Overflow In the Jabber 2.x Server (Jabberd), CVE-2004-0953. https://www.cvedetails.com/cve/CVE-2004-0953/. (2005). Accessed: 05-02-2018.
[29] Tim Dierks and Eric Rescorla. 2008. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. (2008). https://tools.ietf.org/html/rfc5246
[30] Dirk Einecke. 2011. HTTP Headers. https://chrome.google.com/webstore/detail/http-headers/hplfkkmefamockhligfdcfgfnbcdddbg. (2011).
[31] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. 2013. Benchmarking Personal Cloud Storage. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. ACM, 205–212.
[32] Dropbox. 2017. About Dropbox. https://www.dropbox.com/about. (2017).
[33] Dropbox. 2017. Dropbox Terms of Service. https://www.dropbox.com/en_GB/privacy#terms. (2017).
[34] Facebook. 2017. Facebook. https://www.facebook.com/. (2017).
[35] Jon Fingas. 2014. Dropbox Bug Wipes Some Users' Files From the Cloud. https://www.engadget.com/2014/10/13/dropbox-selective-sync-bug/. (Oct. 2014).
[36] Fox-Brewster, Thomas. 2015. Gambling Darling Paysafe Confirms 7.8 Million Customers Hit In Epic Old Hacks. https://goo.gl/xXpkJH. (2015).
[37] Fox-Brewster, Thomas. 2017. Massive WWE Leak Exposes 3 Million Wrestling Fans' Addresses, Ethnicities And More. (July 2017). https://goo.gl/rMyomS
[38] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford University. Advisor(s) Boneh, Dan. AAI3382729.
[39] Git 2018. https://git-scm.com/. (2018).
[40] GitHub. 2017. GitHub. https://www.github.com. (2017).
[41] GitHub. 2017. GitHub Terms of Service. (2017). https://help.github.com/articles/github-terms-of-service/
[42] GitLab.com. 2017. About GitLab. https://about.gitlab.com/. (2017).
[43] GNU project. 2017. The GNU Privacy Guard. https://gnupg.org/. (2017).
[44] Google. 2011. Gmail. https://www.engadget.com/2011/02/27/gmail-accidentally-resetting-accounts-years-of-correspondence-v/. (Feb. 2011).
[45] Google. 2017. Google Docs. https://docs.google.com. (2017).
[46] Google. 2017. Google Drive. https://drive.google.com. (2017).
[47] Google. 2017. Google Terms of Service. https://www.google.com/intl/en/policies/terms/. (2017).
[48] Gough, Valient. 2017. EncFS: an Encrypted Filesystem for FUSE. https://vgough.github.io/encfs/. (2017).
[49] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security '17)*. USENIX Association.
[50] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, 175–188.
[51] Kun He, Jing Chen, Ruiying Du, Qianhong Wu, Guoliang Xue, and Xiang Zhang. 2016. Deypos: Deduplicatable Dynamic Proof of Storage for Multi-User Environments. *IEEE Trans. Comput.* 65, 12 (2016), 3631–3645.
[52] Sean Hollister. 2017. Gmail Accidentally Resetting Accounts, Years of Correspondence Vanish Into the Cloud? https://www.engadget.com/2011/02/27/gmail-accidentally-resetting-accounts-years-of-correspondence-v/. (2017).
[53] Chris Hourihan and Bryan Cline. 2012. A Look Back: U.S. Healthcare Data Breach Trends. https://hitrustalliance.net/content/uploads/2014/05/HITRUST-Report-U.S.-Healthcare-Data-Breach-Trends.pdf. (2012).
[54] Hunt, Troy. 2016. Dating the Ginormous MySpace Breach. https://www.troyhunt.com/dating-the-ginormous-myspace-breach/. (2016).
[55] Gwan-Hwan Hwang and Shih-Kai Fu. 2016. Proof of Violation for Trust and Accountability of Cloud Database Systems. In *The 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '16)*. IEEE, 425–433.
[56] Intel. 2014. Software Guard Extensions Programming Reference, Ref. 329298-002US. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (2014).
[57] Intel. 2016. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk. (2016).
[58] Intel. 2017. *Intel SgxSSL library - User Guide.* Technical Report.
[59] Jake Edge. 2018. TLS in the Kernel. https://lwn.net/Articles/666509/. (2018). Accessed: 31-01-2018.
[60] Dhiru Kholia and Przemysław Węgrzyn. 2013. Looking Inside the (Drop) Box. In *Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT'13)*. USENIX Association, 9–9.
[61] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
[62] Selena Larson. 2017. Verizon data of 6 million users leaked online. http://money.cnn.com/2017/07/12/technology/verizon-data-leaked-online/index.html. (July 2017).
[63] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. USENIX Association, 1–14.
[64] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *OSDI '04*.
[65] lthread. 2017. lthread, a Multicore Enabled Coroutine Library Written in C. https://github.com/halayli/lthread. (2017).
[66] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage With Minimal Trust. *ACM Transactions on Computer Systems (TOCS '11)* 29, 4 (2011), 12.
[67] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. *IACR Cryptology ePrint Archive* 2017 (2017), 48.
[68] mbed TLS 2017. TLS for SGX: a Port of mbedtls. https://github.com/bl4ck5un/mbedtls-SGX. (Feb. 2017).
[69] Microsoft. 2017. Office 365. https://www.office.com/. (2017).
[70] Microsoft. 2017. Windows Azure. https://www.microsoft.com/windowsazure. (2017).
[71] Rene Millman. 2014. OneDrive Users Hit by File Corruption Bug. http://www.cloudpro.co.uk/iaas/cloud-storage/4437/onedrive-users-hit-by-file-corruption-bug. (Aug. 2014).
[72] MySpace. 2016. MySpace blog. https://myspace.com/pages/blog. (2016).
[73] S. Nepal, S. Chen, J. Yao, and D. Thilakanathan. 2011. DIaaS: Data Integrity as a Service in the Cloud. In *IEEE 4th International Conference on Cloud Computing (CLOUD '11)*. 308–315.
[74] H. Nguyen, B. Acharya, R. Ivanov, A. Haeberlen, L. T. X. Phan, O. Sokolsky, J. Walker, J. Weimer, W. Hanson, and I. Lee. 2016. Cloud-Based Secure Logger for Medical Devices. In *IEEE 1st International Conference on Connected Health:*

Applications, Systems and Engineering Technologies (CHASE '16). 89–94.
[75] OnePlus forums. 2018. Credit Card Fraud. https://forums.oneplus.net/threads/credit-card-fraud.747206/. (Jan. 2018).
[76] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference (USENIX ATC '14)*. USENIX Association, 305–319.
[77] OpenBSD Project. 2016. LibreSSL. https://www.libressl.org/. (2016).
[78] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. ACM, 238–253.
[79] ownCloud 2018. ownCloud.org. https://owncloud.org/. (2018).
[80] ownCloud Documents is Collaborative Editing of Rich-text Documents 2017. https://github.com/owncloud/documents. (2017).
[81] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. 2011. Memoir: Practical state continuity for protected modules. In *Security and Privacy (S&P '11), IEEE Symposium on*. IEEE, 379–394.
[82] PayPal 2018. PayPal. http://paypal.com/. (Feb. 2018).
[83] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. 2011. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Technical Conference (USENIX ATC'11)*. USENIX Association.
[84] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. 2014. CloudSafetyNet: Detecting Data Leakage Between Cloud Tenants. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security (CCSW '14)*. ACM, 117–128.
[85] I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram. 2013. Secure Logging as a Service – Delegating Log Management to the Cloud. *IEEE Systems Journal* 7, 2 (June 2013), 323–334.
[86] Reddit.com. 2016. Unauthorized Access Related to LinkedIn / Myspace Leaks? https://www.reddit.com/r/teamviewer/comments/4m5hle/unauthorized_access_related_to_linkedin_myspace. (2016).
[87] Will Reese. 2008. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* 2008, 173 (2008), 2.
[88] Savard, Daniel and Co, Nikolas. 2011. Live HTTP headers. https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/. (2011).
[89] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger. 2013. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In *IEEE Ninth World Congress on Services (SERVICES '13)*. 239–246.
[90] Joshua Schiffman, Hayawardh Vijayakumar, and Trent Jaeger. 2012. *Verifying System Integrity by Proxy*. Springer Berlin Heidelberg, 179–200.
[91] Simon Sharwood. 2017. GitLab.com Melts Down After Wrong Directory Deleted, Backups Fail. https://www.theregister.co.uk/2017/02/01/gitlab_data_loss/?mt=1486066707837. (Feb. 2017).
[92] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. 2017. A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems. *ACM Computing Surveys (CSUR '17)* 49, 4 (2017), 74.
[93] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-scale Distributed Systems Tracing Infrastructure*. Technical Report. Technical report, Google.
[94] Slack Project. 2017. Slack. https://slack.com/. (2017).
[95] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling With Exception-less System Calls. In *Proceedings of the 9th USENIX conference on Operating Systems Design and implementation (OSDI '10)*. USENIX Association.
[96] SQLite Project. 2017. SQLite. https://www.sqlite.org/. (2017).
[97] Squid Project. 2016. Squid Proxy. http://www.squid-cache.org/. (2016).
[98] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security '16). USENIX*.
[99] Stripe 2018. Stripe. https://stripe.com/. (Feb. 2018).
[100] Sync. 2017. Sync. https://www.sync.com. (2017).
[101] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and Justin Cappos. 2016. On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/torres-arias. In *25th USENIX Security Symposium (USENIX Security '16)*. USENIX Association, 379–395.
[102] B. Wang, B. Li, and H. Li. 2014. Oruta: privacy-preserving Public Auditing for Shared Data in the Cloud. *In Proceedings of the IEEE Transactions on Cloud Computing conference (TCC '14)* 2, 1 (Jan 2014), 43–56.
[103] B. Wang, B. Li, and H. Li. 2015. Panda: Public Auditing for Shared Data with Efficient User Revocation in the Cloud. *Proceedings of the IEEE Transactions on Services Computing (TSC '15)* 8, 1 (Jan 2015), 92–106.
[104] B. Wang, H. Li, and M. Li. 2013. Privacy-preserving Public Auditing for Shared Cloud Data Supporting Group Dynamics. In *Proceedings of the IEEE International Conference on Communications conference (ICC '13)*.
[105] C. Wang, Q. Wang, K. Ren, and W. Lou. 2010. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '10)*. 1–9.
[106] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-channel Hazards in SGX. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS '17)*. ACM, 2421–2434.
[107] Washington Post. 2017. A Mysterious Message is Locking Google Docs Users Out of Their Files. https://goo.gl/9ffD9r. (Oct. 2017).
[108] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.
[109] WhatsApp. 2018. https://www.whatsapp.com/. (Feb. 2018).
[110] WolfSSL. 2016. WolfSSL at IDF. https://www.wolfssl.com/wolfSSL/Blog/Entries/2016/8/11_wolfSSL_At_IDF.html. (August 2016).
[111] Victoria Woollaston. 2014. Has Gmail Lost YOUR Emails? Glitch Causes Thousands of Users to Accidentally Delete Messages and Report Others as Spam. http://www.dailymail.co.uk/sciencetech/article-2548010/Has-Gmail-lost-YOUR-emails-Glitch-causes-thousands-users-accidentally-delete-messages-report-spam.html. (Jan. 2014).
[112] XMPP working group. 2018. eXtensible Messaging and Presence Protocol. http://xmpp.org. (2018). Accessed: 31-01-2018.
[113] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy (S&P '15)*. IEEE, 640–656.
[114] S. Zawoad, A. K. Dutta, and R. Hasan. 2016. Towards Building Forensics Enabled Cloud Through Secure Logging-as-a-Service. *IEEE Transactions on Dependable and Secure Computing (TDSC '16)* 13, 2 (March 2016), 148–162.
[115] Qingji Zheng and Shouhuai Xu. 2012. Secure and Efficient Proof of Storage with Deduplication. In *Proceedings of the 2nd ACM conference on Data and Application Security and Privacy (CODASPY '12)*. ACM, 1–12.

A SERVICES INTEGRITY INVARIANTS

This section provides details about the Dropbox and ownCloud invariants, as mentioned in §6.2.

A.1 ownCloud

Log schema. As described in §6.2, updates to ownCloud documents are recorded in a single relation. Relation

```
owncloud(time,docid,sessionid,sndr,rcvr,upd,file_hash)
```

indicates that at logical time `time`, entity `sndr` sent update `upd` about document `docid` to entity `rcvr` with the session identified by `sessionid`. For every tuple within `owncloud`, either `sndr` or `rcvr` refers to the ownCloud server. As values for `sndr` and `recvr`, we leverage the integer values that ownCloud uses to identify its clients. We represent the server by special ID 0. Field `upd` refers to an ordered list of JSON tuples that insert or remove text or annotations. For requests that contain a snapshot sent to the server, `file_hash` contains the hash over the snapshot's content. The same hash is sent and logged as part of responses to new clients that reference the latest snaphsot. Since update messages are always exchanged between the server and one of its clients—but never directly between clients—for each tuple within relation `owncloud` either the value of `sndr` or `rcvr` is 0.

Invariants. On the basis of the above log schema we specify the invariants described in §6.2. The first invariant ensures that snapshots received by new clients match the latest snapshot for the document.

```
SELECT time,docid,sessionid,sndr,rcvr
FROM owncloud o
WHERE file_hash != '' AND
o.sndr = 0
file_hash != (SELECT COALESCE(file_hash, '') FROM owncloud
          WHERE docid = o.docid
```

```
          AND rcvr = 0
          AND file_hash != ''
          ORDER BY time DESC LIMIT 1);
```

The second invariant ensures that document updates sent by the service to a client match a prefix of the aggregated history of updates to the document previously received from all clients.

```
SELECT time,docid,sessionid,sndr,rcvr
FROM owncloud o
WHERE sndr = 0 AND
(SELECT GROUP_CONCAT(upd,'') FROM
    (SELECT docid,upd FROM owncloud
     WHERE docid = o.docid
     AND sessionid = o.sessionid
     AND rcvr = 0 AND time <= o.time
     AND time > (SELECT COALESCE(MAX(time), 0) FROM owncloud
                 WHERE sndr = 0 AND rcvr = o.rcvr
                 AND time < o.time)
    ORDER BY time DESC)
    GROUP BY docid)
NOT LIKE
upd || '%';
```

Trimming. We define two trimming queries for ownCloud. The first query ensures that only data for the most recent session is kept. It also removes all but the latest update sent by the server for each user. The second query ensures that only the most recent snapshot hash of each document is kept.

```
DELETE FROM owncloud
WHERE
EXISTS(SELECT sessionid FROM owncloud o2
       WHERE docid = o2.docid AND
             sessionid != o2.sessionid AND
             time < o2.time)
OR (o.sender = 0 AND o.time != (
       SELECT MAX(time) FROM owncloud o2
       WHERE docid = o2.docid AND
             sessionid = o2.sessionid AND
             rcvr = o2.rcvr))

DELETE FROM owncloud
WHERE file_hash != ''
AND rcvr = 0
AND time != (SELECT MAX(time) FROM owncloud o2
       WHERE docid = o2.docid
       AND o2.rcvr = 0
       AND 02.file_hash != '')
```

A.2 Dropbox

Log schema. §6.2 presents the log schema for Dropbox.

Invariants. The first invariant for Dropbox verifies whether relations `commit_batch` and `list` present a consistent set of files as described in §6.2:

```
SELECT account, file, time, (SELECT MIN(time)
                             FROM commit_batch
                             WHERE size = -1
                             AND account = c.account
                             AND file = c.file
                             AND time > c.time) AS deltime
FROM commit_batch c
WHERE c.size >= 0
    AND EXISTS (SELECT DISTINCT host FROM list l
        WHERE account = c.account AND time > c.time
        AND (time < deltime OR deltime IS NULL)
        AND NOT EXISTS (SELECT 1 FROM list
            WHERE time > c.time
            AND (time < deltime OR deltime IS NULL)
            AND l.host = host AND c.file = file));
```

The second invariant verifies whether for each file being served to the client, the announced blocklist is correct.

```
SELECT account, file, time, (SELECT MAX(time)
                             FROM commit_batch
                             WHERE account = l.account
                             AND file = l.file
                             AND time < l.time) AS
                                 lastupdate
FROM list l
WHERE NOT EXISTS (SELECT 1 FROM commit_batch
    WHERE account = l.account
    AND file = l.file
    AND time = lastupdate
    AND size = l.size
    AND blocks = l.blocks);
```

Trimming. We define two trimming queries to reduce the number of entries in the database. The first one keeps only the most recent `list` entries for each file and host while the second one keeps only the most recent `commit_batch` entries for each file:

```
DELETE FROM list WHERE
      time != (SELECT MAX(time) FROM list l2
            WHERE host = l2.host AND file = l2.file);

DELETE FROM commit_batch WHERE
    time != (SELECT MAX(time) FROM commit_batch c2
          WHERE account = c2.account AND file = c2.file);
```