SecureKeeper: Confidential ZooKeeper using Intel SGX

Stefan Brenner
TU Braunschweig, Germany
brenner@ibr.cs.tu-bs.de

Colin Wulf
TU Braunschweig, Germany
cwulf@ibr.cs.tu-bs.de

David Goltzsche
TU Braunschweig, Germany
goltzsche@ibr.cs.tu-bs.de

Nico Weichbrodt
TU Braunschweig, Germany
weichbr@ibr.cs.tu-bs.de

Matthias Lorenz
TU Braunschweig, Germany
mlorenz@ibr.cs.tu-bs.de

Christof Fetzer
TU Dresden, Germany
christof.fetzer@tu-dresden.de

Peter Pietzuch
Imperial College London, UK
prp@imperial.ac.uk

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

ABSTRACT

Cloud computing, while ubiquitous, still suffers from trust issues, especially for applications managing sensitive data. Third-party coordination services such as ZooKeeper and Consul are fundamental building blocks for cloud applications, but are exposed to potentially sensitive application data. Recently, hardware trust mechanisms such as Intel's Software Guard Extensions (SGX) offer trusted execution environments to shield application data from untrusted software, including the privileged Operating System (OS) and hypervisors. Such hardware support suggests new options for securing third-party coordination services.

We describe *SecureKeeper*, an enhanced version of the ZooKeeper coordination service that uses SGX to preserve the confidentiality and basic integrity of ZooKeeper-managed data. SecureKeeper uses multiple small enclaves to ensure that (i) user-provided data in ZooKeeper is always kept encrypted while not residing inside an enclave, and (ii) essential processing steps that demand plaintext access can still be performed securely. SecureKeeper limits the required changes to the ZooKeeper code base and relies on Java's native code support for accessing enclaves. With an overhead of 11%, the performance of SecureKeeper with SGX is comparable to ZooKeeper with secure communication, while providing much stronger security guarantees with a minimal trusted code base of a few thousand lines of code.

CCS Concepts

•**Security and privacy** → *Distributed systems security;*

Keywords

Cloud Computing, Intel SGX, Apache ZooKeeper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware'16, December 12 - 16, 2016, Trento, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4300-8/16/12…$15.00

DOI: http://dx.doi.org/10.1145/2988336.2988350

1. INTRODUCTION

Cloud computing has become ubiquitous due to its benefits to both cloud customers and providers [1]. Using public cloud resources, however, requires customers to fully trust the provided software and hardware stacks as well as the cloud administrators. This forms an inhibitor when sensitive data must be processed [2, 3].

With Software Guard Extensions (SGX), Intel recently released a new technology [4] for addressing trust issues that customers face when outsourcing services to off-site locations. Based on an instruction set extension, it allows the creation of one or more trusted execution environments—called *enclaves*—inside applications. Thereby, the plaintext of enclave-protected data is only available for computation inside the CPU, and it is encrypted as soon as the data leaves the CPU. This way enclave-residing data is even guarded against unauthorized accesses by higher privileged code and attackers with administrative rights and physical access.

While enclaves were originally designed to host tailored code for specific tasks, e.g. digital rights or password management [5], Baumann et al. [6] proposed Haven that executes unmodified legacy Windows applications inside enclaves. These *application enclaves* are convenient for securing entire legacy applications but have two significant drawbacks: (i) current SGX-capable CPUs restrict the maximum memory footprint of all enclaves to 128 MB. If an application has a larger memory footprint, an SGX-specific form of in-memory paging between trusted and untrusted memory is required, which requires costly re-encryption of enclave pages with a substantial performance overhead; and (ii) by placing whole applications and associated system support inside an enclave, the resulting large trusted code base (TCB) poses a risk of including security relevant vulnerabilities [7].

Based on these observations, we argue that application enclaves that contain entire applications are a poor fit for legacy services if tailored solutions are easy to implement and integrate. In particular, this applies to existing *data-handling* services that receive, store and return data on behalf of clients while performing limited data processing. Examples include simple key-value stores [8], web servers [9–11], and, as highlighted in this paper, coordination services such as ZooKeeper [12]. For such services, data confidentiality and basic integrity can be preserved using small specialized enclaves, which can be integrated with the original code base with only few changes.

© ACM, 2016. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the proceedings of the 17th ACM Middleware conference (2016). http://dx.doi.org/10.1145/2988336.2988350

ZooKeeper is a crucial building block for many distributed applications, e.g. used as naming service, for configuration management, general message exchange and coordination tasks. It therefore manages sensitive application information, i.e., including access tokens and credentials when used for configuration management. Albeit typically not user facing, ZooKeeper constitutes a key component in public cloud infrastructures that must be secured against attacks, especially data theft carried out by insiders.

As a data-handling service, ZooKeeper features a simple file system like API, with some data processing functionality. Thus, data managed by ZooKeeper can be processed in an encrypted state most of the time and only occasionally plaintext access is necessary. This functionality needs to be factored out and executed under the protection of SGX. In contrast to application enclaves, this way all data handling code that does not perform data processing is removed from the TCB. For ZooKeeper, this includes the entire Operating System (OS), the Java runtime environment as well as most of the original service implementation itself. Furthermore, as data stays encrypted most of the time, it can be stored outside of enclaves, only adding to the memory footprint of the enclave when required. This leads to substantially smaller enclaves, staying within the bounds of the 128 MB limit of total enclave memory resulting in good performance.

We describe **SecureKeeper**, an SGX-based ZooKeeper extension that demonstrates how a complex data-handling service can be secured using SGX. SecureKeeper preserves confidentiality and provides basic integrity of service data, even against privileged code and attackers with physical access. These security guarantees are enforced through a minimally-invasive integration of two enclave types: (i) an *entry enclave* is instantiated for each client and responsible for protecting the client-replica connection and the data security of the ZooKeeper data store; (ii) a *counter enclave* is instantiated only once at the leader replica of the ZooKeeper cluster, and handles special write requests of *sequential nodes* that perform data processing. Clients do not need to know the key used by enclaves for encryption towards the ZooKeeper data store. This also allows the exclusion of clients due to misbehaviour.

Based on this design, the remaining ZooKeeper code base and any system software can be excluded from the TCB. SecureKeeper therefore features a small TCB of 3795 Source Lines of Code (SLOCs) apart from the trusted system support provided by the Intel SGX SDK [13]. Besides its small TCB, SecureKeeper only changes *three* lines of the original ZooKeeper code base, making it easily upgradeable to future ZooKeeper versions. The memory demand of the enclaves stays within small bounds, which avoids costly SGX-specific paging with a high client load.

We compare SecureKeeper to ZooKeeper with and without secure communication between replicas and clients on a cluster of SGX-capable Intel Skylake machines. SecureKeeper only adds an average performance overhead of 11.2% to ZooKeeper when secure communication is enabled, while enforcing stronger security guarantees.

In the next section, we describe the background and design considerations that led to the design of SecureKeeper. After that, we introduce its design in Section 4, followed by implementation details in Section 5. We present evaluation results in Section 6 and discuss the obtained security guarantees and limitations in Section 7. Finally, we discuss related work in Section 8 and draw conclusions in Section 9.

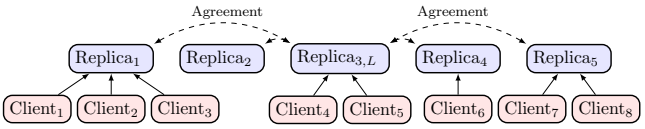


Figure 1: General architecture of ZooKeeper.

2. BACKGROUND

To motivate the design of SecureKeeper, we first give a brief introduction to Apache ZooKeeper thereby discussing its nature as a data-handling service, second we provide essential details of SGX and finally outline an SGX-aware threat model.

2.1 Data handling in Apache ZooKeeper

Apache ZooKeeper [12] is a coordination service, allowing distributed applications the easy implementation of coordination primitives. Such primitives may be naming, configuration management, leader election, group membership, barriers and distributed locks. This exposes ZooKeeper to handle potentially sensitive application data, especially when used for distributed configuration management.

As shown in Figure 1, ZooKeeper is implemented as a fault-tolerant service, typically featuring at least three replicas. Clients connect to one of these and switch over to another in case it crashes. ZooKeeper-managed data is organized hierarchically as a tree that is accessed via a simple file system like Application Progamming Interface (API). This data tree is composed of *znodes*, resembling a mixture of folders and files: znodes can have payload data and other child znodes at the same time.

ZooKeeper globally orders all *write requests* via the leader replica (marked L in Figure 1) using the ZAB [14] agreement protocol. Additionally, ZooKeeper guarantees FIFO ordering of *all* requests of *one* client, allowing replicas to answer read requests from clients connected to them directly.

Most operations provided by ZooKeeper are dedicated towards data handling: In order to retrieve and change the payload of a znode, the `GET` and `SET` operations are used. New znodes can be created using the `CREATE` operation, and deleted via `DELETE`. Also it is possible to access a list of children of a specific znode using the `LS` (getChildren) operation.

An exception to the data handling nature of ZooKeeper builds a variant of the `CREATE` operation that instantiates a *sequential node*. Here data-centric processing is required: If a new sequential node is created, a sequence number is added to the user-provided znode name. This sequence number is monotonically increasing and managed by the parent znode, thus the name of the new znode depends on the internal state of the parent znode.

Since all operations are executed affecting one specific znode in the data tree of ZooKeeper, request messages will usually contain a path to that znode. The payload of a znode will usually be contained in the `GET` response, and the `SET` and `CREATE` requests. The `LS` operation is the only operation that contains paths in the response message – the paths of all child znodes. Application data managed by ZooKeeper comprises the payload as well as the pathnames of znodes. In many cases, the pure existence of a certain path steers processing in a distributed application.

In summary, ZooKeeper primarily performs data handling for clients with the exception of sequential nodes, were actual data processing takes place. Furthermore, payload and path information are relevant to security considerations.

2.2 Intel SGX in a nutshell

SGX [4] is a processor instruction set extension that allows the creation of so called *enclaves*. The enclave memory range Enclave logical range (ELRANGE) is an isolated range inside an application's address space which is confidentiality and integrity protected by the CPU package. Pages of the ELRANGE are backed by the Enclave Page Cache (EPC): a reserved range of system memory of at most 128 MB, used to store all enclave pages.

Basic protection model. SGX prevents to `jmp` or directly `call` to the ELRANGE, instead the enclave must be entered explicitly. Once inside the enclave, the CPU can execute arbitrary code, except for a few instructions that are prohibited inside enclaves. The enclave is allowed to access data outside the ELRANGE, however, it cannot execute untrusted code. Even privileged code is prevented from reading and altering enclave memory: illegal accesses result in CPU exceptions and read access to enclave memory will always return `0xFF`.

Confidentiality of enclave memory is protected by transparent memory encryption done by the CPU, i.e., plaintext is only available inside the CPU package. Valid interaction with an enclave is possible via Enclave calls (ecalls) to enter an enclave and Outside calls (ocalls) to call out of the enclave. An ecall comprises an explicit entering of the enclave and switching to a trusted stack located inside the ELRANGE, as well as moving call parameters into the enclave and calling the according trusted function's code. Returning from an ecall and ocall, equally, requires switching the stack back, moving return values or parameters out of the ELRANGE and an explicit enclave exit.

Memory management. If enclave applications do not fit into the 128 MB of EPC, a special form of EPC-paging must be done. Enclave pages residing in the EPC can be moved out to normal system RAM, similar to paging memory to disk. However, as enclave pages are encrypted by the CPU, first a re-encryption happens to protect the contents of the pages removed from EPC. Then, the page can be migrated to normal RAM and afterwards, is qualified for being paged to disk if required. The procedure of EPC paging involves a mechanism that prevents replaying paged-out enclave pages by generating a so called "version array" that is required to migrate the page back to EPC later and only works once. EPC paging allows enclave applications to use more than 128 MB of memory, however, implies a significant performance impact as we discuss later.

System limitations. As enclaves can be only operated in user space, SGX-based applications still depend on the underlying operating system to cooperate. For example, memory management of the EPC memory is done completely by the untrusted operating system. Hence, SGX naturally cannot prevent denial of service attacks by privileged code. The in-memory state of enclaves is protected against replay attacks, as it is managed by SGX, even if paged out to normal system memory. For persistent state and in the context of application or OS restarts no specific protection is provided.

Intel SDK provided system support. Beside SGX-aware hardware, additional system software support for lifecycle management of enclaves and memory management to handle the EPC is required to operate SGX-enabled applications. Intel offers an Intel SGX SDK to handle these tasks, and on top, provides an interface definition language called Enclave Description Language (EDL) that allows the specification of the ecalls and ocalls an enclave should support. The Intel SGX SDK also comprises a tool called "Edger8r" which is a code generator supporting developers on the recurring task of writing stubs for ecalls and ocalls as well as parameter passing between the enclave and the untrusted code. Enclaves are finally compiled and linked together with the generated code to a shared object, which is loadable by the Intel SGX SDK as an enclave.

2.3 An SGX-aware Threat Model

We assume a typical threat model for SGX enclaves, specifically in an untrusted cloud environment [6, 15]: an attacker has full – even physical – control over the server hardware and software environment. This includes that the attacker can control the OS and all code invoked prior to the transfer of control into the SGX enclave. The attacker's goal is to break confidentiality or integrity of code running in the SGX enclave.

Availability threats, such as crashing an enclave, are not of interest: by design, the hosting OS is able to stop execution of enclaves at any time. Nevertheless, from a fault-tolerance point of view, SecureKeeper – analogous to ZooKeeper, can tolerate a limited number of node crashes.

We do not consider side-channel attacks [16] and assume that enclaves do not possess any security relevant vulnerabilities that may lead to leakage of data or breach its integrity. The latter can be addressed by applying software-hardening techniques that are orthogonal to our approach. We also fully trust the design and correct implementation of the CPU package and the SGX instruction set extension including all cryptographic operations done by SGX.

Finally, clients can read, modify and delete service-provided data and therefore are considered to be trusted in this regard.

3. DESIGN CONSIDERATIONS

The design of SecureKeeper is determined by (i) security considerations but also (ii) performance and (iii) applicability aspects play an important role.

Two basic design variants are compared: using an *application enclave*, which hosts whole ZooKeeper instances, and *tailored enclaves*, as featured by SecureKeeper, used to only protect ZooKeeper-specific sensitive data and its processing.

Application enclaves are attractive for complex legacy applications because they can instantly profit from the SGX-provided security guarantees without additional changes. As a downside, such large enclaves pose a security trade-off (Section 3.2) and may lead to substantial performance degradation (Section 3.3).

3.1 Protecting ZooKeeper data

ZooKeeper as a coordination service is the backbone of many distributed applications, storing essential and sensitive data for their operation. As detailed in Section 2.1, this data is split into path and payload tuples, stored in a tree-like hierarchical database. At least the plaintext of the

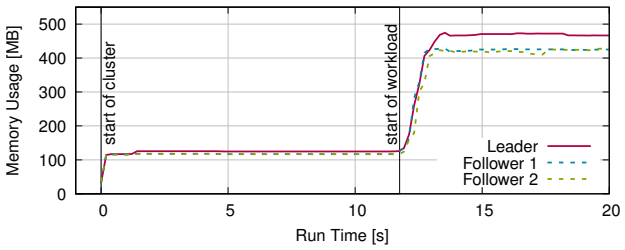


Figure 2: Memory usage of ZooKeeper over time.

path and the associated payload information must always be inaccessible to external entities to enforce confidentiality and additionally guarded to ensure integrity. Thus, this data either needs to be put under the protection of an enclave or stored and processed in an encrypted form. The former is straightforward when using application enclaves, the latter is possible for most core functions with the exception of sequential nodes that require local processing (see Section 2.1). As a consequence, SecureKeeper takes a middle ground and stores and processes data in encrypted form where feasible and redirects plaintext processing to enclaves (see Section 4).

3.2 Security considerations

Since ZooKeeper is implemented in Java, an application enclave software stack would comprise the library OS, an entire Java Virtual Machine (JVM) and the whole ZooKeeper codebase. The library OS of Haven alone comprises millions of code lines [6]. In addition, OpenJDK itself is composed of roughly 4.5 million SLOCs, and was subject to several remote exploitable vulnerabilities in the recent past [17–20]. Furthermore, ZooKeeper is a complex service with several thousand lines of code (see Section 6.4). From a security point of view, such a large TCB is at odds with basic security considerations [7], and therefore a tailored enclave approach, as implemented by SecureKeeper, is more attractive due to its smaller TCB.

In addition, the number of possible ecalls and ocalls composing the enclave interface is of security relevance. Fewer interface functions with a limited number of well defined parameters decrease the attack surface of the enclave. For a hosted system with an application enclave, the application-specific user interface calls and the ocalls that implement system call functionality must be protected. The latter are generic, thus hard to validate, and they have to be guarded by additional checks [6]. In case of tailored enclaves, the interface depends on the provided functionality. For SecureKeeper, we aim for a narrow and minimal interface, enabling a targeted and context-specific parameter validation.

3.3 Resource and performance considerations

Another important aspect is the memory consumption of the JVM and ZooKeeper. As mentioned in Section 2.2, the EPC size is limited to 128 MB, and memory requirements beyond this amount require costly paging from secure SGX memory to untrusted memory after reencryption of the data. Figure 2 shows the memory usage of an unmodified ZooKeeper instance. For this measurement, we applied a realistic workload according to the original ZooKeeper paper [12] to our test cluster which is described in Section 6: asynchronous `GET` and `SET` requests with a 70:30

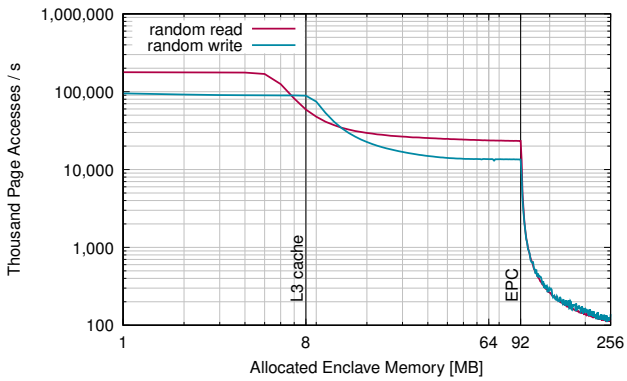


Figure 3: Performance impact of enclave memory size on random reads and writes on a Xeon E3-1230 v5 CPU.

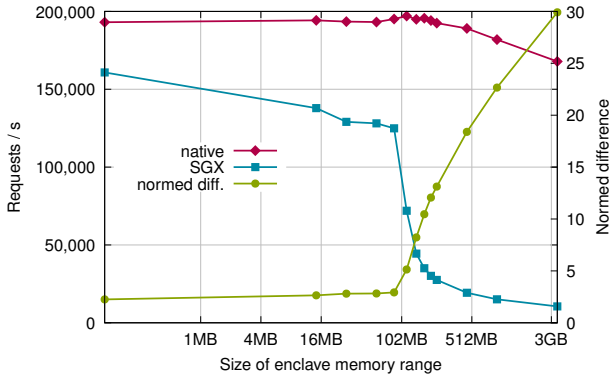


Figure 4: Performance of a key-value store in an enclave for a randomized request pattern.

ratio. The measurement proves that a standard ZooKeeper instance easily exceeds the EPC memory limit, even in idle state. After starting the above workload, the memory usage of ZooKeeper increases significantly. Even though we only add four standard-sized nodes to the initially empty ZooKeeper instance and repeatedly call `GET` and `SET` using four clients, the memory demand rises to more than 400 MB.

To form an intuition regarding the impact of EPC paging, we measure the performance of random reads and writes with variable enclave sizes. Figure 3 shows the performance of accessing single bytes from random pages of a large buffer allocated inside the enclave. By measuring the maximum possible page accesses per second, we find two turning points, as visible in the figure: the first when reaching the L3 cache limit at 8 MB; the second when exceeding the limit of the EPC. As can be seen from the figure, users cannot use the complete 128 MB of the EPC, which we assume is due to the overhead of SGX management data structures. Overall, the performance decreases 5.5× when exceeding the L3 cache, and decreases another 200× when the EPC is exhausted, i.e. the paged-EPC is more than 1000× slower than the L3 cache.

We also implemented a key-value store (KVS) that we execute inside an enclave to obtain a more realistic intuition regarding the performance impact of EPC paging on a real application. We increase the size of the enclave that runs the KVS and limit the maximum number of key-value pairs stored. We then measure the throughput of the KVS by randomly accessing key-value pairs from a remote machine and

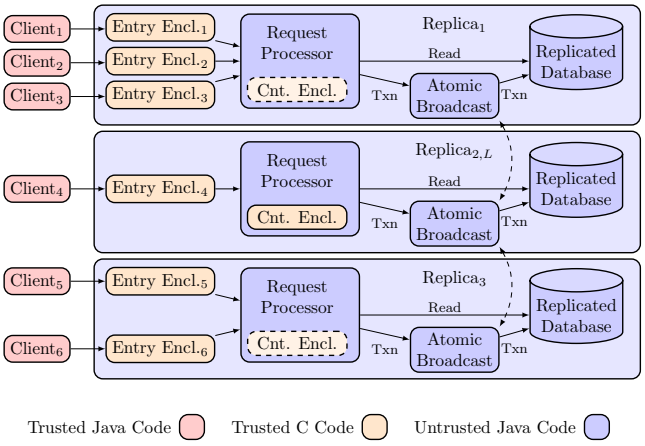


Figure 5: Architecture of SecureKeeper.

find a similar behavior as previously in Figure 3. Essentially, this leads to random memory access to the enclave memory range, which is indicated on the x-axis of Figure 4. As soon as the size of the enclave holding the KVS reaches the EPC limit and the EPC paging starts, the performance drops significantly. The measurements of this experiment are shown in Figure 4, starting at 102 MB a huge performance impact can be seen, which is also visualized as normalized difference between native and SGX in the figure. We explain the difference to Figure 3 with pages that were initially allocated (e.g., stack pages) but never used and after being paged out once, never paged in again.

In summary, EPC paging starts with an enclave memory size larger then 92 MB and reduces the performance of code running inside an enclave significantly. Additionally, an unmodified ZooKeeper instance uses 120 MB in idle state and more than 400 MB of RAM for a realistic workload on a small data set of znodes. These insights support our design decision to implement SecureKeeper using tailored enclaves and only move critical code sections into enclaves as well as aiming at a low memory footprint when processing data.

3.4 Execution Environment Considerations

As our approach uses SGX enclaves, integrated via Java Native Interface (JNI) into ZooKeeper, SGX support is required on all replicas. In addition, access to so called architectural enclaves from the Intel SGX SDK and the Intel SGX driver is required in order to successfully launch and attest enclaves. The use of hardware virtualization technology is orthogonal to using SGX.

4. SECUREKEEPER

Next, we present the design of SecureKeeper that is driven by the earlier outlined security considerations and the goal to craft an efficient solution taking advantage of current SGX-equipped CPUs.

4.1 Basic architecture of SecureKeeper

Figure 5 illustrates how our design integrates into the original ZooKeeper architecture. For each client we have an individual enclave running on the replica as long as the client is connected to which we call *entry enclave*. This enclave maintains the client connection and the encryption of all messages between the client and the enclave.

As we did not want huge changes at the client side, we designed the client-to-replica communication as a simple connection encryption alike Transport Layer Security (TLS). TLS encrypted connections have recently been added to ZooKeeper, thus, the client infrastructure for this already exists. However, for SecureKeeper the endpoint of this secure connection is located inside the entry enclave. The entry enclave uses standard cryptographic functions to decrypt the messages from and to the client with a client-specific session key. Only if this enclave has been remotely attested by the SecureKeeper administrator, the secure key for all encryption towards ZooKeeper is given to the enclave. This key is the same for all enclaves and also the only state held by the enclaves apart from the session key for the client connection. Vice versa if a client can establish a secure connection based on a previously out-of-band received public key, he can trust the entry enclave.

Since the entry enclave exists once per client, there is no need for the enclave to distinguish multiple clients, which helps keeping the enclave's code base low. However, it needs to be able to "understand" the messages exchanged, i.e., be able to serialize and deserialize them. This is required in order to encrypt the sensitive fields of a message towards ZooKeeper with a secret key shared by all enclaves.

Our approach also allows to exclude specific, previously trusted, clients from the cluster in the future, as the clients never see the encryption key used by the enclaves to protect all data stored in the ZooKeeper data store.

In order to support the demands of the aforementioned sequential nodes (see Section 2.1), we designed the *counter enclave*. Even though this enclave is only used on the current ZooKeeper leader replica, it exists once on each replica. This is due to the fact, that on failure ZooKeeper may need to elect a new cluster leader, and thus, a previous follower replica may become new leader. Inside this enclave, the encrypted path and the plaintext sequence number determined by ZooKeeper is merged together into one ciphertext. We describe the implementation details of the counter enclave in Section 4.4 and the consequences regarding our security goals in the discussion section (see Section 7).

All other components of ZooKeeper are untrusted and will always only see the encrypted path names of znodes and the encrypted payload. However, we exploit that they can handle the ciphertext as a blackbox, i.e. the same as plaintext.

4.2 General Message Processing

All messages from clients to the SecureKeeper cluster must pass through the entry enclaves for processing and re-encryption. By this, all sensitive information is encrypted before being stored in the untrusted ZooKeeper database.

We distinguish two notions of encryption: Between clients and the entry enclaves we apply *transport encryption* to the whole messages (cf. TLS). After processing the message, the entry enclaves encrypt sensitive parts towards the ZooKeeper database; we call this *storage encryption*.

In general, for all requests from a client we first decrypt the transport encryption hull. Then, we deserialize the plaintext message and determine which fields contain sensitive information. This usually affects the payload field of the znode and its path name (see Section 4.3). However, not all messages contain both: While `SET` requests always contain a path and a payload and result in an empty response, the `GET` requests contains only the znode path and the response

message will contain the respective payload. After all sensitive fields have been encrypted, we serialize the message and forward it to the untrusted ZooKeeper message processing pipeline. Special treatment is required for creating sequential nodes, involving the counter enclaves (see Section 4.4).

Responses are basically implemented the same but in reverse order: First the message gets deserialized. Then, we decrypt all fields that contain previously encrypted sensitive data. Finally, we apply the transport encryption and forward the buffer to the client via the ZooKeeper message processing pipeline.

ZooKeeper response messages do not contain the operation type of the request they belong to. Hence, in order to determine the operation of a request, we store the operation and request ID for each request. For this purpose we maintain a FIFO queue inside the entry enclaves, because the ordering of all requests of one client always guarantees that the responses arrive in the same order as the requests.

4.3 Path and Payload Encryption

ZooKeeper allows storage of arbitrary data in the payload field of each znode. Since ZooKeeper is never processing this data, it is considered as a blackbox by ZooKeeper and we can just encrypted it.

The encryption of path names of znodes must be done transparently to ZooKeeper. Firstly, the encrypted path must be a valid ZooKeeper path, i.e., it must not contain illegal characters. Also, ZooKeeper must be able to operate on the encrypted path names in all cases. Individual path elements are encrypted separately, retaining the hierarchy of znodes in order to support the `getChildren` operation.

In our design we split paths at each slash (/) and process the chunks one after another We first encrypt each chunk the same as for payload encryption. As an Initialization Vector (IV) we compute a hash of the path starting from the top level hierarchy until (and including) the current chunk. This ensures that we never reuse the same IV [1] The IV and the Keyed-Hash Message Authentication Code (HMAC) of the current chunk are both concatenated with the encrypted chunk, and form the encrypted version of this chunk.

We have to include the current chunk's plaintext in the computation of the hash that we use as IV, because otherwise all child nodes of one parent would use the same IV. However, the decryption of a path clearly requires knowledge of the used IV which is not available from the request message of `LS` requests, as we need to decrypt the paths of the children of the requested znode. Hence, we have to append the IV that we used to the encrypted chunk in order to support the `LS` operation, even though for most operations this would not be required.

We need to tie the encrypted payload of a znode to one distinct znode path. This is due to the ZooKeeper database residing in the untrusted execution environment. The attacker could mix and switch payloads and znode names arbitrarily otherwise. For example, an attacker could set the admin password (if for example stored as payload of a node `/admin-credentials`) to the password of his own non-privileged user. The binding is done by appending the hash of the znode path to the payload before payload encryption. By this, a specific payload is only valid for one specific znode path, and the entry enclave can easily validate this constraint when processing the response from ZooKeeper before forwarding it to the client.

[1] assuming no SHA256 collisions.

4.4 Supporting Sequential Nodes

During creation of a znode with the *sequential* flag set in the request message, ZooKeeper will append a monotonically increasing number to the given znode name. After the sequential node has been created, it is stored in the ZooKeeper database the same as any other node, and is indistinguishable from regular znodes. If we applied the path encryption as described above for sequential nodes, SecureKeeper would try to decrypt the whole pathname later on. In the case of sequential nodes the path decryption would fail, as it would contain the sequence number appended to the given name. If we divided the pathname from the sequential number using an escape character we would change the behaviour of ZooKeeper and violate assumptions of existing applications regarding ZooKeeper's behaviour.

In order to solve this problem, we introduce the counter enclave running once on the ZooKeeper leader replica. The counter enclave decrypts the previously encrypted pathname by the entry enclave, appends the given sequence number and encrypts the whole altered path again. This only happens during the creation of sequential nodes, all other operations will just bypass the counter enclave.

A flag in the payload field marks whether or not a node was created with the sequential flag, which is required for the verification routine. If a node was created as a sequential node, the verification must compare the hash of the znode path without the sequence number to the hash stored in the payload section. This in turn forms a problem to our binding of paths and payload: If the inputs to the counter enclave get replayed, it is possible to forge the sequence number appended to a certain node. We discuss this, and the subsequent integrity properties in the discussion in Section 7.

4.5 Deployment and Key Management

In our design, entry enclaves are not intended to communicate with each other. This reduces complexity, and thus, the TCB of the enclaves according to our security goals. In general, all entry enclaves are equivalent and use the same secret key for cryptography towards ZooKeeper. This is required in order to allow one client's entry enclave to decrypt data created by a different client's entry enclave.

The above design prevents the clients from requiring the key used for encryption towards ZooKeeper, and allows individual session keys for each client connection. However, it is important that entry enclaves verify valid clients and block any communication if the client seems malicious.

This is a common problem of all SGX applications and can be solved easily using bidirectional TLS certificate verification. In order for this to work, a special deployment and bootstrapping procedure is required. The storage encryption key will be provided to the entry enclave only after successful remote attestation [21]. Once the key is received, the correct entry enclave can seal the key and store it persistently on the replica. Other entry enclaves on the same replica can unseal this secret without another remote attestation, as only valid entry enclaves will be able to get the same sealing key. This remote attestation and sealing must be done at least once for each replica.

The TLS private key and the public key of the clients, or the Certificate Authority (CA) key that signed the clients

```
enclave {
 trusted {
  public size_t ec_request(
   [in, out, size=buffer_size] char *buffer,
   size_t msg_len, size_t buffer_size, int id
  );

  public size_t ec_response(
   [in, out, size=buffer_size] char *buffer,
   size_t msg_len, size_t buffer_size, int id
  );
 };
};
```

Listing 1: EDL interface of entry enclave.

keys must be provided to the entry enclaves in the same way. Thus, the entry enclaves can verify correct clients and prove their identity to the clients via TLS.

5. IMPLEMENTATION DETAILS

In our implementation description we describe i.) the integration of enclaves into the code base of ZooKeeper and ii.) the cryptographic operations inside the enclaves.

5.1 Enclave Integration

SecureKeeper is implemented with minimal changes to the Java code of ZooKeeper. Basically, we are intercepting the message processing pipeline (for requests and responses) of ZooKeeper and forward byte buffers of the messages to the entry enclave. The enclave then applies processing to these buffers and hands them back to the request processor where the message will walk through the remaining stages as usual. From ZooKeeper's perspective, the whole message is encrypted before entering the enclave, and is returned from the enclave in decrypted form, such that ZooKeeper is able to process the message without noticing the encrypted path and payload field. The counter enclave is integrated into the message processing pipeline in the same way but at a position that is only executed on the leader replica.

To integrate both types of our enclaves with ZooKeeper, a transition between Java and C-code is required. This is implemented using JNI which allows the transition from Java to a JNI interface written in C followed by an entry into the enclave also written in C. In the ZooKeeper code base we added a Java class that wraps all this, and offers an API that can be conveniently used from the message processing pipeline of ZooKeeper.

From the JNI code we call the enclave's ecalls which are defined by a EDL file that is used to generate the code implementing the ecalls stub code. As shown in Listing 1 the interface of the entry enclave comprises two ecalls that handle the request messages from a client and the response messages from ZooKeeper respectively. Both ecall signatures comprise a pointer to the buffer and the length of this buffer, as well as the length of the message which is different from the buffer length as we describe below. In addition to the generated code, the Intel SGX SDK libraries for cryptographic operations and a specialized `stdlib` for enclaves are linked to the final enclave shared object file. The counter enclave is only required for the creation of sequential nodes, so its EDL interface is similar but comprises only one ecall.

Byte buffers containing the transport encrypted message from the clients have to traverse the path through the JNI interface and via the generated code as an ecall to the enclave. Unfavorably the message length usually increases inside the enclave due to appending the HMAC and the path's hash to the payload and the Base64 encoding. A decrease of message length inside the enclave is not a problem, as the used buffer is larger than required. However, if the length of the message increases inside the enclave, the buffer can not trivially be copied to the outside.

Even though the Intel SGX SDK supports copying buffers in and out of enclaves, increasing a buffer inside an enclave and copying the larger buffer out is currently not supported. This is due to the fact that the enclave is not allowed to execute code outside an enclave, and thus, can not allocate an untrusted memory range outside. An allocation inside the enclave does not help, as the memory range will not be available outside of the enclave. One approach would be to allocate a large buffer outside, and allow memory management of this buffer only by the enclave.

In our case we can predict the amount of additional memory that will be required inside the enclave. Thus, we allocate a slightly larger buffer before executing the ecall, and by this, allow the enclave to append additional data to the buffer. Using an additional length parameter, we indicate the number of bytes actually used for the message in the buffer. By this approach we achieve efficient resource usage, and avoid the usage of a memory allocator for untrusted memory in the enclave at the same time.

5.2 Enclave Cryptography

For cryptographic operations inside the enclaves we use library support as provided by Intel SGX SDK. Amongst others, it supports AES-GCM-128 encryption which is suiting our purposes best, as it is considerably secure and offers additional integrity protection of the ciphertext. As we used the cipher for both, the transport and the storage encryption, all ciphertext is integrity protected by the HMAC which we always append in the end and verify during the decryption. By appending the HMAC to the ciphertext, the size of the payload field increases by a constant overhead. This requires us to update the metadata of the according znode, such that it reflects the correct size of the payload.

The concatenation of the encrypted chunk and the HMAC may contain invalid characters if interpreted by ZooKeeper as a Java String. Thus, we encode the whole chunk with a Base64 encoding scheme variant for URL applications, to also avoid the "/"-character in the encrypted path names. Using this encoding causes an increase of the encrypted pathnames of about 33%, which we discuss in Section 6.2.

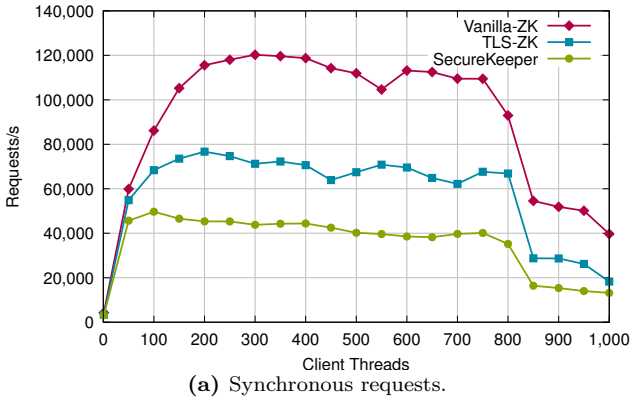
6. EVALUATION

In this section we evaluate the performance of SecureKeeper and compare it against a vanilla ZooKeeper cluster and a version that provides TLS encryption between clients and the replicas as a baseline. All experiments were performed on a cluster of four identical machines[2], where one machine simulates clients while the three remaining machines host ZooKeeper replicas.

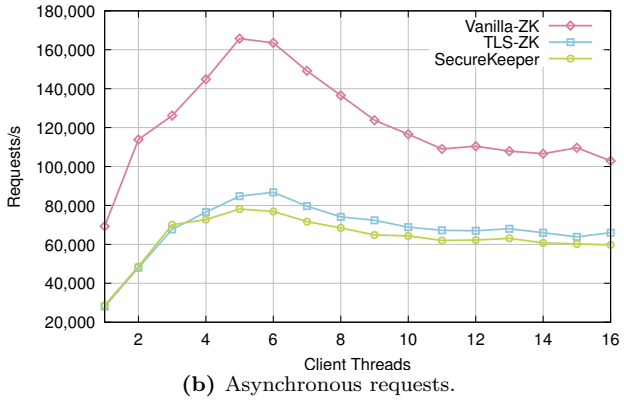
6.1 Methodology

In general, the evaluation is along the lines of the original ZooKeeper paper [12]. For each experiment we compare SecureKeeper against a vanilla ZooKeeper and ZooKeeper

[2]Core i7-6700 @3.4GHz, 24 GB RAM, 256 GB SSD, 4x GbE



(a) Synchronous requests.



(b) Asynchronous requests.

Figure 6: Throughput of 70:30 mixed GET and SET requests for various number of clients and 1024 Byte payload.

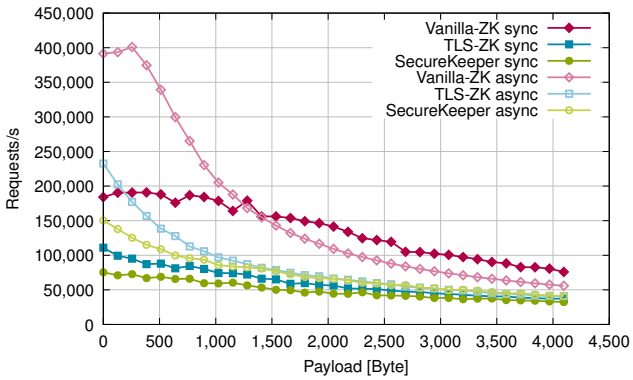


Figure 7: Throughput of sync. and async. GET requests.

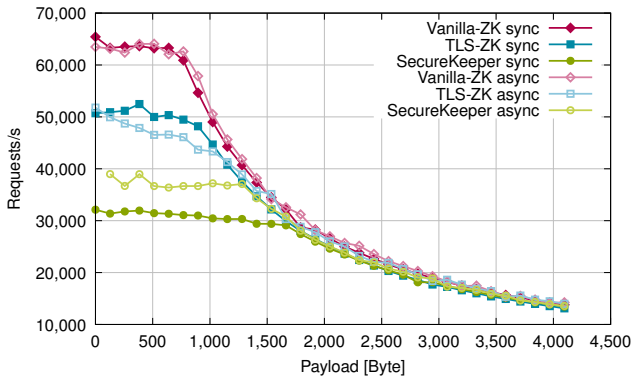


Figure 8: Throughput of sync. and async. SET requests.

with TLS enabled[3]. The three variants are denoted as *SecureKeeper*, *Vanilla-ZK* and *TLS-ZK* in all graphs. The experiments are performed using only asynchronous and again using only synchronous requests. Also we experimented with a wide range of typical payload sizes between 0 and 4096 Bytes. Our evaluator uses various numbers of client threads connected to ZooKeeper replicas issuing requests as fast as possible. We explicitly distribute the clients equally on all replicas, in order to get most stable and reliable results. Only for the fault-tolerance experiment we must allow clients to randomly choose a replica from a list of all replicas (see Section 6.3).

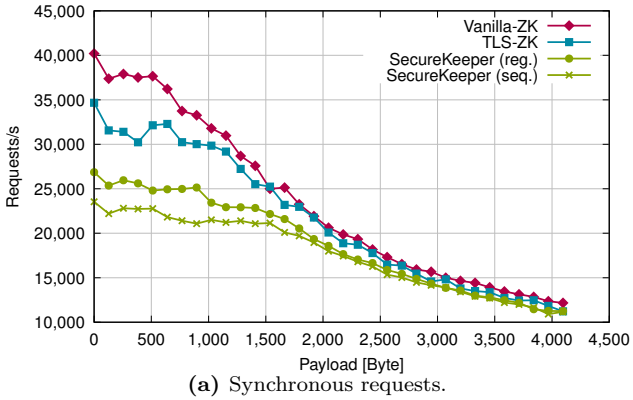
We determine the optimal number of threads (i.e., number of connections) for throughput measurements for a realistic workload, according to the original ZooKeeper paper [12], consisting of a 70:30 mix of GET and SET requests by gradually increasing the number of clients: For asynchronous requests the maximum performance is reached with 5 client threads (see Figure 6b) and 200 pending requests (simultaneous unanswered requests in flight). In contrast, synchronous requests perform best between 200 to 400 client threads (see Figure 6a). As the maximum throughput for synchronous requests is reached at a number of 300 client threads, we consistently used this number for all other synchronous experiments.

[3]https://issues.apache.org/jira/browse/ZOOKEEPER-2125, accessed 5/10/2016.

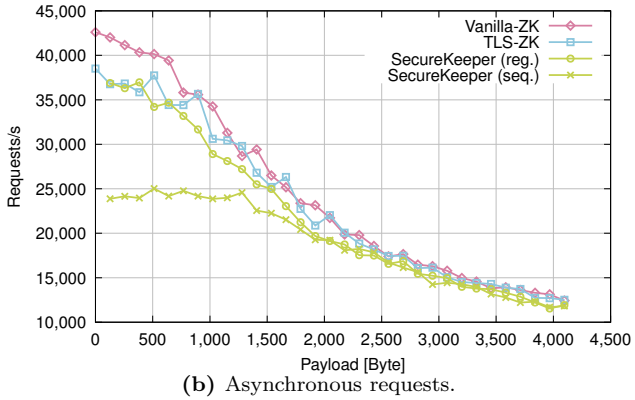
6.2 Throughput of SecureKeeper

In Figure 7 we show the throughput of GET, and in Figure 8 the throughput of SET requests. These are answered by the replica the client is connected to directly, in contrast to SET requests, that are forwarded to the leader replica and passed through the agreement protocol. Initially, for both GET and SET we create one znode for each client thread and then continuously read and write their payload with the GET and SET respectively. As can be seen from the graphs, the performance of SecureKeeper is close to the TLS-based ZooKeeper variant with an average overhead for all measured payload sizes of 7.89% and 10.34% for synchronous requests and 3.12% and 9.85% for asynchronous requests. While the encryption overhead is more visible for low payload sizes, with increasing payload size the throughput of SecureKeeper and TLS-ZK converges. This behavior is due to the constant encryption and enclave entering overhead for each message which gets insignificant for higher payload sizes. The details of the encryption overhead are illustrated in Table 2 and detailed below.

Figures 9a and 9b show the performance for creating znodes in ZooKeeper. We distinguish the creation of *regular* and *sequential* nodes here, as we have a second enclave entry on the leader replica for the latter (see Section 4.4). For all three variants the throughput is a bit lower than the SET throughput, as the internal ZooKeeper state size increases with the growing number of znodes. The overhead of CREATE requests compared to TLS-based ZooKeeper is 9.76% for regular and 11.82% for sequential nodes with synchronous



(a) Synchronous requests.



(b) Asynchronous requests.

Figure 9: Throughput of `CREATE` requests.

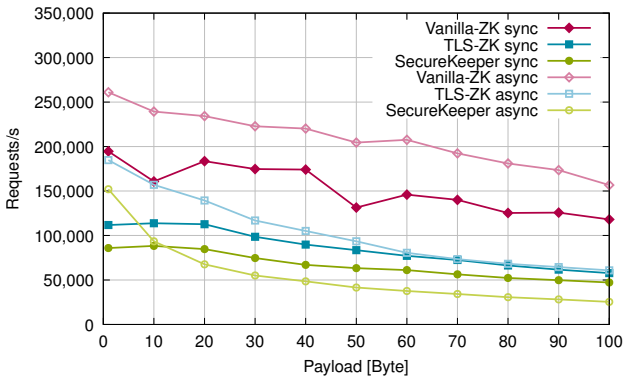


Figure 10: Throughput of sync. and async. `LS` requests.

	Operation	TLS-ZK	SecureKeeper	Δ
synchronous	`GET`	55.71 %	63.60 %	7.89 %
	`SET`	9.12 %	19.46 %	10.34 %
	`LS`	43.17 %	55.98 %	12.82 %
	`CREATE`	6.53 %	16.28 %	9.76 %
	`CREATESEQ`	7.04 %	18.86 %	11.82 %
	`DELETE`	14.48 %	29.64 %	15.16 %
	Average	22.67 %	33.97 %	11.30 %
asynchronous	`GET`	41.50 %	44.62 %	3.12 %
	`SET`	8.45 %	18.30 %	9.85 %
	`LS`	49.58 %	70.97 %	21.38 %
	`CREATE`	3.70 %	11.86 %	8.15 %
	`CREATESEQ`	3.50 %	18.47 %	14.97 %
	`DELETE`	9.04 %	18.12 %	9.08 %
	Average	19.30 %	30.39 %	11.09 %
	Read average	47.49 %	58.79 %	11.30 %
	Write average	7.73 %	18.87 %	11.14 %
	Global average	**20.98 %**	**32.18 %**	**11.20 %**

Table 1: SecureKeeper overhead comparison.

	Request	Response
Transport	−HMAC −IV	+HMAC +IV
Path	+relative Overhead	−relative Overhead
Payload	+HMAC +IV	−HMAC −IV

Table 2: Comparison of encryption overhead.

requests. Asynchronous requests show an overhead of 8.15% for regular and 14.97% for sequential nodes.

We also evaluated the `LS` or `getChildren` operation, listing all child nodes for a specific znode. Listing the children of a znode leads to the decryption of all paths of all child nodes of the znode in question. With our approach the high number of path decryption operations has a notable impact on the throughput of the `LS` operation depending on the number of child nodes. The results are illustrated in Figure 10. Synchronous `LS` requests have an overhead of 12.82% and asynchronous requests 21.38%.

We also evaluated `DELETE` requests, however, as both the request and response messages do not contain payload in this case, we omit plotting a graph. Compared to TLS-based ZooKeeper, the overhead of synchronous `DELETE`s is 15.16% and for asynchronous requests 9.08%.

Table 1 summarizes our evaluation for all operations and for both synchronous and asynchronous requests. We provide the overhead of SecureKeeper and TLS-based ZooKeeper compared to vanilla ZooKeeper in percent, as well as the delta of these numbers. Finally, the table summarizes the overall overhead of our approach as a global average value of 11.20%.

The added confidentiality and integrity guarantees of our approach clearly do not come at no cost when compared with vanilla ZooKeeper. However, our evaluation shows that the throughput of SecureKeeper is scaling and performing similar to TLS-enabled ZooKeeper. The payload and path encryption cause the length of exchanged messages between clients and replicas to increase compared to vanilla ZooKeeper. This increase is essentially due to the HMACs and IVs used for the decryption and encryption operations. Also, messages are lengthened due to the Base64 encoding of the path, as well as the added hash of the znode path to the payload field. Finally, in our approach there is a constant overhead due to the enclave entries and exists, which is clearly not required for TLS-based ZooKeeper. Hence, in general the throughput of our approach is below, though close to, TLS-based ZooKeeper.

Table 2 provides an overview of the message length changes due to our approach. As can be seen, the transport encryption between clients and the entry enclave, causes a constant decrease of the length of requests and a constant increase for the responses. For path encryption the difference in length depends on the "depth" of the path (i.e., the number of slashes in the path): for each chunk of the path, we apply an individual encryption step, that adds an HMAC and IV (see Section 4.3). Paths in requests (specifying which node the operation is applied to) will be increased in length for each request. For response messages the contained paths

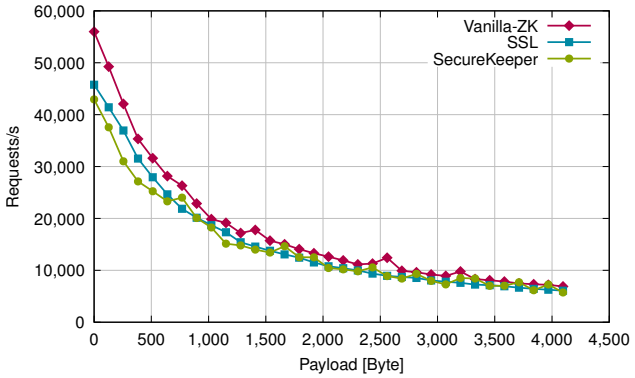


Figure 11: Throughput of synchronous GET and SET operations, performed using the YSCB benchmark suite.

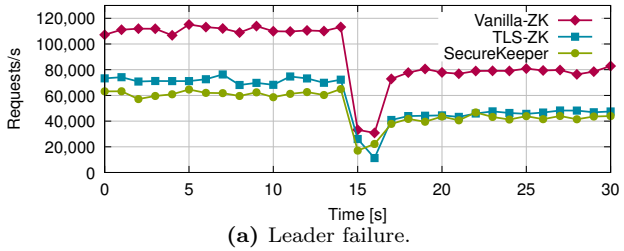
will get the same amount shorter (only in the case of the `getChildren` operation). Finally, the overhead due to the encryption of the payload is exactly the same as for the transport encryption, as we use the same cipher.

In addition to measurements with our own workloads, we also performed throughput experiments using the YCSB benchmark suite. We compiled a mixed workload of synchronous read and write requests, and used 35 threads to connect to our ZooKeeper cluster. Then, for various different payload sizes, we performed 500k operations each. As can be seen from the results in Figure 11, all three variants scale similarly well and SecureKeeper's performance is very close to the TLS variant. The lower baseline of the YCSB benchmark compared to our own, is due to YCSB not able to perform a warmup phase as we do with our evaluator for all experiments. In addition, with YCSB we have much higher client-side CPU load during the measurement, which requires us to decrease the number of simultaneous clients.

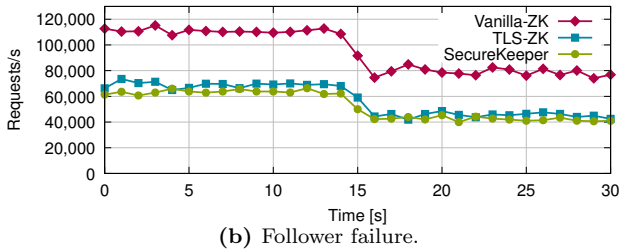
6.3 Fault Tolerance

As ZooKeeper can tolerate replica faults, we evaluated the capability of our system to maintain the fault-tolerance properties of original ZooKeeper. In order to achieve this, we simulated faults of replicas while a constant client load is applied. In all cases we measured the total throughput of all clients on the cluster executing asynchronous GET and SET requests with a fixed payload size of 1024 Bytes every 250 ms and visualize it grouped by timeslots of 1 s. We executed separate experiments for a failure of the leader replica and a failure of one of the follower replicas. After starting a clean cluster, we let the system warmup for a few seconds and inject the fault after 30 s from the cluster start. Then, we keep the cluster running without restarting the replica for another 30 s. We explicitly distribute the clients equally on all replicas in the previous experiments for improving stability of the results. In contrast, for fault tolerance we provide each client with a list of all replicas and let the client choose one replica randomly. Otherwise, automatic failover of clients to another replica is not possible.

As can be seen from Figure 12, most importantly, in all cases SecureKeeper shows the same fault-tolerance as vanilla ZooKeeper. Also, in all cases, the missing replica reduces the total throughput by approximately one third. However, on a leader failure, a leader election is executed by ZooKeeper, which causes the total throughput to drop to zero for a certain amount of time until a new leader has been elected.



(a) Leader failure.



(b) Follower failure.

Figure 12: Fault-tolerance behavior of ZooKeeper variants.

	Component	Language	SLOC
trusted	(De-)Serialization	C	2514
	Counter and entry enclave	C	985
	SDK-generated code	C	572
	Total trusted		**4071**
untrusted	ZooKeeper Server	Java	33851
	Enclave interfaces	Java	154
	Enclave management	C	296
	SDK-generated code	C	262
	Total untrusted		**34563**
	Total SecureKeeper		**4783**
	Total		**38634**

Table 3: Size of code base of SecureKeeper components.

After the leader election has finished, throughput remains at about two thirds as expected, as still the cluster runs without a third replica.

6.4 Size of Code Base

Table 3 compares the size of the original ZooKeeper code base to SecureKeeper. The overall TCB of SecureKeeper comprises about 4,000 SLOCs. More than 62% of this code is used for (de)serialization of messages and taken from the original ZooKeeper C bindings. The remainder of the trusted code base implements parameter passing and glue code for the message flow through the enclaves. The total amount of trusted code of SecureKeeper is roughly 12% of the complete ZooKeeper code base, while we added less than 100 SLOCs to the ZooKeeper client code base. We could not measure the size of the code base of the Intel SGX SDK-provided libraries, such as the `libstd` and the cryptographic library, but we mention their binary size in Section 6.5.

The untrusted code of SecureKeeper comprises creation, initialization and maintenance of the enclaves, parameter passing across the enclave boundary and the JNI interface. Only three lines of the original ZooKeeper code base have been changed, all others have been added, thus, we call our approach minimally-invasive to ZooKeeper.

6.5 Memory Consumption

The total memory footprint of an enclave composed EPC memory and SGX-internal metadata depends on the enclave

code, one stack region for each thread and the heap. Each enclave thread maintains its own stack (default 64 KB), and SGX data structures holding thread metadata and state. The enclave code is basically the compiled and signed shared object which is copied to the ELRANGE during the enclave creation using the Intel SGX SDK API. Finally, configuration of the heap size can be done with an enclave configuration file that is used by the SGX signer tool of the Intel SGX SDK.

The largest and most significant component of enclave memory consumption is the heap and code of the enclave. Due to the cryptographic operations, the heap must be able to hold the largest possible message twice as source and destination of the encryption or decryption is the same size. In addition the list of request types requires space for all pending requests, which can be up to 200 requests in our experiments, but should be negligible as the size of one entry is only a few bytes. In total, based on the above outlined constraints and our enclave shared object size of 436 KB, the required memory for each of our entry enclaves is about 580 KB. This allows more than 150 enclaves to fit into the EPC of *each* ZooKeeper replica without EPC paging.

The counter enclaves memory layout is essentially the same, except for the heap size, which can be a lot smaller because we only process the path of znodes. Hence, the counter enclave with a binary size of 325 KB requires an additional amount of 397 KB memory on the leader replica.

As a result, we can keep the enclave memory consumption below the EPC threshold of 92 MB for each replica, even for high client loads. If necessary, the memory footprint could be further decreased by co-locating multiple or even all clients to one enclave. However, this would increase the code complexity of the entry enclave, i.e. for client session management and add additional synchronization points, which could harm performance. Lastly, enclaves as of this version of SGX cannot dynamically increase their ELRANGE allocated memory size, thus a conservative allocation for the maximum number of clients has to be performed upfront at service startup time.

7. DISCUSSION

In this section, we discuss the security guarantees that SecureKeeper provides and other guarantees that are beyond a minimally-invasive integration with SGX. Furthermore, we explore the general applicability of the proposed methodology to other services.

7.1 Confidentiality and Integrity Guarantees

Our approach ensures the confidentiality of all user-provided data stored in the ZooKeeper database. This comprises the plaintext payload and path names of all znodes. However, operation types, the znode tree structure, as well as node access frequencies and access times are still visible. In this regard, SecureKeeper provides similar protection as filesystem-level encryption [22].

As mentioned, SecureKeeper ensures the integrity of plain znodes and their associated path names. This is achieved by secure authenticated communication between clients and entry enclaves, and also data encryption once znodes are passed to Zookeeper for further processing. This is implemented by an HMAC over the encrypted payload of a znode, which is verified during decryption. Any modification to the payload—be it purposeful or accidental—can be detected during the verification before the data is passed to the client.

Another aspect of integrity is the link between the znode path and the payload. As we encrypt a hash of the path name with its associated payload, the path is bound to the payload. This enables the entry enclave to detect whether a payload belongs to a certain path name or not, before the data is forwarded to the client. Thus, an attacker cannot exchange the payload of a node for that of another node.

This approach, however, does not suffice for sequential nodes. Here the counter enclave receives untrusted input from the ZooKeeper base code in the form of an incremented counter value stored in the metadata of the associated parent node. The passed value can be validated to be a number, but it may be arbitrarily chosen by the outside code. A malicious attacker can choose the tail part of sequential node path names that stands for the sequential node number according to the format convention. In this limited scope, SecureKeeper is susceptible against naming attacks, i.e. no new znode payload information can be crafted nor existing payload information can be modified but only overwritten by payloads with the same sequential node path name prefix.

We considered protective measures but all of them lead to a system design that has to be resilient to rollback attacks of externalized state or even resilient to Byzantine faults [23]. Both aspects, as outlined next, would require substantial architectural changes and, since the primary goal of SecureKeeper is the protection of confidentiality, we do not consider this issue as severe and leave this as a direction for future work that may result in an entirely different service architecture.

7.2 Replay and Denial of Service Attacks

There are two kinds of replay attacks that can be distinguished: first, the communication between a client and the entry enclave may be replayed, but this can easily be prevented using replay-safe encrypted communication (i.e. TLS); second, the inputs provided to the enclaves from the untrusted ZooKeeper data store may be stale and/or replayed, which leads to the clients reading stale values. SGX itself offers no means to prevent replay attacks regarding state that is external to enclaves, be it volatile or persistent [6, 24]. One option would be the use of a Byzantine fault-tolerant agreement protocol [23], but this requires significant changes to the ZooKeeper architecture, which is incompatible with the design goals of SecureKeeper.

7.3 Wider Applicability of the Approach

The general idea of SecureKeeper is the minimally-invasive integration of SGX to achieve confidentiality and integrity under the above outlined restrictions. Despite the complexity of ZooKeeper as a replicated and fault-tolerant coordination service, this is possible due to the fact that ZooKeeper performs only limited data processing. Thus, it can be assumed that other services that mostly handle data but do not process it can be protected similarly. Simple KVS and blob storage may be a good fit because they access data by a unique identifier. If a query API beyond an exact match needs to be supported, e.g. range queries, further enclave support may be needed. This could be addressed, however, by storing keys or a subset of keys inside an enclave, which only would require hosting a fraction of

the service state in the enclave and thus limit performance degradation (see Section 3).

8. RELATED WORK

There have been a number of approaches targeting the protection of applications from unauthorized access. Many of these solutions so far rely on additional system software layers in the absence of appropriate commodity hardware support. Examples are *NGSCB* [25, 26] and *Proxos* [27], which feature the idea of an untrusted and a trusted operating system, running on the same machine isolated by a virtual machine monitor. As a consecutive step several systems proposed a trusted virtualization layer guarding the applications from unauthorized operating system access [28, 29]. All of these approaches come attached with a much larger TCB and if performant typically feature a more coarse-grained partitioning than what SecureKeeper requires.

There has been a large number of systems that utilized trusted hardware to partition software in trusted and untrusted parts. Previous examples are *secure co-processors* [30] and Trusted Platform Modules (TPMs) [31], both offering protection from certain types of physical access. While on the one hand, secure co-processors offer quite limited performance due to thermal problems and are rather expensive, they are primarily used in niche markets. Nevertheless, with TrustedDB [32] there exists an approach for splitting a database between an untrusted commodity system and a secure co-processor.

On the other hand, TPMs are too restricted in their capabilities, e.g. remote attestation, to be useful in complex shared environments. Basing on the functionality of the TPM a number of systems aimed to achieve trusted execution with a small additional TCB [33, 34]. However, they either suffer from the bad performance of the TPM chip [33] or require a larger TCB (i.e., a virtualization layer) and protect systems at a more coarse-grained level.

ARM TrustZone [35] distinguishes only two levels of trust: a secure world and a non-secure world. In essence one has to trust the complete code of the secure world which might include substantial system code. To achieve a more fine-grained trust scheme, Santos et al. [36] proposed so called *trustlets* hosted by a managed runtime inside the secure world. In principle SecureKeeper could be deployed on-top of this framework, but TustZone offers no memory encryption and so far it is mainly supported by embedded systems.

As previously described, SGX features fine-grained deployment of trusted execution environments (TEEs), so a single application can even host multiple enclaves. So far this has been marginally exploited due to the lack of freely available system software and hardware. Baumann et al. [6] proposed Haven, a system that enables execution of unmodified legacy applications inside an enclave. This was achieved by porting a library OS to the enclave and the shielding of system calls that need to be processed by the untrusted host operating system. The consequence of this convenient way to execute legacy applications on top of SGX is, as outlined in detail in Section 3, a considerable performance overhead, and, more importantly, a rather large TCB. *VC3* [15] took a different direction by focusing on the secure execution of MapReduce applications. Only the actual data processing tasks are executed inside enclaves while the framework is kept unchanged in the untrusted environment. To protect the execution inside the enclave, compiler-based hardening techniques have been applied. Finally, the distribution of data is integrity-protected, in order to ensure a malicious host cannot exclude data from a computation without being detected. While SecureKeeper shares the fine-grained partitioning using SGX with VC3, it targets the protection of confidentiality of ZooKeeper as a networked-service. So far neither Haven nor VC3 has been evaluated on top of production hardware, which SecureKeeper has.

Strackx et al. [37] proposed to use of SGX to implement the idea of an inverted cloud. Instead of maintaining huge clusters with machines on low load, mini providers are used. The cloud provider takes CPU power from those mini providers and executes work on them. This leads to the effect, that computing resources can be used more efficiently. However, their work focuses more on conceptual level and does neither address ZooKeeper nor data-handling services.

While the previously mentioned related work focused on generic approaches towards securing applications from unauthorized access, there are systems that specifically target to secure the processing of sensitive data, e.g. in scope of databases or a file system.

One option is to move query processing to the client side [38, 39] or an intermediate proxy [40]. While this can effectively preserve data confidentiality, it demands data storage and query processing at the client side [38, 39]. In a preliminary workshop paper, we proposed the ZPP [40], a trusted proxy intercepting requests between clients and ZooKeeper replicas. Compared to this work, SecureKeeper follows a different direction by integrating secure data processing into ZooKeeper itself. The result is a much leaner TCB as we do not rely on a trusted network stack and need no trusted operating system underneath. It offers better performance as an additional hop that was demanded by putting a physical machine in between the clients and the ZooKeeper instance can be avoided.

Popa et al. [41] proposed the execution of queries over encrypted data. However, the proposed query support is limited and the approach incurs a large performance penalty.

9. CONCLUSION

Trust issues still impede the use of cloud infrastructures when sensitive data needs to be process. In this paper we showed how ZooKeeper, a central component of complex distributed workloads, can be efficiently secured using SGX. The resulting SecureKeeper demonstrates how tailored enclaves can be integrated minimal-invasively to ensure confidentiality of all managed user data and integrity for plain znodes. Due to its lightweight integration of enclaves the induced performance overhead is comparable to using ZooKeeper with TLS encryption enabled.

Acknowledgments

This project received funding from the European Union's Horizon 2020 research and innovation programme under the SERECA (Grant agreement No. 645011) and the SecureCloud (Grant agreement No. 690111) project.

References

[1] IDC, *Worldwide Cloud IT Infrastructure Market Growth Expected to Accelerate [...]* http://www.idc.com/getdoc.jsp?containerId=prUS25576415, 2015.

[2] K. R. Jayaram, D. Safford, U. Sharma, V. Naik, D. Pendarakis, and S. Tao, *Trustworthy Geographically Fenced Hybrid Clouds*, Middleware, 2014.

[3] S. Pearson and A. Benameur, *Privacy, Security and Trust Issues Arising from Cloud Computing*, CloudCom, 2010.

[4] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, *Innovative Instructions and Software Model for Isolated Execution*, HASP, 2013.

[5] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, *Using Innovative Instructions to Create Trustworthy Software Solutions*, HASP, 2013.

[6] A. Baumann, M. Peinado, and G. Hunt, *Shielding Applications from an Untrusted Cloud with Haven*, OSDI, 2014.

[7] Synopsys, Inc., *Open Source Report 2014*, http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf, 2014.

[8] Spotify AB, *Sparkey*, https://github.com/spotify/sparkey-java.

[9] Apache HTTP Server Project, *Apache HTTP Server*, https://httpd.apache.org/, 2016.

[10] Eclipse Foundation, *Jetty*, http://www.eclipse.org/jetty/, 2015.

[11] W. Reese, *Nginx: the High-Performance Web Server and Reverse Proxy*, *Linux Journal*, vol. 2008, no. 173, 2008.

[12] P. Hunt, M. Konar, F. Junqueira, and B. Reed, *ZooKeeper: Wait-Free Coordination for Internet-Scale Systems*, USENIXATC, 2010.

[13] *Intel Software Guard Extensions (Intel SGX) SDK*, https://software.intel.com/sgx-sdk.

[14] F. P. Junqueira, B. C. Reed, and M. Serafini, *Zab: High-performance broadcast for primary-backup systems*, DSN, 2011.

[15] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, *VC3: Trustworthy Data Analytics in the Cloud Using SGX*, SOSP, 2015.

[16] Y. Xu, W. Cui, and M. Peinado, *Controlled-channel attacks: Deterministic side channels for untrusted operating systems*, IEEE, 2015.

[17] *CVE-ID: CVE-2016-0494.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0494, 2016.

[18] *CVE-ID: CVE-2016-0687.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0687, 2016.

[19] *CVE-ID: CVE-2016-3427.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3427, 2016.

[20] *CVE-ID: CVE-2016-3443.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3443, 2016.

[21] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, *Innovative Technology for CPU Based Attestation and Sealing*, HASP, 2013.

[22] M. Halcrow, *eCryptfs: An enterprise-class encrypted filesystem for linux*, *Proceedings of the 2005 Linux Symposium*, 2005.

[23] M. Castro, B. Liskov, *et al.*, *Practical Byzantine fault tolerance*, vol. 99, 1999.

[24] J. Beekman, J. Manferdelli, and D. Wagner, *Attestation transparency: Building secure internet services for legacy clients*, 2016.

[25] A. Carroll, M. Juarez, J. Polk, and T. Leininger, *Microsoft Palladium: A Business Overview*, *Microsoft Content Security Business Unit*, 2002.

[26] M. Peinado, Y. Chen, P. England, and J. Manferdelli, *NGSCB: A Trusted Open System*, Information Security and Privacy, 2004.

[27] R. Ta-Min, L. Litty, and D. Lie, *Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable*, OSDI, 2006.

[28] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, *Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems*, ASPLOS, 2008.

[29] J. Criswell, N. Dautenhahn, and V. Adve, *Virtual Ghost: Protecting Applications from Hostile Operating Systems*, ASPLOS, 2014.

[30] Lindemann, M. and Perez, R. and Sailer, R. and van Doorn, L. and Smith, S.W., *Building the IBM 4758 secure coprocessor*, *Computer*, 2001.

[31] Trusted Computing Group, *Trusted Platform Module Main Specification. version 1.2.*

[32] S. Bajaj and R. Sion, *TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality*, SIGMOD, 2011.

[33] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, *Flicker: An Execution Infrastructure for TCB Minimization*, EuroSys, 2008.

[34] F. Zhang, J. Chen, H. Chen, and B. Zang, *CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization*, SOSP, 2011.

[35] ARM Limited, *ARM Security Technology - Building a Secure System using TrustZone Technology*, 2009.

[36] N. Santos, H. Raj, S. Saroiu, and A. Wolman, *Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications*, ASPLOS, 2014.

[37] R. Strackx, P. Philippaerts, and F. Vogels, *Idea: Towards an Inverted Cloud*, Engineering Secure Software and Systems, 2015.

[38] P. Williams, R. Sion, and D. Shasha, *The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties*, NDSS, 2009.

[39] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan, *Building Web Applications on Top of Encrypted Data Using Mylar*, NSDI, 2014.

[40] S. Brenner, C. Wulf, and R. Kapitza, *Running ZooKeeper Coordination Services in Untrusted Clouds*, HotDep, 2014.

[41] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, *CryptDB: Protecting Confidentiality with Encrypted Query Processing*, SOSP, 2011.