Trusted Execution, and the Impact of Security on Performance

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

Michael Behlendorf
TU Braunschweig
m.behlendorf@tu-braunschweig.de

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

ABSTRACT

Due to increasing success of cloud computing offerings, the demand for sensitive data processing and security in the cloud has also increased. By incorporation of trusted execution technologies such as the broadly available Intel Software Guard Extensions (SGX), applications can be secured. However, software engineers need to align the development process with the capabilities and properties of such a technology, in order to correctly secure applications while achieving good performance.

In this paper, we identify relevant aspects for partitioning applications and discuss two complementary designs optimising for performance or security respectively. Additionally, our contribution comprises a performance and security measurement, at the example of two established real-world applications, that we both partitioned according to the above two distinct design approaches. We consider this paper as a guideline for the partitioning process of mainly data-handling services for usage of trusted execution and as a collection of relevant characteristics during the development of applications with trusted execution environments.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; *Software security engineering*;

KEYWORDS

Intel SGX, Application Partitioning

ACM Reference Format:
Stefan Brenner, Michael Behlendorf, and Rüdiger Kapitza. 2018. Trusted Execution, and the Impact of Security on Performance. In *3rd Workshop on System Software for Trusted Execution (SysTEX '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3268935.3268943

1 INTRODUCTION

Cloud security has been an issue since the early cloud computing offerings, preventing cloud adoption and especially sensitive data processing in the cloud. Distributed application deployments in hybrid models combining public and private clouds are one way to tackle this problem. However, this at least partly impedes some of the benefits of public clouds for the customer, such as the prevention of upfront investment in hardware. With Intel SGX, a new technology for trusted execution has reached widespread availability, even in commodity hardware. It allows sensitive data processing in an encrypted region of main memory securely, requiring to trust only the CPU package, not the whole hardware platform. Companies have already started adopting this technology in real-world (cloud) applications such as the private contact discovery of the *Signal* messenger [13] or Microsoft's Azure Cloud [16]. However, when designing trusted applications to run securely in the cloud, they need to align their development workflow to respect the characteristics of this new technology.

Initially, software engineers tried to naively run whole legacy applications inside trusted execution environments for quite obvious reasons: the existing application does not need to be changed in the optimal case, and every part of it resides in the trusted environment. While this sounds appealing and approaches like SCONE [1], Graphene [19] and Haven [2] have shown the feasibility of this paradigm, the trusted code base is huge—Haven, for example, comprises a whole (library) operating system in the trusted environment. This not only constitutes quite a significant attack surface, and thus, decreases security, but also leads to major performance problems specifically with the Intel SGX technology. As of now, SGX limits the memory space for its *secure enclaves* where transparent encryption and integrity checks are applied to, to a maximum of only 128MB. Memory consumption exceeding this so called Enclave Page Cache (EPC) leads to a significant performance penalty of up to 1000× in the worst case [1].

Hence, in contrast to the above holistic approaches, researchers have already tried to partition applications into secure and insecure parts—so called *application partitioning*. Thereby, they run only sensitive processing of confidential data in a trusted environment and try to decrease the footprint of the secure application, regarding both, trusted memory consumption and the trusted code base. Our earlier work, SecureKeeper [3], is one example for this approach, where we partitioned the ZooKeeper [7] coordination service for usage of trusted execution. The authors of Glamdring [11] even tried to automate this partitioning process by static code analysis in order to derive which source code fragments need to reside in the enclave. While automatic partitioning as of now is not mature enough to result in an optimal partitioning, even manual partitioning demands from the developer to select wisely the design properties of the partitioned application in order to achieve security while retaining good application performance.

In this paper we first investigate various metrics and properties during the partitioning process of an application for usage with the Intel SGX trusted execution technology. We propose a set of various properties that affect the performance and the overall level of security of a partitioned application and discuss and evaluate their impact on two real-world applications. For this purpose we implemented two complementary partitioning approaches, both

SysTEX '18, October 15, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5998-6/18/10…$15.00
https://doi.org/10.1145/3268935.3268943

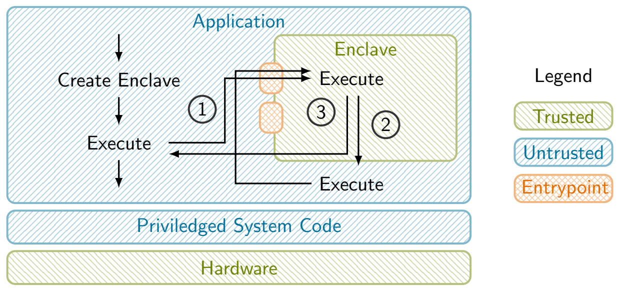

Figure 1: SGX enclave interaction.

at the example of these two example applications and evaluated their performance and security. Thus, the contribution of this paper comprises the following aspects:

- Identification of performance- and security-critical properties relevant for application partitioning, and discussion of their effects on two example applications.
- Suggestion of two different partitioning designs that optimise performance or security as needed.
- Partitioning of two real-world applications—both according to two distinct partitioning approaches—and evaluation and discussion of their resulting performance and security properties.

The paper's structure comprises a background section (Section 2), followed by the analysis of enclave design metrics (Section 3) and a detailed description of two different enclave designs (Section 4). Next, we describe our two use case applications that we partitioned with the identified metrics in mind (Section 5), and evaluate their performance and security (Section 6). Finally, we discuss related work (Section 7) and conclude the paper (Section 8).

2 BACKGROUND

In this section we describe relevant background of this paper, comprising the Software Guard Extensions (SGX) technology, and the two services we partitioned for using trusted execution.

2.1 Intel Software Guard Extensions

Intel SGX [15] is a processor instruction set extension that allows the creation of an x86-based Trusted Execution Environment (TEE)—a so called *secure enclave*. Confidentiality of enclave memory is protected by transparent encryption done within the CPU, i.e., plaintext is only available inside the CPU package. Valid interaction with an enclave is only possible via explicit enclave calls (ecalls) to enter an enclave and outside calls (ocalls) to call out of the enclave. Figure 1 depicts the interaction of an untrusted application part with an enclave, comprising an ecall ①, an ocall ②, and finally, an ecall-return ③ to the untrusted application.

In general, enclaves are bound to a user process—even multiple enclaves in one user process are possible. However, if all enclaves on a platform do not fit into the so called EPC a special form of memory swapping to regular RAM is required. As this process requires re-encryption of the page's contents, as well as measures to ensure integrity and prevent replay and rollback attacks, it induces a high performance overhead to the application.

As enclaves can only be operated in user space, SGX-based applications still depend on the underlying Operating System (OS) to cooperate. This includes the enclave's memory management inside the EPC by the untrusted OS. Hence, SGX naturally cannot prevent denial of service attacks by privileged code.

Each thread requires an available Thread Control Structure (TCS) page during enclave entry for storing the thread's state inside the enclave. Depending on the number of TCS pages added to an enclave, this allows simultaneous enclave entries from multiple threads.

Intel offers an SDK to handle the enclave's life cycle and memory management. It also comprises the Enclave Description Language (EDL) that allows the specification of the ecalls and ocalls that the enclave supports and the "Edger8r" tool which acts a code generator for ecall and ocall stubs. Finally, enclaves are compiled and linked together with the generated code to a shared object, which is loadable as an enclave by using SDK functions.

2.2 An SGX-aware Threat Model

We assume a typical threat model for SGX enclaves [2], specifically in an untrusted cloud environment: 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 the enclave's code running in the SGX enclave.

Availability threats such as crashing an enclave are not of interest, as, inherently to the SGX paradigm, the hosting OS can stop enclave execution arbitrarily and at any time.

We do not consider side-channel attacks [21] and trust the design and correct implementation of the CPU package and the SGX instructions including all cryptographic operations done by SGX. While the recent Meltdown [12] and Spectre [9] bugs have shown that even hardware security is not impeccable, these bugs can be fixed by microcode and system software updates, and investment in future CPU generations.

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

2.3 Apache ZooKeeper

Apache ZooKeeper [7] is a coordination service for distributed systems, and itself a distributed system. Coordination with ZooKeeper is realised by implementation of coordination primitives such as locks or barriers using the ZooKeeper API that allows management of so called znodes. Znodes represent folders and directories of a virtual in-memory file system at the same time, i.e. they can have children and store payload data. Additional features (e.g. callbacks) and consistency guarantees such as global write ordering allow the implementation of more complex coordination primitives.

2.4 Voldemort

Voldemort is a high-performance key value store implemented in Java, used at LinkedIn and an open source clone of Amazon's Dynamo store [5]. It supports distributed setups with sharding of the stored data across multiple hosts and features a horizontal scalability of both, read and write accesses. However, Voldemort is not a relational database system, as to its clients it offers a relatively simple `get`/`put` API. According to its authors it represents a “big, distributed, persistent, fault-tolerant hash table”[1].

3 METRICS OF ENCLAVE DESIGN

Securing services with SGX enclaves can be done in various different ways. While in our earlier work [3] we proposed the *application partitioning approach*, this paper builds upon that and aims at the investigation of performance- and security-critical enclave design aspects. In the following we discuss these aspects and their impact when securing a service with the application partitioning approach.

3.1 Enclave Performance Metrics

The performance of a partitioned application with SGX significantly depends on at least two properties: the memory footprint of the enclave and the amount of context switches to and from the enclave. In order to achieve high performance of partitioned applications, both, the memory footprint of the enclaves and the frequency of context switches to and from the enclaves should be minimised.

3.1.1 Trusted Memory Footprint. Current SGX architectures are limited to 128 MB of EPC memory for all enclaves on a platform. Therefore, the enclave footprint inside the EPC must me minimised for good performance. Otherwise, the costly SGX paging mechanism causes dramatic performance degradation of up to 1000× as shown in earlier work [3].

Of special interest is the absolute *working set* size of an enclave, as pages containing rarely used code can be swapped out of the precious EPC memory range. But, if all frequently used pages do not fit inside the EPC, the SGX paging mechanism starts swapping pages in and out continuously, causing a huge performance drop.

Furthermore, (common) libraries that are frequently an ingredient of all enclaves impose another problem: as sharing of pages between enclaves is not possible in general [14], code pages holding the same libraries in multiple enclaves reside in memory as duplicates. Especially, if such a library is frequently used, as it is the case for the cryptographic library supporting TLS encryption for example, the respective pages can not be swapped out as they would soon be required again. This limits the maximum number of enclaves that can be operated with good performance on a platform.

3.1.2 Enclave Execution Mode Changes. In addition to context switches caused by the OS scheduler and interrupts in regular applications, partitioned SGX applications also experience delays by execution mode changes: entering and leaving the enclave causes a constant performance overhead. In SGX applications this happens for each ecall and ocall, and also for asynchronous enclave exits due to interrupts. Therefore, improving the performance of enclavised applications also requires minimising the ecall interface and frequency of an enclave by implementing just the right subset of the application logic inside the enclave.

Delays caused by entering and leaving an enclave are due to necessary microarchitectural tasks of the CPU, such as additional checking of enclave page permissions and flushing the TLB. Therefore, the number of ecalls and ocalls per second is a parameter to be considered when trying to achieve good performance. Not completely under control of the developer, however, is the number of asynchronous exits of an enclave, as these are caused by interrupts and CPU exceptions at runtime.

The performance penalty of these context switches is constant and in the order of magnitude of a process switch—about 8000 cycles have been measured [18]. Therefore, in some cases it is even beneficial to move insensitive code into an enclave to reduce the number of enclave transitions [11].

Recently, Intel introduced so called *switchless calls* for enclaves in their SGX SDK. This allows to circumvent the transition overhead for frequent short-running calls by a set of threads that stay inside the enclave to process requests from the outside. However, this feature not only requires suitable workloads but also fine-grained configuration and tuning to gain a stable performance benefit.

3.1.3 Exploiting Parallelisability. As multiple threads can enter the same enclave when enough TCS pages are available, computation can be accelerated by using multiple threads. Accesses to shared data structures, however, naturally require the usage of synchronisation primitives such as mutexes in order to ensure consistency.

The SGX SDK provides a mutex implementation allowing synchronisation of critical sections in enclaves. However, as no system calls are available inside an enclave, these are implemented as spin locks inside the enclave, in order to prevent unnecessary enclave exits. When waiting for the acquisition of a mutex, the spin lock implementation will eventually cancel and exit the enclave falling back to an untrusted system call-based mechanism. This prevents exiting the enclave for short waiting periods and staying in the spin lock forever in case of long waits. However, it inevitably imposes higher cost of synchronisation in enclaves, especially on high lock contention, and needs to be respected by the enclave developer.

3.2 Enclave Security Metrics

In the previous section, we described properties that affect an application’s performance. Some of these properties also affect the security of a partitioned application. We discuss these and other security-specific properties in the following.

3.2.1 Size of Trusted Code Base. The size of the Trusted Code Base (TCB) is an important security-related metric. As studies have shown, more lines of code usually lead to more exploitable security vulnerabilities [4, 10]. Therefore, the amount of code in a TEE should be minimised. This can be done by offloading only security-critical parts of the application logic to a TEE, while keeping everything else in the untrusted environment. In this context, security-critical parts are characterised by explicitly requiring access to the plain text of the data in order to function properly. This applies to application logic that processes, alters or analyses the user data, whereas platform functions such as the network stack, do not need access to plain text to transmit it via the network but can also work with cipher text.

3.2.2 Enclave Interface. Since the OS is untrusted in our scenario, all ecall arguments as well as return values from ocalls must be considered as potentially malicious (c.f. IAGO attacks [2]). Consecutively, bounds checking and other measures alike need to be implemented for each ecall and ocall to ensure correct and defined

[1] http://www.project-voldemort.com/voldemort/

behaviour of the enclave in the face of arbitrary or malicious input from the untrusted world. Therefore, the attack surface of an enclave is also dependent of the number of available ecalls and ocalls and their signatures in the interface description, and should be considered during the design phase of the enclave in order to optimise security. In general, ecalls and ocalls should be as specific as possible to allow only little margin for manipulation, at the same time the number of available calls should be minimised as well.

3.2.3 Fault Isolation. When designing enclaves, sensitive code can either be comprised in one single enclave or split up and spread across multiple enclaves that interact with each other, and in their entirety constitute the trusted application. Both approaches have different pros and cons, however, splitting an enclave can be advantageous as vulnerabilities are isolated and the harm of their exploitation is limited. Partitioning code for multiple enclaves leads to hardware-based isolation between the enclaves in contrast to only software-based isolation when consolidating all trusted code in a single enclave. This allows the isolation of distinct shards of application data, as well as isolating ramifications of software bugs.

4 SINGLE VS. MULTI USER ENCLAVES

According to the performance and security metrics introduced in the last section, typical interactive cloud-based services for large numbers of users and equipped with secure enclaves can be designed in different ways. In this section we discuss the pros and cons regarding security and performance of two possible enclave designs: one that optimises security by minimising the TCB, and one optimising performance by reducing the trusted memory footprint.

The prevailing attacker model for SGX applications, that we also chose for this work, not only considers the network as untrusted but also the OS. Consequently, when securing an Internet service, exchanged network messages need to be protected, for example by using TLS. For this to work, the encryption endpoint must terminate inside the enclave, while a complete network stack is not required to be trusted. For managing the context of those TLS connections, there are two fundamentally different approaches with contrary advantages and disadvantages, that we describe in the following.

4.1 Single User Enclave (SUE)

The simplest approach from an enclave development point of view, instantiates for each client connection a new enclave which is responsible for only a single client—we call this a Single User Enclave (SUE). This design results in a low TCB and memory footprint of the enclave, and isolates the individual clients from each other in different enclaves. In this design the enclave is agnostic of client connections as the connection management is done completely outside of the enclave, i.e. the untrusted world assigns client connections to enclaves. This does not pose a security issue, as only the correct enclave possesses the right key for message decryption and otherwise will simply not work.

4.2 Multi User Enclave (MUE)

An enclave can also be designed to maintain connections of multiple client connections at once—we call this Multi User Enclave (MUE). In this design, the enclave is aware of individual connections and stores their associated meta data in an appropriate data structure. This, may comprise the negotiated TLS session parameters and keys for each client, and other application-specific meta data.

The MUE approach allows the usage of multiple identical enclaves handling distinct subsets of all connections. From a programming point of view, this does not change the enclave design but only the required heap usage per enclave. Except for exactly one enclave, the MUE approach still requires the untrusted application to assign connections to enclaves.

4.3 Trading Performance for Security

The above metrics for achieving high security or high performance are sometimes conflicting. In this section we discuss the individual properties and their conflict potential.

Single User Enclave. The SUE enclave design leads to a high level of security due to the low TCB, and also high performance for a small number of clients, at least as long as all enclaves still fit inside the EPC. However, when the number of clients increases, the performance does not scale well with this approach as the equal code pages of the enclave instances can not be deduplicated in memory. For example, each enclave will require EPC memory for storing the cryptographic library, which is also necessarily part of the working set as each received message needs to be decrypted. Hence, the according pages can not be swapped out of the EPC—at least not permanently—as they are actively used.

Multi User Enclave. The MUE design leads to a much higher and also a more dynamic heap usage inside the enclave, as it is directly dependant on the number of simultaneously connected clients. As of now, the available enclave memory can not be changed once the enclave is running. Therefore, this approach requires an estimation of the maximum heap usage at compile time (more precisely at enclave signing time) by the developer. An overestimation does not immediately harm the performance as unused pages can be swapped, but delays enclave start-up as even empty pages need to be added to the enclave upfront.

Performance Metrics. An advantage of the MUE approach is, that connected but inactive clients will not occupy large portions of EPC memory with their personal enclave, but instead they require only some space on the heap for their stored context in the shared enclave. This at the same time, is one of the most important advantages of this approach as libraries in memory such as the cryptographic library can now effectively be used for multiple clients. Thus, the overall memory footprint of MUE enclaves is much lower compared to SUE enclaves. However, synchronisation of multiple threads inside the same enclave is more critical with MUE enclaves, as more threads enter the same enclave simultaneously. This puts more load on the synchronisation primitives—in the worst case leading to more ocalls as mutexes may more frequently use the untrusted scheduler outside of the enclave.

Security Metrics. In the MUE enclave design, an exploitable security vulnerability will inherently affect multiple clients as the isolation between clients is only software-based, whereas the SUE approach offers hardware isolation between clients as they reside in individual enclaves. In addition, SUE can reduce the TCB as only a single connection is handled, leading to less code and a leaner attack surface, and thus, higher security. In contrast, MUE enclaves require the implementation of data structures for management of connection-specific data such as the TLS context. This increases the enclave code base, but allows a more efficient memory usage which improves the performance. In our two use cases the EDL interface of the enclaves contains the same amount of ecalls for both MUE and SUE enclaves. However, the MUE design requires an additional parameter for indicating the connection ID potentially leaving more opportunities to an attacker.

Summary of Discussion. From the above discussion we conclude, that SUE enclaves will be smaller in memory and code size, with client connection management code only outside of the enclave and a predictable heap usage. With the leanest attack surface and TCB, SUE enclaves offer the highest level of security. However, with this approach EPC memory usage increases quickly with the number of clients. In contrast, MUE enclaves require more trusted code and a higher memory footprint *per enclave*. Therefore, MUE enclaves provide a more stable and better performance especially for high numbers of clients. Finally, MUE enclaves offer a lower level of fault isolation, as exploiting a bug in one enclave endangers multiple users at once. Hence, an enclave developer must always trade the significance of security versus the performance requirements.

5 USE CASE APPLICATIONS

This section describes the implementation details of the two partitioned services that we chose as use cases to demonstrate the effects of our defined metrics on a real-world service.

SecureKeeper. In earlier work we proposed *SecureKeeper* [3], a secure variant of Apache ZooKeeper coordination service with SGX enclaves. We initially implemented SecureKeeper following the SUE principle with only one client per enclave.

In this paper, we augmented its design to additionally support the MUE approach by integration of a hash table that stores connection contexts of different clients in the enclave. Just as with the original SecureKeeper implementation, the enclave still contains only functionality that is imperatively required to be trusted in order to offer strong security.

Dumbledore. Our second use case application is the Voldemort key value store, that we partitioned for usage of SGX from scratch for this paper and that we call *Dumbledore*.

In general, the partitioning process of Dumbledore is similar to the one of SecureKeeper. However, in Dumbledore no application-specific state is required due to the properties of the original Voldemort service. This also leads to no necessity of synchronisation primitives inside the enclaves, as all data required for processing a message can be held on the respective thread's stack, which simplifies the enclave implementation.

6 EVALUATION

In this section we present the evaluation of our two use case services *SecureKeeper* and *Dumbledore* and display their characteristics regarding the metrics we introduced in Section 3.

6.1 Methodology

In order to evaluate the performance of our two services, we have implemented an evaluation tool allowing fine-grained evaluation

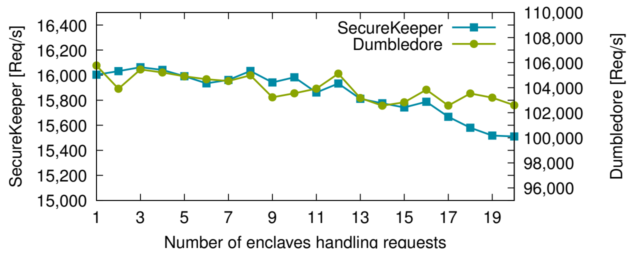

Figure 2: SecureKeeper and Dumbledore performance.

parameter tuning. The measurements of our evaluation were made using multiple threads on each client machine, simultaneously issuing requests to the target service. Our graph visualises aggregate values of multiple runs, each with its own warm-up time and a freshly started test subject.

We ran our evaluation tool on machines with 24 E5-2430 v2 cores, and the SecureKeeper replica and the Dumbledore server on an SGX-capable E3-1230 v5 server with secure enclaves in SGX hardware mode and the EPC size set to the maximum of 128 MB.

6.2 Performance Evaluation

For both of our use case services we measured the performance difference when using various numbers of MUE enclaves to handle the same amount of client connections. We used one client thread per available CPU core on each client machine (i.e. 24 threads per client machine). The client connections were equally spread across all enclaves using the modulo operator. Both services show quite a similar behaviour as can be seen in Figure 2, which illustrates the performance of the SecureKeeper coordination service and the Dumbledore key value store. In both cases, the performance decreases slightly with higher numbers of enclaves as the CPU caches can be used less effectively then. As can be seen in the graph, the performance of SecureKeeper drops by 3.2% from one enclave to 20 enclaves, and by 3.1% for Dumbledore respectively. High numbers of enclaves would eventually cause the SGX driver to swap EPC pages leading to dramatic performance drop, so we can conclude that the usage of fewer enclaves per platform is faster.

In Figure 2, usage of the SUE approach would resemble the special case of using 72 enclaves to handle the connections (not visible in the graph), as in that case each client connection is mapped to a dedicated enclave responsible only for this single connection. Obviously, in this case the overall throughput is much lower compared to the MUE approach with a reasonably (low) number of enclaves.

6.3 Enclave Memory Footprint

By monitoring segmentation faults after removing all permissions to the enclave pages, we obtained the working set size of the target enclave in the EPC. After the system has warmed up and all connections are established, the working set memory footprint decreases compared to a higher EPC memory usage during enclave start-up. While SecureKeeper requires 322 during start-up and drops to 94 later (1.26 MB/0.37 MB), the Dumbledore KV store starts with 295 and falls back to 67 (1.15 MB/0.26 MB). Our measurements emphasise the small memory footprint of enclaves in partitioned applications in general, and the low working set memory footprint compared to the larger binary size of the compiled enclave which is quite close to the start-up memory usage outlined above. In addition to the code pages from the compiled enclave binary, the TCS pages, the enclave stack and heap are also part of its runtime memory footprint. For both services we also measured an increase of the enclave's memory footprint of about 5% when a new client connects, which leads to the conclusion that the MUE approach offers a favourable EPC usage efficiency. Note that an increase of memory usage of only 5% in the MUE approach, is analogous to the creation of a completely new enclave in case of SUE.

6.4 TCB Evaluation

In order to quantify the overhead for implementing MUE enclaves instead of SUE enclaves, we also measured the TCB of the enclaves. The application logic of the enclaves is relatively small with 4239 SLOC for SecureKeeper and 702 SLOC for Dumbledore. Due to required message serialisation code from the original ZooKeeper project in the enclave (c.f. SecureKeeper [3]), the SecureKeeper's TCB is larger. For both applications the SDK libraries are required inside the enclave adding roughly 18k SLOC. We conclude that the additional trusted code required to implement the multi-user capability is relatively small (only 83 SLOC for Dumbledore), and is worth it despite the increased TCB. In both use cases, the MUE enclave requires an additional map data structure for storing connection contexts and individual encryption keys per client.

7 RELATED WORK

Existing works can be categorised into platforms for legacy applications and partitioning approaches. Haven [2] and Graphene-SGX [19] feature execution of unchanged binaries in enclaves based on a Library Operating System (LibOS) inside the enclave. SCONE [1] offers a container abstraction inside SGX enclaves similar to Docker. However, these approaches lead to a very large enclave size, both, regarding their TCB and their memory footprint, which will impact performance negatively.

The authors of Ryoan [8] try to limit the enclave size targeting execution of generic modules in an enclaved sandbox. Panoply [17] is also a LibOS approach focusing on the reduction of the TCB to provide stronger security. Glamdring [11] offers support for partitioning applications on the source code level in order to design minimal enclaves. Weisse et al. [20] propose a faster enclave interaction scheme by relying on threads inside the enclave polling memory shared with the untrusted world using spin locks.

The above approaches comprise various approaches for running legacy and new code components inside enclaves and speeding up interaction with enclaves. The goal of our investigation and discussion of enclave security and performance metrics, however, is a guideline regarding enclave design principles and is orthogonal to these approaches.

Additional details regarding the characteristics of the SGX hardware and the official SGX kernel module are given by Gjerdrum et al. [6]. In their paper, the authors present measurements of enclave transitions and properties of the SGX kernel module's eviction strategies. However, neither do they evaluate the performance of real-world applications as we do, nor do they investigate scalability behaviour and multi-threading scenarios which are particularly important in a cloud setting as we do in this paper.

8 CONCLUSION

In this paper, we identified and discussed metrics affecting the security and performance of applications incorporating SGX enclaves. Derived from these metrics we presented two possible enclave design approaches for mainly data-handling services, applied them to two different use cases and evaluated their characteristics. By this we conclude that an enclave designer has to pay for security by sacrificing performance—at least to a certain degree.

9 ACKNOWLEDGEMENT

This project received funding from the German Research Foundation (DFG) under grant no. KA 3171/9-1.

REFERENCES

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*.
[2] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*.
[3] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Middleware*.
[4] Coverity Report 2014. Coverity Scan Open Source Report 2014. http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf.
[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*.
[6] Anders T. Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. 2017. Performance of trusted computing in cloud infrastructures with Intel SGX. In *CLOSER*.
[7] Patrick Hunt, Mahadev Konar, FP Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *ATC*.
[8] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *OSDI*.
[9] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. (January 2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203
[10] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. 2014. Quantifiable Run-time Kernel Attack Surface Reduction. In *DIMVA*.
[11] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rudiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *ATC*.
[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv* (2018). arXiv:1801.01207 http://arxiv.org/abs/1801.01207
[13] Moxie Marlinspike. 2017. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/.
[14] Frank McKeen. 2014. Intel Software Guard Extensions (Specification).
[15] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP*.
[16] Mark Russinovich. 2017. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.
[17] Shweta Shinde, D Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*.
[18] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: A Library Operating System Optimized for Intel SGX. In *CF*.
[19] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. *ATC*.
[20] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls. In *ISCA*. ACM Press.
[21] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*.