This is the author's version of the work. For personal use only, not for redistribution.
The definitive version will be published in the proceedings of the 2018 19th Annual Middleware Conference (MIDDLEWARE).

*E*Actors: Fast and flexible trusted computing using SGX

Vasily A. Sartakov
TU Braunschweig
sartakov@ibr.cs.tu-bs.de

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

Sonia Ben Mokhtar
INSA Lyon
sonia.benmokhtar@insa-lyon.fr

Sara Bouchenak
INSA Lyon
sara.bouchenak@insa-lyon.fr

Gaël Thomas
Telecom SudParis
gael.thomas@telecom-sudparis.eu

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

ABSTRACT

Novel trusted execution support, as offered by Intel's Software Guard eXtensions (SGX), embeds seamlessly into user space applications by establishing regions of encrypted memory, called *enclaves*. Enclaves comprise code and data that is executed under special protection of the CPU and can only be accessed via an enclave defined interface. To facilitate the usability of this new system abstraction, Intel offers a software development kit (SGX SDK). While the SDK eases the use of SGX, it misses appropriate programming support for inter-enclave interaction, and demands to hardcode the exact use of trusted execution into applications, which restricts flexibility.

This paper proposes *E*Actors, an actor framework that is tailored to SGX and offers a more seamless, flexible and efficient use of trusted execution – especially for applications demanding multiple enclaves. *E*Actors disentangles the interaction with enclaves and, among them, from costly execution mode transitions. It features lightweight fine-grained parallelism based on the concept of actors, thereby avoiding costly SGX SDK provided synchronisation constructs. Finally, *E*Actors offers a high degree of freedom to execute actors, either untrusted or trusted, depending on security requirements and performance demands. We implemented two use cases on top of *E*Actors: (i) a secure instant messaging service, and (ii) a secure multi-party computation service. Both illustrate the ability of *E*Actors to seamlessly and effectively build secure applications. Furthermore, our performance evaluation results show that securing the messaging service with *E*Actors improves performance compared to the vanilla versions of JabberD2 and ejabberd by up to 40×.

CCS CONCEPTS

• **Security and privacy → Trusted computing**;

KEYWORDS

Actors, Intel SGX, Trusted Execution

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 '18, December 10–14, 2018, Rennes, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5702-9/18/12...$15.00
https://doi.org/10.1145/3274808.3274823

ACM Reference Format:
Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. *E*Actors: Fast and flexible trusted computing using SGX. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3274808.3274823

1 INTRODUCTION

Enforcing data privacy in cloud infrastructures is a primary concern. Fortunately, with the advent of trusted execution support as offered by Intel's Software Guard eXtensions (SGX) [37], application code and data can be protected from unauthorised access, including privileged code such as the operating system or the hypervisor. This is achieved by providing *enclaves*, which resemble regions of encrypted memory that can be seamlessly embedded into user space applications and are only decrypted while processed inside the CPU. For ease of use, Intel offers a software development kit (SGX SDK) [27], which provides an enclave definition language and an associated code generation process to bootstrap enclave provisioning and facilitate enclave access.

While the SGX SDK offers a convenient and intuitive programming model, it has several shortcomings. Calls to and out of an enclave require the execution of SGX specific call instructions and an associated execution mode transition (i.e. from normal to trusted mode or vice versa). Such a call is costly as it requires between 8000-9000 CPU cycles [39]. Since a thread has first to leave an enclave to perform a system call, the cost of these execution mode transitions drastically slows down an application that often interacts with the operating system. This is typically the case for multi-threaded applications, because the SGX SDK offers synchronisation constructs, like mutexes, that lead to frequent system calls. The execution mode transition cost becomes also very expensive for applications that use multiple enclaves to further strengthen resiliency of a system by a more fine-grained compartmentalisation. Finally, the SGX SDK forces developers to explicitly specify the use of enclaves during the early phases of the development and restricts thereby the freedom to decide on trusted execution based on the deployment scenario. Here, more flexibility could ease development and deployment.

Over the recent years some of the aforementioned issues have been identified. SCONE [6] and Haven [8] focused on executing whole legacy applications on top of SGX. Besides other things, these approaches reduced synchronisation cost by using user-level threading. However, the downside is an increased trusted computing base (TCB), which widens the attack surface. SCONE, HotCalls [52], and Switchless Calls [28] provide an asynchronous call interface, which eliminates the cost of execution mode changes as it is inherent to

the SGX SDK. Panoply [49] discusses multi-enclave applications but does not offer mechanisms to speed-up inter-enclave communication. In essence there is currently no system that addresses all the shortcomings of the SGX SDK and especially the support for fast inter-enclave communication is essentially missing.

Such support would allow to push the boundaries of building SGX-based privacy-preserving systems much beyond its current status, where all the data is processed inside a single enclave (e.g., [38, 47]) and enable fine-grained compartmentalisation. Indeed, supporting multi-enclave applications would allow building applications that compute aggregate results from data provided by multiple distrusting parties. In this case each party would store its data inside a separate enclave and the aggregated result would be computed using secure multi-party primitives (e.g., [18]). Furthermore, efficiently supporting multi-enclave communications would also allow building applications that split sensitive data into insensitive parts to be processed independently (e.g., [22, 40]). Compared to the single enclave application model there is an opportunity to establish a stronger adversarial model, where the assumption that the code running inside an enclave is fully trusted, can be relaxed.

In this paper we propose the *E*Actors framework that offers dedicated support for implementing efficient multi-enclave applications and makes trusted execution a matter of compile-time configuration. Thereby, it avoids a large TCB and addresses the outlined shortcomings of the SGX SDK. *E*Actors achieves this by proposing the actor model as an alternative programming abstraction for implementing applications on top of SGX. The *E*Actors framework features $_{e}actor$s. In line with the classical actor model, an $_{e}actor$ is self-contained and communicates with other $_{e}actor$s purely via messages. Thereby, $_{e}actor$s can be executed in parallel and typically don't perform blocking operations. Together these properties match the demands of the SGX execution model quite well, as asynchronous interaction has been identified as performance friendly, while blocking operations (e.g. due to synchronisation) need to be avoided. Furthermore, the actor model naturally encourages the splitting of functionality in small independent and self-contained tasks, which simplifies the assignment of tasks to one or more trusted execution contexts and builds the basis for a flexible deployment.

In sum the *E*Actors framework makes the following contributions:

- **Fast interaction and multi-enclave support.** *E*Actors implements the actor model in the context of SGX. Thereby, special focus has been put on using non-blocking communication for message exchange between $_{e}actor$s inside the same and different enclaves, as well as mixed trusted/untrusted settings. This avoids costly execution mode changes and enables to vary the number of utilised threads to flexibly scale parallelism as needed.
- **Resource efficiency.** The *E*Actors framework features a small trusted computing base and avoids dynamic memory allocation to keep the memory footprint of *E*Actors within the performance-friendly bounds of SGX.
- **Flexible use of trusted execution.** All important system settings, including trusted or normal execution as well as co-location or separation of $_{e}actor$s, are configurable.
- **Versatile applicability.** There is a set of system $_{e}actor$s that handle networking and storage. We also implemented a secure instant messaging service and a secure multiparty computation service that feature the use of multiple enclaves. The evaluation results show that the *E*Actors-based messaging service outperforms the vanilla versions of JabberD2 and ejabberd between 1.11× and up to 40× for a secure chat scenario.

The remainder of this paper is organised as follows. In Section 2, we motivate the need for multi-enclave applications, introduce the current system support of SGX and outline our multi-enclave aware attacker model. Section 3 outlines the core concepts of *E*Actors and details the programming model. Networking and storage support are explained in Section 4. In Section 5, we describe a secure instant messaging service and a multi-party computation service as use cases for the *E*Actors framework. Section 6 discusses the performed evaluation. Section 7 details related approaches, while Section 8 finally concludes the paper.

2 TOWARDS MULTI-ENCLAVE SCENARIOS

To motivate our actor framework, we first discuss the need of multi-enclave applications and present a secure instant messaging service scenario that features the use of multiple enclaves (Section 2.1). Then, we present a background on existing SGX-based system and application development support (SGX SDK) and discuss its drawbacks (Section 2.2). Finally, we describe the assumed attacker model, which addresses multi-enclave settings (Section 2.3).

2.1 The case for multi-enclave applications

The privacy-preserving processing of sensitive data in distributed systems can be done using multiple means. The most well-known approach is the use of cryptographic techniques such as homomorphic encryption(e.g., [20]), which allows performing operations over encrypted data.

As these techniques incur a heavy overhead, other solutions based on data partitioning have emerged. The principle behind data partitioning is the splitting of a sensitive piece of information into insensitive parts and making these parts processed by independent servers. For instance, to process a user request in a location-based service, one can split this request into three pieces: the user identifier (e.g., IP address), the user location (e.g., GPS coordinates) and the search query (e.g., medical doctor in an area of 500m around the user location). Next, these can be processed by three non-colluding servers (e.g., KOI [22]). Similar services based on data-partitioning have been proposed for private Web search [40], private recommender systems [21] or online voting [16].

However, one of the limitations of these solutions is that they rely on a weak adversarial model where the collaborating servers are assumed not to collude. A mean to alleviate this assumption is to rely on trusted computing hardware were the sensitive data get processed in secure enclaves. Various solutions have been proposed in this direction (e.g., [38, 41, 47]) but these solutions rely on a single enclave. Relying on a single enclave necessarily builds on the assumption that the code running inside the enclave is fully trusted. However, this might not be the case as the code base running inside the enclave might be relatively large and thus difficult to verify.

Furthermore, the need for multi-enclave applications is even more appealing for applications involving a set of data providers that do not trust each other and that request to have their data isolated from the one of other providers. These applications that generally rely on secure multi-party computation schemes (e.g., [18]) are natural candidates for multi-enclave environments, where they would benefit from a stronger adversarial model enabled by trusted hardware. However, developing multi-enclave applications was limited up to now due to the inadequate programming support provided by the SGX SDK as further explained in Section 2.2.

A secure instant messaging service. To further motivate the need of such system support, we discuss a secure instant messaging service. Instant messaging builds one the of main communication mechanisms of the Internet. It is used for exchanging all kind of private and business critical information that needs to be protected.

While end-to-end encryption between two users is getting more and more common, secure group chat functionality requires additional measures as data may be processed in a plain form on the server side [44]. Trusted execution as offered by SGX has been proposed for protecting user profiles and match making for the Signal instant messenger [2]. However, a fully-fledged solution for a secure instant messaging service is missing so far.

In principle, all privacy sensitive data of an instant messaging service can be processed inside a single enclave. However, as soon as a single flaw can be exploited in the trusted computing base of the enclave the whole service together with all user data is at risk. Thus a setting where in the extreme each user is confined to a single enclave, plus dedicating each group chat to a separate enclave, improves security. Here, if a user could trigger an exploit in her own enclave, this does not necessarily imply she would right away gain access to sensitive information of other users. However, designing this application with multiple enclaves by using the SGX SDK is not practical as further discussed in the following section.

2.2 Background on SGX

As aforementioned, SGX can create one or more isolated contexts, called enclaves, inside an application. Enclaves feature the following properties: (i) an enclave is isolated from other enclaves, other untrusted applications and higher-privileged entities such as the operating system through memory access control mechanisms enforced by the CPU; (ii) memory encryption is used at all times to defend against physical attacks; and (iii) enclaves support remote attestation by which the identity of an enclave and its integrity can be proven to a remote party.

Until now, two different approaches for programming applications with enclaves have been used: the standard approach makes use of the Intel SGX SDK, which provides its own interface definition language for code generation of stubs that enable interaction between the untrusted application and its associated enclaves. Alternatively, there are projects offering runtime support in the form of a library operating system [8, 15], in order to execute entire legacy applications inside enclaves without any changes.

SGX SDK Programming model. The SGX SDK by Intel offers all the necessary tool support to create enclaves and embed them into an application. In order to interact with an enclave, a developer has to declare the enclave interface, i.e. the enclave functions available to be called by the untrusted application, using the Enclave Description Language (EDL) [27]. This implies a programming model with several characteristics that we describe in the following.

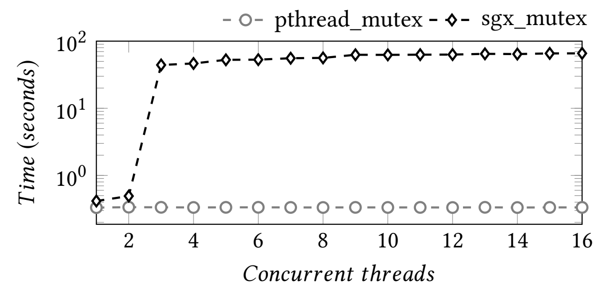


Figure 1: Concurrent dequeuing of elements from a stack

Costly enclave interaction. The SDK allows the generation of code handling the data movement across the enclave boundary in a way similar to RPC systems. This includes its ability of performing the necessary memory copy operations during the transition to and from the enclave. In addition to the natural overhead of copying data to and from the enclave, the transition into the enclave induces an inevitable overhead of approximately 8000 CPU cycles [52].

Hardcoded partitioning. Using the EDL and the code generation workflow of the SGX SDK also forces the developer to explicitly define which functionality needs to be placed inside the enclaves during compile time, thereby leaving little to no flexibility for late design and deployment decisions.

Costly synchronisation. The SGX SDK offers basic synchronisation primitives such as mutexes, allowing to synchronise access to shared data structures, as enclaves can be concurrently entered by multiple threads. However, as threads cannot be suspended by the operating system inside an enclave when waiting for a condition to be fulfilled, either spin-locking needs to be performed or the thread has to step out of the enclave, which requires a costly execution mode switch. The current solution of the Intel SGX SDK is to spin lock for a defined (short) time period before eventually leaving the enclave, in order to prevent enclave exits for very short waiting periods that would otherwise lead to the above described transition delay. In order to quantify the effects of this, we measured the concurrent dequeuing of 1000000 elements from a mutex synchronised stack for a variable number of consumer threads. Compared to a pthread-based untrusted implementation, the SGX SDK-based variant is several orders of magnitudes slower. Figure 1 highlights the dramatic impact of this approach with costly transitions.

Missing fast inter-enclave communication. Another critical aspect of the SGX SDK is the lack of an efficient enclave-to-enclave communication mechanism. In essence, this has to be implemented using standard calls via the EDL interface, implying the high cost of entering and exiting an enclave. Essentially, a thread has to exit the source enclave and enter the target enclave.

Scarce memory resources. Enclave creation and its memory layout are handled by the SDK and the SGX kernel module. During enclave creation, the enclave code and data are copied page-by-page into the SGX-protected memory range, called Enclave Page Cache (EPC). In current CPUs the EPC is limited to 128 MB whereof only 93 MB

are available for usage due to SGX-internal data structures [6]. If the memory combined usage of all enclaves on a machine exceed the available EPC, the SGX kernel module executes a special paging process that allows moving EPC pages to normal system memory and which implies their (re-)encryption, leading to a severe performance degradation [11, 39, 46].

Runtime support for legacy applications. Haven [8], SCONE [6] and Graphene-SGX [15] enable the execution of unchanged legacy applications inside an enclave. Approaches of that kind require extensive runtime support inside the enclave in the form of a library operating system [8, 15], or at least an extended libc [6]. Thereby, securing the interface between the untrusted environment and the enclave poses an additional challenge.

While the execution of a legacy application on top of SGX is tempting, the resulting additional trusted code base is not negligible. Hence, such an approach has security implications, but the required runtime support also consumes large amounts of the precious EPC space. For complex applications with relevant in-memory state this will likely lead to performance-degrading EPC paging. As a consequence, we argue that if an existing application needs to be secured, these approaches may have their merits. However, for newly developed applications, a tailored solution that aims at a small trusted computing base and that targets minimisation of resource usage — especially regarding memory usage — is preferable.

2.3 Attacker model

In line with previous works, we assume a powerful adversary who has superuser privileges and physical access to the hardware. The adversary can control the entire software stack, including privileged code, the Operating System (OS) kernel, and other system software. This enables the adversary to replay, record, modify, and drop any I/O between the environment and an enclave. We assume the implementation of the hardware and its trusted execution support (i.e. SGX) are correct and that the associated system software (i.e. utilised parts of the SGX SDK) does not contain flaws of any kind. We also assume that the utilised cryptographic protocols are correct and that the adversary is computationally bound and therefore is unable to execute brute force attacks successfully.

We do not address denial of service or side-channel attacks that exploit timing [12, 32] and page faults [54]. Furthermore, mitigation strategies are under development [48] and one can assume that the hardware manufacturer will provide appropriate fixes over time.

Unlike previous work, we do not fully exclude programming bugs or inadvertent design flaws in enclaves (i.e. in the application code), and it has already been shown that enclaves can be attacked [31, 51]. Indeed, *E*Actors is intended to facilitate the design of multi-enclave applications, where the sensitive data is spread across multiple enclaves. In this context, if an enclave is compromised only parts of the data is exposed to the attacker—the extent of which is subject to the application design.

2.4 The actor model as starting point

For *E*Actors we assume an *actor model* that is stripped-down to the core principles. An actor is a computational entity that can in reaction to a received message send messages to other actors. Thereby, multiple actors can work concurrently and don't possess shared execution state. In the next section we will outline how such a generic abstraction enables us to address the aforementioned shortcomings of the SDK, while at the same time requiring less memory compared to the outlined support for executing entire legacy applications on top of SGX.

```
1  struct state {struct channel chan[2]; int first;}
2
3  void aping(struct actor* self) {
4    if(self->state->first) {
5      self->state->first = 0;
6    } else {
7          /* receive a pong */
8      char* msg = recv(&self->channel[0]);
9      if(msg == NULL)
10       return;
11   }
12   /* send a ping */
13   send(&self->channel[1], "ping");
14 }
15
16 void aping_ctr(struct actor* self) {
17   self->state->first = 1;
18   connect(self->channel[0]);
19 }
```

Listing 1: Pseudo-Code of an $_{e}actor$.

3 *E*ACTORS OVERVIEW

The *E*Actors framework offers an actor-inspired programming model and has the following design goals in response to the identified shortcomings of the SGX SDK (see Section 2.2):
Firstly, the *E*Actors framework enables fast message exchange between $_{e}actor$s – the notion of actors inside the *E*Actors framework. This especially applies for $_{e}actor$s communication across enclave boundaries, which facilitates the use of multiple enclaves.
Secondly, our *E*Actors programming model avoids costly fine-grained synchronisation, because actors do not rely on shared state but instead exchange messages.
Thirdly, the *E*Actors framework offers a flexible use of trusted execution. We separate the code of an $_{e}actor$, which is implemented in a standard programming language, from the associated deployment policy. This policy defines the assignment of a given $_{e}actor$ to computational resources (i.e. processors and threads) and especially enclaves. This is facilitated by providing uniform communication primitives, which transparently select adequate communication mechanisms, regardless where an $_{e}actor$ is placed (i.e., inside or outside of an enclave) or with whom it communicates. As a result, we can deploy an $_{e}actor$ either in an enclave or outside of an enclave without further modifications to its application logic.

In the remainder of this section, we first present the *E*Actors programming model. Next, we outline the *E*Actors runtime and its support for dynamic configurations. Finally, we explain how $_{e}actor$s efficiently communicate and detail the devised uniform communication primitives.

3.1 *E*Actors programming model

The *E*Actors framework features a lean actor programming model. In order to implement an $_{e}actor$, the developer has to provide the body function of the actor, which contains the application logic, and a constructor function. The purpose of the latter is to initialise communication channels to other $_{e}actor$s and initialise the private state of the $_{e}actor$ at startup time. Thus, in essence, connections are

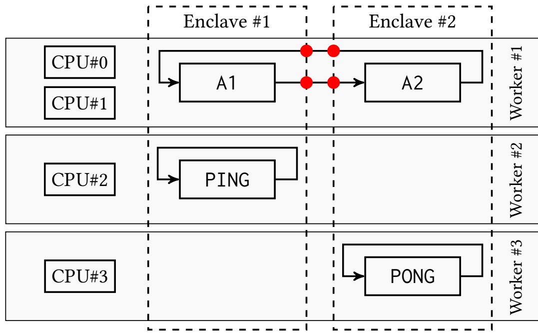


Figure 2: Deployment of $_e$*actors*, workers, and enclaves

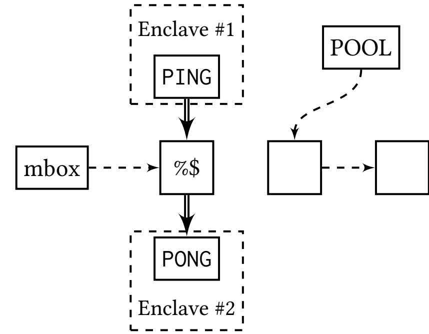


Figure 3: Message exchange between two $_e$*actors*

statically assigned and we avoided additional naming and resolving mechanisms with the aim of a small trusted computing base.

Listing 1 presents a simplified example of an $_e$*actor* written in C. The $_e$*actor* sends a `ping` message when it receives a `pong` message from a `PONG` $_e$*actor*. The latter works analogous to the presented example. The structure `state` declares the private state of the actor, the `aping` function is the body function, and the `aping_ctr` is the constructor. Once called by the runtime the constructor function initialises the field `first` to `1` (line 17), and connects the `PING` $_e$*actor* to a `PONG` $_e$*actor* via the communication channel `0` provided by the runtime. Next, the *E*Actors runtime regularly executes the body of the $_e$*actor*. When the runtime executes `aping` for the first time (line 4), `aping` simply generates an initial `ping` message (line 13) in order to start the `ping`/`pong` message exchange. Each time the body function `aping` of the $_e$*actor* is executed either a message is received via the channel and `ping` is emitted or the $_e$*actor* simply returns as no data needs to be processed.

The example outlines the lean API of our framework, which turned out to be well suited to implement the targeted use cases and is in line with the aim of designing a framework featuring a small trusted computing base.

3.2 *E*Actors runtime

At its core the *E*Actors runtime enables to map computational resources in terms of CPUs and threads to $_e$*actors*. More importantly, it allows to define if an $_e$*actor* should be executed in a trusted execution context (i.e. an enclave) or as part of the untrusted application. Thereby, the use of multiple enclaves is supported.

Figure 2 illustrates an example deployment. It defines $_e$*actors* (`A1`, `A2`, `PING` and `PONG`) and two enclaves. `A1` and `PING` are located in the first enclave, while `A2` and `PONG` are located in the second enclave. To execute these $_e$*actors*, three workers are utilised. A worker is a the framework abstraction to manage a POSIX thread. The first worker is bound to the CPUs 0 and 1, and executes the $_e$*actors* `A1` and `A2` in round-robin. The second worker is bound to the CPU 2 and executes only the $_e$*actor* `PING`, while the third worker is bound to the CPU 4 and executes only the $_e$*actor* `PONG`. If all $_e$*actors* assigned to a worker are confined to the same enclave, the worker does not leave the enclave (e.g. `Worker#2` for `PING` $_e$*actor*). Otherwise, as in the case of $_e$*actor* `A1` and `A2` the worker has to migrate from enclave to enclave in order to execute the body functions, which will result in costly execution mode transitions. Such an approach usually should be avoided but can be used if $_e$*actors* are dispersed over multiple enclaves and are infrequently activated.

To implement the outlined scenario and more complex ones, the developer defines the necessary mapping of computational resources and trusted execution contexts of $_e$*actors* in a special configuration file. This file builds the basis for a custom build process and leads to the generation of the source code tree. These generated files are used together with the SGX SDK for the independent compilation of binaries for the untrusted application and the enclaves that implement the envisioned deployment.

When the application is started, the generated *E*Actors runtime creates the enclaves, allocates the private state, calls the constructors of the actors and creates as well as starts the workers. Each worker then executes the body functions of its assigned $_e$*actors* in a round-robin order.

3.3 Uniform communication primitives

As outlined in Section 3.1, $_e$*actors* are in the common case connected using bi-directional communication channels. These channels provide uniform communication primitives to enable the outlined flexible deployment support of $_e$*actors* (see Section 3.2).

To provide channels, the lower layer of the *E*Actors framework offers so called `nodes`, `pools` and `mboxes`. A `node` is a memory object, which consists of two elements: a *header* and a *payload*. The payload is a memory region used to transfer *E*Actors messages. The header consists of a set of data pointers to manage `nodes`. A `pool` is an abstraction, which refers to a set of empty `nodes`. The framework preallocates private and public `pools` at system start. A `mbox` is an abstraction, which refers to a set of linked nodes used for message exchange. `Mboxes` and `pools` are organised in the form of bi-direction double linked lists implemented on top of Hardware Lock Elision [42], but have slightly different APIs and semantics: `mboxes` offer FIFO semantic, while `pools` implement LIFO semantic.

To achieve uniform communication, the channels hide the location of the communicating $_e$*actors*. If both $_e$*actors* are located in the same enclave there is no need for message encryption. To send a message, an $_e$*actor* firstly needs to dequeue one `node` from a `pool`, fill the payload of the node, and enqueue it to a `mbox` (Fig. 3). To receive, an $_e$*actor* should poll the `mbox`, and upon reception of a message read the content and return the `node` back to the `pool`. If $_e$*actors* are located in different enclaves, messages are encrypted

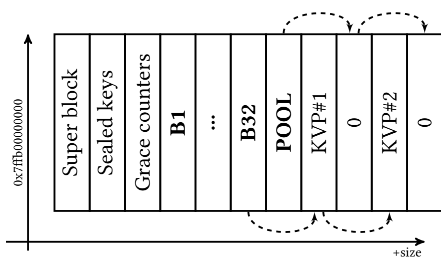


Figure 4: Memory layout of POS

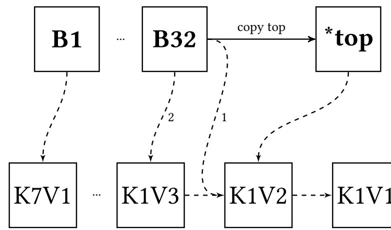


Figure 5: get(K1) example

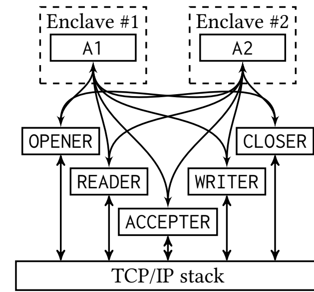


Figure 6: Networking by system actors

in order to protect the $_{e}actor$s against a malicious runtime. To establish encrypted communication between enclaves, the local attestation support of the SGX SDK is used [27].

$_{e}actor$s initialise a communication channel in two phases. During the first phase, an $_{e}actor$, called the initiator, connects to a channel (see line 18 of Listing 1). The runtime simply records the initiator. During the second phase, a second $_{e}actor$, called the client, connects to the channel. If the initiator and the client are located in two enclaves, the communication channel transparently encrypts the messages, except if the channel is configured as non-encrypted. Otherwise, the communication channel simply pushes and pops non-encrypted messages on the underlying `mbox`.

4 SYSTEM COMPONENTS

To facilitate fast and scalable $_{e}actor$s applications and because an enclave cannot directly interact with the operating system, the *E*Actors framework provides support for two system components: storage and networking.

4.1 Persistent object store

The main motivation of the provided persistent storage support is to offer a lean concurrently accessible abstraction that is easily accessible to all $_{e}actor$s to handle configuration and application data. It is provided as a Persistent Object Store (POS) and is based on a memory-mapped file that utilises the page cache of the kernel [35, 43]. This allows us to avoid system calls besides infrequent calls to make the in-memory state actually persistent (i.e. using `sync`). In essence, objects can be accessed and stored using their assigned key. The keys are mapped to a configurable number of stacks that share their implementation with the aforementioned `pool` abstraction.

The current design favours writes and reads of frequently changed data. If a common file system storage is required, *E*Actors can be extended similarly to the networking support described in Section 4.2 by implementing dedicated untrusted $_{e}actor$s that execute the necessary system calls.

Storage life cycle. The framework initialises the storage at boot time. It starts by mapping a storage file (or a device) at a virtual address. Because the POS uses virtual addresses inside its internal objects, the storage file has to be mapped at a fixed address.

During the first boot, the framework initialises a 'super block' with basic data (version of the storage, sizes of the pages etc.) (see Fig. 4). The framework also splits the storage into fixed sized regions and assigns them to a *storage pool*, which is again realised by a stack.

At runtime, an $_{e}actor$ pops an entry from the storage pool, fills it with a key-value pair, and inserts the entry with a *set(k,v)* operation. Later, the $_{e}actor$ can retrieve the entry with a *get(k)* operation.

Set and get. The POS uses a configurable number of stacks to manage its objects. In order to preserve linearisability, an outdated key-value pair remains in the stack while a new version is inserted. The *set(k,v)* operation thus simply pushes the new key-value pair on top of the stack associated with the hash code of the key.

The *get(k)* operation scans the stack associated with the hash code of the key. Since the first key-value pair is the latest inserted one, the *get(k)* operation simply returns the first value associated with the first occurrence of the key in the stack. As a result, our implementation provides fast writes, and relative fast reads. Moreover, more frequently changed objects can be retrieved faster than less frequently changed ones. Because the *set(k,v)* operation does not remove old keys, our implementation is linearisable. In the worst case, as presented in Fig. 5, if a *get(k)* operation (see arrow number 1, Fig. 5) starts before a *set(k, v)* operation (see arrow number 2, Fig. 5), *get(k)* returns the value that was associated with the key at the beginning of the *get(k)* operation.

While inserts are lightweight and fast, outdate entries will accumulate over time in the POS. These are removed by a housekeeping $_{e}actor$ – the *Cleaner*. To enable a safe deletion of outdated objects from the stack, all read operations of the outdated entries need to have finished. This can be detected by monitoring that all $_{e}actor$s connected to the POS have been executed at least once since the update that invalidated the object in question. The marking of outdated values is performed immediately after updates in the course of a *set(k,v)* operation to ease cleaning.

Storage encryption. The POS supports encryption for the key-value pairs. If the key is encrypted, the hash code is computed from the deterministically encrypted version of the key, which means that the storage does not have to decrypt the key to retrieve the value associated with a key: the storage simply compare the encrypted keys. Additionally, to preserve the integrity of the pairs, the POS does not store the keys and the values separately. Instead, it stores the encrypted pairs as combined values.

At runtime, an $_{e}actor$ can store its encryption key in its private state. To secure encryption keys across reboots they can be stored as sealed data inside the POS using the support of the SGX SDK. Handling reboot and fork attacks is out of scope but could be addressed by adopting an approach such as LCM [9] or ROTE [36].

4.2 Networking

The main target of the devised networking support is to horizontally scale with the number of established network connections and the requested processing of the application logic. Thereby, both layers should be independently scalable. To achieve this goal, we internally use the `mbox` abstraction to connect the $_e$*actor* responsible for basic network functions with the upper application logic. This is beneficial as the `mbox` abstraction enables concurrent access by multiple readers and multiple writers.

The actual networking support enables TCP via five untrusted actors (see Fig. 6). The `OPENER` $_e$*actor* creates a non-blocking socket as an instance of a server socket or of a client socket. The `ACCEPTER` $_e$*actor* is used for server sockets to accept connections from clients. The `READER` and `WRITER` $_e$*actor*s are respectively used to receive and send messages by using the `recv` and `send` system calls. Finally, the `CLOSER` $_e$*actor* closes the sockets.

When a client $_e$*actor* creates a new socket with an `OPENER`, it indicates a `mbox`, which is used by the `OPENER` to return the socket identifier. For a client socket, the client $_e$*actor* sends to the `READER` the socket identifier and a `mbox`, which is then used by the `READER` to forward the incoming messages. For a server socket-based interaction a similar workflow is performed.

While typically the outlined networking support will be suitable and the necessary actors will be instantiated during the application startup, sometimes custom network support is required. For these cases the application can implement custom and therefore optimised network actors as done in the context of our messaging service use case (see Section 5).

5 *E*ACTORS USE CASES

We implemented two use cases on top of *E*Actors: (i) a secure instant messaging service, and (ii) a secure multi-party computation service. Both feature the use of multiple enclaves and exercise the core components of our framework.

5.1 XMPP instant messaging service

Instant messaging is used to exchange privacy-sensitive information. Trusted execution builds a means to make it more secure (see Section 2.1). Accordingly, we designed and implemented an *E*Actors-based variant of an instant messaging service that exercises all the outlined features. This includes flexible configuration, uniform communication primitives, including its underlying mechanism and the devised system support for networking and storage. Beside security based on the use of multiple enclaves, performance and scalability were prime design goals.

The developed XMPP instant messaging service implements core parts of the XMPP protocol [45] and supports two types of communication: One-to-One (O2O) and One-to-Many (O2M). O2O allows end-to-end encrypted messaging between two participants, which resembles the *de facto* approach for modern messengers. In principle, these type of connections can all be managed inside a single enclave as only the information about online users and statistics have to be secure. For One-to-Many (O2M) the situation is different as it offers support for group chats. Here the server decrypts the messages of each user and re-encrypts for all members of the group. While in principle also a single enclave could be used our implementation supports a dedicated enclave for each group chat to improve isolation.

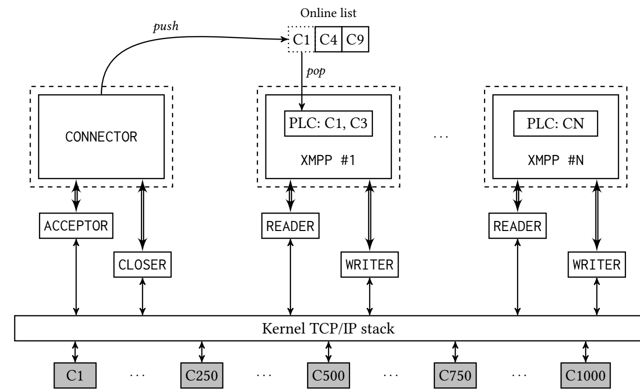


Figure 7: XMPP service architecture

5.1.1 *Architecture.*

In the following we present a simplified version of the overall architecture implemented on top the framework. As shown in Fig. 7 the service is decomposed in two parts. The enclaved `CONNECTOR` manages incoming connection with the help of an `ACCEPTOR` network $_e$*actor*. Thereby the `CONNECTOR` stores all established connections in the `Online list`. This list is shared with the second part of the service: the enclaved XMPP $_e$*actor*s and its associated *READER* and *WRITER* networking actors. This second part implements the XMPP protocol logic. It can be instantiated multiple times, as shown in the figure, to achieve scalability but also to confine group chats in dedicated XMPP $_e$*actor*s, which are assigned to dedicated enclaves.

5.1.2 *Communication workflow.*

While the services feature a number of functionalities, we decided to focus on the message exchange between an `XMPP` instance and the connected users. The *XMPP* $_e$*actor* fetches users, i.e. their socket descriptors, that it plans to serve from the `Online list`. Next, it locally stores the list in the `PCL` (private client list) structure and requests to read data from all connections using a batch request that is sent to the assigned `READER` actor. Once the `READER` $_e$*actor* starts processing it queries all provided client sockets. The results are fed back to the *XMPP* $_e$*actor*, which processes them one by one. To achieve scalability at the messaging level, each entry in the batch is equipped with an `mbox` that is devoted to a specific user. The `READER` fills these `mbox` if the associated connection provides data. Sending data to the client is implemented in a similar fashion.

As a design decision we omitted the use of channels in this particular case, this does not break the proposed configurability as the networking $_e$*actor*s always need to be executed outside an enclave to perform system calls. This way communication stays unencrypted at the framework level but is encrypted at the service level independent of the fact if the *XMPP* is executed inside or outside of an enclave.

5.1.3 *Deployment of* $_e$*actors.*

The deployment shown in Fig. 7 is a result of a careful evaluation of multiple alternatives and associated measurements. This was enabled by the flexible configuration support of the *E*Actors framework.

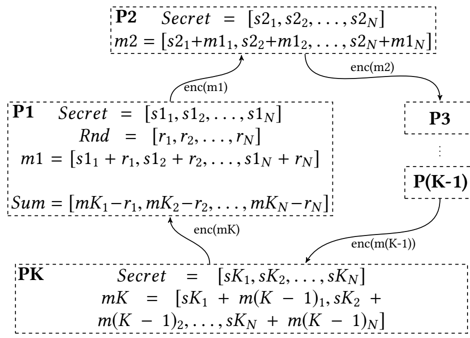


Figure 8: SGX-based secure multi-party computation

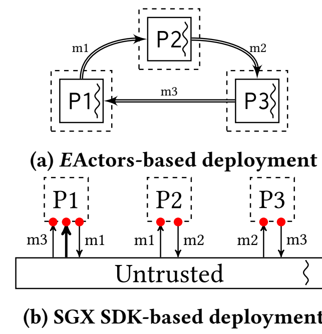


Figure 9: Deployments of the secure sum protocol

At the level of computational resources, the CONNECTOR and its attached untrusted $_e actor$s are executed by two threads: one for the trusted CONNECTOR and the other for the networking $_e actor$s. This way the CONNECTOR assigned thread never needs to leave the enclave. A similar pattern has been proven useful for the XMMP $_e actor$ and its attached networking $_e actor$s. It has to be noted that the number of threads is scaled with the number of XMMP and networking $_e actor$s.

5.2 Secure multi-party computation service

There exist in the literature multiple protocols enforcing secure multi-party computations. In this use case, we selected a secure sum protocol [17, 34]. This protocol aims at securely computing the sum of all the inputs of a set of participants without revealing the individual values. Usually the protocol targets a distributed setting where the individual participants exchange message over the network. With the support of trusted execution all participants can be represented by enclaves that are co-located on a single machine. This way costly network-based communication between the participants can be avoided. Furthermore, our use case generalizes the proposed protocol by performing the sum of private vectors instead of individual values.

Figure 8 shows an overview of our secure-sum service. In this figure, k parties (i.e., P1, P2,..PK), are connected to each other in a ring structure. Each party has its own enclave and stores a secret input vector (Secret). On demand, the Secure multi-party computation (SMC) scheme computes the sum of the secret vectors. To do that, the first party P1 starts by generating a vector of random values Rnd of the same size as the secret vector. Then, P1 generates a message vector m1, which is the sum of Rnd and the secret vector Secret. After encryption, this message is delivered to the second party P2. The second party decrypts the message, sums it with its own secret vector, encrypts the resulting message and sends it to the next node in the ring. This process is repeated until the last party PK delivers the mK message to the first party P1. Finally, P1 computes the result of the sum by subtracting the Rnd vector from the latest received vector mK. This result is then shared among all the participants.

To highlight the benefits of *E*Actors we designed two different variants of this protocol as shown in Fig. 9. The top part of the figure shows the SMC use case implemented with *E*Actors while the bottom part shows the same use case implemented with the SGX SDK. In the first case, each party is implemented as an independent actor with its own worker and an SGX enclave image. Communication between the parties is implemented as encrypted communication channels. As such, communications between different parties are protected, and neither malicious parties nor an adversary listening to the network can obtain the temporary sum or guess any secret. In the second case, each party is also implemented as an SGX enclave but only a single thread executes the protocol by entering and leaving one enclave after another. As will be shown in the evaluation (see Section 6.3) the *E*Actors design features more parallelism and less time-consuming switches between trusted and untrusted execution mode.

6 EVALUATION OF *E*ACTORS

In the following, we briefly describe the experimental setup, before evaluating *E*Actors inter-enclave performance, and the impact of various configurations and deployments on our two use cases.

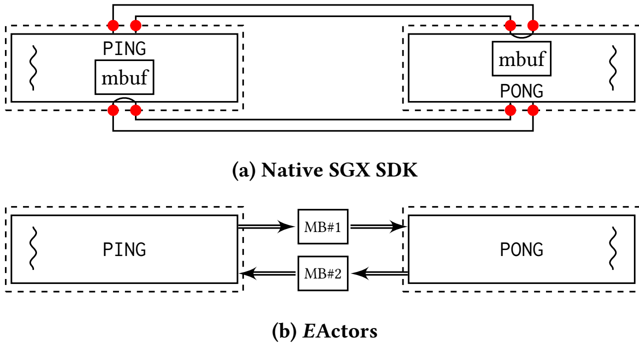
6.1 Evaluation system

The following experiments are conducted on Intel Xeon CPU E3-1230v5 (3.40 GHz, 4 cores, 8 hyper-threads), equipped with 32 GiB RAM, and a Mellanox MT27520 RoCE RDMA controller (10 GbE, RDMA capabilities were not used). The used software consists of Intel SGX SDK 1.8 and the SDK builtin Intel IPP library, with all modules built using the "-O2" optimisation, and running on Ubuntu 16.04.3 with a Linux kernel version 4.4.0-109.

The framework, implemented in C, contains roughly 6200 lines of code. The part of the framework embedded in an enclave contains 3278 lines of code and some of the third-party libraries shipped with the SGX SDK. As a result, for an application such as the XMPP server, an enclave uses roughly 500 KiB of memory.

6.2 Inter-enclave communication

We consider a simple *pingpong* application that consists of a PING component and a PONG component. The PING sends a message to the PONG, and the PONG replies to the PING with a message. Two variants of *pingpong* were implemented and compared: an *E*Actors-based implementation and a native SGX SDK-based approach, as described in Fig. 10. In the native SGX SDK-based scenario, the PING and the PONG components are located inside different enclaves. In the *E*Actors-based scenario, PING and PONG are designed as two



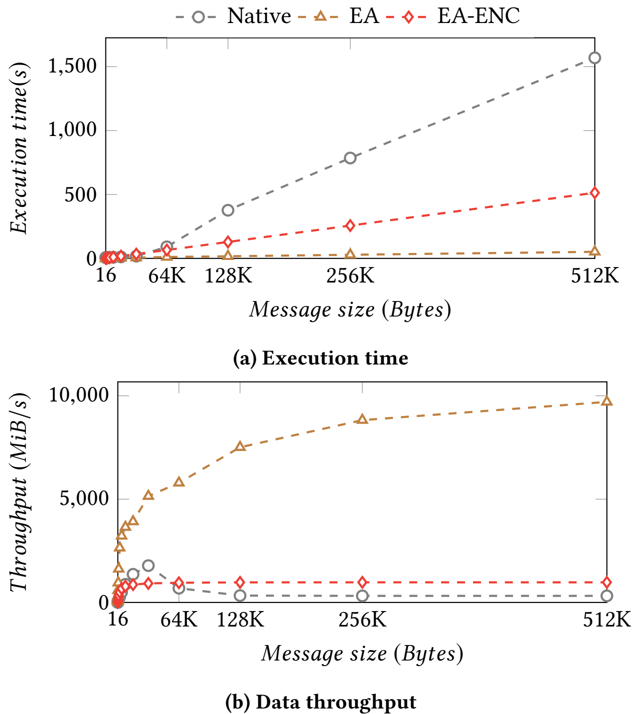
(a) Native SGX SDK

(b) *E*Actors

Figure 10: Micro-benchmark scenarios

$_e$*actors* hosted in two different enclaves. Here, $_e$*actors*'s threads are bound to different CPU cores, and non-encrypted `mboxes`.

A total of 1,000,000 `PING-PONG` pairs of operations was executed. `PING-PONG` operations include not only the time necessary to send messages, but also the time needed to fill the messages with payload data. The payload data are pseudo-random generated strings. Each reported result is an average of runs. Fig. 11 presents the performance results of these experiments, with, respectively, the execution time in Fig. 11(a), and the data throughput in Fig. 11(b).



(a) Execution time

(b) Data throughput

Figure 11: Inter-enclave performance

The results clearly show that *E*Actors (EA) outperforms the native SGX SDK (Native) in terms of execution time and data throughput. Fig. 11 shows that the native SGX SDK reaches its peak throughput near 32 KiB. This is explained by the fact that for each OCall, the SDK allocates a memory space in which the sent message is copied. But after reaching the L1 data cache size, which is 32 KiB in Intel Skylake Core [1], memory copy becomes slow. Furthermore, for a finer analysis of the difference between *E*Actors and the native SDK, *E*Actors is also evaluated with an encrypted communication channel. Obviously, encryption induces a non-negligible overhead, with a data throughput that is up to 10 times lower than without encryption (EA). Even with encryption, and thanks to its optimised inter-enclaves communication primitives, *E*Actors still provides a data throughput 3 times higher than the native SDK.

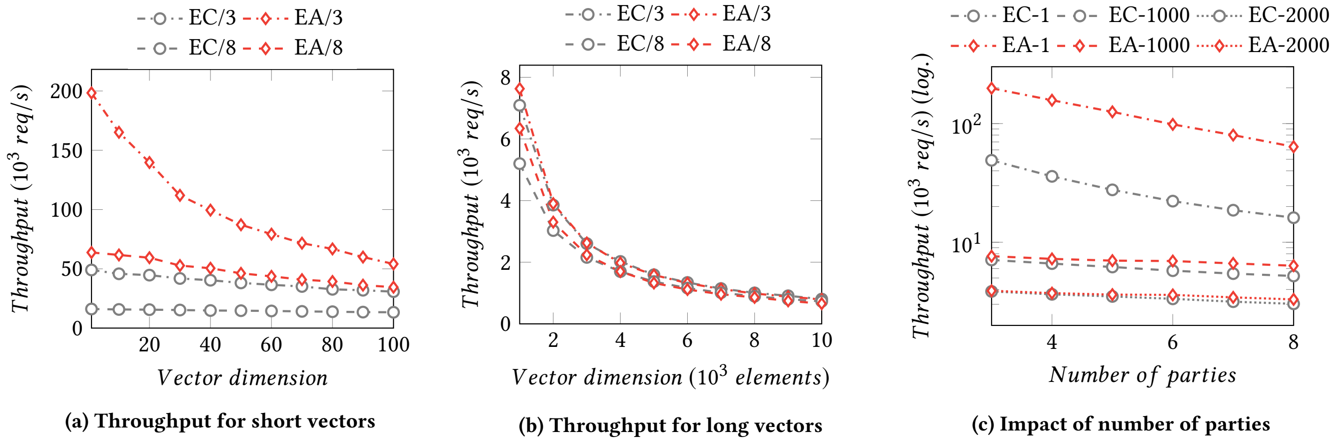
6.3 Secure multi-party computation service

To evaluate our second use case, we implemented the two deployment variants of the secure multi-party use case as depicted in Fig. 9: an *E*Actors-based implementation and a SGX SDK-based version. For a predefined number of parties and vector dimensions, we generated 10.000 invocations of the secure sum scheme, and measured the response time. We repeated each experiment at least three times and computed the average. The measured throughput depends on multiple factors, such as the speed of encryption/decryption of messages, number of parties, vector size and the duration for generating random numbers. The last factor is crucial because the first party needs to refill the `Rnd` vector on each request.

6.3.1 Case#1: plain protocol. In our first experiments we concentrated on the plain execution of the protocol. Fig. 12a and Fig. 12b show the performance of the SMC service for short (below 100 elements) and long (between 1000 and 10.000 elements) input vectors. We used two extreme configurations: three and eight participating parties. Fig. 12c shows the performance impact depending on the number of parties. For this benchmark, we used three different vector sizes: 1, 1000, and 2000 elements.

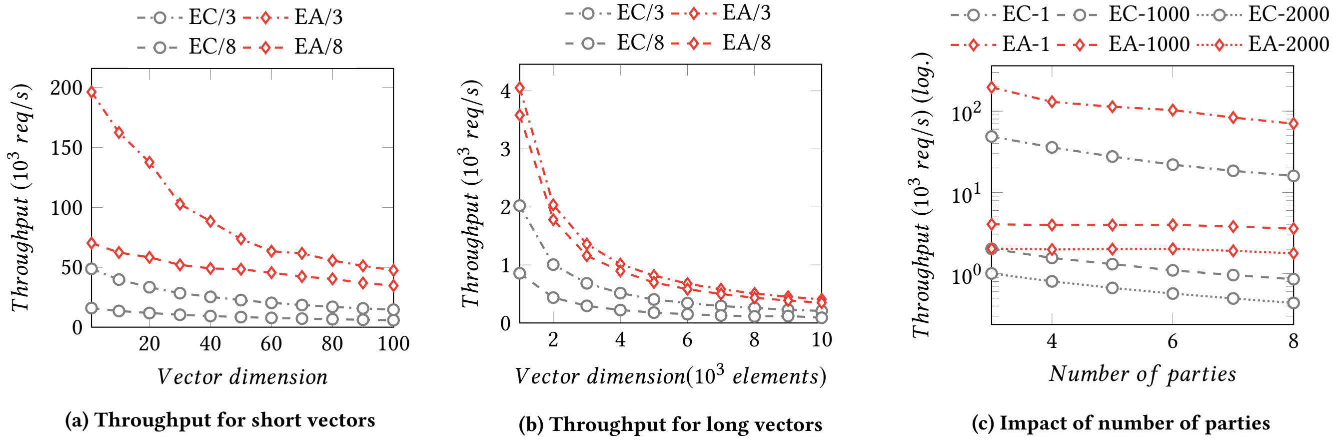
Firstly, the throughput of the *E*Actors-based implementations is higher than the throughput of the SDK-based implementation, especially for short vectors. Increasing the vector size leads to the degradation of the throughput. The same applies for increasing the number of parties. Secondly, as it is shown in Fig. 12b, for long vectors the difference between the two implementations is not so severe as for short vectors. For example, the difference in throughput for three parties and 1000 elements (EC/3, EA/3) is 8%, and it becomes negligible for vectors longer than 2000 elements.

We identified three sources of performance degradation. The first one is transition costs entering and leaving trusted execution mode during ECall/OCall use. The implementation of the SDK-based SMC scheme uses ECalls efficiently, i.e. transition costs do not involve copying of memory, and thus, the transition costs do not depend on the vector size. However, the number of parties increases transition costs proportionally. The second source of performance degradation is encryption and decryption of messages. The vector size impacts linearly the encryption/decryption demand. However, as it is shown in the previous benchmark (Fig. 11b), the encrypted ping-pong application reached a throughput of 1GiB/s, which is roughly 35 times bigger than the throughput of the EA/3 configuration for 1000 elements ($7092 * 1000 * sizeof(uint32_t) \approx 27\ MiB/s$), thus bandwidth is not the limiting factor. A detailed analysis revealed the source of the performance degradation is a slow `sgx_read_rand()` SGX SDK function. In accordance with the protocol, the first party has to generate a vector of random values before each request. An



(a) Throughput for short vectors **(b) Throughput for long vectors** **(c) Impact of number of parties**

Figure 12: Plain SMC execution



(a) Throughput for short vectors **(b) Throughput for long vectors** **(c) Impact of number of parties**

Figure 13: SMC scheme with dynamically computed input vectors

increase of the vector size leads to an increased usage of the trusted random number generator.

6.3.2 Case#2: dynamically computed vectors. In the previous evaluation we assumed that the parties do not perform any computation beside the bare protocol. In contrast to the previous scheme, in this case, the parties update their internal secrets after each computation of the secure sum. Fig. 13 shows the performance of the two SMC systems with such an additional workload applied.

As one can see, this additional computation significantly impacts the performance. For example, for a vector size of only one, the performance benefit for *E*Actors has grown to 4× for three parties and to 4.4× for eight parties. For lager vectors, the difference grows faster. For example, as Fig. 13c shows, for 2000 elements, the difference in throughput grows from 2× (three parties) to 4.1× (eight parties). In addition, and in contrast to the plain execution of the protocol, experiments with larger vectors also show a significant difference in throughput. Fig. 13b demonstrates, that for eight parties and any vector size in the tested range, the *E*Actors-based implementation is at least 3.88× faster.

6.4 Evaluation of messaging service use case

The messaging service outlined in Section 5.1 has been evaluated in a personal chat and a group chat scenarios. Thereby, we compare our *E*Actors-based implementation with vanilla versions of JadderD2 (version 2.3.4) and ejabberd (version 16.01-2) servers. The former is written in C and has a multi-process architecture, while the latter is implemented in Erlang. For both services we used the default configuration and deployment settings in the evaluation.

6.4.1 Performance of one-to-one communication. In the following, we evaluate the scalability of an *E*Actors-based XMPP service with concurrent emulated clients (c.f., Section 5.1), where a thread is spawned for each client. A client is based on the *libstrophe* library to implement XMPP client behavior [3]. We first consider an XMPP service with One-to-One (O2O) client communication, where, after connecting to the service, a client sends a message to another client, receives a response message back from that client, and repeats this inter-client communication during 1 minute. Message payloads are pseudo-random generated strings of maximum 150 bytes. In the experiments, half of the clients is senders of messages, and the other

half is receivers. A sender client randomly selects a receiver client to which it will send messages. To evaluate scalability, we measure the request throughput, i.e., number of pairs of sent/received messages per second. Fig. 14 presents the evaluation results when varying the number of concurrent clients. It compares the baseline JabberD2 (JBD2) messaging service, the baseline ejabberd (EJB), and various configurations of *E*Actors-based XMPP service depending on the number of $_e$*actors* used for deploying the service. For instance, EA/3 *E*Actors-based XMPP is deployed with 3 $_e$*actors*, i.e., an enclaved XMPP $_e$*actor* with own worker, an untrusted READER $_e$*actor* and an untrusted WRITER $_e$*actor* with worker (c.f., Section 5). EA/3 uses thus as many threads as the communication process of JBD2. JBD2 and EA messaging systems rapidly reach their steady state and scale well when the number of clients increases, with a maximum throughput for *E*Actors EA/3 that is 81% higher than JBD2. EJB messaging systems reaches its steady state at approximately 600 clients. For this number of clients, the corresponding throughput for *E*Actors EA/3 is 2.42× higher.

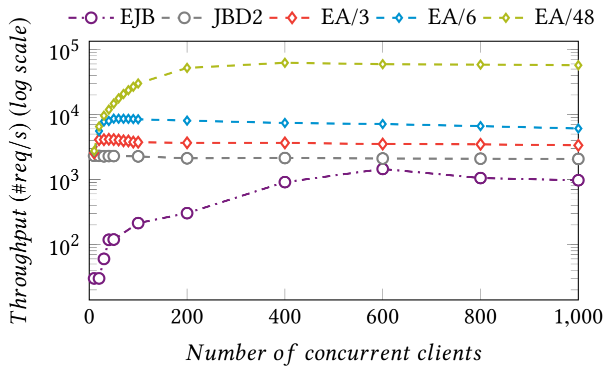


Figure 14: Scalability with different #$_e$*actors*

We also consider other deployments of the *E*Actors-based XMPP service such as EA/6 that involves a total of 6 $_e$*actors* consisting of two enclaved XMPP $_e$*actors*, two untrusted READER $_e$*actors* and two untrusted WRITER $_e$*actors*, and similarly EA/48 with a total of 48 $_e$*actors*. Fig. 14 shows that deploying *E*Actors-based XMPP with more enclaves significantly improves scalability. Here, EA/48 outperforms other systems with a factor of up to respectively 40 when compared to EJB, 29 when compared to JBD2, 17 when compared to EA/3, and 8 when compared to EA/6.

6.4.2 Performance of one-to-many communication. The following section considers the O2M case. In this experiment, we define group chats, and, for each group chat, one of the clients sends a new message to the group chat when it receives its previous message.

With *E*Actors, a group consists of an XMPP $_e$*actor* as well as its READER and WRITER $_e$*actors*. A worker is associated to each of these three $_e$*actors*. As a first experiment, we observed that the throughput does not change when we increase the number of groups. This result is expected since each group works almost in isolation. For this reason, we report in Fig. 15 the throughput of the O2M experiment when we increase the number of clients of a single group. We evaluate two configurations: in the EA/trusted

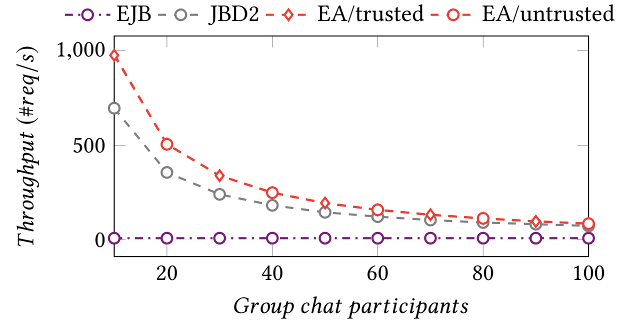


Figure 15: Group communication, trusted vs. untrusted

experiment, the XMPP $_e$*actor* runs in an enclave, while in the EA/untrusted experiment, the XMPP $_e$*actor* runs in untrusted memory. Since we run our XMPP server with a single thread, we can also compare the performance of our XMPP server with the single-threaded JabberD2 (i.e., baseline JBD2) when SSL is enabled and when it uses the MU-Conference modules to handle chat groups.

We can observe that EA/untrusted and EA/trusted have exactly the same performance, and that they slightly outperform the baseline JabberD2 server. This result shows that, regardless of SGX, the *E*Actors actor model is competitive as compared to the multi process model used by JabberD2. Moreover, while EA/untrusted and JabberD2 provide the same security guarantee (if the server is compromised, the server can read the messages), EA/trusted allows to protect the messages in a compromised server. Moreover, when we use a configuration with more enclaves and more groups, our XMPP server also ensures that if an attacker is able to compromise one of the enclaves, the other enclaves remain safe under the assumption that there is no a common flaw that can be exploited. Thus, *E*Actors can achieve efficiency while ensuring privacy.

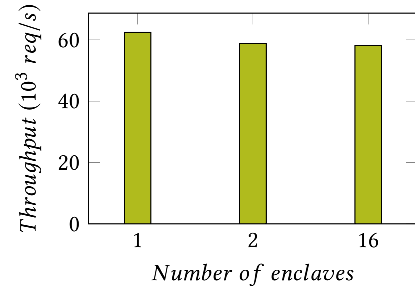


Figure 16: Impact of number of enclaves

6.4.3 Impact of number of enclaves. In the following, we evaluate the impact of the number of enclaves on the performance of the underlying application. We consider an *E*Actors-based XMPP service that consists of a total of 48 $_e$*actors*, i.e., 16 XMPP $_e$*actors*, 16 READER $_e$*actors* and 16 WRITER $_e$*actors*. In these experiments, 400 clients concurrently access the service in a One-to-One (O2O) communication mode as described earlier. Fig. 16 presents the client request throughput in the following deployment scenarios: all 48 $_e$*actors* are located inside the same enclave, 2 enclaves host

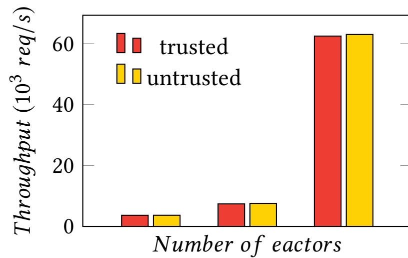


Figure 17: Trusted mode vs. untrusted mode

each 24 $_e$*actor*s, or 16 enclaves are used each one hosting the three XMPP, `READER` and `WRITER` $_e$*actor*s. The two latter scenarios have similar performance, whereas the former scenario's throughput is 6.2% better. Indeed, in case of a single enclave the data shared between $_e$*actor*s is directly stored and accessed inside the enclave memory without encryption.

6.4.4 Overhead evaluation. In order to evaluate the cost of using trusted environments, Fig. 17 compares the performance of *E*Actors-based XMPP when the underlying $_e$*actor*s are hosted in enclaves vs. when they run in untrusted mode. As in previous experiments, we consider an XMPP service deployed in a total of 3, 6 or 48 $_e$*actor*s (respectively `EA/3`, `EA/6` or `EA/48`), running in trusted or non-trusted mode. Here, 400 concurrent clients access the service. Fig. 17 shows very similar performance results for enclaved vs. non-enclaved $_e$*actor*s, with no perceptible overhead.

7 RELATED WORKS

Frameworks and programming languages that feature the actor model have a long history [4, 5, 29, 30]. To our knowledge there is no related work that proposed a framework that is tailored towards the use of trusted computing. For example, the CAF framework [13], which evolved from the libcppa [14] project, is an actor-based programming framework written in C++. It offers a high-performance, lightweight messaging system implemented using atomic Compare-and-swap (CAS) operations and supports heterogenous actors, which can interact with devices such as GPUs. The lowest communication layer of the *E*Actors framework shares some similarities with the queue implementation of CAF. However, the CAF framework misses any support for trusted execution, as well as central ideas of *E*Actors like effective enclave-to-enclave communications, flexible reconfiguration and more. Java extensions, like Kilim [50] and Akka [23], as well as languages like Erlang [5] offer an actor-based execution model based on lightweight threads and fast communication using queues. None of these frameworks supports trusted execution and porting such heavyweight runtimes like a JVM or the Erlang VM to SGX is a challenge in itself.

As previously presented, a couple of works already identified shortcomings of the SGX SDK and aimed to address specific aspects: for example HotCalls [52] offers hand-crafted spin-locks for SGX to reduce the synchronisation overhead, and SCONE [6] as well as Eleos [39] aim at avoiding costly enclave exits. However, none of these works addresses multi-enclave settings. The same applies to Glamdring [33], which enables automated partitioning of legacy applications but also does not offer support for multiple enclaves.

More related to the presented framework is Panoply [49], which offers lightweight enclave-based *microns*. However, their inter-enclave communication support requires a custom mapping using a reference monitor and is as costly as using the SGX SDK. Furthermore, Panoply misses configurability as offered by *E*Actors.

There is a growing number of middleware-alike systems that utilise SGX: VC3 enables map reduce [47], SecureStream provides tailored support for stream processing [26], and SecureVertex enables secure Cloud micro services [10]. None of these works offers fine-grained and fast support for multi-enclave programming. Instead, these systems use standard communication mechanisms such as the socket interface (i.e. VC3 and SecureVertex).

In line with our secure multi-party computation use case there is a limited number of works that specifically focused on the combination of SMC and trusted computing. Iron [19] provides practical functional encryption scheme, which involves several enclaves. To achieved this intensive message exchange is required, initiated and managed by different components of the Iron platform. Communication between enclaves is performed using standard ECalls. Accordingly Iron could profit from *E*Actors's fast inter-enclave communication support.

Another SGX-based protocol for multi-party computation was introduced by Bahman et al. [7]. This protocol has two phases: a preparation phase and an online phase. During the preparation phase, parties, represented by SGX enclaves, establish encrypted communication channels. During the online phase they evaluate built-in functions. Overall, their proposed SGX multi-party computation protocol and our SMC use case implemented in Section 5.2 share some similarities, however the work of Bahman et al. misses support for fast inter-enclave communication and system support as provided by *E*Actors.

At a higher level, *E*Actors is inspired by the concept of separation of mechanisms and policies [24, 53] and supports the ideas of *lateral trustworthy apps* [25].

8 CONCLUSION

The *E*Actors[1] frameworks enables multi-enclave programming and interaction at low costs. This is achieved by an SGX-tailored implementation of the actor model that prevents costly execution mode transitions, offers uniform communication primitives, and can be flexibly configured for different deployments. *E*Actors offers a lean programming interface that can be further extended but also leads to a framework with a small trusted computing base of less than 3.3K lines of code plus some additional libraries provided by SGX SDK. Together this paves the way for novel privacy preserving multi-party computing schemes and strengthens the security of privacy critical services such as the presented use cases. Our evaluations show that previously considered costly multi-enclave programming comes almost for free and that off-the-shelf instant messaging services can be outperformed between 1.11× up to 40×.

ACKNOWLEDGEMENT

This project received funding from the German Research Foundation (DFG) under grant no. KA 3171/9-1 and from the National Agency for Research (ANR) under grant no. ANR-17-CE25-0017.

[1] Available at https://github.com/ibr-ds/EActors

REFERENCES

[1] 2017. *6th Gen Intel Core X-Series Processor Family Datasheet, Vol. 1.*

[2] 2017. Signal taps up Intel's SGX to (hopefully) stop contacts falling into hackers, cops' hands. https://www.theregister.co.uk/2017/09/27/signal_turns_to_intels_sgx_to_lock_down_contacts_from_spying_eyes/

[3] 2018. "libstrophe - An XMPP library for C". http://strophe.im/libstrophe/

[4] Gul Agha, Ian A Mason, Scott Smith, and Carolyn Talcott. 1992. Towards a theory of actor computation. In *International Conference on Concurrency Theory*. Springer, 565–579.

[5] Joe Armstrong. 1996. Erlang – a Survey of the Language and its Industrial Applications. In *Proc. INAP*, Vol. 96.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[7] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. 2017. Secure multiparty computation from SGX. In *International Conference on Financial Cryptography and Data Security*. Springer, 477–497.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[9] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN'17)*. 157–168.

[10] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. 2017. Secure Cloud Micro Services using Intel SGX. In *Proceedings of the 17th International IFIP Conference on Distributed Applications and Interoperable Systems*. Springer.

[11] Stefan Brenner, Colin Wulf, Matthias Lorenz, Nico Weichbrodt, David Goltzsche, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*. ACM.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*.

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C Schmidt. 2014. CAF – the C++ Actor Framework for Scalable and Resource-efficient Applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. 15–28.

[14] Dominik Charousset, Thomas C Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*. 87–96.

[15] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.

[16] Nikos Chondros, Bingsheng Zhang, Thomas Zacharias, Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Alex Delis, Aggelos Kiayias, and Mema Roussopoulos. 2013. A distributed, end-to-end verifiable, internet voting system. In *International Conference on Distributed Computing Systems (ICDCS'13)*.

[17] Chris Clifton, Murat Kantarcioglu, Jaideep Vaidya, Xiaodong Lin, and Michael Y. Zhu. 2002. Tools for Privacy Preserving Distributed Data Mining. *SIGKDD Explor. Newsl.* 4, 2 (Dec. 2002), 28–34.

[18] Wenliang Du and Mikhail J Atallah. 2001. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*. 13–22.

[19] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the ACM Conference on Computer and Communications Security (SIGSAC)*. 765–782.

[20] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford University.

[21] Rachid Guerraoui, Anne-Marie Kermarrec, Rhicheek Patra, Mahammad Valiyev, and Jingjing Wang. 2017. I know nothing about you but here is what you might like. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 439–450.

[22] Saikat Guha, Mudit Jain, and Venkata N Padmanabhan. 2012. Koi: A location-privacy platform for smartphone apps. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 14–14.

[23] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* 410, 2-3 (2009), 202–220.

[24] Per Brinch Hansen. 2001. The evolution of operating systems. In *Classic operating systems*. Springer, 1–34.

[25] Hermann Härtig, Michael Roitzsch, Carsten Weinhold, and Adam Lackorzynski. 2017. Lateral Thinking for Trustworthy Apps. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 1890–1899.

[26] Aurélien Havet, Rafael Pires, Pascal Felber, Marcelo Pasin, Romain Rouvoy, and Valerio Schiavoni. 2017. SecureStreams: A Reactive Middleware Framework for Secure Data Stream Processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 124–133.

[27] Intel. 2017. Intel® Software Guard Extensions SDK for Linux* OS , Revision 2.0. https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-developer-reference.

[28] Intel. 2018. Intel® Software Guard Extensions SDK for Linux* OS , Revision 2.2. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf.

[29] Dennis G. Kafura and Keung H Lee. 1989. *ACT++: Building a Concurrent C++ with Actors.* Technical Report. Blacksburg, VA, USA.

[30] Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, Vol. 28. 91–108.

[31] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*. 523–539.

[32] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. 557–574.

[33] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (USENIX ATC 17)*. 285–298.

[34] Yehuda Lindell and Benny Pinkas. 2008. Secure Multiparty Computation for Privacy-Preserving Data Mining. *IACR Cryptology ePrint Archive* (2008), 197.

[35] Robert Love, So Here We Are, Along Came Linus, and Before We Begin. 2005. *Linux kernel development second edition.* Novell Press.

[36] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. [n. d.]. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1289–1306.

[37] 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. *HASP@ ISCA* 10 (2013).

[38] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. 2017. X-search: revisiting private web search using intel SGX. In *Proceedings of the 18th International Middleware Conference*. 198–208.

[39] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 238–253.

[40] Albin Petit, Thomas Cerqueus, Sonia Ben Mokhtar, Lionel Brunie, and Harald Kosch. 2015. PEAS: Private, Efficient and Accurate Web Search. In *Trustcom*, Vol. 1. 571–580.

[41] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. 2016. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. 10:1–10:10.

[42] Ravi Rajwar and James R Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. 294–305.

[43] Alessandro Rubini and Jonathan Corbet. 2001. *Linux device drivers.* " O'Reilly Media, Inc.".

[44] Paul Rösler, Christian Mainka, and Jörg Schwenk. 2017. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. Cryptology ePrint Archive, Report 2017/713.

[45] Peter Saint-Andre. 2011. Extensible messaging and presence protocol (XMPP): Core. https://xmpp.org/rfcs/rfc6120.html

[46] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rüdiger Kapitza. 2018. STANlite–a database engine for secure data processing at rack-scale level. In *IEEE International Conference on Cloud Engineering (IC2E)*. 23–33.

[47] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*. 38–54.

[48] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[49] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[50] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *ECOOP*, Vol. 8. Springer, 104–128.

[51] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS'16)*.

[52] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. 81–93.

[53] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM (CACM)* 17, 6 (June 1974), 337–345.

[54] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), IEEE Symposium on*. 640–656.