Erlang SGX

Protecting Confidential Erlang Workloads with Intel SGX

Master's thesis in Computer Science and Engineering

Emil Hemdal & Eliot Roxbergh

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Erlang SGX

Protecting Confidential Erlang Workloads with Intel SGX

Emil Hemdal & Eliot Roxbergh

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Erlang SGX
Protecting Confidential Erlang Workloads with Intel SGX
Emil Hemdal & Eliot Roxbergh

Advisor: Mikael Eriksson, Ericsson AB
Supervisor: David Sands, Computer Science and Engineering
Examiner: Alejandro Russo, Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Erlang SGX
Protecting Confidential Erlang Workloads with Intel SGX
Emil Hemdal & Eliot Roxbergh
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Secure enclaves, such as Intel SGX, provide a trusted execution environment which offers integrity and confidentiality guarantees to supported applications. In this thesis, we show how Erlang can be protected to harden telecommunication workloads by utilizing hardware-based Intel SGX. First, we demonstrate how an untrusted Erlang runtime can execute trusted C code inside of SGX via Erlang C Nodes and NIFs. A possible use case is the protection of cryptographic functionality which is demonstrated with OpenSSL inside of SGX, callable from Erlang. Second, to protect the Erlang runtime itself, a number of alternatives are explored as to enable execution of Erlang code inside of the enclave. However, Erlang ships with BEAM, an advanced virtual machine, which performs frequent syscalls and IO activity that drastically increases the complexity of porting it to SGX enclaves. Therefore, two prominent third-party frameworks are tested which aim to support generic applications inside of SGX: the Library OS Graphene and the shim-layer solution SCONE. Third, alternatives to implement a custom solution are discussed which could yield performance and security benefits over the generic frameworks while protecting an Erlang runtime. The complete source code for this thesis is available under a permissive BSD 3 license.[1]

Keywords: Erlang, BEAM, Intel-SGX, Graphene, SCONE, secure-enclave, trusted-computing

[1]`https://github.com/Erlang-Enclave-Thesis/sgx-erlang-extension`

Acknowledgements

First, we would like to thank our supervisors, David Sands from Chalmers and Mikael Eriksson from Ericsson. Thank you Mikael for suggesting the topic and for helping us get in contact with industry experts at Ericsson.

Second, we will not forget our desk buddy Torbjörn Andréasson for his supportive talks and for sharing his knowledge of the inner workings of the thesis process.

Finally, we really appreciate the help received from the experts; We would like thank the employees at Decentriq for reaching out to us, and in particular to Stefan Deml for sharing his knowledge on porting applications to SGX. Christof Fetzer at Scontain for his help in making Erlang run in SCONE. Dmitrii Kuvaiskii at Graphene for his help with debugging Erlang in Graphene and providing guidelines going forward. As well as the Erlang and security experts at Ericsson, you know who you are. Thank you, all.

Emil Hemdal & Eliot Roxbergh, Gothenburg, June 2020

Contents

1
Introduction

Erlang is a functional programming language that is widely used in telecommunication and other back-end systems [22, 5][13, p. 6]. As these communication systems become increasingly cloud-based, new opportunities for attackers emerge: both for conventional attacks as well as for the theft of intellectual property. To counter this, so-called secure enclaves could be introduced as proposed by the major CPU vendors. The secure enclave feature is often supported by modern CPUs and it provides functionality to protect sensitive processes or parts of running programs [3, 35]. One such secure enclave technology is the *Intel Software Guard Extensions* (Intel SGX) which will be utilized in this thesis. The aim of secure enclaves is to protect highly sensitive data and workloads, including both their integrity as well as their confidentiality, and even if the adversary is executing in kernel mode [17]. These security guarantees, are in Intel SGX achieved by two main methods; First, the secure enclave encrypts and integrity-protects the memory of the running process. Second, the untrusted system can only communicate with the enclave via specific CPU instructions, thereby greatly restricting its access to the running process.

Due to the limited interface exposed by Intel SGX enclaves, there is no native method to perform syscalls inside of the enclave. As a result, the possible language support in SGX is greatly limited, where the official support only includes C and C++ with the Intel SGX SDK (simply referred to as Intel SDK in this thesis). There are third-party frameworks that support the use of syscalls and thereby could enable deployment of legacy applications (such as a language runtime) inside of an SGX enclave. This would be the quickest method to allow for the execution of arbitrary Erlang code inside of SGX, however, these frameworks come with performance and security penalties. The alternative is to create a custom solution, either by changing the Erlang runtime itself or by adding a layer in SGX to perfectly fit Erlang by only supporting the external communication and syscalls required. However, the optimized solution would require a substantial amount of work since Erlang relies on a complex runtime with many syscalls and a high IO usage. On the other hand, for these very reasons, a third-party framework could come to struggle with performance, increasing the possible benefits with a custom solution.

The recommendation from Intel is to not perform any syscalls from within SGX which would need to be forwarded to the untrusted OS. To follow this advice would make it extremely difficult to port the Erlang BEAM runtime[1], instead an alternative would be to call secured C code inside of SGX from the untrusted Erlang runtime. By not porting the Erlang runtime itself, and to only make calls to separate C functions without external dependencies, would provide very strict security of the enclaved process and it would not reduce the performance of Erlang code. However, as the Erlang runtime in this case would still be running in a completely untrusted OS there would not be any security guarantees made on this side, which could open up for attacks.

The thesis is structured as follows; Chapter 1 introduces the reader to the problem, motivates its importance, and proposes a possible solution. In chapter 2, technical details will be explained and earlier work in the field will be surveyed. Chapter 3 will explain the method and overview the possible designs which could achieve the earlier laid out goals. The implementation specific details are presented in chapter 4. The implemented designs will be evaluated in chapter 5, and the findings further discussed in chapter 6. Finally, the thesis will be summarized in chapter 7 with final thoughts and recommendations going forward.

[1]However, there could exist ways of otherwise running Erlang in SGX without syscalls: such as to port a simpler Erlang runtime or if it would be possible to compile Erlang code to another intermediate representation (i.e. bytecode) which is better fit for this purpose.

1.1 Relevance

For Ericsson, one of the company's dominating telecommunication applications is using the Erlang language. As these applications are expected to run in different cloud environments, partly due to the 5G standard [4, 10], there is a need for new security features. A significant weakness in this new infrastructure is that it is no longer possible to trust other running processes, or even the hypervisor or kernel itself [33]. A proposed mitigation is with the introduction of secure enclaves, such as Intel SGX, which provide developers with the tools for running software securely on operating systems that themselves might be hostile. This is an issue which is now becoming reality, sensitive code may come to be executed in datacentres owned by a competitor or on computational resources shared with the public (i.e. public cloud). Intel SGX aims to protect sensitive data that the software is working on as well as the running process itself [14]. An example of such data is a private key which could be captured by a privileged process if the system does not utilize a secure enclave, something which would covertly undermine the attacked process' security completely. Additionally, the executing binary itself might be confidential which the underlying cloud-vendor should neither be able to view nor modify.

1.1.1 Usefulness of Erlang SGX

Intel SGX can either be used to enable secure function evaluation (i.e. evaluate a publicly known function without disclosing the inputs) [28] or to execute confidential functions where both the code itself and the data it works on should be kept secret. Consequently, there are many possible use cases which have been proposed to be especially fit for Intel SGX [51]. Two prominent examples are Machine Learning (ML) where the inputs are secret and cloud workloads as hinted at earlier which may involve secret code and data. Traditionally, Erlang has not been widely used in ML and in this thesis the focus will be on cloud use cases which is where Erlang currently has a market share.

Erlang is used in back-end services such as the database CouchDB [6, p. 12], the messaging application WhatsApp [46, 59], as well as in the telecommunications system at Ericsson [22]. In all these areas there is a need to provide additional security features to harden the Erlang workloads, as to maintain security and confidentiality while deployed in commodity cloud solutions. Thereby, it is possible to lower cost and in some cases increase vendor neutrality.[2] Therefore, it would be beneficial if Erlang could be supported to run inside of secure enclaves, as they offer security guarantees for these cloud use cases.

1.2 Goal

The goal of this thesis is to investigate how the Erlang runtime system can be modified in order to support the execution of Erlang processes, or some sensitive parts of these processes, in Intel SGX enclaves. Additionally, this extends to the investigation of whether C code can be executed securely inside the enclave, initiated from and communicating with an untrusted Erlang runtime (specifically the BEAM virtual machine). This work consists of both theoretical and practical parts, including the creation of several prototypes to showcase their respective feasibility.

To give an idea of how to utilize Erlang with SGX and as to fine-tune the goals of this thesis, we introduce a short case study of lawful interception in 5G. Based on this, two concrete scenarios are proposed which function as examples of how Erlang and SGX can be used together to aid this system. Consequently, it is possible to evaluate any prototypes created: both to test their performance, and to discuss their usefulness in relation to these requirements.

[2]For instance, in telecommunications there is an additional push for cloud claiming increased vendor neutrality as compared to hardware-based products. As a result of the migration to the cloud, Intel responded with the initiative *Open Mobile Evolved Core* proposing a 4G telecommunication infrastructure secured with Intel SGX: `https://www.opennetworking.org/omec/` [51].

1.2.1 Case Study: Lawful Interception

Lawful Interception for telecommunications is specified in the 3GPP standard [2] and is required by law in many jurisdictions, such as the European Union. The idea being that law enforcement should be able to locate and listen in on certain individuals for the sake of national security or to investigate serious crimes [27]. Naturally, this functionality is considered highly sensitive and could be a target for foreign nations and companies - especially as it can track and monitor anyone in real time [21]. However, since telecommunications companies are deploying more functionality to the cloud these features become increasingly hard to secure. Questions arise, such as what if the public cloud vendor themselves would attack this system, or if other hosts inside the data center could pose a security risk [51]. To protect the application from the host itself is exactly what secure enclave technologies such as Intel SGX aims to achieve. Earlier research has shown different methods of utilizing and optimizing for secure enclaves, but what is lacking is support for executing Erlang workloads in these systems.

1.2.2 Use Case I: Call C Functions in SGX

Many Erlang workloads make calls to C code in the background, often via natively implemented functions (NIFs) which is used by many libraries. The inclusion of C code can be due to performance reasons or for instance if a certain cryptographic standard is not available natively in Erlang. One such example is TLS, which is often used within the 5G infrastructure to secure inter-service communication. Therefore, in the first use case, the goal is to create a minimum viable product that could execute the cryptographic functionality (written in C) required by TLS inside of SGX. By this method, it could potentially be possible to replace these sensitive libraries on the back end and thereby protect the unmodified Erlang workloads which depends on them.

Aims

1. C code in SGX enclaves should be callable from Erlang
2. Include TLS inside of the enclave to protect cryptographic keys and functionality
3. Support remote attestation to verify correct execution and maintain confidentiality of the binary

1.2.3 Use Case II: Erlang in SGX

Erlang is an important component of Ericsson's telecommunication platform which requires additional protection when executed in the cloud. Therefore, in this use case, we aim to add support for BEAM in SGX with the functionality necessary to run legacy Erlang workloads.

Aims

1. Add support to execute Erlang functions in SGX
2. Support remote attestation to verify correct execution in SGX
3. Provision the enclave as to provide integrity and confidentiality of Erlang binaries

1.3 Limitations

To accomplish the task to move Erlang processes into an enclave there exist several constraints imposed by the enclave technology as well as from the Erlang environment itself. One such constraint, is the limited interface that is used to interact with the enclave which results in the lack of native support for performing syscalls inside of SGX. Additionally, the Erlang programming language should still follow the specifications and maintain backwards compatibility. In this thesis the focus will primarily be on the official BEAM runtime. The ideal design would allow an unmodified BEAM instance to run, enabling legacy Erlang code to be ported easily. However, running Erlang code in SGX is seemingly not possible without a method of resolving syscalls inside the enclave as these are requested by BEAM. Thus there are two alternatives, when BEAM is used, either to include a custom layer to handle specific syscalls or to use a third-party framework which allows for the use of syscalls in generic applications.

For the purpose of this thesis, both changes to the runtime itself and the creation of a custom layer to resolve syscalls will not be done due to time constrains. Instead, we investigate if the major third-party frameworks are 'good enough' to support Erlang for the use cases provided and we then discuss a number of ways on how to best go forward from here. Moreover, we limit the scope to Intel SGX as it currently is the most mature enclave platform, although other promising alternatives exist. The work will be done on a Linux machine which will be used to build and modify Erlang. Furthermore, the focus is on the area of telecommunications, although we believe other industries would share the same security concerns and therefore would benefit from a more secure Erlang runtime.

1.3.1 Ethical Considerations

The aim of this project is to secure sensitive telecommunication applications. In this context we see no direct ethical issues. However, secure enclave technology is often used for Digital Rights Management (DRM) which in some industries is surrounded by controversy, it is a double-edged sword and might be used to restrict the users' freedom [19]. Regardless, for this use case we argue that DRM is beneficial as it protects sensitive intellectual property from other companies and nation states, which could protect this vital communication infrastructure. Additionally, the legal spying (lawful interception) on citizens could be debated, but this is not a technical problem. As long as this is required by law it is beneficial to secure this infrastructure as to ensure that only authorized personnel may use it.

2

Background

In this chapter we will first introduce the background of Erlang and then Intel SGX to the reader. Last, we present three methods of utilizing SGX and within these categories a comparison is made between several frameworks according to earlier work in the field. The comparison is done to introduce these frameworks to the reader and to further show the possibilities as well as what trade-offs can be made for general SGX solutions.

2.1 Erlang

Erlang is a functional programming language conceived at Ericsson in the eighties for use in the telecommunications industry [5]. The main focus of Erlang is concurrency which is achieved through lightweight processes and message passing. But Erlang on its own is just a programming language, it is first when paired with an Erlang Runtime System (ERTS), such as the official BEAM compiler and virtual machine (VM), that compiling and running Erlang code becomes possible [60]. Largely thanks to using this Erlang system, an uptime of nine-nines (99.9999999%) could be achieved in Ericsson's famous *AXD301 ATM* (*Asynchronous Transfer Mode*) switch [12, p. 10]. Therefore, Erlang's success can not only be attributed to the language itself, but also to the ERTS (i.e. the BEAM VM and its accompanied compiler) which is part of the so called *Open Telecom Platform* (OTP). OTP contains BEAM together with a collection of libraries, tools, and design principles developed and maintained by the OTP team at Ericsson [13]. OTP is a crucial piece in the development process of Erlang applications and is together with the Erlang language called Erlang/OTP.

To run Erlang code, the compiler first compiles the code into BEAM bytecode which can then be executed by the BEAM VM[1] [60]. This VM works by creating *schedulers*[2] and *dirty schedulers* on which the work is distributed to [13, p. 32]. Dirty schedulers are special schedulers meant for long-running tasks, such as external interfaced code, which otherwise could make the regular schedulers malfunction. [13, p. 409].

Erlang is a language highly fit for distributed computing as it allows for workloads to be executed on VMs running on external servers [12, p. 7]. These calls are performed through the same message passing that is used for inter-process communication on the local system with the exception that you need to specify the node name, hostname, and a secret cookie [13, pp. 48-49].

2.1.1 To Interface with Other Languages

The BEAM VM can easily communicate with other BEAM instances and in Erlang lingo each such instance is called a node. Additionally, there are a number of ways for Erlang (nodes) to interface with other programming languages, mainly C. For instance, performance critical functions can be implemented in C and then called from Erlang. There are a few methods of achieving this: namely by using C Nodes, NIFs, Ports, or Port Drivers.

[1]This is similar to how Java creates and executes Java virtual machine code.
[2]Usually the same amount as the amount of cores on the system.

C Node

C Node is a technology for writing C applications that interface with Erlang like it was a regular Erlang node (i.e. just another BEAM VM instance). An Erlang developer can use the regular message passing functions when communicating with the C application [23]. This interface allows for arbitrary C applications to perform work, pass messages to, and receive messages from Erlang processes. Another benefit is that the C Node can be on a separate machine (just like a regular Erlang node) which can be useful in order to distribute the application. The *ei* library included in Erlang/OTP gives C developers access to functions for connecting, sending, and receiving data to and from Erlang nodes [12, p. 342].

NIF

Erlang NIF is a method of creating external C functions that can be executed inline by regular Erlang code. Cesarini and Vinoski describe in their book *Erlang Programming*, that NIFs execute directly within the Erlang runtime "on the runtime's scheduler threads" [13, p. 408]. Therefore, it is important to avoid running programs that take a long time since this will interfere with the scheduler's operation. Although, dirty schedulers can be configured to mitigate the risks of running slower functions as a NIF [13, p. 409]. NIFs are created by compiling a library with the needed C functions together with Erlang NIF library instructions to allow Erlang to become aware of the exposed functions. Then to use the NIF, the Erlang code must contain matching function definitions as well as to load the NIF using special instructions [24]. In many of Erlang's libraries, the usage of NIFs is common for tasks that are either computationally intensive or that require external libraries (e.g. cryptography or sorting libraries). In these cases, NIFs are often preferred to e.g. C Nodes as they have less overhead. However, if the NIF code would misbehave the whole BEAM instance can crash or become unstable.

Port

Another approach for interfacing is to use Ports which enables Erlang to communicate "with the external world", as described in the Erlang manual [26]. In order to communicate with a Port the developer needs to configure the Port and prepare the data for encoding. Communication with an external program can then be performed by sending and receiving bytes on the Port. Thereby, Erlang can communicate with external programs, written in any language, as long as it encodes the data correctly [12, p. 349].

Port Driver

Port Drivers (sometimes called linked-in drivers) work like Ports but if they crash they will also crash the Erlang VM. The reason for this is that the Driver is linked into the Erlang VM which removes certain protections, however it also means that there are no context switches [25]. This is sometimes useful since it allows for fast interaction between the Driver and the ERTS, which is why it is used when performance is of the essence [12, p. 352].

2.2 Trusted Computing with Intel SGX

To protect execution of sensitive functions and storage of data (often encryption keys) is not a new idea, on the contrary, hardware-assisted trusted computing designs have been available for years. These designs can provide a number of benefits, such as memory isolation, storage and memory integrity and confidentiality, as well as attestation to prove that software was indeed executed securely inside the hardware module. Thereby, it is for instance possible to provide data-in-use protections. This category of trusted computing can be divided into two parts: *Trusted Platform Module* (TPM) which is physically separated from the CPU, and *Trusted Execution Environment* (TEE) which is embedded in the CPU itself [17]. Naturally these topics are very broad, and there exist a number of TPM and TEE architectures which vary greatly in how protections are achieved [14]. Compared to TPM, TEE has two important advantages [56, 14]. First, since the functionality is embedded inside the CPU there is no possibility for bus sniffing and cold boot attacks. Second, the TEE can act as an isolated execution environment for any sensitive code and data - whereas TPMs usually only provide a standardized set of included functions.[3] The wide difference between different TEE architectures becomes evident when comparing the three major solutions: namely Intel SGX, AMD SEV, and ARM TrustZone. While Intel SGX aims to only store some specific functions and their data in the enclave, AMD SEV's goal is to run a whole virtual machine therein [3] and ARM TrustZone has an even more granular view with only two compartments, trusted and untrusted. ARM TrustZone is also claimed to be especially fit to protect intellectual property [14].

A reason for the increased popularity of TEE solutions is due to the proliferation of cloud use cases and the public cloud, here these solutions could be used to separate user workloads and sensitive data. Formally, TEE is being introduced to cloud environments to solve this issue of *secure remote computation* with the use of *remote attestation.* Historically, Intel has had a majority of the server market share which has prompted cloud vendors such as Microsoft to adopt Intel SGX for their cloud solution Azure, where it is referred to as confidential computing [47]. It is due to their prevalence as well as their numerous features that Intel SGX is used in this thesis, and it is on SGX the focus lies going forward. Additionally, Intel SGX can provide stronger security protections than AMD SEV especially for interacting with security-sensitive data [48].

2.2.1 Technical Overview

Intel SGX enclaves is in many ways similar to a regular process. Basically, each enclave exists as an encrypted memory region in RAM where data such as program code, heap, and thread states is stored - ready to be used by threads on the CPU. Unlike regular processes, a thread may only enter the enclave with a context switch, initiated by the EENTER CPU instruction, whereby the CPU decrypts the enclave memory allowing for subsequent execution. Additionally, enclaves are restricted in many ways to improve security: some CPU instructions are forbidden, syscalls cannot be performed natively, and communication is done via a limited interface or by shared memory. Although initially restricted, with SGX2, support was added for dynamically adding memory and threads to the running enclave[4] [40, 41]. Since enclaves are executed on the same CPU as regular applications (and since encryption is performed natively in hardware) it can in some cases reach the same performance [51]. However, the creation of new enclaves, I/O, and context switches can be extremely costly and should be avoided or planned for.

To verify the integrity of functions executed in the enclave, Intel SGX relies on Remote Attestation (RA) where the function's hash can be signed and verified, both before and during execution [41]. With RA, the remote party can be confident that they are communicating with a certain function (hash) running in the secure enclave. To ensure confidentiality for these functions, SGX uses an encrypted and restricted[5] part of RAM. First when created the enclave cannot hold any secrets, it is only after a successful RA that the enclave could share a public key which then can used by a remote host to push secrets to the enclave [9]. Moreover, communication from outside the enclave comes via special CPU instructions (ring 3), reducing the attack surface. Intel also provides an interface (written in C), which by calls can be made both to the enclave (ECALL) and to the outside world (OCALL) [14]. Another security feature present in SGX is that, while an enclave is executing, if a trap call would be made the enclave will first erase the CPU registers before a context switch [17].[6]

Even though the kernel has little control over functions which run in the enclave, it still has some responsibilities. One such task is to handle the enclave memory in RAM which is done with special CPU instructions (ring 0). It is thereby possible for the kernel to allocate memory as well as to move the encrypted enclave memory to untrusted space while it is not actively running. It is by this mechanism that the system can perform paging: this includes swapping out the process if memory would run out [61].

[3]For instance as specified in ISO/IEC 11889-1:2015
[4]This is enabled by Intel EDMM
[5]This memory is only accessible with special CPU instructions and otherwise not addressable.
[6]Security features like these also explain why context switches can be relatively slow.

2.2.2 Technological Limitations

Intel SGX can provide confidentiality and integrity for a running process in the enclave, but there exist a number of limitations and possible issues with this new technology.

A potential issue with SGX, is the overhead imposed on the protected process; First, there is a set up time to prepare the enclave before first use, this can be quite slow as it is usually not only required to create the enclave but also to perform RA to verify it and then push confidential code or data.[7] Second, when using the enclave, OCALLs and ECALLs can be very expensive as they usually result in a context switch.[8] Therefore, the recommendation from Intel is to avoid context switches to improve performance. This can be done either by the use of shared memory[9] or by utilizing dedicated worker threads to perform so called Switchless Calls. Switchless Calls have dedicated threads on both the untrusted and the enclave side waiting and ready to receive OCALLs and ECALLs respectively. These calls are stored in shared buffers and asynchronously processed by these worker threads [62]. Thus, the enclave can operate without context switches, although it raises the possibility of idle threads [41]. Switchless was introduced in Intel SDK 2.2 (july 2018) [39, 40].

It is important to note that Intel do not consider side-channel attacks to be in their threat model for SGX which raises the question of possible attacks [42]. Additionally, Intel SGX is vulnerable to attacks that exploits the microarchitecture such as Spectre and Meltdown, although this can be patched and mitigated to some degree (either in software or by microcode updates). While the Spectre exploit only affected the confidentiality and not the integrity of the enclaved process [66], there exist exploits such as Foreshadow which were able to break both confidentiality (in their case extracted from L1 cache) and integrity [11]. As Intel SGX is almost completely implemented in microcode, exploits can mostly be mitigated with updates from Intel [17, p. 28].[10] Although this varies depending on the type of attack, as mitigation is limited due to the fact that the underlying microarchitecture is implemented in hardware and therefore unmodifiable. Additionally, the framework used - such the Intel SDK - can also be modified to provide some mitigation as well [14, 61, 66].

A downside with Intel SGX is the increased developer overhead. Since only a sensitive subset of a program is meant to be run inside the SGX enclave, there is a need to rethink and separate the application accordingly. It is therefore not obvious how to quickly make legacy code work while only utilizing the Intel SDK, especially as it is not possible to make syscalls from within it [9]. Compare this to AMD SEV and ARM TrustZone, they act by separating whole applications and not only specific functions, which makes it easier to port applications to run in their enclaves [14]. Intel SGX on the other hand has on paper a smaller attack surface and footprint, but a larger developer overhead. The overhead comes from the fact that code modifications are required to move sensitive functions from the application into the enclave, for instance due to the lack of native syscall support. However, there exist several unofficial frameworks that can provide the possibility of performing syscalls, enabling seamless porting of whole applications - at some cost of performance and security as described in the following section. Moreover, since there is a big difference between different TEE solutions it is not obvious how to migrate between vendors. Although cross-platform frameworks such as OpenEnclave exist, TEE solutions were created for vastly different use cases and threat models that can result in platform specific benefits being lost when using such cross-platform solutions.

Once a process has been enclaved there are still possible improvements to make, even if there are no conventional security flaws (e.g. buffer overflows) in the process. For instance, it is possible to mitigate some side-channel attacks not fully protected by Intel SGX, namely the observation of memory access patterns or branches made in the enclave. However, in practice it is cumbersome for the developer to prevent these attacks, as this is done by avoiding branches in critical

[7] Felsen et al writes in their paper *Secure and Private Function Evaluation with Intel SGX* [28], that this overhead is around two seconds in their case. They continue to add that although this is very slow, the enclave could thereafter be used multiple times and as a result it would not be a critical issue for most applications.

[8] Each context switch has been shown to take between 8k and 14k CPU cycles [65, 50].

[9] The enclave can address untrusted memory which can be used for different methods of communication.

[10] Such as with the patches:
`https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html`,
`https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html`

code sections [45] and by maintaining the same memory access pattern regardless of the action taken [28].[11] It is therefore important to note that a process which has its vital functions placed in the Intel SGX enclave cannot be assumed to be impenetrable, both conventional direct attacks (e.g. buffer overflows) and more advanced side-channel attacks should be mitigated in the enclave code. It is also due to these security risks it is recommended to minimize the size of the payload executing in the enclave - the so called Trusted Code Base (TCB).

2.2.3 Attestation and Provisioning

A vital part of Intel SGX is *attestation* which includes local and remote methods. Basically, attestation is a method where either an enclave on the same machine or an external machine can validate that the enclave is running, as well as validating certain parameters or performing key-exchange [52]. This can later be used for *provisioning*, which in simple terms means providing secrets or data to the enclave. Provisioning is essential in order to enable confidentiality, as the enclave cannot initially hold any secrets. In the following paragraphs, a more in-depth explanation is presented of how the Intel attestation and provisioning methods function.

Local attestation is supported on Intel SGX supported systems, it provides special CPU instructions that can be executed in order to generate information about the state of the enclave. This information is called a *measurement* and is described as *"an accurate and protected recording of [the enclave] identity"* in the Intel SGX Developer Guide [15]. These measurements can then be generated and shared with other enclaves in order to set up secure channels between them. Local attestation is used when separate applications (sharing one machine, i.e. intra-platform) need to cooperate on a task or share secrets between them.

When an enclave is requested by a service provider to attest remotely (with RA), it first performs local attestation with a *quoting enclave.* This quoting enclave is created and signed by Intel and its purpose is to create *quotes* for the enclaves on a system. When receiving a measurement, the quoting enclave validates the measurement and generates a quote, which basically is a signed measurement. Thereafter, the quote is sent to the application where it is then forwarded to the service provider for verification through one of the two verification techniques. The first type of verification relies on Intel for verification, this is called Enhanced Privacy ID (EPID) verification [15]. The EPID service is not free, and it also requires the enclave to have internet access during the RA process which might be an issue for some setups [37, 38, 34]. The second type of verification, on the other hand, requires a custom verification infrastructure which utilizes software from Intel called *Data Center Attestation Primitives* (DCAP) [16]. A benefit with DCAP, is that it does not require constant internet access for the enclave machine. After verifying the enclave with one of these methods, the service provider can be certain that the enclaved process is running properly on an Intel SGX system.

By sharing cryptographic keys with RA, it is possible to provision the enclave. Provisioning is the process of providing an enclave with the secrets it needs to be able to function, one such secret could be cryptographic keys for decrypting an encrypted binary. Thereby, the enclave can by using provisioning execute confidential binaries or in other ways send or receive secrets.

2.2.4 Earlier Work: Methods of Utilizing Intel SGX

In this section, earlier work in the field is surveyed as to show what methods exist to port applications to SGX. Additionally, this also aims to show what possibilities and limitations there are with Intel SGX.

It is possible to divide methods of utilizing Intel SGX into three categories based on how syscalls are handled. First (I), as envisioned with Intel SDK one can implement a single function or a smaller program in an enclave. Syscalls are completely avoided and a very slim TCB can be maintained at the cost of usability. The defining factor for this method is the total lack of syscalls which can be quite limiting and it makes porting especially difficult. As a result, two additional methods (II, III) were created to enable the use of syscalls inside of the SGX enclave. The second method (II) uses a shim layer to handle syscalls. This layer forwards these calls to the outside world resulting in more external requests, which can be slow depending on how many context switches

[11] Exemplified by Signal: `https://signal.org/blog/private-contact-discovery/`

are required. Additionally, this method requires a wider interface to be exposed to the outside world as well as more computation performed by the untrusted OS, this contributes to a larger attack surface. Lastly (III), with the use of a so called *Library* OS inside of the enclave, syscalls can be caught and resolved locally, this results in much less interaction with the untrusted process on the other side - but resulting in a substantially larger TCB. However, as many syscalls require a response or action from the outside world, some communication must necessarily take place. Still, the benefit with this method (III) is that these requests are fewer and less computation is performed in the untrusted area. Since both solutions (II, III) expose a larger interface and require additional communication with the outside world, especially for performing syscalls, they should provide some type of shielding support inside of the enclave to validate these untrusted requests [63, p. 647] [43, 63].

Both methods (II) and (III) enables the developer to more easily put their current application into an enclave, however, it is important to reiterate that any code put into an enclave does not necessarily make it secure. It must by itself be hardened, a design pattern which is enforced by the stringent Intel SDK but not as much with these frameworks [43]. Additionally, these larger third-party frameworks do not always support the official SGX RA [61], some have no support at all while others provide their own custom solution for this crucial task [54].

In this section a lot of comparison will be made between different method's required Lines of Code (LoC). This metric is mentioned in multiple papers cited in the following sections and is used to approximate the risk of bugs in the system which may result in security vulnerabilities as well as to indicate the difficulty of performing a code review to find such issues. Although rather limited, the metric can still provide a rough estimate of the risk, as shown below some frameworks in the second method (II) add 100K LoC while some frameworks in the third (III) add 1M LoC - a significant difference of one order of magnitude.[12]

2.2.4.1 Method I: Partitioning

The idea from Intel is to move small sensitive functions out of the codebase and re-implement them to be run in enclaves. In this manner, a large project can be partitioned into trusted and untrusted areas [36]. Thereby, a small enclave with a limited interface to the outside can be maintained, minimizing the attack surface of the vital functionality placed therein. However, the enclaved application is therefore limited as it is supposed to be as small as possible while reducing expensive calls to the outside (OCALLs). No syscalls can be made from within the enclave which restricts the trusted application further and as a result the Intel SDK only supports C and C++ code. The lack of official language support is a limiting factor and a reason why unofficial frameworks have been developed (according to methods II and III, described in the two following subsections) [43]. However, when additional features are not necessary it may be prudent to utilize only the official SDK, or care must be taken otherwise. The official SDK has support for remote attestation, it receives patches for attacks such as Spectre and Meltdown, it limits the use of unsafe C functions, and it utilizes hardware instructions (AES-NI, RDRAND) to a large extent - the same cannot be said for all frameworks or SDKs [61].

There exist a number of projects that build upon this idea of using Intel SDK without syscalls while maintaining a small TCB. Below we summarize three papers which aid such implementations.

Brenner et al - in *A Practical Intel* SGX *Setting for Linux Containers in the Cloud* [9] - created an implementation to allow Java functions to call trusted C/C++ code (Intel SDK) in the enclave via a Java Native Interface. They argue that even though a complete Java Virtual Machine could be implemented in the enclave this would not be advisable as the larger TCB would increase the attack surface and cause additional performance overhead. They noted that there is a constant overhead of entering and exiting the enclave (i.e. context switch), resulting in a 83% overhead for small functions (0-512 bytes) and 23% for large payloads (512B-256KB).

It is also possible to allow support for new programming languages while adhering to this method (I), as seen in Rust-SGX. Rust-SGX was introduced in the paper *Towards Memory Safe Enclave Programming with Rust-*SGX [64]. With it, Rust code can be run in the enclave by the use of a layer between Rust and the Intel SDK. This layer maps these worlds together allowing for a trusted area

[12]However, when compiled, the program does not necessarily need to include all available libraries which could result in vastly different binary sizes, making this distinction less clear.

running Rust code, without any syscalls. However, this framework adds almost 40K LoC inside the enclave due to the fact that the SGX port of the Rust standard library takes 35K LoC. Additionally, it is not possible to use the full language functionality as long as no syscalls are allowed.

Another framework which could be placed in this category is EActors[13] by Sartakov et al [50]. EActors is a framework which use Intel SGX to enable an "actor-inspired programming model". These actors can be located in a shared enclave, in separate enclaves as well as in the untrusted OS. The main feature of the framework is to allow for quick communication between these actors with message passing. Message passing is implemented with special memory-mapped files as to only require infrequent syscalls, and therefore improved performance. Communication between actors can be encrypted, in these cases the key-exchange is performed with local attestation supported by Intel SDK. EActors requires an additional 3.3K LoC in the enclave. All three projects listed above also require the Intel SDK which adds a few thousands LoC to the enclave [8].

2.2.4.2 Method II: Shim Layer

To support larger codebases without major rewriting, a potential solution is to use a *shim layer* to transparently forward calls to the outside world. It is then possible to make syscalls from within the enclave, these calls are then translated by the shim layer unbeknownst to the trusted application itself. This solution adds some size to the TCB (usually from a few thousands up to roughly 100K LoC [58, 7]) but primarily the overhead originates from the many external calls made. As mentioned earlier, OCALLs are expensive and by this method (II) each syscall present in the code is simply forwarded to the outside. However, this overhead is decreasing as Intel SGX matures, with protected shared memory or with Switchless Calls which was introduced in SGX2. Regardless, a possible issue here is that any syscall made will be executed in the untrusted OS and therefore vulnerable to attack [7], this also exposes a larger external interface. To mitigate parts of these security issues a shielding layer should be included, although this will add to the TCB.

Two examples of frameworks which work by a shim layered approach are SCONE, for secure containers, and Panoply, for regular Linux applications. Although these papers are a bit dated (2016 and 2017), they serve as an example of the trade offs possible in a shim layered approach. SCONE is commercial and partly proprietary, where a full deployment requires yearly subscription [53, 55]. Panoply is open source (Apache License 2.0), however it is no longer maintained [57]. A third project possibly in this category, is the commercial framework Fortanix which is not compared here due to lack of information, additionally the publicly available material only supports Rust [29, 30].

SCONE provides a full C library (*musl*) in the enclave which is the system call interface the application in the enclave interacts with. By including a C library in the enclave, it enables SCONE to better protect the application against adversarial calls from the OS. Additionally, SCONE offers shields which encrypts files, console output, and network communication. The network shield requires the inclusion of a TLS library inside the enclave which adds significantly to the TCB, this is done to facilitate a trusted TLS endpoint which enables a secure tunnel between clients and the secure container. The SCONE shim layer's *asynchronous system call interface* provides a request queue and a response queue for system calls, performed by the SCONE provided kernel module. SCONE improves the performance as compared to a fully shim layered design by providing a pool of enclave and non-enclave threads which may consume the events on these queues, resulting in fewer enclave transitions. A downside with SCONE is the increased TCB due to the inclusion of the *musl* C library (11K LoC) and the shielding layer (99K LoC including the TLS library). Moreover, some system calls are not supported in SCONE such as *fork*, *exec*, and *clone*. Since the initial SCONE paper [7], Intel SDK has included features similar to the main ones as provided by SCONE, except for the shim layer itself. Specifically, the Switchless feature which uses a task pool and dedicated worker threads, they also provide the option to build the SDK with the official Intel SSL crypto library, and the *Protected File System* is provided to ensure confidentiality and integrity protection of files [40].

The framework Panoply is overall similar to SCONE, but there are a number of differences, partly as different use cases are in mind; Panoply is not aimed at containers but rather conventional Linux applications [58, 7]. Panoply has a smaller TCB at only 20K LoC as it does not include any larger

[13] `https://github.com/ibr-ds/EActors`

library such as libc or a TLS library inside of the enclave. Instead, calls made to Panoply's POSIX API are forwarded to a libc implementation in the untrusted system. Similarly to how SCONE has a dynamic thread pool, Panoply can dynamically scale the number of threads assigned to an enclave by their use of "virtual threads". A separate scheduler process keeps track of the requested threads, and it spawns more enclaves as needed increasing the total number of threads in use for the application (no longer necessary in SGX2). Unlike SCONE, Panoply allows for multi-process applications where the application is split over multiple enclaves to allow for greater segmentation, however these enclaves do not share address space which makes porting harder if this feature is to be used. With this in mind Panoply also supports the syscalls *fork*, *exec*, and *clone*. When an enclave is forked, it is the OS that creates the empty child enclave. Since the OS cannot read the enclave memory it is Panoply which provides methods to encrypt and copy the parent memory to the child, this process is thereby significantly slower than if performed natively in the untrusted OS. Similar to SCONE, communication *between* enclaves is secured with authentication, freshness checks, and reliable delivery - to protect from attacks by the OS. Additional protections such as SCONE's file, network, and console shield is not present in Panoply, although this provides the benefit of a smaller TCB.

2.2.4.3 Method III: Library OS

It is possible, instead of forwarding calls, to mostly resolve the syscalls internally by moving the required libraries into the enclave itself with a *Library* OS (III). As with the shim layer approach (II) the application can (ideally) be put into the enclave without modification, but in this case fewer external calls are required and the external interface can be made smaller. Thereby, it is possible to lower latency as well as to reduce the reliance upon the untrusted OS at the cost of drastically increasing the size of the TCB, as seen with the frameworks Graphene and Haven [8, 63, 43]. The argument against these types of solutions is that by increasing the TCB by ten or a hundred times (measured in LoC) it increases the risk of exploitable bugs which could be used to nullify the security of the enclave. However, it has been claimed that it would be possible to create a Library OS similar in size to the larger shim layer solutions such as SCONE (100K LoC), although this is yet to be seen [63]. Additionally, since these frameworks usually require much more code it could be the case that this method is harder to implement in practice.

Below we compare two highly cited papers and their proposed frameworks which utilize a Library OS inside of the enclave: Haven and Graphene-SGX. Haven was a research paper from Microsoft and little information has been released since. Graphene on the other hand is both active and open source (LGPL 3) [31].

Both Haven and Graphene-SGX work by two layers. First, a Library OS exposes an interface to the enclaved application, such as a POSIX API (in Graphene-SGX) or a Windows 8 API (in Haven), whereby the user application inside of the enclave can remain unmodified. Thereafter, the library OS can resolve calls internally or if necessary make requests down towards the underlying OS. Since the OS cannot be trusted, a shielding layer is provided in between which exposes an internal application binary interface (ABI) to the Library OS layer as well as an untrusted interface to the OS. The shielding layer tries to protect the enclave by validating the calls in either direction, this can be done by ensuring that the calls adhere to the standard, additionally some calls can be further verified. In Graphene-SGX the shielding layer also provides additional support for file protection, encrypted communication between enclaves, and system calls such as *execve* and *fork* [8, 63]. Furthermore, *manifest* files are used to configure which processes may fork and which files can be loaded into the enclave. These manifests are then protected through signatures and checksums in order to prevent injections of unauthorized code [18]. Similarly, in Haven, support is provided for file protection as well as threading features, including a scheduler to protect from host attacks. Both frameworks are quite large in the millions of LoC. The library OS in Haven stands at multiple million lines of code (a whopping 209MB), and its shield module at 23K LoC. Graphene-SGX is not as large, and by itself the Library OS takes only 34K LoC however it requires an additional 1.3M LoC with the inclusion of glibc, their shielding layer takes 22K LoC. Since there exist much smaller libc implementations than glibc it has been claimed that the number of LoC in Graphene-SGX could be reduced by an order of magnitude [8, 63]. However, this has not yet been implemented as these smaller C libraries has more limited functionality.

After the release of the initial paper, Graphene-SGX has added support for multiple new features such as Switchless Calls [44], EPID and DCAP attestation [20] as well as support for a new applications, among them Redis and Tensorflow [49].

3
Method

3.1 Technical Input from Ericsson

After an initial research period, a meeting was held at Ericsson (2020-02-27) as to determine the best way to go forward with the implementation. Both Erlang and security experts at Ericsson provided a basic requirements analysis and their technological options on how this could be achieved, a summary of the meeting follows.

Overview

SGX can provide a number of benefits, mainly integrity protect sensitive functions and confidentiality for intellectual property. A specific use case is the protection of vital TLS endpoints within the telecommunication's infrastructure. For this use case it would be beneficial if an Erlang workload could call secured C code inside of SGX which would host the TLS endpoint. Additionally, to call secured C code would be an easier task than protecting the whole Erlang runtime and therefore a good starting point.

Porting BEAM manually to SGX should probably not be pursued in this thesis due the immense number of syscalls and IO activity performed by BEAM. Instead, the frameworks Graphene, SCONE, and possibly Fortanix, were suggested as these were known to the exports. Graphene was said to be the most interesting candidate due to its open licensing even though it was claimed to be buggy.

Erlang in SGX

In theory, the Erlang runtime could be shrunk down to around 10MB if libraries are excluded as the VM itself is pretty small. Therefore, size should not directly pose any problems.

A possible issue with porting Erlang is the hefty amount of filesystem operations and syscalls performed by BEAM. The *initial* IO can be reduced by limiting the dynamic code loading by BEAM which is possible by statically loading modules. Eventually, almost all IO could be eliminated by utilizing heavy preloading in BEAM. In practice, the idea would be to initialize a minimal enclave and only later during provisioning push an encrypted BEAM binary with all the required functions preloaded. Thereby, it could be possible to achieve both integrity and confidentiality of the whole Erlang runtime, including sensitive Erlang workloads. Additionally, the enclave could make use of encrypted shared memory for some IO, to communicate with the outside avoiding context switches. Syscalls on the other hand are harder to limit overall, and need to be investigated individually whether they can be ignored or replaced. The only general solution to limiting syscalls would be to replace BEAM completely with another Erlang runtime. Some alternative runtimes have been in development but none fits the requirements for Ericsson or are as well maintained as BEAM.

Interfacing with Erlang in SGX

There are a number of ways to interact with the C code that could run inside of an enclave. NIFs are superior to Ports, and C Nodes are also preferred if the SGX implementation would be unstable since that would not crash BEAM and the SGX process could be restarted.

One idea was whether it would be possible to separate a single BEAM scheduler such that a single Erlang process would run inside of the enclave while the complete BEAM would run in the untrusted area. This was deemed to be very difficult based on the inner workings of BEAM, additionally this could have unforeseen security consequences. Another rejected idea was to put the enclaved BEAM inside of the untrusted BEAM, something which was not well received by the Erlang experts who stated that it was unadvised and had never been done before. A third rejected idea was to send whole functions between BEAMs. In reality, it *is* possible to tell another BEAM instance to make a certain function call but this function must be available and prepared beforehand to be able to run it.

3.2 Design Choices

There are multiple ways to achieve the goals laid out for the two use cases presented earlier in section 1.2. In this section, different approaches will be presented of how to call trusted C code as well as to partially or fully run Erlang/OTP in an enclave. Finally, last in this section the four design chosen to be implemented are summarized. Some of the methods involve running C code inside of an enclave while the actual Erlang/OTP application is outside of the enclave, while others wrap the whole (BEAM) runtime in order to run a limited subset of Erlang/OTP applications inside an enclave.

Rewriting BEAM

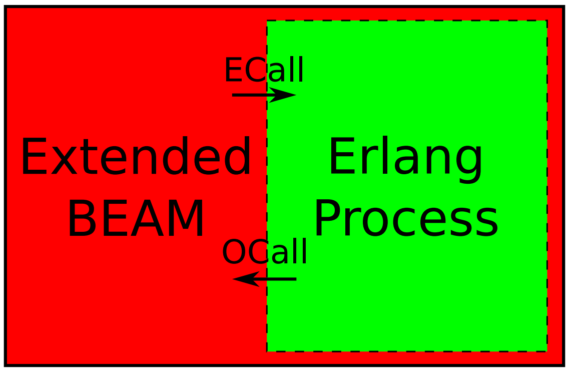

Figure 3.1: Suggested design, later abandoned due to time restraint: separated Erlang process from BEAM *(untrusted code is shown in red and enclaved code in green)*

An approach that was abandoned early was to rewrite or extend how BEAM works in order to allow for Erlang processes to run inside an enclave without dependency on additional third-party libraries. Initially we had a few design ideas how to achieve this and we believed this to be an efficient solution. While this approach is not completely implausible, it would require a very in-depth knowledge of BEAM and the Erlang/OTP experts consulted advised against it for a project of this length.

The approach could be similar to method (I) as described in 2.2.4, however as the current runtime requires many system calls this would not be possible with BEAM. In theory it could be possible to create a (I) solution if Erlang could be compiled to C (or another representation) and then run directly in the enclave, however this would be hard to maintain and all benefits with the powerful BEAM VM would be lost.

Additionally, another design involved restricting the runtime and placing this bare minimum execution environment in the enclave which would belong to an untrusted BEAM as shown in 3.1. This design would basically place a minimal trusted BEAM *inside* of another untrusted BEAM, something which is not usually done with and could show to be a near impossible feat. Additionally, the runtime is by default quite large and it is not obvious how to move certain parts out of BEAM to a smaller barebones VM, and how this in-turn would be dependent on functionality in the external BEAM instance. Separating vital functionality to the untrusted area can also break security promises. For instance, the scheduler could be run in the untrusted area to lower the TCB size it is assumably not very secure and it would leak a lot of data about the trusted execution.

Lastly, by rewriting BEAM it could also be possible to integrate the whole runtime into the enclave similar to what is shown below in figure 3.2 where a third-party framework is used. However, in this case the same problem would be solved by a custom layer specifically for Erlang which would take care of the syscalls used by BEAM. The custom framework would only expose the syscalls used by Erlang and could include other optimizations as well, either by using a shim layer or a library OS like approach. We deemed this solution to be the most promising and it is later discussed in section 4.3.

Neither of these custom solutions, which require changing the BEAM runtime itself, were deemed reasonable to implement during this project due to time constraints.

Wrapping BEAM

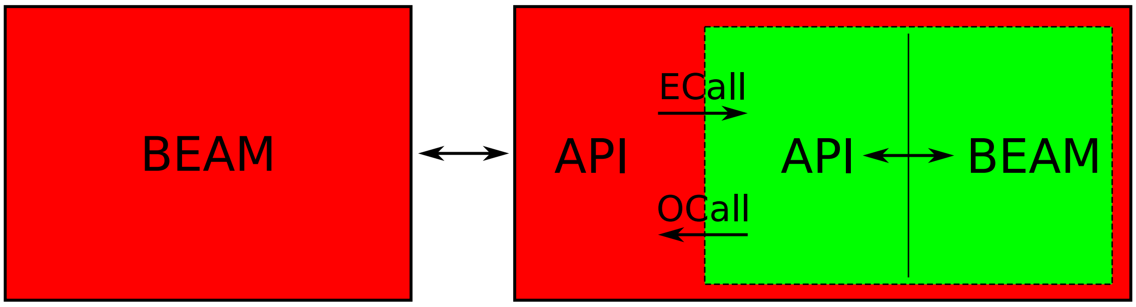

Figure 3.2: Chosen solution: possible to implement with third-party frameworks

Creating a wrapper around BEAM inside of SGX, which could capture and handle the necessary syscalls seems like a possible option. To accomplish this task the frameworks SCONE and Graphene will be investigated. These two frameworks were chosen because they utilize different methods of wrapping (shim layer method II and LibOS method III respectively), and they are still active projects unlike Panoply and Haven. The implementation would look something like figure 3.2 where the API parts are covered by the wrapper inside the enclave. Additionally, some functionality might need to be added in the untrusted area to facilitate communication between Erlang nodes (i.e. between BEAM instances).

Partitioning Safety Critical Parts In C

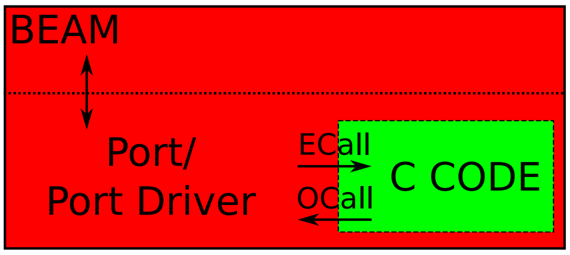

Figure 3.3: Suggested design, later abandoned (in favor of better alternatives): run C code from Erlang with Port or Port Driver

A solution which belongs to method I is possible to achieve by calling secure C code from an untrusted BEAM instance. This is enabled by the functionality in Erlang to communicate with C functions, and as described in section 2.1.1 there are several ways to do this: namely with a NIF, C Node, Port, or a Port Driver.

After the discussion with the Erlang/OTP experts at Ericsson (mentioned earlier in section 3.1), designs for implementations using Port and Port Driver (shown in figure 3.3) were scrapped since these were considered to have no benefit over implementations utilizing NIF and C Nodes. The decision was therefore made to continue with C Node and NIF designs.

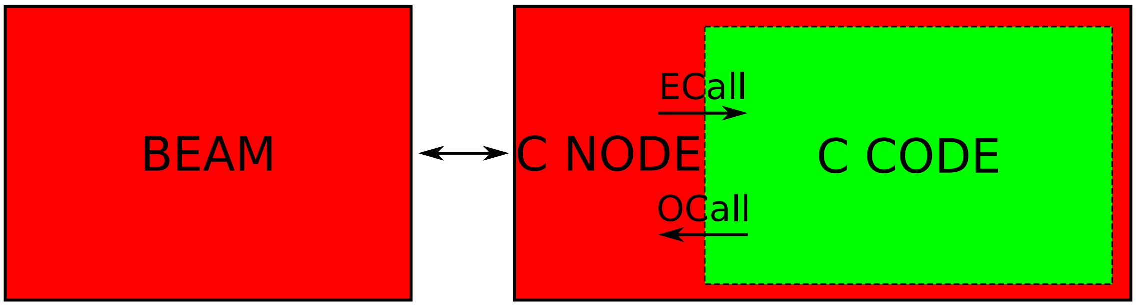

Figure 3.4: Chosen solution: run C code from Erlang with a C Node

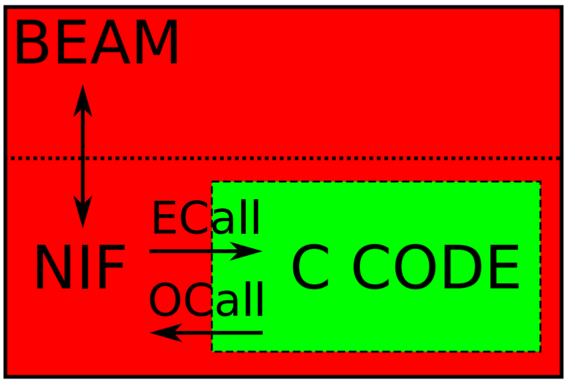

Figure 3.5: Chosen solution: run C code from Erlang with a NIF

A design of a C Node that have safety critical parts inside of an enclave will be created, as shown in figure 3.4. Similarly, a NIF design will be developed as can be seen in figure 3.5. As described in 2.1.1, the C Node is external to the untrusted BEAM instance and can be communicated with using Erlang message passing functions. The NIF on the other hand belongs to the untrusted instance and is executed as a regular function call, similarly to how many libraries are called from Erlang.

3.2.1 The Four Designs Chosen

Four general designs were chosen and are summarized below.

The first two designs adhere to method (I), i.e. they utilize partitioned applications in SGX which do not perform any syscalls from within the enclave. These designs execute secure C code directly in Intel SGX, via C Nodes or NIFs. The idea here is to implement all sensitive functionality in C which would then be called from the untrusted Erlang runtime. Thereby, the sensitive C workloads themselves would be very well protected, whereas the Erlang runtime would not be. As mentioned in the first use case, the NIF design could be useful when wanting to perform secure cryptographical operations, many of which are already implemented in C and called via a NIF in Erlang.

The final two designs use two different frameworks, the shim layer SCONE and the LibOS Graphene. Basically, they utilize different methods of resolving syscalls inside of the enclave which then could enable the support of BEAM with arbitrary Erlang payloads therein (according to the second use case). In this solution, the hope is that BEAM would not need to be modified, although it could be to gain some performance or security advantages: for example by lessening the number of syscalls and the amount of IO activity.

4 Implementation

The implementation of the four designs began with the deemed easier method I which consists of a C Node and a NIF. Once a simple prototype function was completed for both C code designs, the frameworks for method II and III were tested. Finally, the first method's NIF design was extended with OpenSSL support to test the first use case. All implementation details and measurements can be found in the source code[1] for this thesis which is available under a permissive BSD license.

4.1 Method I: Partitioning

For method I an SGX sample[2] from Intel was utilized in order to reduce the implementation time. The sample had a basic enclave and application part which we rewrote from C++ to C and subsequently cleaned up to remove any parts not needed to simplify Erlang interfacing with it. For the C Node and NIF implementation, two basic functions were added to the enclave code to test the implementation: *increment* and *return*. These two functions enabled the simple functionality of incrementing and returning a secret integer inside of the enclave. Additionally, limited RSA functionality was later added to the NIF with the inclusion of OpenSSL in SGX. The application code exposes bindings to the SGX-protected functions through ECALLS and the enclave code can return the result to the application with OCALLS.

Although the idea was to support RA, this was not completed in the prototype provided. The implementation was started but had to be abandoned in favor of the OpenSSL, SCONE, and Graphene implementations. Additionally, RA for native C code running in SGX has been done before and the supervisor at Ericsson suggested more time should be spent on Erlang specific areas.

4.1.1 Basic C Node

To get a C Node version working, the cleaned sample code from Intel was modified, adding parts that allowed an external Erlang application to call into it by executing code as in listing 4.1. The C code within the C Node then receives the call from the Erlang application in which it then invokes the ECALL to pass the desired operation into the enclave. The enclave processes this request and then return the result to the C Node through an OCALL which is then relayed to the calling Erlang node.

```
call_cnode(Msg) ->
    {any, 'nodename@cnodehostname'} ! {call, self(), Msg},
    receive
            {cnode, Result} ->
                Result
    end.
```

Listing 4.1: Erlang C Node calling code

[1] https://github.com/Erlang-Enclave-Thesis/sgx-erlang-extension
[2] https://github.com/intel/linux-sgx/tree/master/SampleCode/SampleEnclave

4.1.2 Basic NIF

Unlike C Nodes, NIFs are integrated directly into the VM and therefore execute on the schedulers threads, this means that the built binaries for the application have to be linkable. Changes to the code were made, for instance moving the main function content from the C Node approach into initialization functions needed for proper communication with the NIF. This resulted in a working NIF that could be loaded and called from BEAM, similar to how regular Erlang functions can be called.

4.1.3 OpenSSL NIF

The NIF prototype was later extended with cryptographic functionality, this was chosen in favor of a C Node approach as Erlang generally makes cryptographic calls by this method. To support the use case to protect TLS sessions with SGX, the official Intel SGX SSL was chosen which brings the popular OpenSSL library into SGX.[3]

As to allow for comparison between regular and SGX-enabled OpenSSL (version 1.1.1d), both projects are supported in our Makefile which performs the necessary linking. For a simple prototype we added support for four RSA functions via Erlang NIFs: generate key pair, get public key, sign data, and encrypt data. These functions perform calls to the C back end which communicate with the standardized OpenSSL API, either in untrusted space or inside of SGX. The RSA functionality created was based on the examples in Intel SGX SSL with the following settings; An RSA key length of 4096 bits is used, encryption is performed with RSA_PKCS1_PADDING and the signature uses NID_sha message digest algorithm. This minimum viable product shows basic RSA functionality which demonstrates how cryptographic functionality can be offloaded from Erlang onto an enclave. RSA was chosen as it is a common algorithm which can be used in TLS. To support a full TLS endpoint should be possible as the full OpenSSL library is available within the enclave, but could not be finished due to time constraints.

4.2 Method II and III: Resolving Syscalls

Before starting exploring method II and III it was necessary to measure the syscalls performed by BEAM as these need to be handled by a potential framework. Thereafter, we investigated the most prominent frameworks as earlier described in section 2.2.4. Graphene was our main focus as it was the only fully open source framework out of these, additionally we also investigated SCONE and Fortanix. However, Fortanix was quickly abandoned as they did only provide an image for Azure.

4.2.1 Syscalls Used by BEAM

The syscalls performed by BEAM were measured to get an idea of what the overhead could be for running Erlang in SGX as well as to determine which syscalls need to be supported inside of the enclave. The measurements were made with *strace* to determine which and how many syscalls are performed by BEAM for simple workloads, including all calls made by any children the process creates. The same measurements were also made with a functionally equivalent Java workload (for JVM) as a reference point since that is a VM runtime which has received some attention in earlier work as well as a runtime which is supported in SGX with SCONE.

Table 4.1: Syscalls performed by BEAM and JVM

Workload	**Unique syscalls**	**Total amount of syscalls (average)**
Erlang/OTP 23		
Hello World	67	25k
Hello World (threaded)	67	25k
Java JVM 11.0.7		
Hello World	48	2.4k
Hello World (threaded)	48	3.3k

[3]`https://github.com/intel/intel-sgx-ssl`

The workloads tested consist of two simple '*hello world*' programs, the first only performs a simple print while the second launches five additional threads which all print a message. The compilation of the programs is not included in these measurements. As seen in table 4.1, Erlang performs vastly more syscalls for the basic '*hello world*' program, around 40% more types of syscalls and 940% more calls in total. It is worth noting that while the calls made by the JVM were quite stable, the total number of calls for BEAM peaked at 40k the first execution to vary between 15-25k for subsequent calls. For four executions the amount of syscalls made by BEAM averaged around 25k, but this is only an estimation as the numbers varied greatly between executions.

The Java runtime differs in many ways from BEAM. As shown in the table, when we introduce multi-threading to the '*hello world*' program, the JVM performs 37.5% more syscalls while BEAM does not make any more calls than before. The reason for this is because BEAM by default creates multiple schedulers regardless of the program executed while Java is assumably more conservative. It is important to note that far from every call is vital for BEAM to function, as a majority of calls performed is done by the scheduler, e.g. probing CPUs (with *sched_yield*) and sleeping workloads for microsecond amounts (with *timerfd*).

4.2.2 Graphene

For Graphene, most syscalls performed by BEAM for our '*hello world*' program are supported, all but 9 out of the 67 according to the Graphene documentation [32]. An additional five[4] were claimed to have partial support. However, when cross-referenced with their source code[5], not nine but only seven syscalls were not implemented at all. The seven completely unsupported syscalls in Graphene were: *madvice*, *sysinfo*, *timerfd_settime*, *timerfd_create*, *prlimit64*, *statfs*, and *prctl*.[6] Seemingly, most of the unsupported syscalls could be ignored at the cost of performance and stability which was partly confirmed by Graphene [1].[7] *madvice* is usually called to achieve performance improvements and *sysinfo* only returns load, memory, and swap statistics. *timerfd_settime* and *timefd_create* are used by the scheduler to put processes to sleep for microsecond amounts, and in our test build this call could be ignored completely. In the case of the '*hello world*' program, *prlimit64* is used to limit the allowed stack size and number of file descriptors which can be opened for the calling process.[8] For our measurements *statfs* was not necessary, and it was only used to determine if *SELinux* is in use on the machine. Finally, *prctl* was used to set the name of the different scheduler threads within BEAM.

Since Graphene is fully open source, acquiring the code needed to build Erlang/OTP was straightforward. Graphene uses so-called manifest files to configure and sign the binaries (and the associated dependencies) so that they are able run in the enclave securely [18]. A manifest file and the associated Makefile from the Graphene examples were modified to allow Erlang to run in the Graphene environment. However, this was a trial-and-error-prone approach since BEAM both depended on more files than expected and as it wanted to start several processes during its lifetime such as (but not limited to) *beam.smp*, *inet_gethost*, and *erl_child_setup*. Since Graphene requires manifest files and signatures for each binary it executes, it denied BEAM from loading these which resulted in it crashing. By adding separate manifests for each binary required, the execution could progress further. Unfortunately the lack of support for certain syscalls and certain primitives, caused BEAM to not progress beyond certain a point. By cross-referencing the supported syscalls by Graphene with the source code of Erlang, we discovered that the syscall *timerfd_create* could be disabled. As a result, a script was created to force BEAM not to use the syscall (*timerfd_create*), as shown in listing 4.2. Thereby, BEAM could progress further when executing inside of Graphene, but it could still not fully initialize.

```
sed -i "s/define HAVE_SYS_TIMERFD_H 1/undef HAVE_SYS_TIMERFD_H/" $(ERLANG_SRC)/erts/configure
```

Listing 4.2: Part of the Makefile that patched one of the configure files for Erlang/OTP

[4]Partially supported by Graphene: *clone, fcntl, readlink, bind, setsid*

[5]`https://github.com/oscarlab/graphene/blob/master/LibOS/shim/src/shim_syscalls.c`

[6]The two additional calls not listed as supported but seems to be supported in the source code were *sched_getaffinity* and *getppid*

[7]`https://github.com/Erlang-Enclave-Thesis/graphene-erlang/issues/3`

[8]*prlimit* is similar to *getrlimit* and *setrlimit*, however only *getrlimit* has partial support in Graphene

The Graphene developers were asked for assistance in understanding how the issues could be resolved which led to progress in making Erlang executable in Graphene as well as discovering some bugs in Graphene.[9] The Graphene developer Dmitrii Kuvaiskii provided a thorough *strace* analysis of which syscalls are required by an Erlang '*hello world*' program, comparing it to what features are not yet supported in Graphene.[10] Based on this analysis, it became evident that although some syscalls could be ignored or later supported by Graphene, the main issue is BEAM's use of MPMC pipes instead of point-to-point communication in some cases. There is no way for Graphene to determine if the pipes are used for inter-thread or for inter-process communication.[11] As a result, Graphene chooses to TLS encrypt all pipes although he notes it could be possible to introduce a keyword in Graphene if it is certain that *all* pipes are only used within one process and therefore not necessary to encrypt. The Erlang experts at Ericsson remarked that these pipes are used for two things. Namely, they are used to wake up schedulers sleeping in *poll/epoll/select* configured in *erl_poll.c* as well as by Erlang Ports by the function *open_port({spawn,_}).* The later uses UNIX domain sockets and pipes to create child processes such as *intel_gethost* and it was claimed that the modification of this feature would require much more work in *sys_drivers.c.*

4.2.3 SCONE

As we had rely on prebuilt Docker images from a private repository without available source code, most work we did on SCONE is based on personal communications with the their team and especially Christof Fetzer. First, a request had to be made to the SCONE team to acquire access to their Docker images. Access was granted and a Dockerfile was created that utilized their SCONE crosscompiler image to compile Erlang/OTP for SCONE's shim layer. Building Erlang/OTP proved to be quite difficult because Erlang requires quite a lot of syscalls and the runtime allocates a lot of memory if no options are provided. A patch was produced to force Erlang to use less memory by default, by adjusting the default values, but this still did not help since Erlang just could not start in the SCONE environment to completely build all components. The SCONE developers were asked for support and they supplied an experimental image specifically for BEAM which they had been working on, with a partially working Erlang executable. This image could unfortunately not communicate with external Erlang processes because certain fork operations did not succeed which caused network related operations to fail, but it could support some basic Erlang functionality. The functionality was ascertained in SGX simulation mode by running a number of tests provided by SCONE.[12] Unfortunately, the combination of the hardware, BIOS, and driver used was not able to execute the image on SCONE's enclave running in hardware mode, therefore only the simulation mode was used to test the functionality of the image. The SCONE developers claimed to have succeeded in running Erlang in the hardware enclave on their systems and did not know why it could not run in our hardware mode. As little progress could be made without the source code or by heavily relying on the SCONE developers, no further progress was made on this front. To summarize, the SCONE team is working on supporting Erlang officially but it is currently not ready for deployment in production.

4.2.4 Fortanix

Although we had initially disregarded Fortanix due to lack of publicly available information, we still contacted them as to ascertain if their solution could support BEAM on our hardware platform. They provided access to their proprietary Enclaved OS built for Microsoft Azure. It was not clear whether it would be able to support BEAM or if it could be modified to run directly on our hardware. Consequently, mainly due to time constraints, we did not pursue Fortanix further.

[9] https://github.com/oscarlab/graphene/pull/1483
[10] https://github.com/Erlang-Enclave-Thesis/graphene-erlang/issues/2
[11] https://github.com/Erlang-Enclave-Thesis/graphene-erlang/issues/2
[12] https://github.com/scontain/erlang-examples/

4.3 Explored Alternatives

Additional to the four designs which were attempted in this thesis, two more alternatives were also explored. These alternatives could provide a way to port a Erlang runtime to BEAM, primarily utilizing method II and III (shim layer and LibOS).

4.3.1 Creating a Custom Layer

After reaching out on social media, we were contacted by the company Decentriq which is working with confidential computing and SGX. At a subsequent meeting (personal communication: 2020-04-22 with Stefan Deml), they showed concern for the security and performance risks enabled by utilizing the general frameworks which we have explored previously. Instead, they clearly recommended for the creation of a custom later specifically created for BEAM which we present some ideas and conclusions on in this section.

As seen with Rust-SGX, it is possible to port an interpreter to SGX without immense frameworks or forwarded syscalls. Rust makes this possible as it is a compiled language with garbage collection on compile time and in many ways similar to C which is already supported in SGX enclaves. However, with Erlang this is much more complicated as it is executed in a VM with its own scheduler and garbage collector. Additionally, the VM (by default BEAM) requires many syscalls and IO operations which would then need to be ported or somehow removed from the runtime to run inside of SGX. We believe it would not be recommended to separate the components of BEAM (e.g. the scheduler) between secure and untrusted areas, as this would leak a lot of data and create new attack vectors from the untrusted side.

As discussed earlier, it is possible to create a solution whereby a custom layer improved performance and security can be achieved while running the whole BEAM inside of SGX. The custom layer would do a number of things; The IO calls could be made in the back end with memory-mapped IO, creating a virtual file system, and the related syscalls caught and resolved appropriately. A major component of the custom layer would be to create the appropriate logic for each syscall, depending on whether it can be ignored, resolved locally in the enclave, or needs to be forwarded to the outside OS. Since *pthreads* are now supported by SGX2 it could be possible for the scheduler to function as before with multiple threads and scheduling, here the whole BEAM instance and all its threads could be in one enclave. If additional segmentation is necessary it would be natural to have multiple BEAMs running as separate processes both in the untrusted area and in multiple enclaves as communication between these is normal in Erlang execution. The solution could also be extended to include a modification of the BEAM code-base to reduce slow and risky calls, however this would most likely be an immense task which would also require a forking of the BEAM code base.

As earlier mentioned in 3.1, the Ericsson experts provided some valuable insight in how BEAM could be ported to SGX which also extends to a custom layer. The main point would be to statically preload all required modules into BEAM which could remove the need for almost all IO. Additionally, context switches could be removed with the use of shared memory or Switchless Calls. However, the unavoidable fact is that BEAM requires many syscalls which will always be a possible risk with BEAM inside of SGX.

4.3.2 Replacing the BEAM Runtime

It could be possible to run Erlang without BEAM which would create new opportunities for secure execution. One question is whether Erlang code can be compiled directly to C, or an easier executed intermediate representation (such as bytecode), and in general if a simpler runtime than BEAM could be used. However, BEAM is a major reason for Erlang's performance and reliability, and consequently its success.

The issue with porting Erlang is partly because of the nature of the language itself, this includes automatic garbage collection during runtime and native support for abstract primitives such as multithreading. However, as described in section 4.2.1, BEAM is not designed with SGX in mind nor does it try to minimize the amount of syscalls made. On the contrary, regardless of the workload, BEAM starts multiple schedulers which all perform thousands of syscalls each while Java's JVM was much more conservative (for the tested '*hello world*' programs), both in the the total amount of syscalls and the number of unique syscalls performed. As JVM is also a powerful VM and still only performs a fraction of syscalls as compared to Erlang's BEAM, it becomes apparent that it could be possible to reduce the number of syscalls if BEAM were to be replaced or greatly modified. By doing it so, it would reduce the work required to port the runtime either to a custom layer or to a general framework as well as to possibly provide security benefits as less and fewer types of syscalls are made to outside. A number of lightweight alternatives to BEAM are in development[13], but it is unclear whether they would be able to make the same promises as BEAM does.

[13] Projects include: `https://github.com/kvakvs/ErlangRT`, `https://github.com/archseer/enigma`, `https://github.com/bettio/AtomVM`, and `https://github.com/lumen/lumen`

5

Evaluation

In this chapter, we present the evaluation which was performed to benchmark the performance of the prototypes for method I, calling C code in SGX. These results are later discussed in chapter 6 in regards to the earlier proposed use case of securing TLS with OpenSSL in SGX enclaves. Since SCONE and Graphene were not working on our hardware setup, method II and III could not evaluated.

5.1 Method I: Partitioning

The execution time of the prototype functions in C from Erlang was measured for both the C Node and the NIF, with SGX as well as similar functionality completely without SGX. The basic functionality which provides the ability to increment and return a number from inside of the enclave was tested for both designs. Additionally, the RSA functionality with OpenSSL was tested for the NIF. The tests were conducted using a simple wrapper function which calls Erlang timer:tc[1]. These tests are available as *eval_increment* and *eval_rsa* in the source code which we provide (*enclave_communicator.erl*). The full test output is available in appendix A and B. The tests conducted only measure the execution time of said Erlang functions and the results may vary depending on other load on the device that moment or whether the data was in cache at the moment of execution, especially as we are counting microseconds. Regardless, general conclusions can be drawn from this data about the delay caused by using SGX without optimizations. However, as we measure from time of call on the untrusted side, until the enclave returns, the majority of delay is due to context switches (similar conclusions have been mentioned in earlier work, section 2.2.4). The data does therefore not show the effectiveness of executing *inside* of SGX which should yield similar performance as it is executed in the untrusted area on the CPU with encryption performed in hardware. What it does show is the worst case overhead, caused by SGX for a given workload. These measurements do not take into account the time it takes for the initial creation of the enclave or the compilation of the Erlang program.

5.1.1 Basic C Node and NIF

Table 5.1: C Node execution time for basic workload

Tested functionality	**In total**	**Excluding first call**
Increment		
Without SGX	538.40 μs	76.24 μs
Simulated SGX (Release Mode)	564.93 μs	111.14 μs
Hardware SGX (Release Mode)	612.50 μs	104.48 μs
Overhead Hardware SGX / Without SGX	13.76%	37.04%
Return		
Without SGX	76.17 μs	75.24 μs
Simulated SGX (Release Mode)	101.83 μs	101.66 μs
Hardware SGX (Release Mode)	67.50 μs	67.28 μs
Overhead Hardware SGX / Without SGX	-11.38%	-10.58%

[1]`http://erlang.org/doc/man/timer.html#tc-3`

Table 5.2: Average NIF execution time for basic workload

Tested functionality	**In total**	**Excluding first call**
Increment		
Without SGX	0.13 μs	0.00 μs
Simulated SGX (Release Mode)	22.43 μs	19.69 μs
Hardware SGX (Release Mode)	31.50 μs	28.07 μs
Overhead Hardware SGX / Without SGX	24,130.77%	At least 280,000.00%
Return		
Without SGX	0.07 μs	0.00 μs
Simulated SGX (Release Mode)	16.63 μs	16.48 μs
Hardware SGX (Release Mode)	28.00 μs	27.86 μs
Overhead Hardware SGX / Without SGX	39,900.00%	At least 280,000.00%

As seen in table 5.1, the first time called a connection is set up to the C Node resulting in a drastic overhead. We therefore show the average execution time excluding the first call to the right in the table. Comparing this with the NIF measurements shown in table 5.2, it becomes evident that even though the NIFs are much faster than C Nodes the overhead with SGX is of the same magnitude, often around $30\mu s$. However, why the function 'return' is faster for the C Node in SGX than without is unclear as it is stored in encrypted RAM and still quicker, both the average and the quickest function call.

To summarize, to make function calls to SGX is slightly slower than without when we are required to perform context switches as done in our prototypes. However, this overhead seems to be quite low as it often resulted in around $30\mu s$ additional execution time, although this varied. As the overhead is quite static between workloads, it implies that the context switches account for a majority of the overhead by SGX which also matches the earlier work described in chapter 2. Thereby, if this is the case, it would not be viable to place an Erlang runtime inside of SGX with around 25,000 syscalls without optimizations as to avoid context switches. Finally, we note that the worst-case overhead in these tests were roughly $30\mu s$ showing a baseline for what we can expect by executing simple C functions in SGX, without any optimizations.

5.1.2 OpenSSL NIF

Table 5.3: Average NIF execution time for RSA workload

Tested functionality	**For first execution**	**For second execution**
RSA key generation		
Without SGX	1,292,506.40 μs	1,329,274.13 μs
Simulated SGX (Release Mode)	1,568,162.87 μs	1,608,280.63 μs
Hardware SGX (Release Mode)	1,306,003.00 μs	1,266,289.87 μs
Overhead Hardware SGX / Without SGX	1.04%	-4.74%
RSA signing		
Without SGX	13,888.03 μs	13,890.00 μs
Simulated SGX (Release Mode)	13,938.80 μs	14,020.46 μs
Hardware SGX (Release Mode)	14,137.27 μs	14,137.50 μs
Overhead Hardware SGX / Without SGX	1.79%	1.78%
RSA encryption		
Without SGX	228.00 μs	225.63 μs
Simulated SGX (Release Mode)	367.17 μs	366.57 μs
Hardware SGX (Release Mode)	373.37 μs	376.20 μs
Overhead Hardware SGX / Without SGX	63.76%	66.73%

OpenSSL was used both inside of the enclave (Intel SGX SSL) and outside in the regular untrusted OS, although these two versions might differ in some ways they both utilize the OpenSSL 1.1.1d source code and support the same functionality. The average execution times of three of the implemented RSA functions are shown, as can be seen in table 5.3. To get an idea of the difference between runs, two consecutive runs of 30 executions each are shown side-to-side. RSA signing and encryption had a quite stable execution time when comparing the average. Based on the data of both runs included here, the overhead in SGX for signing was around 1.8%, and for encryption a hefty 63 to 67%. In absolute numbers, the overhead was between 145 and 250μs. RSA key generation on the other hand could take anywhere between 100kμs and 2Mμs (based on raw data as shown in appendix B), a wide span partly due to the generation of pseudo random data. Over the 30 runs executed, the average difference between the execution time in hardware SGX and no SGX was a few percentage points and therefore not enough to draw any conclusions of which were faster. The quicker functions had a larger percentage overhead which could indicate that the main contributor to this delay are the context switches, which as mentioned earlier can be greatly optimized. A possible explanation to the larger constant overhead, is that Intel SGX SSL can perform multiple calls between the trusted and untrusted areas during our single function call.

6
Discussion

In this section we discuss the current status of our research, if the implementations achieved the goals laid out, and what our recommendations are going forward.

6.1 Achievement of Goals

To recap, the goal of this thesis was to investigate how Erlang can be secured with Intel SGX as to protect a cloud-based telecommunications system. We presented a short case study of the lawful interception capabilities necessary in these systems which shows a need for TEE-technologies such as SGX. Based on this scenario, two use cases were created which could aid this system and these were later partly implemented. The first use case consists of executing Erlang in an untrusted environment, which in turn calls trusted TLS (C-based) functionality in SGX. The larger second use case, aims to support the execution of Erlang code inside of SGX. To achieve the first use case, three requirements were set up; Specifically, to create a prototype to call C functionality in SGX from BEAM (i) and adding support for TLS inside of SGX (ii) which then could be verified with RA to achieve integrity and confidentiality of the SGX workload itself (iii). Similarly, the second use case also had three requirements; To protect the Erlang workload it would be necessary to enable execution of an ERTS (here BEAM) in SGX (i), add support for RA to ensure correct execution in SGX (ii) and thereafter to provision the enclave with verified and confidential Erlang binaries (iii).

For the first use case a basic prototype was created supporting cryptographic (including TLS) functionality inside of SGX callable from a regular Erlang runtime (i, ii). Moreover, although we worked on RA support (iii) it was abandoned in favor of supporting the other aims which were more interesting from a research point of view. Basically, RA for C code is supported on SGX and can be implemented based on earlier work in the field and by instructions from Intel.

The second use case was shown to be much more difficult due to the vast number of syscalls performed by BEAM. Although we investigated a number of potential methods which could achieve this use case, no functioning prototype was created due to a number of difficulties, for instance as the most promising framework Graphene would require modification of the BEAM runtime itself. Regardless, we showed that the first aim (i), to run Erlang in SGX, is possible as SCONE currently provides basic support for BEAM in SGX enclaves and they aim on providing official support in the future. As the first aim was not fulfilled, it was not possible to implement or test RA and provisioning for BEAM SGX (ii, iii). Although, we note that RA is supported by the aforementioned frameworks (and could be supported by a custom solution as well). Additionally, a rough idea of how to provide integrity and confidentiality of a Erlang workload in SGX was discussed with Ericsson as mentioned in section 3.1.

6.2 Status of Erlang SGX

In this section we present the status of the approaches which we have explored and discuss some recommendations going forward.

6.2.1 Method I: Partitioning

In this thesis, we successfully demonstrated two methods of calling sensitive C/C++ payloads inside of SGX by using the Erlang interfaces C Node and NIF. Based on our solution, it would be possible to quite easily call secure code, either C/C++ or (we assume) any unofficially supported language such as Rust, from an untrusted BEAM runtime. Although the prototype created does not perform optimally, since each call from and to the enclave requires a context switch, this could be mitigated. In particular, it would be possible to use Intel Switchless Calls which reserves threads on each side, rather than the same thread switching context for each OCALL or ECALL performed, which could significantly improve performance [62].

To call secured C code brings a lot of potential upsides since BEAM is on the back end already using such NIFs to perform certain operations including cryptography.[1] Therefore, we believe that these sensitive functions could be ported to SGX and then called seamlessly from Erlang based on the prototype NIF which we provide. With remote attestation it would be possible, as with any native SGX function, to enable confidentiality and integrity of the binary and to ensure that the enclave is running properly. However, it is important to note that as long as the call is made from the insecure Erlang runtime, the security guarantees made by SGX are greatly affected. In this situation, Erlang makes the call to SGX via an untrusted C function which in turn calls the trusted code. Anywhere in the untrusted chain of execution, a malicious kernel could modify or drop the request or the response as well as replaying messages and other classic man-in-the-middle attacks [58]. Therefore, even though this solution would provide a number of benefits there will always be vectors of attack as long as Erlang itself is not running securely. Future work for this method could consist of investigating methods of cryptographically secure the communication between these sides, however, as Erlang itself can at anytime be attacked by the kernel according to our threat model we believe this to be somewhat limited.

6.2.2 Method II and III: Resolving Syscalls

We came to the important conclusion that BEAM is not far from being usable inside an SGX enclave with the main frameworks presented in this thesis: SCONE and Graphene.

SCONE was the framework which had the most support for BEAM, we believe this is partly due to how it passes through the syscalls to the operating system which makes supporting the calls needed for BEAM easier.[2] However, this also opens up for a larger interface towards the untrusted system and many syscalls are performed by this possibly malicious system. Additionally, SCONE is not fully open source, and the unknown code required for execution could greatly effect the trust model. However, as security was not directly evaluated in this thesis, future work is required as to determine the effects these frameworks have on Erlang.

We discovered that Graphene could most likely support BEAM with one caveat, either BEAM must be modified to only use (TLS-encrypted) point-to-point communication and not MPMC pipes, or an option must be added in Graphene to allow insecure pipes while ensuring these pipes are only used intra-process (inter-thread). To remove MPMC pipes in BEAM is probably the best approach as it would not risk unencrypted pipes being used inappropriately. These pipes could be replaced with a fair bit of work, as explained in 4.2.2. Although some syscalls performed by our Erlang '*hello world*' program were not supported by Graphene, this should not be an issue for two reasons: the syscalls lacking support were not vital and the Graphene community was helpful and seemed positive to adding more functionality to their framework if needed. To conclude, based on our communication with the Graphene developers, it seems like the removal of these pipes is the only thing that stands in the way for basic support of BEAM in Graphene. However, this needs more research as other obstacles could be discovered further down the road.

6.2.3 Alternative Solutions

The Intel documentation, as well as previous research presented in section 2.2.4.1, state that only sensitive functions should be ported to SGX to maintain strict security requirements. However, BEAM is very far from being able to execute completely in an enclave without external syscalls. In this case, a very deep knowledge of BEAM would have been necessary to perform the identification and rewriting most of the runtime. As we see it, it would be easier to base the work on a minimal

[1]For instance the *crypto* module: `https://github.com/erlang/otp/blob/master/lib/crypto/src/crypto.erl`

[2]Although, we must not forget that SCONE also had worked specifically to support BEAM while no such work had been done prior for Graphene

runtime, as mentioned earlier in 4.3.2, or write a whole new runtime from scratch. Looking at the vast amount of code and complexity of the Erlang/OTP project, it is clear that if such an endeavor should be accomplished a team of developers have to spend quite some time to create a SGX-ready runtime.

Instead, a more reasonable approach would be to create a custom layer for the current BEAM runtime. Although, the work could perhaps be reduced if the runtime could also be modified as to remove or minimize certain syscalls. The custom layer would function similar to third-party frameworks, utilizing either a shim layer or a LibOS approach, but be customly fit for BEAM. First it would be necessary to explore which syscalls are either required, recommended, or optional for BEAM. It could be the case that many syscalls can be resolved inside of the enclave with some simple logic, the remaining calls would be resolved either in a shim layer or a LibOS manner forwarded to the host OS as necessary. We believe this to be the best approach, as a custom solution could be made more performant and secure than general frameworks. However, we cannot make any promises of what the performance penalty or how many syscalls would need to be executed in the untrusted area enabling for possible attacks. To create a custom solution, either from scratch or forking an existing open source framework, would require extensive work but with great potential. This solution was not a major part of our thesis, but our research and suggestions were earlier mentioned in 4.3.1.

A risk with creating a custom solution for SGX, or by using a SGX-only framework, is the vendor lock-in to Intel - this is a limitation of our work. Currently Intel dominates the server market, their TEE solution is the market leader and other prominent solutions do not support all the critical RA features it does. However, this could change which would require porting BEAM to additional platforms if this is desired. The future is hard to predict and even SGX2 has taken years and is still not supported on consumer available server platforms, indicating that other vendors might take years to be competitive for our specific use case.

6.3 Is Erlang Fit for SGX

In this thesis we showed how it is possible to utilize Intel SGX to secure an Erlang workload. However, it is important to remember that other programming languages such as C/C++ and Rust can be executed in the enclave without immense frameworks and external syscalls. We need to remember that these applications are still limited if no syscalls are to be made, but this results in great performance and security benefits. In theory, the best way forward would be to rewrite many existing workloads and use a safe and SGX-enabled language such as Rust, however this would be detrimental for time-to-market. Additionally, the issue will always be that current codebases are not optimized for Intel SGX and therefore we can never reach the same security and performance guarantees.

Based on our research, Erlang can be executed in SGX and there are still many optimizations which can be applied. Such as a custom layer described in section 4.3 as well as to use features from Intel such as Switchless Calls and shared memory. Regardless, Erlang's BEAM is a quite complex runtime with its own scheduler, garbage collection, etc. and as such it requires many syscalls including heavy IO utilization. Additionally, it requires more syscalls than Java although this could possibly be tuned to reach similar levels.

We argue that the issue with porting legacy applications to SGX will always be a problem and the forwarding of syscalls to the outside will always come at a risk. Erlang in itself is not the problem, only the ERTS BEAM which is not currently optimized for SGX. We believe that by modifying BEAM, tuning its build options[3], and by resolving some calls locally, Erlang could be better fit for SGX and at least reach similar levels to Java JVM. However, if new applications would be written specifically for SGX the question arises if Erlang should be used. If these applications does not need to perform syscalls, a runtime not using syscalls either (e.g. Rust-SGX or C/C++) would create a much more secure solution as it could completely avoid using syscalls altogether greatly reducing

[3]Build options could be explored to reduce the number of syscalls: by increasing ETHR_YIELD_AFTER_BUSY_LOOPS or by setting '*+sbwt none +sbwtdcpu none +sbwtdio none*'. Explained here: `https://stressgrid.com/blog/beam_cpu_usage`. Additionally, more information regarding the Erlang processes can be read out with so called microstate accounting: `https://erlang.org/doc/man/msacc.html`

the attack surface towards the OS.

7 Conclusion

The goal of this thesis was to investigate how Erlang can be secured with Intel SGX as to protect a cloud-based telecommunications system. The practical work consisted of two general approaches as concluded from earlier work in the field which correspond to the two use cases presented in section 1.2. The first approach involved calling C code in Intel SGX secure enclaves from the regular Erlang BEAM runtime. The second approach utilizes frameworks (based on either shim layer or Library OS methods) which are included together with the BEAM runtime inside of SGX, to support the execution of regular Erlang binaries. Our practical work for this approach utilizes the third-party frameworks SCONE and Graphene which were tested with mixed results, additionally, custom solutions were explored but not implemented.

With the first approach it was possible from Erlang, to call SGX-secured C code through a C Node and a NIF, including the cryptographic library OpenSSL for the latter (with Intel SGX SSL). For these prototypes, the overhead was around 30 to $250\mu s$, seemingly due to context switches. The second approach however, was harder than initially anticipated due to the vast number of syscalls performed by BEAM and especially its internal scheduler. Regardless, we showed that it is indeed possible to enable support for Erlang inside of SGX, and suggested a number of ways to achieve this. Although the vital syscalls performed by BEAM are supported by Graphene, we discovered a major obstacle that Graphene cannot support MPMC pipes which are used internally by BEAM. For SCONE on the other hand, BEAM aims to be officially supported in later versions, however the experimental image tested could only support very limited Erlang functionality. Since the project, or at least the Erlang/OTP SCONE image, is proprietary it makes it harder to evaluate or develop this solution further. Regardless, by using SCONE we showed that Erlang can be supported in SGX. Finally, investigations are made on how to develop a more secure layer by creating a custom solution, which would provide benefits as compared to general third-party frameworks. However, we made no attempt at implementing a custom solution as this would require immense work including both examining the syscalls made by BEAM and then create a custom layer (by either a shim layer or Library OS approach) to resolve the necessary calls.

Each approach has its place, and we provide the following recommendation although an in-depth security evaluation is recommended before deployment. With the first approach, it is possible to call C functions in SGX from Erlang. A potential use for this approach could be to port the C libraries used by Erlang to SGX, which would then allow for some increased security while supporting legacy Erlang applications. However, more research is needed to investigate this security model, as the Erlang runtime itself and the communication between these two worlds can be attacked. Therefore, to limit these attacks, it could be beneficial to move the whole Erlang runtime inside of SGX as made possible with the second approach. However, to execute Erlang code in SGX will require syscalls and therefore more communication with the untrusted OS which will lessen the security guarantees of SGX. Regardless, secure enclaves can still provide valuable protection from a malicious system and placing Erlang, or the C workloads called from Erlang, in SGX would therefore be advantageous although the potential performance loss needs to be evaluated further.

Bibliography

[1] 2: system calls - linux man pages. https://linux.die.net/man/2/ Accessed on: 2020-05-06.

[2] 3rd Generation Partnership Project. *3GPP TS 33.127 - Technical Specification Group Services and System Aspects; Security; Lawful Interception (LI) architecture and functions (Release 16)*, 12 2019.

[3] AMD. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/ Accessed on: 2019-12-07.

[4] Jari Arkko. Service-Based Architecture in 5G, 2017. https://www.ericsson.com/en/blog/2017/9/service-based-architecture-in-5g Accessed on: 2020-01-30.

[5] Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1 – 6–26, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[9] Stefan Brenner, Tobias Hundt, Giovanni Mazzeo, and Rüdiger Kapitza. Secure cloud micro services using intel sgx. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 177–191. Springer, 2017.

[10] Gabriel Brown. Service-Based Architecture for 5G Core Networks, 2017. https://www.3g4g.co.uk/5G/5Gtech_6004_2017_11_Service-Based-Architecture-for-5G-Core-Networks_HR_Huawei.pdf Accessed on: 2020-01-30.

[11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[12] Francesco Cesarini and Simon Thompson. *Erlang Programming.* O'Reilly Media Inc., Sebastopol, California, 2009.

[13] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP.* O'Reilly Media Inc., Sebastopol, California, 2016.

[14] L Coppolino, S D'Antonio, G Mazzeo, and L Romano. A comprehensive survey of hardware-assisted security: From the edge to the cloud. *Internet of Things*, 6(100,055), 2019.

[15] Intel Corporation. *Intel® Software Guard Extensions(Intel® SGX) - Developer Guide 2.9.1*, 2020. https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Developer_Guide.pdf Accessed on: 2020-04-29.

[16] Intel Corporation. *Intel® Software Guard Extensions(Intel® SGX) Data Center Attestation Primitives: ECDSA Quote Library - DCAP 1.5*, 2020. https://download.01.org/intel-

sgx/sgx-dcap/1.5/linux/docs/Intel_SGX_ECDSA_QuoteLibReference_DCAP_API.pdf Accessed on: 2020-04-29.

[17] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016. https://eprint.iacr.org/2016/086.pdf Accessed on: 2019-12-07.

[18] Graphene Developers. Graphene Manifest Syntax - Graphene Documentation. https://graphene.readthedocs.io/en/latest/manifest-syntax.html Accessed on: 2020-05-29.

[19] Frederick W Dingledy and Alex Berrio Matamoros. *What is Digital Rights Management?* Rowman & Littlefield, 2016.

[20] Chihyun Song Dmitrii Kuvaiskii. [Pal/Linux-SGX] Add support for the DCAP SGX driver versions 1.5-, 2020. https://github.com/oscarlab/graphene/commit/b4d3cbccc621746a6dc93aa54d438b293272d0d4 Accessed on: 2020-04-20.

[21] Ericsson. A guide to 5G network security, 2018. https://www.ericsson.com/en/security/a-guide-to-5g-network-security Accessed on: 2020-04-06.

[22] Ericsson. Erlang celebrates 20 years as open source, 2018. https://www.ericsson.com/en/news/2018/5/erlang-celebrates-20-years-as-open-source Accessed on: 2019-12-07.

[23] Ericsson AB. *C Nodes*, 2019. https://erlang.org/doc/tutorial/cnode.html Accessed on: 2020-02-24.

[24] Ericsson AB. *erl_nif*, 2019. https://erlang.org/doc/man/erl_nif.html Accessed on: 2020-02-24.

[25] Ericsson AB. *Port Drivers*, 2019. https://erlang.org/doc/tutorial/c_portdriver.html Accessed on: 2020-02-24.

[26] Ericsson AB. *Ports and Port Drivers*, 2019. https://erlang.org/doc/reference_manual/ports.html Accessed on: 2020-02-24.

[27] European Union Law (EUR-Lex). *31996G1104: Council Resolution of 17 January 1995 on the lawful interception of telecommunications*, 11 1996.

[28] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and private function evaluation with intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, pages 165–181, New York, NY, USA, 2019. ACM.

[29] Fortanix. Rust EDP documentation. https://edp.fortanix.com/docs/ Accessed on: 2020-04-20.

[30] Fortanix. Fortanix: Rust-SGX, 2020. https://github.com/fortanix/rust-sgx Accessed on: 2020-04-20.

[31] Graphene. Graphene-SGX, 2020. https://github.com/oscarlab/graphene Accessed on: 2020-04-20.

[32] Graphene. Supported system calls in graphene, 2020. https://graphene.readthedocs.io/en/latest/supported-syscalls.html Accessed on: 2020-05-06.

[33] Yu-Lun Huang, Borting Chen, Ming-Wei Shih, and Chien-Yu Lai. Security impacts of virtualization on a network testbed. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 71–77. IEEE, June 2012.

[34] Intel. Attestation Service for Intel® Software Guard Extensions (Intel® SGX): API Documentation. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf Accessed on: 2020-04-20.

[35] Intel. Intel® Software Guard Extensions. https://software.intel.com/en-us/sgx Accessed on: 2019-12-07.

[36] Intel. Intel® software guard extensions sdk for linux description. https://01.org/intel-softwareguard-extensions Accessed on: 2020-04-02.

[37] Intel. Register Your Production Enclave. https://software.intel.com/en-us/sgx/request-license Accessed on: 2020-04-20.

[38] Intel. Strengthen Enclave Trust With Attestation. https://software.intel.com/en-us/sgx/attestation-services Accessed on: 2020-04-20.

[39] Intel. Intel sgx sdk release notes linux 2.2, 2018. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf Accessed on: 2020-04-02.

[40] Intel. Intel sgx sdk release notes linux 2.7, 2019. `https://download.01.org/intel-sgx/sgx-linux/2.7/docs/Intel_SGX_SDK_Release_Notes_Linux_2.7_Open_Source.pdf` Accessed on: 2020-04-02.

[41] Intel. Intel® 64 and ia-32 architectures, software developer's manual volume 3d: System programming guide, part 4, 2019. `https://software.intel.com/sites/default/files/managed/7c/f1/332831-sdm-vol-3d.pdf` Accessed on: 2020-04-02.

[42] Matthew H. (Intel). Look both ways and watch out for side-channels!, 2015. `https://software.intel.com/en-us/blogs/2015/05/19/look-both-ways-and-watch-out-for-side-channels` Accessed on: 2020-02-17.

[43] Kubilay Ahmet Küçük, David Grawrock, and Andrew Martin. Managing confidentiality leaks through private algorithms on software guard extensions (sgx) enclaves. *EURASIP Journal on Information Security*, 2019(1):14, 2019.

[44] Dmitrii Kuvaiskii. [Pal/Linux-SGX] Add exitless system calls, 2019. `https://github.com/oscarlab/graphene/commit/58c53ad747579225bf29e3506d883586ff4b8eee` Accessed on: 2020-04-20.

[45] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 557–574, 2017.

[46] Cade Metz. Why whatsapp only needs 50 engineers for its 900m users, 2015. `https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/` Accessed on: 2020-06-03.

[47] Microsoft. Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/` Accessed on: 2020-02-02.

[48] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel sgx and amd memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[49] Golem Project. Graphene v1.0 has been released!, 2019. `https://medium.com/golem-project/graphene-v1-0-has-been-released-cca5443f0887` Accessed on: 2020-04-20.

[50] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using sgx. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 187–200, New York, NY, USA, 2018. ACM.

[51] Vincent Scarlata. Innovative uses of intel software guard extensions (intel sgx), 2019. `https://www.platformsecuritysummit.com/2019/speaker/scarlata/` Accessed on: 2020-04-01.

[52] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for intel® sgx with intel® data center attestation primitives. Technical report, Intel Corporation, 2018. `https://software.intel.com/content/www/us/en/develop/download/supporting-third-party-attestation-for-intel-sgx-data-center-attestation-primitives.html` Accessed on: 2020-06-02.

[53] Scontain. SCONE - A Secure Container Environment. `SCONE-ASecureContainerEnvironment` Accessed on: 2020-04-20.

[54] Scontain. SCONE attestation (CASConfiguration). `https://sconedocs.github.io/CASConfiguration/#attestation` Accessed on: 2020-04-02.

[55] Scontain. WELCOME TO SCONE PLATFORM, 2020. `https://sconedocs.github.io/` Accessed on: 2020-04-20.

[56] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. Establishing mutually trusted channels for remote sensing devices with trusted execution environments. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–10, 08 2017.

[57] Shweta Shinde. Panoply, 2018. `https://github.com/shwetasshinde24/Panoply` Accessed on: 2020-04-20.

[58] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[59] Erlang Solutions. 20 Years of Open Source Erlang: OpenErlang Interview with Anton Lavrik from WhatsApp, 2018. `https://www.erlang-solutions.com/blog/20-years-of-`

open-source-erlang-openerlang-interview-with-anton-lavrik-from-whatsapp.html Accessed on: 2020-06-03.

[60] Erik Stenman. The erlang runtime system, 2018. https://blog.stenmans.org/theBeamBook/ Accessed on: 2020-01-29.

[61] Dave Tian, Joseph I Choi, Grant Hernandez, Patrick Traynor, and Kevin RB Butler. A practical intel sgx setting for linux containers in the cloud. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 255–266, 2019.

[62] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel sgx. pages 22–27, 10 2018.

[63] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.

[64] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2333–2350, 2019.

[65] Nico Weichbrodt, Pierre-Louis Aublin, and R. Kapitza. sgx-perf: A performance analysis tool for intel sgx enclaves. pages 201–213, 11 2018.

[66] Dr. Greg Wettstein. Sgx after spectre and meltdown: Status, analysis and remediations, 2018. ftp://ftp.idfusion.net/pub/sgx/sgx-spectre-meltdown.pdf Accessed on: 2020-03-15.

A
Run Time Measurements Basic C Node and NIF

Setup

Ubuntu 18.04, NUC7PJYH, DCAP 1.4 driver, SGX SDK 2.9.1
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10] [kernel-poll:false]
Eshell V9.2
Pre-compiled Erlang from Ubuntu repository.
GCC / G++ 7.5.0-3

Execution

SGX versions are located directly in the root of the repo (erlang-c-node, erlang-nif), while the non-SGX versions are to be found in without-sgx/.

```
erl -sname e1 -setcookie "Very Secret Cookie"
c(enclave_communicator).
enclave_communicator:run_tests().
```

This function performs a simple (time) evaluation of calling C code in Erlang with C Node and NIF. In the background timer:tc is used in Erlang to measure time.

C NODE

Equivalent functionality without any SGX use

```
cd sgx-erlang-extension/without-sgx/erlang-c-node
make SGX_DEBUG=0

Calling increment, 30 times
Average time: 538.4 microseconds
Average time excluding first element: 76.24137931034483 microseconds
Raw data: [13941,88,85,84,84,84,89,84,85,84,85,91,71,71,
        71,71,70,72,70,70,70,70,70,70,72,70,70,70,70,70]

Calling return, 30 times
Average time: 76.16666666666667 microseconds
Average time excluding first element: 75.24137931034483 microseconds
Raw data: [103,93,91,91,91,96,91,71,70,70,70,70,70,71,71,
        70,70,72,71,71,70,70,71,71,71,70,76,71,71,71]
```

Hardware mode (release)

```
cd sgx-erlang-extension/erlang-c-node
```

```
make SGX_DEBUG=0

Calling increment, 30 times
Average time: 612.5 microseconds
Average time excluding first element: 104.48275862068965 microseconds
Raw data: [15345,157,149,145,142,144,143,165,143,118,110,109,110,
        146,143,117,103,67,66,67,66,70,66,66,69,68,70,70,71,70]

Calling return, 30 times
Average time: 67.5 microseconds
Average time excluding first element: 67.27586206896552 microseconds
Raw data: [74,67,82,71,71,66,69,65,66,70,71,66,65,65,
        65,65,69,66,65,65,65,65,71,66,66,65,65,69,65,65]
```

Hardware mode (debug)

```
cd sgx-erlang-extension/erlang-c-node
make SGX_DEBUG=1

Calling increment, 30 times
Average time: 578.8666666666667 microseconds
Average time excluding first element: 112.27586206896552 microseconds
Raw data: [14110,117,104,103,102,108,102,105,199,154,147,125,137,105,
        102,102,102,106,102,102,101,119,102,102,101,102,101,101,102,101]

Calling return, 30 times
Average time: 102.56666666666666 microseconds
Average time excluding first element: 102.20689655172414 microseconds
Raw data: [113,103,108,102,101,102,102,102,101,101,102,101,102,101,101,
        102,102,102,102,101,103,102,101,101,111,103,101,101,101,102]
```

Simulation mode (release)

```
cd sgx-erlang-extension/erlang-c-node
make SGX_MODE=SIM SGX_DEBUG=0

Calling increment, 30 times
Average time: 564.9333333333333 microseconds
Average time excluding first element: 111.13793103448276 microseconds
Raw data: [13725,129,124,124,122,123,122,122,123,123,109,109,101,133,130,
        109,101,104,101,101,101,101,101,101,101,101,102,103,101,101]

Calling return, 30 times
Average time: 101.83333333333333 microseconds
Average time excluding first element: 101.65517241379311 microseconds
Raw data: [107,103,102,101,101,101,101,103,101,101,101,101,101,101,101,
        101,115,101,101,100,101,102,101,101,101,101,101,101,101,101]
```

NIF

Equivalent functionality without any SGX use

```
cd sgx-erlang-extension/without-sgx/erlang-nif
make SGX_DEBUG=0

Calling increment, 30 times
```

```
Average time: 0.13333333333333333 microseconds
Average time excluding first element: 0.0 microseconds
Raw data: [4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

Calling rtrn, 30 times
Average time: 0.06666666666666667 microseconds
Average time excluding first element: 0.0 microseconds
Raw data: [2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Hardware mode (release)

```
cd sgx-erlang-extension/erlang-nif
make SGX_DEBUG=0

Calling increment, 30 times
Average time: 31.5 microseconds
Average time excluding first element: 28.06896551724138 microseconds
Raw data: [131,29,28,28,28,28,28,26,27,28,28,28,28,28,28,28,28,28,28,28,28,30,
           28,28,28,30,28,28,28,28]

Calling rtrn, 30 times
Average time: 28.0 microseconds
Average time excluding first element: 27.862068965517242 microseconds
Raw data: [32,28,28,28,28,27,27,29,28,27,27,28,28,28,27,28,28,28,28,28,28,28,
           28,30,27,27,28,28,28,28]
```

Hardware mode (debug)

```
cd sgx-erlang-extension/erlang-nif
make SGX_DEBUG=1

Calling increment, 30 times
Average time: 30.866666666666667 microseconds
Average time excluding first element: 27.75862068965517 microseconds
Raw data: [121,29,28,28,28,27,28,28,27,28,27,27,28,28,28,27,28,28,27,28,28,29,
           28,27,27,28,28,28,28,27]

Calling rtrn, 30 times
Average time: 27.8 microseconds
Average time excluding first element: 27.655172413793103 microseconds
Raw data: [32,28,28,28,27,27,27,27,27,27,28,28,27,28,27,28,28,28,28,28,28,28,
           28,28,28,27,28,28,27,28]
```

Simulation mode (release)

```
cd sgx-erlang-extension/erlang-nif
make SGX_MODE=SIM SGX_DEBUG=0

Calling increment, 30 times
Average time: 22.433333333333334 microseconds
Average time excluding first element: 19.689655172413794 microseconds
Raw data: [102,18,17,17,17,17,16,16,17,17,17,17,16,16,100,18,17,16,17,17,16,
           19,17,16,17,16,18,16,16,17]

Calling rtrn, 30 times
Average time: 16.633333333333333 microseconds
Average time excluding first element: 16.482758620689655 microseconds
Raw data: [21,17,16,16,17,16,16,17,17,18,16,16,16,16,17,16,17,16,16,17,16,17,
           16,18,16,16,17,17,16,16]
```

B

Run Time Measurements SSL NIF

Setup

Ubuntu 18.04, NUC7PJYH, DCAP 1.5 driver, SGX SDK 2.9.1
Erlang/OTP 23 [erts-11.0] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1]
Eshell V11.0
Erlang compiled from source without HiPE
GCC / G++ 10.1.0-2

Execution

Both SGX and non-SGX versions are located in erlang-nif_SSL, this includes old increment functionality as well as some new RSA functions performed in OpenSSL 1.1.1d.

```
erl
c(enclave_communicator).
enclave_communicator:eval_rsa().
enclave_communicator:eval_increment().
```

These functions perform simple (time) evaluations for calling Erlang NIF functions. In the background timer:tc is used in Erlang to measure time.

HW Release Mode

```
make sgx SGX_DEBUG=0 SGX_MODE=HW
```

eval_rsa

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1349624.2666666666 microseconds
Average time excluding first element: 1389217.2413793104 microseconds
Raw data: [201428,710525,1329479,1212152,1521520,1229589,426689,1111910,
           2531939,1262472,1320354,2756141,3195494,1187874,1705678,2048262,
           1622328,1320962,1317989,2733033,611052,92785,2238959,1213102,
           510368,335672,1546954,1170406,1204547,819065]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1306003.0 microseconds
Average time excluding first element: 1323085.551724138 microseconds
Raw data: [810609,1354884,150391,3993619,1353936,576288,2429567,1487026,
           910381,2898635,818858,1645158,1795175,151063,843480,526788,2496977,
           1504991,567941,318076,2499582,1060961,1388320,877149,1838692,
           869747,1370621,902044,702278,1036853]
```

```
Calling rsa_sign, 30 times
Average time: 14137.266666666666 microseconds
Average time excluding first element: 14033.896551724138 microseconds
Raw data: [17135,14066,14094,14021,14011,14020,14010,14117,14010,14020,14010,
           14066,14124,14021,14009,14024,14007,14019,14009,14021,14011,14111,
           14010,14020,14009,14020,14009,14019,14075,14020]

Calling rsa_encrypt, 30 times
Average time: 373.3666666666667 microseconds
Average time excluding first element: 371.55172413793105 microseconds
Raw data: [426,372,370,370,372,384,370,371,369,370,373,369,372,371,369,369,
           380,369,368,370,370,373,370,370,369,370,369,384,371,371]
```

eval_rsa - 2nd run for comparison

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1535870.4666666666 microseconds
Average time excluding first element: 1518466.2758620689 microseconds
Raw data: [2040592,735562,685916,2308531,3744627,1772549,1003087,
            393524,2439300,802722,3131870,510473,3869326,4702860,
            845700,193362,1254721,1386996,769186,686550,1245634,
            1204138,987392,752503,1270258,259779,1821499,618760,2323616,2315081]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1266289.8666666667 microseconds
Average time excluding first element: 1215949.896551724 microseconds
Raw data: [2726149,1085886,1195178,1019812,1169233,693907,660521,
            1511599,4202745,1619789,334555,334232,1870819,1161935,
            635276,1119226,1027553,2054949,1244194,2230131,1436832,
            226257,2285472,935589,159021,92400,1996572,1337872,919178,701814]

Calling rsa_sign, 30 times
Average time: 14137.5 microseconds
Average time excluding first element: 14035.758620689656 microseconds
Raw data: [17088,14209,14156,14058,14007,14017,14007,14128,
            14007,14016,14006,14018,14007,14017,14006,14016,14006,
            14106,14007,14019,14010,14006,14016,14006,14018,14119,
            14018,14006,14017,14008]

Calling rsa_encrypt, 30 times
Average time: 376.2 microseconds
Average time excluding first element: 372.62068965517244 microseconds
Raw data: [480,390,384,369,369,371,370,371,370,370,373,372,369,
            381,371,372,372,372,371,371,370,372,373,376,371,
            371,369,373,372,371]
```

eval_increment

```
Calling increment, 30 times
Average time: 76.26666666666667 microseconds
Average time excluding first element: 59.44827586206897 microseconds
Raw data: [564,60,60,59,60,59,59,59,59,60,59,59,59,60,
            59,60,59,59,59,59,62,60,59,59,59,59,60,60,60,59]
```

```
Calling rtrn, 30 times
Average time: 60.56666666666667 microseconds
Average time excluding first element: 60.44827586206897 microseconds
Raw data: [64,60,60,59,59,59,59,59,60,59,59,59,59,59,
           91,59,59,59,59,59,59,59,59,60,60,60,60,60,60,60]
```

eval_increment - 2nd run for comparison

```
Calling increment, 30 times
Average time: 65.0 microseconds
Average time excluding first element: 61.55172413793103 microseconds
Raw data: [165,60,60,73,113,59,59,59,59,59,59,59,59,59,59,
           59,59,60,59,59,59,59,59,59,59,60,60,59,59,60]

Calling rtrn, 30 times
Average time: 60.5 microseconds
Average time excluding first element: 60.310344827586206 microseconds
Raw data: [66,59,60,60,59,59,59,59,59,59,59,59,59,59,59,59,
           59,59,59,60,60,90,60,59,60,59,60,59,59,59]
```

Simulated Release Mode

```
make sgx SGX_DEBUG=0 SGX_MODE=SIM
```

eval_rsa

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1604126.6 microseconds
Average time excluding first element: 1594696.2068965517 microseconds
Raw data: [1877608,157455,1794176,2078710,363619,364691,901833,
           950866,3378818,2430897,575533,809807,860904,1495821,
           1512164,644843,2296674,1297911,6678954,2287462,2620055,
           1370473,752419,983299,2837715,909766,1562568,2488515,848487,991755]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1568162.8666666667 microseconds
Average time excluding first element: 1596015.6551724137 microseconds
Raw data: [760432,1444788,2163204,3130905,1239269,1463464,
           1619029,2773439,1585385,3615827,1412065,1642588,
           645041,3815667,561359,446065,222694,1988400,759810,
           1164147,710071,1429162,991400,2236859,2682164,966256,
           1982170,355189,1411992,1826045]

Calling rsa_sign, 30 times
Average time: 13938.8 microseconds
Average time excluding first element: 13836.068965517241 microseconds
Raw data: [16918,13866,13861,13983,13818,13823,13818,13825,
           13818,13818,13825,13818,13823,13818,13822,13817,
           13824,13818,13867,13819,13819,13824,13818,13825,
           13818,13823,13819,13938,13818,13863]

Calling rsa_encrypt, 30 times
Average time: 367.1666666666667 microseconds
Average time excluding first element: 363.44827586206895 microseconds
Raw data: [475,365,362,361,364,363,361,362,370,363,365,363,
           364,362,362,362,364,363,362,371,363,366,362,
           362,363,361,362,365,365,362]
```

eval_rsa - 2nd run for comparison

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1317200.1666666667 microseconds
Average time excluding first element: 1353760.7931034483 microseconds
Raw data: [256942,776776,677506,789187,1008455,2567675,1891370,
           1173047,216239,2346717,949700,991435,1792516,826267,
           1337456,504754,1040253,1397231,764368,1882935,2757453,
           2660051,677823,2156786,198882,2578817,2295692,
           999238,901122,1099312]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1608280.6333333333 microseconds
Average time excluding first element: 1580508.0344827587 microseconds
Raw data: [2413686,1033028,2747680,1396000,173831,611257,1990077,
           1874887,1718523,2707728,1288109,421440,1263411,
           2419685,2153043,894208,1714094,1065674,1015189,
           2363838,231616,3331024,652955,1024361,1463516,
           986123,5064900,1503018,1858271,867247]

Calling rsa_sign, 30 times
Average time: 14020.466666666667 microseconds
Average time excluding first element: 13923.931034482759 microseconds
Raw data: [16820,13832,13869,13831,13827,13925,13827,
           13936,13827,14119,13831,13866,13871,13827,13832,
           13826,13833,14107,13837,13865,13921,13828,13831,
           14138,14351,14249,13832,13827,13831,14298]

Calling rsa_encrypt, 30 times
Average time: 366.56666666666666 microseconds
Average time excluding first element: 363.86206896551727 microseconds
Raw data: [445,389,363,362,360,365,361,363,361,361,362,363,
           368,365,361,363,363,363,364,361,364,361,362,
           373,363,363,362,362,362,362]
```

eval_increment

```
Calling increment, 30 times
Average time: 47.4 microseconds
Average time excluding first element: 45.10344827586207 microseconds
Raw data: [114,47,46,44,45,45,45,45,45,45,45,45,
           45,45,45,45,45,45,45,45,45,46,45,45,
           45,45,45,45,45,45]

Calling rtrn, 30 times
Average time: 47.7 microseconds
Average time excluding first element: 47.62068965517241 microseconds
Raw data: [50,46,45,45,45,118,46,45,45,45,45,45,
           45,45,45,45,45,45,45,45,45,45,45,45,
            45,45,45,45,46,45]
```

eval_increment - 2nd run for comparison

```
Calling increment, 30 times
Average time: 47.266666666666666 microseconds
Average time excluding first element: 44.93103448275862 microseconds
Raw data: [115,45,45,45,45,44,45,45,45,45,45,45,45,45,
            45,45,45,45,45,45,45,45,44,45,45,45,45,45,45,45]

Calling rtrn, 30 times
Average time: 46.733333333333334 microseconds
Average time excluding first element: 46.55172413793103 microseconds
Raw data: [52,46,46,47,45,85,46,45,45,45,45,45,45,45,
            45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45]
```

No SGX

```
make no-sgx SGX_DEBUG=0
```

eval_rsa

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1317428.3 microseconds
Average time excluding first element: 1319427.448275862 microseconds
Raw data: [1259453,739225,828763,179403,2799639,755797,2092528,
            966774,244620,1915815,2646888,876961,1769421,1323146,
            852805,3326962,884995,1031088,901734,625799,1129793,
            1347458,876946,2059292,1696289,1517870,2005642,812400,1137832,917511]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1292506.4 microseconds
Average time excluding first element: 1287413.6896551724 microseconds
Raw data: [1440195,1388633,1071691,1648425,324967,999555,105894,
            803759,81732,1087840,2077240,3312600,861716,1201191,
            365716,2588802,1713756,2914882,1655547,730733,105991,
            2060986,1160684,2661524,641192,1297437,771691,1339033,1517258,844522]

Calling rsa_sign, 30 times
Average time: 13888.033333333333 microseconds
Average time excluding first element: 13796.379310344828 microseconds
Raw data: [16546,13793,13810,13792,13785,13830,13787,13784,13790,
            13786,13790,13785,13790,13785,13898,13786,13784,13792,13786,
            13789,13829,13790,13785,13790,13786,13790,13785,13786,13789,13823]

Calling rsa_encrypt, 30 times
Average time: 228.0 microseconds
Average time excluding first element: 225.6206896551724 microseconds
Raw data: [297,231,223,219,233,223,225,227,237,227,227,225,
            223,226,223,228,224,231,230,220,233,221,221,223,
            226,227,225,221,219,225]
```

eval_rsa - 2nd run for comparison

```
Running RSA keygen, sign, and encrypt together
Calling rsa_basic_operations, 30 times
Average time: 1513691.4333333333 microseconds
Average time excluding first element: 1533443.7586206896 microseconds
Raw data: [940874,625616,1087950,398365,1128800,747870,
           2459919,4231213,739626,5294203,601548,991249,
           1705166,1590873,649895,884926,2012816,1128373,
           1250649,414814,1640043,560785,2589272,1103892,
           584868,1023207,2986426,2385394,1704265,1947846]

Running RSA these functions separately
Calling rsa_key_gen, 30 times
Average time: 1329274.1333333333 microseconds
Average time excluding first element: 1309351.2413793104 microseconds
Raw data: [1907038,1079442,1103932,414072,1225231,933053,
           1169128,2474677,1858674,3384006,1143906,1175951,
           438309,2409540,495116,1022524,511557,1047545,
           389711,1077948,2378005,916990,1841772,1428910,
           1347865,1980021,2109508,1573849,633718,406226]

Calling rsa_sign, 30 times
Average time: 13890.0 microseconds
Average time excluding first element: 13798.551724137931 microseconds
Raw data: [16542,13859,13796,13790,13794,13790,13790,13797,
           13790,13794,13832,13794,13797,13795,13790,13796,
           13790,13828,13795,13789,13795,13789,13794,13789,
           13833,13790,13789,13791,13788,13794]

Calling rsa_encrypt, 30 times
Average time: 225.63333333333333 microseconds
Average time excluding first element: 224.17241379310346 microseconds
Raw data: [268,224,224,224,225,221,226,223,227,230,228,221,
           218,223,226,227,218,221,223,230,220,223,
           223,222,223,221,223,240,225,222]
```

eval_increment

```
Calling increment, 30 times
Average time: 0.03333333333333333 microseconds
Average time excluding first element: 0.0 microseconds
Raw data: [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

Calling rtrn, 30 times
Average time: 0.03333333333333333 microseconds
Average time excluding first element: 0.0 microseconds
Raw data: [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

eval_increment - 2nd run for comparison

```
Calling increment, 30 times
Average time: 0.13333333333333333 microseconds
Average time excluding first element: 0.0 microseconds
Raw data: [4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
           0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
Calling rtrn, 30 times
Average time: 0.13333333333333333 microseconds
Average time excluding first element: 0.06896551724137931 microseconds
Raw data: [2,0,0,2,0,0,0,0,0,0,0,0,0,0,0
            ,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```