Improving Security of Edge Devices by Offloading Computations to Remote, Trusted Execution Environments

Carlos Bilbao Muñoz

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Angelos Stavrou
Ruslan Nikolaev

December 14, 2021
Blacksburg, Virginia

Keywords: Edge computing, ISA-heterogeneous computing, Systems security

Improving Security of Edge Devices by Offloading Computations to Remote, Trusted Execution Environments

Carlos Bilbao Muñoz

(ABSTRACT)

In this thesis we aim to push forward the state-of-the-art security on instruction set architecture (ISA) heterogeneous systems by adopting an edge-computing approach. As the embedded devices market grows, such systems remain affected by a wide range of attacks and are particularly vulnerable to techniques that render the operating system or hypervisor untrusted. The usage of Trusted Execution Environments (TEEs) can help mitigate such threat model(s) immensely, but embedded devices rarely count with the hardware support required. To address this situation and enhance security on embedded devices, we present the RemoteTrust framework, which allows modest devices to offload secure computations on a remote server with hardware-level TEEs. To ease portability, we develop the framework on top of the open-source hardware-agnostic Open Enclave SDK. We evaluate the framework from a security and performance perspectives on a realistic infrastructure. In terms of security, we provide a list of CVEs that could potentially be mitigated by RemoteTrust, and we prevent the Heartbleed attack on a vulnerable server. From a performance perspective, we port C/C++ benchmarks of SPEC CPU 2017, two overhead microbenchmarks and five open-source applications, demonstrating small communication overhead (averaging less than 1 second per 100 remote single-parameter enclave calls).

Improving Security of Edge Devices by Offloading Computations to Remote, Trusted Execution Environments

Carlos Bilbao Muñoz

(GENERAL AUDIENCE ABSTRACT)

We develop software that can be leveraged to secure an embedded device (reduced-size computer) using features only present in more powerful systems such as a server. This requires developing and extending source code for ISA-heterogeneous (different instruction sets) systems. Our thesis is then evaluated on a realistic setup, using the type of device (a Raspberry Pi v4) that the framework is intended for. We demonstrate our framework can help secure devices without paying a high price in performance.

Dedication

I dedicate this work to my parents.

Acknowledgments

This work would not have been possible without the support of the Systems Software Research Group (SSRG), Dr. Binoy Ravindran (Virginia Tech) and Dr. Xiaoguang Wang (Virginia Tech).

I would also like to thanks the members of my Master's committee, Dr. Ruslan Nikolaev and Dr. Angelos Stavrou.

This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-18-1-2022 and N00014-19-1-2493.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There exists an increasing interest in the systems community to combine ISA-heterogeneous architectures. This is done in an attempt to leverage the best of both worlds: The power efficiency and portability provided by embedded devices, and the heavy computational capabilities of server mainframes.

Unfortunately, this is extraordinarily challenging in practice, particularly from a security and performance perspectives. In terms of security, embedded devices are specially vulnerable to attacks that render the operating system (or hypervisor) untrusted, due to their lack of hardware-based security features. In particular, whenever an attacker infects the system to the point where we cannot trust the operating system itself, using Trusted Execution Environments (TEEs) can help protect code and data from everyone else. From a performance perspective, leveraging the combined efforts of ISA-heterogeneous systems requires taking into account the communication overhead between them, as well as profiling the workloads to determine what architecture(s) is better suited to carry out certain workloads. In simpler terms, even though certain tasks might finish their execution sooner if we combine two or more architectures, the time required to synchronize the machines (usually, in terms of memory coherency) might make the total execution time even greater than when running in a single system.

In this thesis we intend to address security on ISA-heterogeneous systems without discarding performance from the equation. In particular, we include extensions to the hardware-agnostic

open-source project Open Enclave [2] that abstracts the creation of Trusted Execution Environments (TEEs), with the goal of offloading security enclave operations from ARM embedded devices into a remote x86 server in an edge-computing approach. The extensions to the Open Enclave SDK and the software infrastructure that makes them possible are merged into the RemoteTrust framework. The design, implementation and evaluation of each of the components that shape RemoteTrust are covered in this thesis.

1.1 Motivation

It is a well-known fact that embedded devices are increasingly more part of our everyday life, with solutions ranging from self-driving cars to voice recognition integrated devices. Market reports consistently project tremendous growth of the global embedded system market [15], fueling the consensus in the systems community that the embedded trend will only accelerate. Other key factors that evidence the incoming growth of embedded devices are the arrival of the 5G technology [3] that eases the rise of the Internet of Things (IoT), the increase in demand for Cyber-Physical Systems (CPS) such as wearable devices [19] and the current industry efforts on electric vehicles with driver-assisted systems [29].

Considering this, it is no surprise that embedded devices are an increasingly popular target for attackers [14, 20]. However, the manufacturers of these systems rarely focus on security, since other factors such as reduced electric consumption take a protagonist role. These power constraints affect all layers of the design, effectively limiting the computational capabilities and hardware security features provided, rendering the devices more vulnerable [13, 17, 22]. Certain security threats require reducing the Trusting Computing Base (TCB) to the point where not even the operating system or hypervisor can be trusted. In order to address them, chip manufacturer have included in the recent years trusted execution environments

(TEEs) [23] that generate safe computing zones where all the memory and instructions of specific threads are encrypted and isolated from the rest of the software stack. As of today, the most prominent examples are Intel Software Guard Extensions (SGX) [16] and ARM TrustZone [4]. Other companies have also tackled hardware data encryption with the needs of Cloud in mind, as it is the case of IBM Z's secure enclaves or AMD EPYC-based confidential computing. These features can be enormously useful for security purposes and are being increasingly leveraged by systems researchers [5, 10, 24, 27].

Of course, server solutions such as Intel SGX are not suitable for embedded devices, even though they can potentially be victims of the same attacks these features aim to prevent. For this reason, in this thesis we provide an edge-computing framework with which embedded devices can offload secure operations on a remote server that will have access to these hardware features, in a Security as a service (SECaaS) approach. In particular, relying on enclaves to protect the sensitive operations on embedded devices protects the systems against the widespread memory corruption vulnerabilities [26] rooted in the absence of built-in memory safety on the popular system programming languages C and C++.

In addition, the integration of enclave features should be taken care of without hardening the work of developers that rely on these trusted environments for their software. For this reason, we leverage the open-source hardware-agnostic library Open Enclave that is widely used by the security community to abstract the enclave creation, manipulation and attestation. We do not use PCIe to communicate client-server as that would be hardly realistic on an industry implementation of the framework.

1.2 Thesis Contributions

The RemoteTrust framework is our response to the challenges outlined on the previous section 1.1, for which we make the following contributions, covered in this thesis:

1. We design RemoteTrust, an edge-computing framework that generates an SSL/TLS server-client infrastructure for embedded devices to offload hardware-based trusted execution environments such as Intel SGX. This process is done transparently since it does not require source code modifications.

2. In order to ease the implementation for developers, we extend the popular hardware-agnostic open-source Open Enclave SDK. All intermediate code is autogenerated and scripts are provided to ease the usage of the framework. A user-space distributed shared memory (DSM) is provided to allow synchronization between server and client using the userfaultfd system call.

3. We overcome the challenges of compiling the necessary tools from the Open Enclave SDK on the ARM architecture, which is currently not officially supported. This requires adapting compilation scripts, Makefiles, and providing workarounds for the missing symbols and errors or warnings produced by the different tools and shared libraries.

4. We evaluate RemoteTrust from a security and a performance perspective. We provide an array of CVEs that RemoteTrust could help mitigate, and we prevent the Heartbleed attack (CVE-2014-0160) by securing an SSL server with our framework. From a performance standpoint, we port the C/C++ applications from SPEC CPU 2017, as well as two overhead microbenchmarks and five applications of the public domain: Three use cases provided as samples of the Open Enclave SDK, an open-source ar-

tificial intelligence-based virtual assistant for Linux [25] and OpenFABMAP [11], a visual place recognition application for robotics automation. The evaluation of these applications on an ISA-heterogeneous setup with a Raspberry Pi shows that the communication overhead falls within acceptable boundaries.

The RemoteTrust framework combined with the extensions to the Open Enclave project make a total of approximately 4,000 lines of code.

1.3 Thesis Organization

The remaining of this thesis is organized as follows: Chapter 2 provides the required technical background for the reader. Chapter 3 covers the design of the RemoteTrust framework, covering its main abstractions. Chapter 4 gets into implementation details of the source code. Chapter 5 discusses the evaluation of the framework from a security and performance perspectives. Finally, Chapter 6 concludes with insights extracted from the work.

Chapter 2

Background

In this chapter we provide the necessary technical background required to understand this thesis in its entirety. This includes an introductory discussion on the hardware-level Trusted Execution Environments (TEEs), the Intel SGX (Software Guard Extension) technology and the Open Enclave software development kit (SDK) that we used to build the framework RemoteTrust.

2.1 Background on TEEs

Trusted Execution Environments (TEEs) provide a high level of isolation and security guarantees to applications that rely on them to execute sensitive operations. The reduction of the attack surface inversely correlates with the likelihood that a vulnerability in the system will affect their software, since the executed code -known as the Trusted Application TA- is isolated via hardware with cryptographic keys.

Trusted Applications are explicitly divided into two components, a host and an enclave. The enclave's code and data are protected from third-party reads or modifications. If the host wants to access enclave data, it will require explicit authorization from the enclave, that exposes a list of functions that can be called from the outside (ecalls). Similarly, the host side of the application has a list of functions that can be called from the enclave (ocalls). Furthermore, implementations of TEEs like Intel SGX use this structure to offer

other services such as attestation over the software running in the enclave.

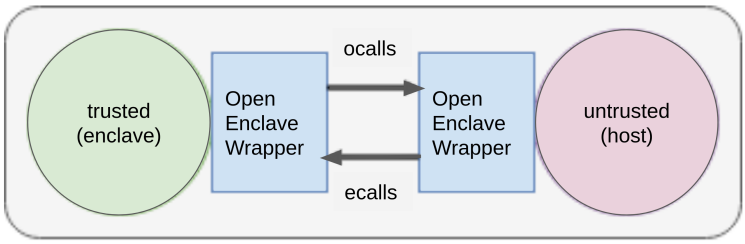

Figure 2.1: Basic flow of secure application using Open Enclave to abstract a Trusted Execution Environment. Enclave calls (ecalls) get directed to the trusted side, and host calls (ocalls) to the untrusted side, known as the host, at the right side of the diagram.

However, the wide range of enclave technologies and their integration complexity make the work of security developers more challenging. This is the motivation behind the Open Enclave SDK [2], an open-source project that provides an API surface agnostic to particular enclave technologies. Open Enclave quickly won traction thanks to their enclave generalization ability, among other perks such as its standardization capabilities (removing hardware specific requirements for verification and signing) or their multi-platform compatibility potential.

The Open Enclave library wraps the enclave and host external calls, as shown in Figure 2.1, to minimize the hardware/software specific concepts that the developer needs to be concerned about. This marshaling code is automatically generated by the tool `Oeedger8r` [21], which is based on Intel's SGX SDK edger8r tool. The most important concept of this technology are the `edl` files, used for the definition of special functions that can be accessed from the outside, both for the secure and unsecure sides. Figure 4.2 illustrates the basic format of edl files, where the methods for both host and enclave are defined, along with their parameter constraints.

```
1 enclave {
2  trusted {
3     public oe_result_t enclave_foo(void);
4  };
5  untrusted {
6     oe_result_t host_bar([user_check]char *str);
7  };
8 };
```

Figure 2.2: Example of the .edl file format

2.2 Background on Intel SGX

Intel SGX is the most prominent example of a TEE implementation and the focus of the Open Enclave project, even if the platform can potentially be used for other hardware-level security technologies. Since we use Intel SGX to test our framework, it is appropriate for the reader to be familiarized with its main abstractions.

Intel SGX was first introduced in some Intel processors back in 2015 with the Skylake micro-architecture. Using micro-code -slightly more abstract than raw assembly- they introduced 18 new instructions (see Table 2.1), 5 for external users (as for example to request to enter the enclave) and 13 for the supervisor (once inside).

At the same time, diverse structures were included to handle the secured pages information, the enclave signatures, the thread states, etc. From all of the additions of SGX, the most important one likely is the Enclave Page Cache (EPC), where the enclave code and data are placed and encrypted using a dedicated chip, the Memory Encryption Engine (MEE). It is interesting to notice that the size of the EPC is limited by the size of the EPCM, which is set at BIOS with a maximum value of 128 MB. Noticeably, this is the first case of real-world cryptographic memory protection added on commodity (general-purpose) processors [12].

Memory management is particularly important on Intel SGX. The traditional process for fetching pages is extended to secure the EPCs. In particular, once the linear address is

Table 2.1: Most important SGX instructions

Instruction	Issuer	Description
EADD	Supervisor	Add a secure EPC page
EBLOCK	Supervisor	Block an EPC page
ECREATE	Supervisor	Create the enclave
EENTER	External	Enter the enclave
EEXIT	External	Exit the enclave
ERESUME	External	Enter again on the enclave

converted to physical address using the standard page walk, additional security checks must be conducted. If it is not an access to the enclave, the address should not be in the EPC, and -more importantly- if it is an enclave access but the address is not on the EPC, or the Enclave Page Cache Map (EPCM) doesn't show the appropriate permissions, a signal fault will be generated.

2.3 Background on the Open Enclave SDK

As can be seen, the internals of Intel SGX can easily get complicated, and this is not different on other TEE implementations. At the end of the day, these are hardware-level solutions and their usage (at the source code level) was not conceived for the general audience. Many enclave operations require tuning parameters, or adjusting configuration variables. This is complex and time consuming, even for cybersecurity experts. For this reason, the community started the Open Enclave SDK project [2], an open-source hardware-agnostic effort that eases things for developers.

The Open Enclave SDK takes care of many TEE details "under the hood" and abstracts complicated operations. However, possibly more important than that is the potential ability of Open Enclave to work with different TEEs. As of today, most of the support is centered on Intel SGX, but the design of Open Enclave was, from its very beginning, focused on

developing a framework that could be potentially used with other implementations of Trusted Execution Environments. This is very promising because developers would not have to worry about porting their secured applications between TEE implementations as in from Intel SGX to Arm TrustZone, to name an example.

Chapter 3

Design of RemoteTrust

The main design challenges that the framework had to address are as follows:

1. RemoteTrust needs to execute the Open Enclave binary in both client and server. The communication between these two parties must be secure, and transparent from the execution of the enclave or host operations. This challenge is covered in the following subsection 3.1.

2. Even though the binary is running on the client side too, the enclave functions should never be executed there; nor can they, if their architecture does not support the hardware environment. Both in the server and in the client, the sensitive function calls must be wrapped and redirected to the other side.

   Whenever the client wants to execute an enclave operation this will have to be redirected to the server side as a function call request, as explained in section 3.2. The server will have to handle this request and communicate with the Open Enclave binary running on the server side to request execution and share function parameters.

3. Since some function calls include pointers, the heap memory of the binaries running on server and client must be synchronized. This has to be done from user space. In order to provide this memory guarantee covered in subsection 3.3, as well as the function requests, server and client will exchange and handle messages, whose design is discussed in subsection 3.4.

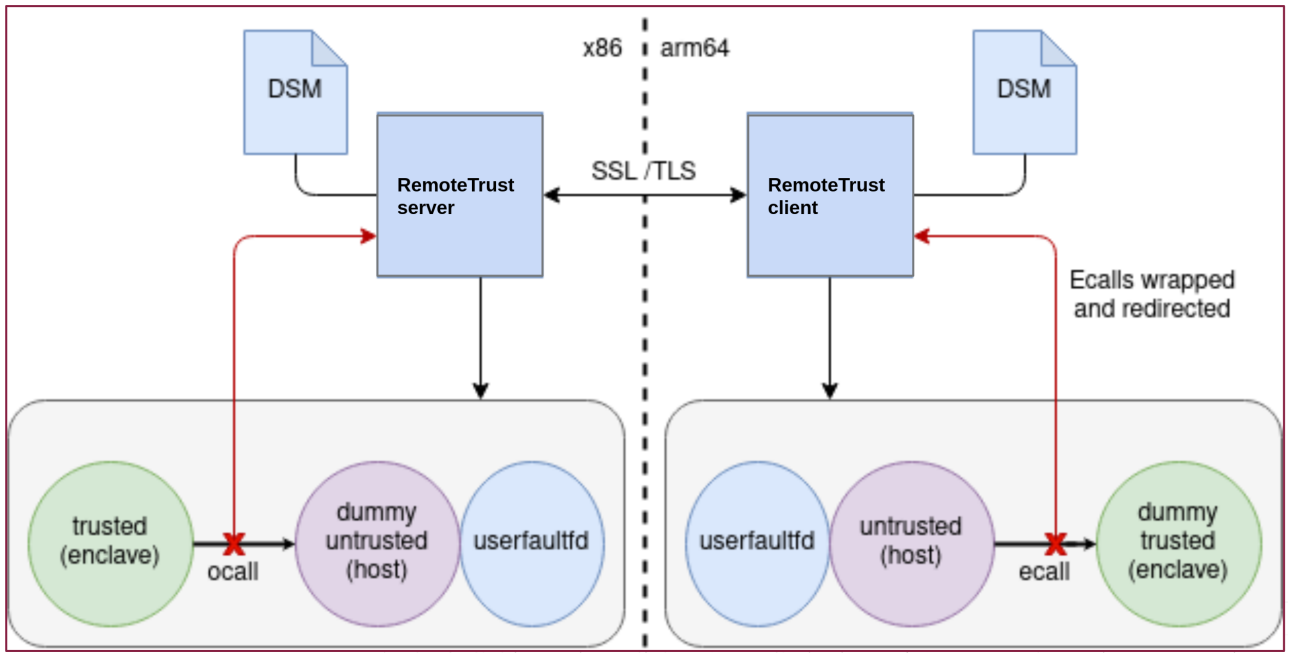

Figure 3.1: Complete design of the RemoteTrust infrastructure, with an x86 server on the left side and an arm64 client on the right. Both RemoteTrust sides have a forked child that contains an userfaultfd handler and the two sides of a trusted execution environment (host and enclave). On the client side, enclave calls (ecalls) to the dummy enclave are redirected using extensions on the open-source hardware-agnostic Open Enclave project. Whenever the userfaultfd handler receives a page fault, it relies on RemoteTrust to retrieve page contents.

3.1 Enclave execution on client side

RemoteTrust is conceived as an SSL/TLS server-client infrastructure (Figure 3.1) with synchronized heap pages of the secured binaries. Whenever the client -presumably an embedded device- needs to execute an enclave call, it will be wrapped -through extensions on the Open Enclave library- and a function call request will be sent to the server side. On both sides of the communication, the binary *RemoteTrust* is executed to establish a secure communication and send or receive the messages required for the heterogeneous execution of the enclave operations. This binary, no matter if it is on the client or in the server, will have a child forked to execute the secure program that uses Open Enclave. In the server, this child will be a dummy host (wrapper) and the real enclave, as seen on the left side of Figure 3.1. In the client, the child will be the real host, and the enclave just a wrapper, as in the right side of Figure 3.1.

In both cases, parent and child communicate via user-defined signals and shared sockets. In addition, both children request help to their parents via user-defined signals [1]: They ask them for enclave functions when they cannot execute them. In the case of the server, these will be calls to the host, and in the client these will be enclave calls wrapped with the marshaling tool Oeedger8r [21]. They also send their parents their heap information to allow DSM between the sides; since function calls may include pointers. Finally, they ask them for pages when they page fault. The children register userfaultfd handlers, and their parents synchronize and share page contents.

[1] To avoid third-party intrusions, inside the user-defined signal handler the parent makes sure that the sender of the signal was the child.

3.2 Wrapping of enclave projects

Whenever the untrusted side calls an enclave function (this is, the children of the client side, that `execve` to the provided binary) it will be wrapped using the Oeedger8r library, part of the Open Enclave SDK effort. In normal conditions, this wrapper will just ask the enclave for the execution of that particular function, but in our case further steps are required, since the enclave is actually on the server's side.

The enclave function wrapper is instead a request to the parent, via signals, of the execution of that specific function with certain parameters. This is illustrated on Figure 3.1 with the red arrows. The client side sends a function execution request to the server via SSL/TLS. The server receives the request, acknowledges it back, and sends a custom signal to its children (that `execve` to the binary). The children will handle the signal by calling the real enclave function. This is accomplished by modifying the Oeedger8r library, that contains an array of function pointers `pop_calls` to the enclave, automatically generated. It is worth noticing that all the modifications inside the Open Enclave library are in fact autogenerated, taking into account the name of the provided project and the functions it implements.

3.3 Memory synchronization component

Enclave and host function calls contain value and pointer arguments. The values can be easily copied and sent to the other side, the client if it is a host function or server if it is an enclave call. However, arguments passed by reference are more tricky, since the other side does not have access to the same virtual memory regions. Consequently, RemoteTrust also needs to synchronize the heaps on both sides.

For this purpose, the RemoteTrust library -called inside the children binaries- creates a

thread in charge of handling heap page faults using userfaultfd. Whenever the binary has a page fault in the heap region, the handling is not done in the kernel but instead in the binary itself. When that happens, the binary will ask the parent, that will either be server or client, for the missing page. The parent process is in charge of retrieving that page and imposing memory synchronization. If the page is in a valid state, it is enough to share it with the child via shared socket. If however, the page is in an invalid state according to the parent's DSM logic, it will have to request this data to the other side of the communication.

3.4 Message sharing component

The main components of RemoteTrust can be covered by enumerating the type of messages that the server or client might need to handle throughout their execution:

Memory-synchronization Messages to establish a DSM communication between the heaps of the secure children running on both sides. This is required to handle the function call parameters that are passed by reference, as the binary on the other side has a different virtual address space. Hence, the heap pages on both sides need to be managed, shared on page faults (`PAGE_REPLY`) and invalidated if necessary (`INVALIDATE`).

RemoteTrust relies on userfaultfd for this purpose - see Figure 3.1-, and page contents are shared between the framework and the child with a shared socket opened before the fork. Noticeably, the enclave binary does not select the page contents or performs pages (in)validation, as that would be a security flaw, but relies on RemoteTrust for this purpose.

Function call requests The client will wrap enclave calls and request the server to execute them with messages of type `CALL_REQUEST`. These requests will be handled on the other

side by signaling the child with user-defined signals and sharing with it the function names and parameters using shared files.

Special remote requests Two special message are reserved to terminate the enclave execution remotely and share the untrusted binary with the client side, `TERMINATE_ENCLAVE` and `SEND_UBIN`, respectively.

All these messages have their corresponding ACK counterpart allowing the other side can confirm reception.

Chapter 4

Implementation of RemoteTrust

We can divide the implementation of RemoteTrust into three equally important components: The extensions on the Open Enclave project, the RemoteTrust header included in the enclave binary and the server-client infrastructure used to establish communication, and handle data or function requests. The extensions on Open Enclave allow the redirection of function calls to the other side of the communication, with requests that are sent via the RemoteTrust header, and ultimately handled with the server-client software.

4.1 Extensions on Open Enclave

For the extensions on Open Enclave, we use the C++ implementation of the Oeedger8r marshaling tool [21]. In its source we find the logic to automatically generate code that wraps the enclave and host calls for both the host and enclave. Whenever a Makefile invokes the Oeedger8r tool, the first step is to parse the shared flags. These can include options such as `--trusted` to indicate we are generating marshaling code for the enclave, `--experimental` for simulation purposes, or the flag we include `--pop-sgx` to identify projects that intend to use the RemoteTrust framework. Whether it is for the host or the enclave, both intermediate C header and source file will get autogenerated. For RemoteTrust, we include our custom functions `emit_pop_t_c()` and `emit_pop_u_c()` to generate the trusted and untrusted code, respectively.

```
1 void emit_pop_u_c():
2     create file name = (edl_name + "_u.c")
3     inside the file:
4         /* Trusted source code generation: */
5         include trusted function IDs
6         include trusted function names
7         include ECALL marshalling structs
8         include trusted function arrays generated by RemoteTrust
9         include RemoteTrust ECALL function wrappers
10        /* Untrusted source code generation: */
11        include untrusted function IDs
12        include untrusted function names
13        include OCALL marshalling structs
14        include OCALL functions
15        include OCALL function table
16        include function to redirect ecalls to RemoteTrust header
```

Figure 4.1: Pseudo-code for `emit_pop_u_c()`

```
1 void emit_pop_t_c():
2     create file name = (edl_name + "_t.c")
3     inside the file:
4         /* Trusted source code generation: */
5         include trusted function IDs
6         include ECALL marshalling structs
7         include ECALL functions
8         include ECALL function table
9         /* Untrusted source code generation: */
10        include untrusted function IDs
11        include OCALL marhaslling structs
12        include RemoteTrust OCALL function wrappers
```

Figure 4.2: Pseudo-code for `emit_pop_t_c()`

For obvious reasons, on the untrusted side the first function that will be called is the creation of the enclave. However, for the purposes of RemoteTrust we do not want the instance of the binary running on the server side to execute this, but instead we want it to wait for instructions of the server RemoteTrust on enclave calls to execute. Hence, we modify this function to alter the behavior depending on if we are on the client or the server side after calling `init_RemoteTrust()`; provided by the RemoteTrust header that is described on the next subsection 4.2.

On the client side, we will ask RemoteTrust to request the creation of the enclave, using `RemoteTrust_call()`, whereas on the server side we will register a signal handler for server requests and busy wait. Whenever the untrusted side running on the server receives a signal from the server (the parent process) it will retrieve the function name and parameters from a shared file, and call the corresponding ecall using an auxiliary table `pop_calls[]` that is automatically generated as well.

On the client side, the implementations of the wrapping functions will also differ, as we do not want to call the enclave functions -as that would be catastrophic on the embedded device- but instead they need to ask RemoteTrust to request the ecall to the server, using `RemoteTrust_call()` again.

4.2 The RemoteTrust header

The RemoteTrust header gets included in the enclave binaries using the intermediate auto-generated headers, as well as in the server-client infrastructure. It acts as the middle-man for the parent-child communication whenever the server needs to request the child for function calls - see Figure 4.3, or the child needs the parent to request an external call to the other side or handle memory synchronization. It is important to remark that the function call

requests are a two-way dynamic: The binary might need to ask the parent for a function request (in the host side, if it is an ecall), and the parent will also ask the child for functions (whenever the server receives a function request).

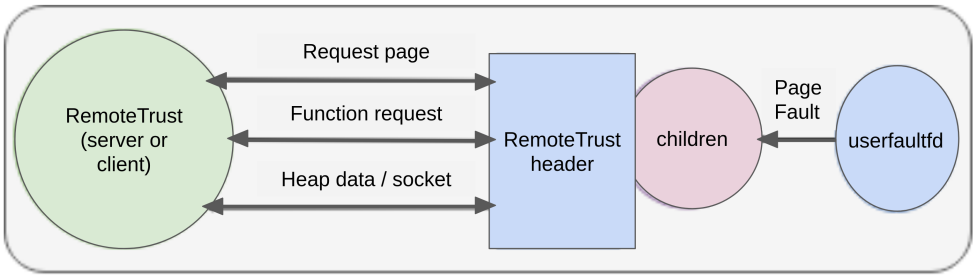

Figure 4.3: Main communication instances between RemoteTrust and the header included in the generated binary that used Open Enclave. Among the main requests we find of special importance the request of pages when a page fault is registered by the userfaultfd handler, function requests and the exchange of sockets of heap information.

The most important services that the RemoteTrust header provide are the following:

Initialization of RemoteTrust Done via `init_RemoteTrust()`. Initializing the environment involves obtaining the shared socket created by the parent before fork as well as registering the userfaultfd handler for the heap pages and sharing them with the parent process.

Requesting function calls Using `pop_call()`, the binary running on the client side will use this logic to obtain the results of an ecall from the server side. The RemoteTrust header will obtain the function name and arguments from a shared file, and will then signal the parent process for the execution of the function. After that, the header will busy wait until the parent signals reception of function return value(s).

Requesting faulting pages In the handler of the page faults, the RemoteTrust header will check if the event message was a page fault, and if so it will request the parent for

the page contents. Noticeably, the header delegates on the parent process to handle the DSM logic, obtaining the page and performing the synchronization logic. After obtaining the page contents from the parent using a shared socket, it will copy them into the faulting region.

4.3 Server-client infrastructure

The RemoteTrust server or client binary will be in charge of sending and receiving messages, handling the DSM of heap pages, establishing the SSL/TLS communication, forking into the children processes and providing optional debugging and tracing utilities.

Whenever the `RemoteTrust` binary is executed, it will determine if it is running on the client or the server side by attempting to connect to the other node. It will then create a thread in charge of handling bus messages using an SSL/TLS connection. If RemoteTrust is running on the server side, it will share with the client the untrusted binary -by pieces- for the corresponding architecture.

When the client receives the binary, it can execute choosing the option (s)gx from the terminal. This will trigger the ”parenthood” logic in which the children process will be forked after creating a socket that can be used to share heap page contents. The parenthood logic is specially important since it includes the registration of a signal handler for the children, in which RemoteTrust will use a shared file to determine if the children needs a function call, is asking for a page when an address faulted or is simply sharing some other information such as the heap virtual address range.

For optional tracing purposes, RemoteTrust includes the possibility of using **ptrace**, with the parent process as tracer and the Open Enclave binary as ”tracee”. This includes logic to

start tracing the child, waiting for its status update or reading its address space. This utility is disabled by default to enhance performance. Similarly, RemoteTrust includes optional debugging logs with an API that contains different degrees of emergency (from informative or alarm messages).

A bash script that takes care of the compilation of every source file is also provided, as well as replacing the Oeedger8r binary (from the Open Enclave project) with the custom binary, that automatically generates all the intermediate code that is required. Noticeably, only two binaries get generated: *RemoteTrust* and the custom binary. The behavior of both files will vary depending on where they get executed (server or client). As an example, if the project is called "Helloworld", we can do:

```
$ ./create_remote_trust generate helloworld
```

which will generate *RemoteTrust*, for both server and client, as well as *hellowordhost*, originally only on the server side. When the secure SSL/TLS communication is established via certificate exchange, the server sends to the client (untrusted side) the helloworld binary required for the `execve` that follows the child's forking.

Chapter 5

Evaluation of RemoteTrust

We perform a double evaluation of RemoteTrust: From a security perspective, and focusing on overall performance and overhead. In this Chapter we discuss both, with the security evaluation on section 5.1 and performance on 5.2 (covering in different subsections each of the ported applications).

In terms of performance, we evaluated the framework by securing and analyzing the generated communication overhead of the C/C++ benchmarks of the SPEC CPU 2017 benchmark suite, two overhead microbenchmarks and five applications of the public domain: Three Open Enclave samples (including a program related with file encryption), a Linux AI-based virtual assistant [25] and a visual place recognition software for robotics automation [11]. We introduce each of these applications in following subsections in this chapter, to then conclude with a discussion focused on performance.

To arrange a server-client infrastructure according to our conceived scenario (high-end mainframe, embedded device) we rely on a Raspberry Pi 4 Model B as client and a Lenovo ThinkPad as server, resulting in the experimental setup shown in Table 5.1. Since the Open Enclave package is not available for ARM yet, we adapted their source code to be able to build locally the necessary code on the embedded side and instructed the linker to ignore unresolved enclave-related symbols in object files. In addition, we slightly modified the most recent version of the Linux kernel for the Raspberry Pi (v5.10.63-v7l+) on its configuration to include the `userfaultfd()` system call. Some contributions to the open-source repository

of Open Enclave were also made along the way.

Table 5.1: Experimental setup

Description	Core i9-9880H	Cortex-A72 (ARMv7/v8)
Vendor	Intel	Arm Ltd.
Cores	8 (16 HT)	4
Clock (GHz)	2.3	1.5
RAM	16 GB	8 GB
Intel SGX	Yes	No
Linux kernel	v5.11.0	v5.10.63
Bits	64	32 / 64
Interconnect	Network Layer (IPv4/IPv6)	

5.1 Security evaluation

In terms of security, we provide a list of CVEs (vulnerabilities) related with embedded devices and IoT that our framework could potentially help mitigate – see Table 5.2, that contains seven examples. We also leverage RemoteTrust to protect a custom server from the well-known Heartbleed attack.

An alternative view of the list of the vulnerabilities collected is shown in Figure 5.1, in which each example of vulnerability is mitigated with a security additions that RemoteTrust allows edge devices to use. The usage of the debug malloc tool (to mitigate CVE-2021-22547) is covered later in the performance section, using one of the most popular Open Enclave use samples, ported to RemoteTrust. The other two security additions (Protection of credentials, encryption of files with sensitive data) are arguably the two most common examples of trusted execution environments' benefits.

Heartbleed (CVE-2014-0160) is an OpenSSL vulnerability. In particular, it is a buffer-overflow that exposes the server's memory. There exists many tools online to audit target websites, usually in the form of small Python scripts. The flaw is triggered when we tell

Example of vulnerability	Description
CVE-2021-22547	An implementation of calloc() that doesn't have a length check in IoT SDK, allowing an attacker access to the other parts of the heap.
CVE-2020-26079	A vulnerability in Cisco IoT Field Network Director (FND) could allow an authenticated, remote attacker to obtain hashes of user passwords on an affected device. The vulnerability is due to insufficient protection of user credentials. A successful exploit could allow the attacker to obtain hashes of user passwords on an affected device.
CVE-2018-7811	An Unverified Password Change vulnerability exists in the embedded web servers in all Modicon M340, Premium, Quantum PLCs and BMXNOR0200 which could allow an unauthenticated remote user to access the change password function of the web server.
CVE-2017-10718	Recently discovered on IoT devices in Shekar Endoscope that allows a malicious user to connect to the device and change the default SSID and password.
CVE-2016-10512	Passwords are stored unencrypted of its LDAP configuration, and the credentials are retrieved by the system when the LDAP configuration page is opened.
CVE-2015-1005	IniNet embeddedWebServer before 2.02 uses cleartext for password storage, which allows context-dependent attackers to obtain sensitive information via unspecified vectors.
CVE-2013-3734	The Embedded Jopr component in JBoss Application Server includes the cleartext datasource password, which might allow attackers to obtain sensitive information.

Table 5.2: Survey of example vulnerabilities that relate to embedded and Internet-of-Thing devices that the RemoteTrust framework could potentially help address

Examples of vulnerabilities	Base	Security additions			RemoteTrust
		Debug malloc tool	Protection of credentials	Encryption of files with sensitive data	
CVE-2021-22547	×	✔	×	×	✔
CVE-2020-26079	×	×	✔	×	✔
CVE-2018-7811	×	×	✔	×	✔
CVE-2017-10718	×	×	✔	×	✔
CVE-2016-10512	×	×	✔	×	✔
CVE-2015-1005	×	×	×	✔	✔
CVE-2013-3734	×	×	×	✔	✔

Figure 5.1: Collection of example CVEs in which to show how each security addition can help mitigate the vulnerability with RemoteTrust

the server that we are going to send a particular message that is of certain amount of bytes long, but we actually send a shorter message; an un-patched version of the OpenSSL library will not check if we really sent the aforementioned bytes of data. The server will store our message, then proceed to read the amount of bytes of data from its memory (it reads the memory region where our message should be stored) to then send that read message back to the malicious client.

We first replicate the attack with a vulnerable version of the OpenSSL tool (OpenSSL version 1.0.1) acting as server, an a custom evil client written in C. Subsequently, we repeat the experiment but using as server our own C code, so that we can protect it later with RemoteTrust. The attack over our vulnerable server is shown in Figure 5.2.

```
11:05 [host]  (master)$ ./Heartbleedhost
Accepting connections on port 4433 (localhost)
11:06:10 INFO  bus_functions.c:82: Waiting for connections
11:06:11 INFO  bus_functions.c:88: Connection established with client
Receiving HELLO
Sending header...
Receiving HB...
11:06 [host]  (master)$
```

(a) Vulnerable custom server

```
11:06 [heartbleed-client]  (master)$ ./heart
Connected to server via port 4433
Sending HELLO
Receiving HELLO
Sending HB
Receiving HB
00 18 03 02 40 00 0e 00 00 1b 10 37 a5 3b 56 00   ....@......7.;V.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00   ............
11:06 [heartbleed-client]  (master)$
```

(b) Heartbleed malicious client written in C

Figure 5.2: Heartbleed attack replicated with custom malicious client written in C and server with vulnerable OpenSSL version

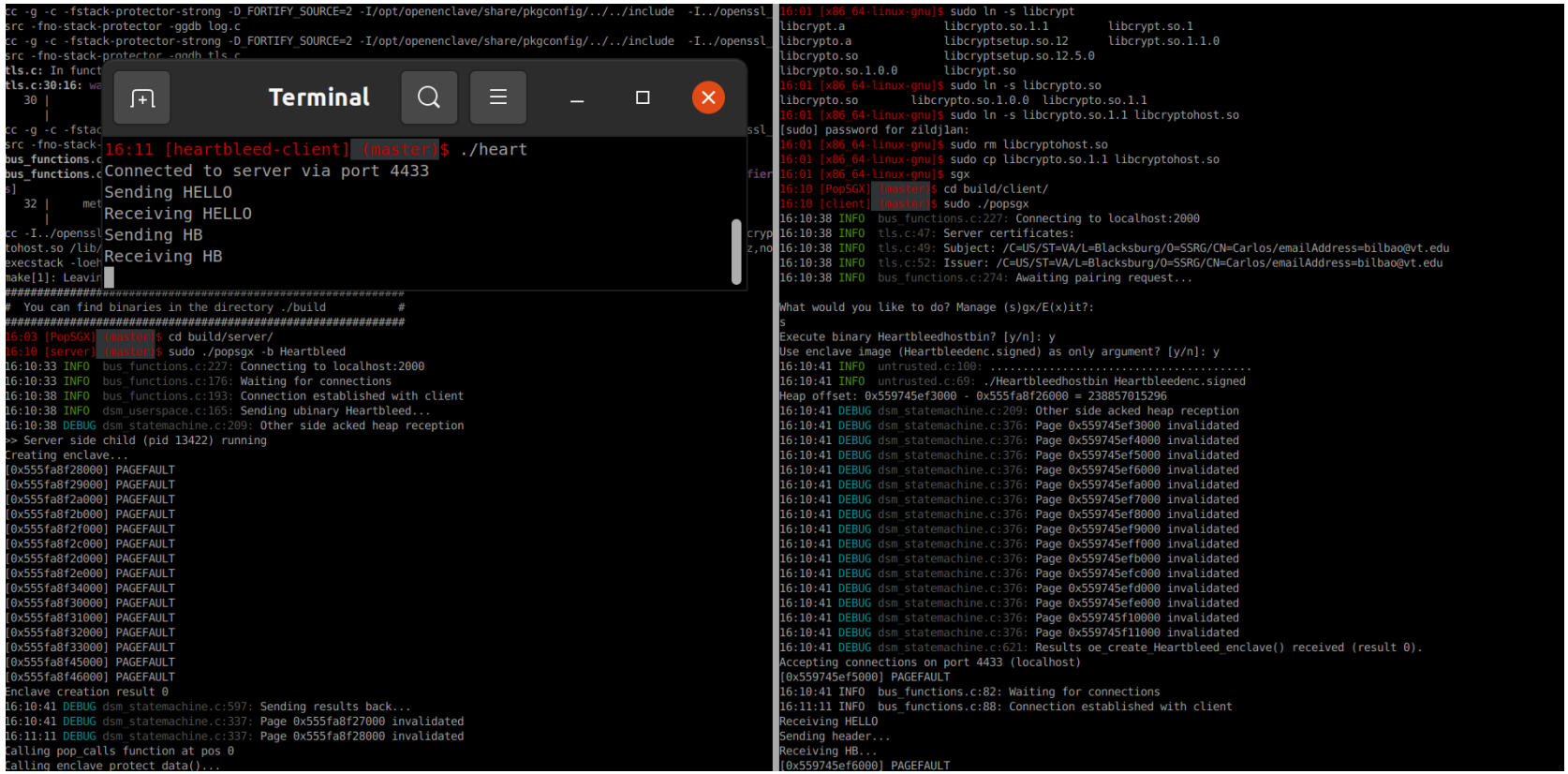

Figure 5.3: Preventing Heartbleed attack using RemoteTrust on a vulnerable server acting as enclave-client

Finally, we secure the server communication with the malicious client using RemoteTrust (see Figure 5.3). In particular, this requires making the vulnerable custom server a client of the framework, and written an external server that can backup the vulnerable memory via trusted enclave protection. Using RemoteTrust to create a server that backups the memory of the client and then sends it back after the `OpenSSL_read()` effectively protects the vulnerable server from the Heartbleed attack.

5.2 Performance evaluation

The RemoteTrust framework provides security benefits to edge devices, but overall performance should also be taken into account. In particular, the overhead created on the communication and synchronization between server and client should fall within acceptable boundaries, as such devices usually depend on a timeliness response to external inputs. To put it in simple terms, an extremely-secure smart fridge is no good if the ice cream is melted by the time the server is done figuring out the new fridge temperature.

In order to perform a comprehensive evaluation of RemoteTrust, we port and evaluate the performance of a diverse set of applications; As explained in the following subsection, we port the C/C++ benchmarks of SPEC CPU 2017 (as the applications written in higher level languages are not supported at the moment).

Subsequently, we execute two overhead microbenchmarks, one for raw-latency of number of enclave calls, and other for parameter-passing overhead, following the approach of previous security researchers on SGX enclaves [8]. Finally, we secure popular open-source applications: Three Open Enclave SDK samples, an AIbased Linux Vitrual Assistant [25] and OpenFABMAP [11], a visual place recognition software for robotics automation.

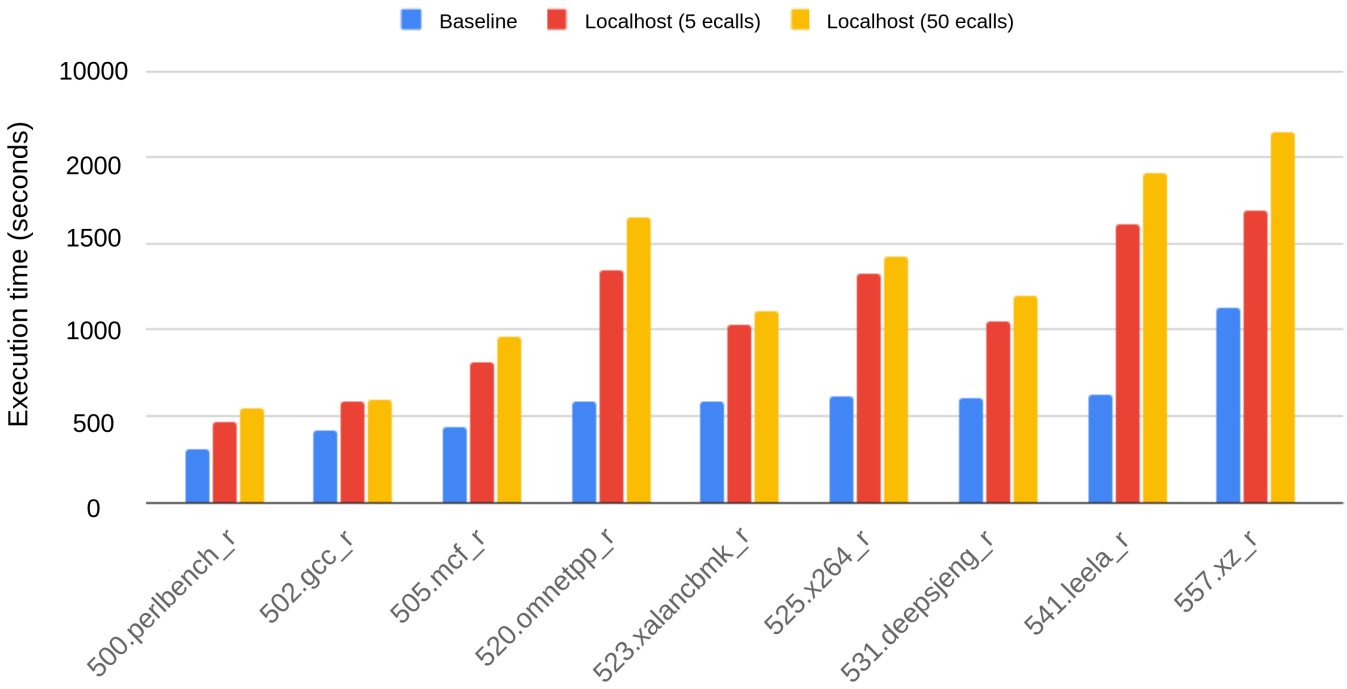

Figure 5.4: Execution of the ported C/C++ applications of SPEC CPU 2017 under an homogeneous setup (both client and server are x86_64 systems) for baseline, 5 and 50 remote enclave calls.

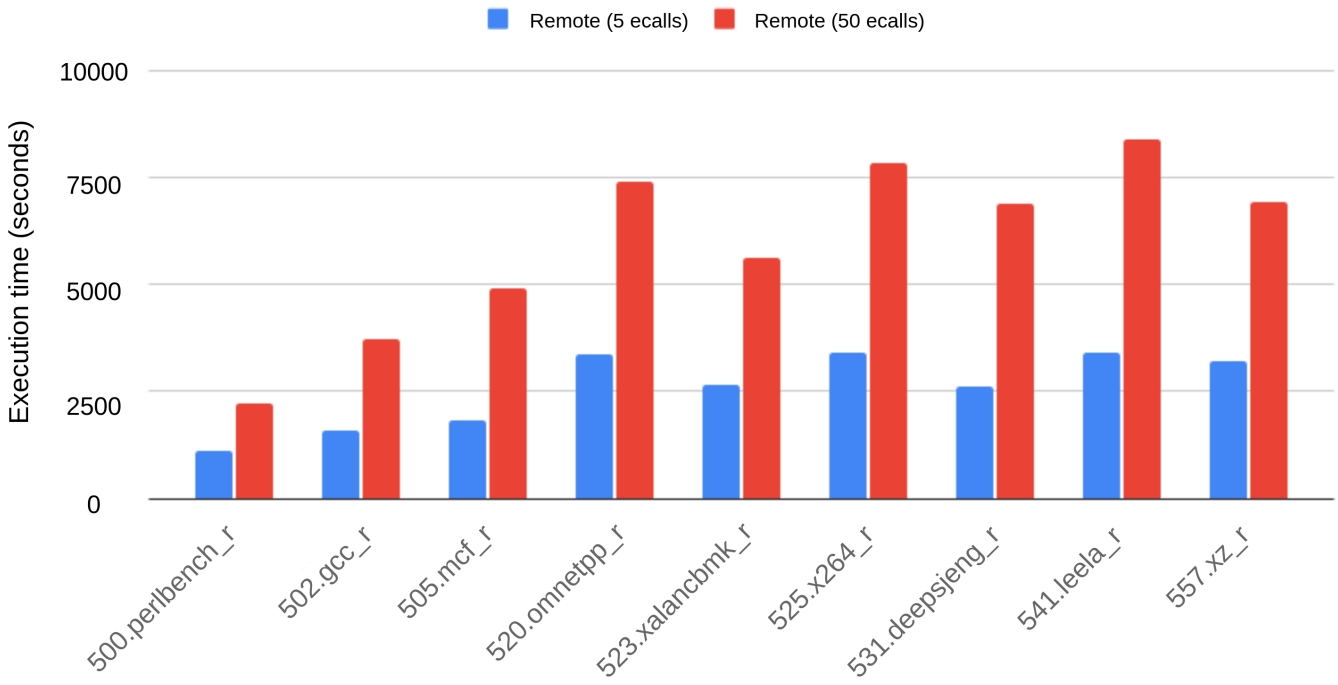

Figure 5.5: Execution of the ported C/C++ applications of SPEC CPU 2017 under an heterogeneous setup (client is a Raspberry Pi and server is an x86_64 system) for 5 and 50 remote enclave calls.

Figure 5.6: Execution of the Raw-overhead microbenchmark for different number of remote enclave calls (from 1 to 10,000) on homogeneous and heterogeneous setups, and the standard deviations for a 20-runs sample, all of them below one.

5.2.1 SPEC CPU 2017

The SPEC CPU benchmark suite(s) usually need very little introduction among system researchers. It is one of the most popular industry-standardized, CPU-intensive benchmark suite, and it is used to measure and compare systems' performance. This is achieved via stressing different components of the device, ranging from the processor to the memory.

We port all the C/C++ applications from SPEC CPU 2017 (500.perlbench,gcc, 505.mcf, 520.omnetpp, 523.xalancbmk, 525.x264, 531.deepsjeng, 541.leela and 557.xz). We use the opportunity to compare the overhead on them produced by RemoteTrust, both in homoge-

Figure 5.7: Execution of the Parameter-passing overhead microbenchmark for different number of parameters in the enclave functions (from 1 to 20) on homogeneous and heterogeneous setups, and the standard deviations for a 20-runs sample, all of them below one.

neous (server and client are x86) and heterogeneous setups (Raspberry Pi and x8_64 Lenovo server). The results from these experiments are shown in Figure 5.4 and Figure 5.5. Furthermore, the execution on an homogeneous setup provides the opportunity to compare the overhead of RemoteTrust with TEEs and the baseline (the benchmark running as usual, without enclaves), which provides a better picture of the total overhead on heterogeneous, highly affected by the network overhead itself on Figure 5.5. One thing to notice is the variability of the benchmarks, which is highly affected by the way they interact with the memory; Page faults mean an even higher price in performance than usual, due to the communication overhead between both sides.

5.2.2 Overhead microbenchmarks

In order to gain a better understanding of the root causes of the overhead produced by the RemoteTrust framework, we develop two microbenchmarks: One to measure the latency and overhead due to an expotential increase of the number of remote enclave calls, and other to understand the effect of parameter-passing on such function calls.

The results from the Raw-latency overhead microbenchmark are shown in Figure 5.6, and the values from the Parameter-passing microbenchmark in Figure 5.7. Noticeably, the effect of executing one, ten or even a hundred enclave calls is very small, due to the fact that most of the execution time consist of the creation of the enclave on the server side, which happens only once no matter the number of enclave calls that follow after that (unless, of course, there is a request to destroy and create a new enclave, but that is not the case on these experiments). We run each of the scenarios twenty times, both on homogeneous and heterogeneous setups, and then compute the standard deviations of the samples, which is always smaller than one ($\sigma < 1$).

5.2.3 Open Enclave SDK samples

We first leveraged RemoteTrust to port three well-known sample use cases presented in the public repository of the Open Enclave project: HelloWorld, DebugMalloc and he File encryptor. The two first samples are written in the C language and the third is a C++ implementation.

HelloWorld sample. The first use case does not carry out any special computation other than performing the most common enclave operations: Creation of the enclave, request for an enclave call, execution of that enclave call, and destruction of the enclave on termination. This sample can be easily compiled using the RemoteTrust extensions with the flag `-pop-sgx`, and its execution flow observed on Figure 5.8 from both the server and client sides. This simple use case demonstrates how to build, sign and run an Open Enclave image with RemoteTrust; supporting both normal execution and the Intel SGX simulation mode.

Normally, on the client side of Figure 5.8 information of the SSL/TLS certificates (Subject, Issuer) is displayed, but we delete them for the sake of simplicity. The hyphens (-) mark the execution of the secured application, when server and client fork a children with the generated binary. After that, the server side (top on Figure 5.8) will start displaying the enclave function call requests received and the results of those operations that are sent back to the embedded devices. The client also informs of the reception of each function call, although that behavior is limited to the debug mode to improve performance.

It is important to remark that even though there is no output information about that in Figure 5.8, this entire process is possible thanks to the SSL/TLS protocol and the transparent DSM synchronization of the applications' heaps, that requires communication between server and client for diverse requests such as page invalidation.

DebugMalloc sample. The second use case shows how to enable debug malloc using

```
$ ./create_remote_trust.sh generate helloworld
$ ./RemoteTrust -b helloworld
INFO Server waiting for connections
INFO Client connected, sending binary helloworld
INFO Server side children (pid 5503) running
---------
Creating enclave... (result 0)
Calling enclave_helloworld()...
Hello world from the enclave!
Enclave function successful (result 0)
Calling oe_terminate_enclave()...
Enclave function successful (result 0)
---------
```

```
$ ./RemoteTrust
INFO Connected to server with valid certificates
INFO Saving binary helloworldhost
What would you like to do? (s)gx/E(x)it?: s
Use enclave image as only argument? [y/n]: y
INFO Forking ./helloworld helloworldenc.signed
---------
DEBUG Other side acked heap info reception
DEBUG Results of create enclave received
DEBUG Results of ecall received (0)
DEBUG Results of ecall received (0)
---------
```

Figure 5.8: Execution of the HelloWorld sample from server (top) and client (bottom).

compiling options, as well as the debug malloc public API. This tool can be used to identify memory leaks on the enclaves, helping with memory allocation tracking in both "global detection" and "local detection". As such, it can help mitigate a variety of memory-leak-related vulnerabilities such as CVE-2021-40797 or CVE-2021-39282.

File Encryptor sample. We select this C++ application to show how to leverage file cryptographic operations inside an enclave; using a subset of the features offered by the open-source mbedTLS library [1]. The usage of file encryption inside a trusted execution environment offers the combined effort of "data-in-use" and "data-at-rest" protection, especially against an untrusted host in the threat model of enclaves. Hence, such feature can be used to solve multiple non-encrypted file vulnerabilities like in the case of CVE-2019-11064

and CVE-2018-6975.

The execution of the File Encryptor use case from the embedded device requires the remote request of the following enclave operations:

1. Creation of the enclave on the RemoteTrust server side, which holds the real trusted side of the generated application.

2. Initialization of the encryptor. Similarly, this use case shows how to derive a key from a string -the password- using PBKDF2.

3. Encryption of a file passed as argument by the user, and then check for correct encryption comparing the resulting contents of the documents.

4. Decryption of the file, which has to be requested from the client to the server side. Of course, the server side will use the previously created enclave.

5. Termination of the enclave whenever all encryption and decryption of files have taken place.

5.2.4 AI-based Linux Virtual Assistant

Many embedded devices apply artificial intelligence as it is remarkably useful for tasks that require recognition of patterns, such as identifying faces and traffic signs or monitoring patients health state. However, some of these tasks deal with specially sensitive information from the users, so securing them on embedded devices is imperative.

Virtual assistants [9] rely on artificial intelligence to analyze user requests and execute the command that should better accommodate the user needs. An interesting implementation in C language can be found open-sourced online [25]. In this particular case, the application

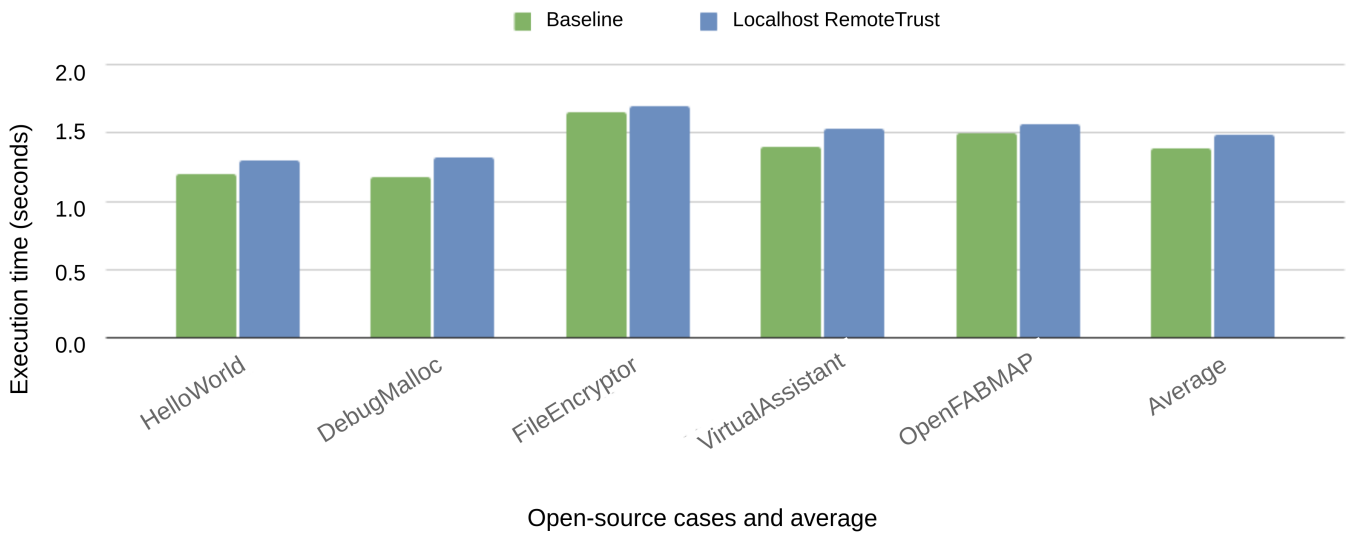

Figure 5.9: Average execution times (in seconds) for each of the open-source applications before and after porting them to RemoteTrust, executing server and client locally on the same x86_64 device.

uses the text inputs to execute commands by following a simplified multinomial Naive Bayes algorithm [28]. Such an elementary AI classification allows to distinguish between several categories of user requests: weather information, media playing, opening the calendar, etc. by assigning them a score after tokenizing the sentences into words and reviewing a keyword database.

From all the requests the virtual assistant reacts to, sending emails is specially sensitive from a security perspective. In particular, during the configuration process both the email address and password are encoded by the assistant using Base64 [18]. We leverage RemoteTrust to secure these operations by offloading them to a central server with Intel SGX capabilities. Hence, we move the handling of the email request to the enclave side, and accordingly extend the host of the virtual assistant to request remote creation of the enclave and execution of the ecalls whenever the user triggers the email request. The function `b64_enclave_encode()` is developed in the enclave and included into `virtualassistant.edl`. This way, RemoteTrust

can be used to generate the server/client infrastructure and offload the encryption of email and password into the server. We do so and collect execution time measurements that are analyzed in the subsection 6.0.4 below.

5.2.5 OpenFABMAP: visual place recognition

Embedded devices frequently include software to perform some type of bio-metric operation such as image or voice recognition. These applications usually handle sensitive user information and as such require special security considerations. With this in mind, the last application we port to the RemoteTrust framework is OpenFABMAP [11], a visual place recognition software for robotics automation. The application, written in C++, is an open-source implementation of a simultaneous localisation and mapping algorithm that performs matches between places, using camera images as only input of the system, that then process them using OpenCV [7] extraction methods.

We use RemoteTrust to secure the first step of OpenFABMAP: Opening a YML settings file that is used to specify details of the desired operation. Hence, two enclave calls are included in the main function of the application: Creation of the enclave and remote opening of the settings file. RemoteTrust automatically terminates the enclave on the server once the application running on the client side concludes.

5.2.6 RemoteTrust performance trade-offs

In terms of performance, the time it would take to execute the ported applications solely on the Raspberry Pi cannot be measured; the embedded device is not capable of executing these applications and otherwise the framework itself would not have a reason to be. However, we can analyze the impact of the framework's intervention in the overall performance when

running both server and client inside the same x86_64 machine. Furthermore, since this effort is closely related with edge-computing, it is imperative to investigate the degradation in performance when the distance between client and server increases [6]. Hence, the aim of this evaluation was to understand:

1. The impact of the framework in comparison with vanilla execution of the enclave operations.

2. The performance degradation when moving from a local area network (LAN) into a realistic remote network connection.

Both evaluation directions are addressed on Figure 5.9 and Figure 5.10, respectively. Figure 5.9 presents the average execution time of all applications compared to their localhost execution on RemoteTrust and Figure 5.10 the execution times on a local network as opposed to remotely with public IPs. Table 5.3 shows the percentage in performance degradation for both cases: the RemoteTrust framework had a 7% impact compared to the local execution while remote IPs averaged 11% compared to LAN, which was to be expected from the overhead introduced on the network layer communication.

It is important to remark that the execution times of both the LAN and remote settings do not only suffer from the communication overhead between server and client, but are also subject to the hardware limitations of the Raspberry Pi. In other words, the time required to execute the applications with or without RemoteTrust locally (Figure 5.9) cannot be compared with the average measurements obtained from LAN or remote execution (Figure 5.10) that constitute an ISA-heterogeneous setup. In the second case, most of the workload is handled by the Raspberry Pi, which is very limited on hardware resources compared to our server.

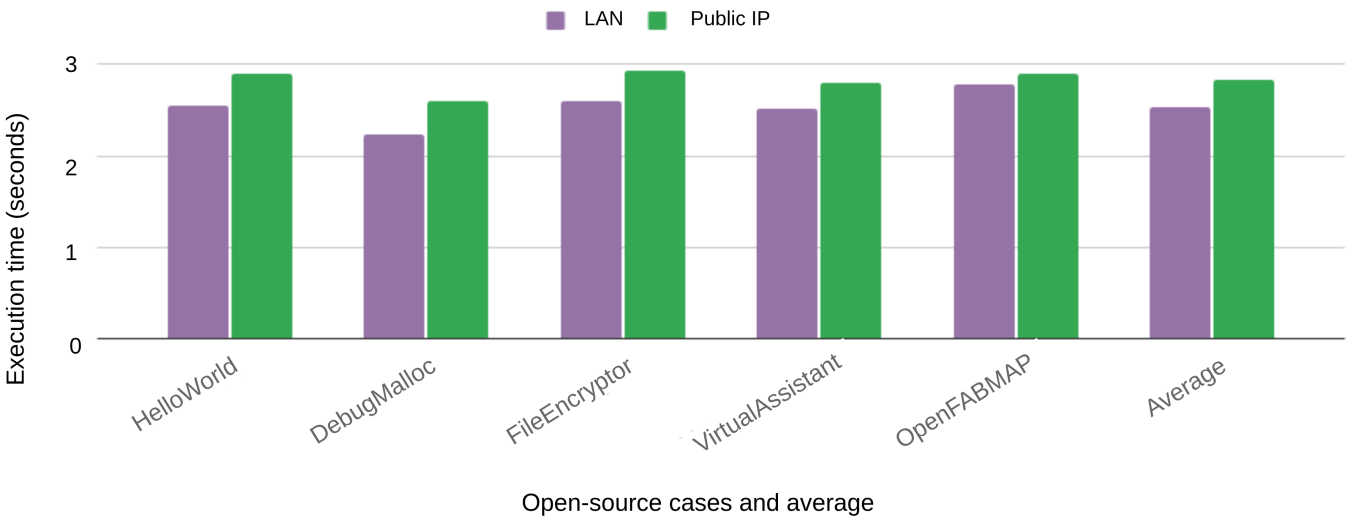

Figure 5.10: Average execution times (in seconds) for each of the open-source applications ported to RemoteTrust, comparing the performance of a server-client infrastructure where both sides reside on the same local area network (LAN) and remotely (with public IPs).

Table 5.3: Performance degradation percentage for each of the samples when executed with RemoteTrust on the same machine (Vanilla vs RemoteTrust) and when moved from a local area network to remote connection (LAN vs Remote)

Sample	Vanilla vs RemoteTrust	LAN vs Remote
Hello World	8.3%	13.9%
Debug Malloc	11%	16%
Virtual Assistant	9.5%	10.9%
OpenFABMAP	4.2%	4.3%
File Encryptor	3.2%	13%
Average	**7%**	**11%**

Chapter 6

Conclusions

In this thesis we aimed to advance the state-of-the-art on heterogeneous setups from an edge-computing security perspective but also taking performance into consideration. To provide better security capabilities on embedded devices, we develop RemoteTrust, a framework that can be leveraged to offload hardware-based secure computations on a remote server with better resources – probably an x86_64 machine.

We evaluated our solution with a realistic setup (x86 server, Raspberry Pi client) and real-life use cases (four open-source applications) as well as performance-intensive benchmarks with SPEC CPU 2017 and overhead microbenchmarks to gain a better understanding of the cost in performance; averaging less than 1 second per 100 remote single-parameter enclave calls. Furthermore, we analyzed the framework from a security perspective providing a list of CVEs that could be potentially mitigated and preventing the Heartbleed attack.

To the best of our knowledge, this is the first framework that extends the Open Enclave SDK to provide Trusted Execution Environments (TEE) to edge devices. We understand that providing better security mechanisms to such devices is critical, since the embedded devices market keeps growing at a tremendous pace, but with a design focus on power-consumption for practical reasons.

There is still a lot of work left to do in the field of systems in regard of these notions. In terms of parallel performance, improving the communication layer between nodes would be

very beneficial for the overall performance of the system. This is also true on security for RemoteTrust, since there is an overhead in remotely requesting execution of enclave functions using the network layer. Enhancing the communication channels would be incredibly beneficial in both cases, but there would still be a need for better algorithms and underlying methods for scheduling and work distribution.

6.1 Trust Base and Limitations

Granted, during the development of the RemoteTrust framework there were some actors, specially in terms of infrastructure, from which we rely for a successful -and secure- offloading of the remote trusted executions. If any of these components is compromised, the security of RemoteTrust could not be guaranteed. For this reason, it is important to enumerate these parts, in order to understand our limitations - as well as potential ways to make RemoteTrust more secure in future iterations.

We rely on SSL/TLS as communication channel between server and client. Needless to say, if a vulnerability is found on this protocol, the entire RemoteTrust framework would be compromised. This is however fairly unlikely, or at least less likely than the exploitation of other components we put our trust on.

The most important trusted base that could be compromised is the messaging layer. In fact, the messaging between server and client could potentially be used to generate a denial-of-service (DoS) scenario. In particular, if a malicious agent gains control of the untrusted binary, it could potentially invalidate pages following the DSM system, effectively rendering the framework useless.

Hence, both the communication channel and the messaging layer are the two main compo-

nents we rely upon for a safe communication. The good news is that, even if one or even both of them were compromised, the security of the enclave operation should theoretically remain intact. This is, the framework might stop being useful under attack, but it should not be unsafe.

6.2 Future work

One major improvement of RemoteTrust would be to address its limitation to synchronize only anonymous pages for the heap of the trusted and untrusted sides of the connection. This is due to the usage of userfaultfd, which is generally a very good approach for managing page faults on user space but lacks support for file-backed pages. Other limitation is the support for C and C++ language, while many modern embedded devices carry out Artificial Intelligence operations that are mostly written in Python. Hence, it would be very good to adapt the Open Enclave SDK and include support for Python projects. Porting the SDK to other languages would allow for an experimental evaluation with the totality of the SPEC CPU 2017 benchmarks.

The Open Enclave SDK has the potential to support other Trusted Execution Environments than Intel SGX. Taking into account the layer of abstraction provided by RemoteTrust and the library, their implementation and evaluation should be straightforward. For example, the inverse experiments could be conducted where the client is x86 and the server an ARM with support for ARM TrustZone.

Furthermore, as soon as the Open Enclave SDK community finish supporting ARM, the combined efforts of ARM TrustZone and Intel SGX could be achieved with RemoteTrust. The merging of both TEE technologies could be beneficial in unforeseen ways, even as a method to add an extra layer of protection. For example, the SGX enclave creation using

RemoteTrust in an edge-computing device could be handled inside ArmTrustZone, or the other way around. The combined efforts of heterogeneous architectures is better documented in terms of performance than from a security perspective as of today, and the possibilities are endless.

It would also be interesting to provide automatic attestation of the enclaves created using RemoteTrust. This should not be very challenging, since the attestation can be carried out inside the wrapping code that communicated server and client on the initial exchange of messages. However, that would also impact performance (as you need an extra enclave for attestation), so it should be an option, and not the default behavior of the framework.

In addition, interesting insights could result from experiments on RemoteTrust with several clients and one server. For instance, the server could reuse the same enclave to carry out the trusted operations -of the same type- requested by all the clients. Intuitively, this would reduce the overall execution time since all the clients but the first one would not need to either create nor destroy a dedicated enclave.

Bibliography

[1] mbedtls library. https://github.com/ARMmbed/mbedtls.

[2] Open Enclave SDK. https://openenclave.io/sdk/.

[3] Joyce Ayoola Adebusola, Adebiyi Ayodele Ariyo, Okeyinka Aderemi Elisha, Adebiyi Marion Olubunmi, and Okesola Olatunji Julius. An overview of 5g technology. In *2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*, pages 1–4, 2020. doi: 10.1109/ICMCECS47690.2020.240853.

[4] Thaynara Alves and D. Felton. Trustzone: Integrated hardware and software security. 01 2004.

[5] 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. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[6] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: The case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 73–87, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi: 10.1145/3381052.3381321. URL https://doi.org/10.1145/3381052.3381321.

[7] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[8] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. Towards efficiently establishing mutual distrust between host application and enclave for sgx, 2020.

[9] Hyunji Chung, Michaela Iorga, Jeffrey Voas, and Sangjin Lee. Alexa, can i trust you? volume 50(9), pages 100–104. Computer, 2017. doi: 10.1145/2948618.2954331.

[10] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 841–858. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters.

[11] A. Glover, W. Maddern, M. Warren, S. Reid, M. Milford, and G. Wyeth. Openfabmap: An open source toolbox for appearance-based loop closure detection. In *The International Conference on Robotics and Automation*, St Paul, Minnesota, 2011. IEEE.

[12] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security Privacy*, 14(6):54–62, 2016. doi: 10.1109/MSP.2016.124.

[13] Hongmei He, Carsten Maple, Tim Watson, Ashutosh Tiwari, Jorn Mehnen, Yaochu Jin, and Bogdan Gabrys. The security challenges in the iot enabled cyber-physical systems and opportunities for evolutionary computing & other computational intelligence. pages 1015–1021, 07 2016. doi: 10.1109/CEC.2016.7743900.

[14] Ángel Longueira-Romerc, Rosa Iglesias, David Gonzalez, and Iñaki Garitano. How to quantify the security level of embedded systems? a taxonomy of security metrics. In

2020 IEEE 18th International Conference on Industrial Informatics (INDIN), volume 1, pages 153–158, 2020. doi: 10.1109/INDIN45582.2020.9442219.

[15] MarketsandMarkets.com. Embedded system market by hardware (mpu, mcu, application-specific integrated circuits, dsp, fpga, and memories), software (middleware, operating systems), system size, functionality, application, region - global forecast to 2025, 2020. URL https://www.marketsandmarkets.com/Market-Reports/embedded-system-market-98154672.html.

[16] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347693. doi: 10.1145/2948618.2954331. URL https://doi.org/10.1145/2948618.2954331.

[17] Vijeet H. Meshram and Ashish B. Sasankar. Security in embedded systems : Vulnerabilities , pigeonholing of attacks and countermeasures. 2016.

[18] Wojciech Muła and Daniel Lemire. Base64 encoding and decoding at almost the speed of a memory copy. *Software: Practice and Experience*, 50(2):89–97, Nov 2019. ISSN 1097-024X. doi: 10.1002/spe.2777. URL http://dx.doi.org/10.1002/spe.2777.

[19] Aleksandr Ometov, Viktoriia Shubina, Lucie Klus, Justyna Skibińska, Salwa Saafi, Pavel Pascacio, Laura Flueratoru, Darwin Quezada Gaibor, Nadezhda Chukhno, Olga Chukhno, Asad Ali, Asma Channa, Ekaterina Svertoka, Waleed Bin Qaim, Raúl Casanova-Marqués, Sylvia Holcer, Joaquín Torres-Sospedra, Sven Casteleyn, Giuseppe Ruggeri, Giuseppe Araniti, Radim Burget, Jiri Hosek, and Elena Simona Lohan. A

survey on wearable technology: History, state-of-the-art and current challenges. *Computer Networks*, 193:108074, 2021. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2021.108074. URL https://www.sciencedirect.com/science/article/pii/S1389128621001651.

[20] Dorottya Papp, Zhendong Ma, and L. Buttyán. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152, 2015.

[21] The Open Enclave SDK project. Oeedger8r open-source tool. URL https://github.com/openenclave/oeedger8r-cpp.

[22] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3): 461–491, August 2004. ISSN 1539-9087. doi: 10.1145/1015047.1015049. URL https://doi.org/10.1145/1015047.1015049.

[23] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015. doi: 10.1109/Trustcom.2015.357.

[24] Fabian Schwarz and Christian Rossow. SENG, the sgx-enforcing network gateway: Authorizing communication from shielded clients. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 753–770. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/schwarz.

[25] Ritwik Sharma and Riya. Linux based virtual assistant in C. In *International Journal of Advance Research, Ideas and Innovations in Technology*, volume 6.1. IJARIIT,

January 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/schwarz.

[26] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.

[27] Johannes Winter. Experimenting with arm trustzone – or: How i met friendly piece of trusted hardware. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1161–1166, 2012. doi: 10.1109/TrustCom.2012.157.

[28] Shuo Xu, Yan Li, and Wang Zheng. Bayesian multinomial naïve bayes classifier to text classification. pages 347–352, 05 2017. ISBN 978-981-10-5040-4. doi: 10.1007/978-981-10-5041-1_57.

[29] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and K. Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8: 58443–58469, 2020.