POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

On the applicability of software attestation techniques to embedded systems.

Supervisors
Prof. Cataldo Basile
Prof. Antonio Lioy
Alessio Viticchiè, Ph.D

Candidate
Marco Zudettich

Academic year 2018-2019

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been a vast diffusion of technology depending on software in the last few years. Not only personal computers, tablets and smartphones, but also numerous different devices (so called "embedded systems") are used today to control almost anything, from vehicles to industrial systems. Embedded systems have the advantage of being cheap and they can be deployed almost everywhere in a massive amount. The problem in having so much software running everywhere is that security breaches take place daily. The more the software is in charge of controlling something, the more the consequences of a security failure will affect people and companies.

Different solutions are available to protect systems and the software running on them. Most of them only works given assumptions that can't always stand.

First of all, not all systems are under the physical control of the people who are trying to protect them. It is common to have hardware owned by a business but under the physical control of a potentially not reliable third party. Most mobile devices, sensors and remote controls, in general, are placed in locations that are not under strict surveillance and control. This poses a serious problem, since ensuring the trustworthiness of a system under adversary control can be impractical. If a potential attacker owns the device itself, tampering cannot be prevented. The adversary can open the device, attach a debugger and potentially modify the software present on it in any way. The only possible defense is to detect these attacks and take the convenient countermeasures.

Attestation is a process in which the device's owner performs an integrity check of the software running on the device itself. Checking the integrity of a program can be done in different ways. Various papers about software attestation have been written. There are two approaches to this problem. The first one requires the integration of an additional piece of hardware. This is the solution proposed by the Trusted Computing Group [1]. A chip called TPM (Trusted Platform Module) is combined in the system and it is used for cryptographic operations and key storage. The problem with TPM is its cost. On large scale production devices, every single piece of hardware can increase the cost by a considerable amount. Another thing to consider is the integration with the software component, which makes the attestation code less portable. The other possible solution is an attestator requiring only software components. It does not require additional hardware, but it is by design less protected from tampering. In this research, only the second type of attestation is considered.

Generally speaking, every protection system is bypassable given enough time. The goal of a protection mechanism is to increase the time and the resources required for an attack to take place. In the end, every system is vulnerable to a determined and persistent attacker. For this reason, a device should never be trusted completely even if it is being attested. In the design phase of every system, security failures should be considered, especially on devices controlled by third parties. It is compulsory not to rely fully on information coming from an untrusted source, but also have some checks on the trusted side.

Also, the fact that only the endpoint security is addressed by such an attestator mechanism has to be analyzed. Attacks carried over the network are still possible, so the communication protocols used should be secured as well.

The idea of this thesis comes from a project called ASPIRE [2]. This project offers a framework (ASPIRE Compiler Tool Chain or ACTC) capable of automating protection mechanisms on software. Starting from an unprotected application's source code, it is possible to create an executable secured with various types of protections, both offline and online. These protections comprehend also a remote attestator.

The general structure of the ASPIRE project and its dependencies create a problem in terms of portability. This thesis will attempt to address two questions, both related to the use of fully-software attestation techniques in less powerful devices, such as embedded systems. The first one is about the possibility of solving the ASPIRE remote attestator's portability issues. The second one is about to what extent the attestator can be ported to different platforms, with a particular focus on embedded devices. The flow of the discussion followed by this research is outlined in Chapter 3.

1.1 Thesis organization

The following paragraphs describe the sequence through which the narration will be developed.

Chapter 2 contains an explanation about the ASPIRE attestator's functionality and an analysis of the previous literature in regards to software attestation. This initial analysis is crucial to understand the basic concepts of remote attestation and it will serve as the base for the examination of the ASPIRE attestator's portability.

Chapter 3 points out the goals of this research and further explains the narration that will be developed within every chapter.

Chapter 4 contains an outline about the specific requirements the ASPIRE framework poses on its attestator module.

To extend the portability of the ASPIRE attestator, an attempt at separating it from the entire ACTC will be made in Chapter 6.

The extraction of the attestator from the whole framework is not a trivial task. To mimic the behavior of the ACTC an understanding of the inner functionality of ELF executable is needed. In particular, an examination of ELF executable's memory layout and relocations is required for properly substitute the framework task. Chapter 5 includes a study about the ELF file format and an evaluation of useful tools for analyzing and manipulating it. This chapter will not examine the topic of attestation directly, but it is necessary to address some of the issues this thesis is trying to resolve.

Once the attestator has been extracted from the ASPIRE framework, an examination of some obfuscation options is done in Chapter 7. This chapter will contain a review of the most common obfuscation techniques, and it will analyze some obfuscation alternatives to the ASPIRE framework in this regard.

Chapter 8 holds a study about the portability of the mentioned attestator. In particular, this chapter focuses on the possibility of porting the attestator to the most used operating systems and architectures in the field of embedded devices.

Finally, Chapter 9 draws conclusions about the work.

Appendix A.1 contains the user and the developer manual for the software developed during this study.

Chapter 2

Background

This chapter presents the background regarding software attestation required for this research. The first section is an analysis of the ASPIRE attestator's functionality. The second section contains a summary of the previous work related to the topic of software attestation. The last section exposes some security problems related to the subject.

The ASPIRE project is a framework that offers a complete protection suite. Given the source code for a program, it is possible to create a protected binary. During this work, only the remote attestator functionality has been considered, but the framework also implements some other interesting methods. The complete project can be found on GitHub[1]. A comprehensive description of the protection techniques of ASPIRE can be found in its reference architecture guide [3]

2.1 The ASPIRE remote attestator

As already mentioned, the ASPIRE project offers different protection mechanisms. The following security analysis concerns only the remote attestator's functionality. The idea behind a remote attestator is to perform an integrity check of the code running on an untrusted device. The device is considered untapered only if this control is passed. In the ASPIRE attestator, integrity is ensured by computing a checksum of the protected code's memory areas. The checksum is computed by code operating on the target device itself, but is checked by a trusted server. Figure 2.1 shows the general structure of the ASPIRE attestator.

The attestation service can be split into multiple components.

The server component

The server is the part of the attestator that is executing on a platform considered trusted. It can be divided into three subcomponents: the back-end database, the supervisor and the verifier.

- **The back-end database component.**

  The database holds a list of coupled nonce-checksums. At each nonce corresponds the correct computed checksum for the program the attestator has to check. The database has to be filled continuously, to avoid re-using the same nonce-checksum more than once.

- **The supervisor**

  The supervisor is responsible for periodically sending requests to the monitored devices to prove themselves. At each request, the supervisor sends to the device a different nonce, taken from a backend database it shares with the verifier.

[1] https://github.com/aspire-fp7/framework

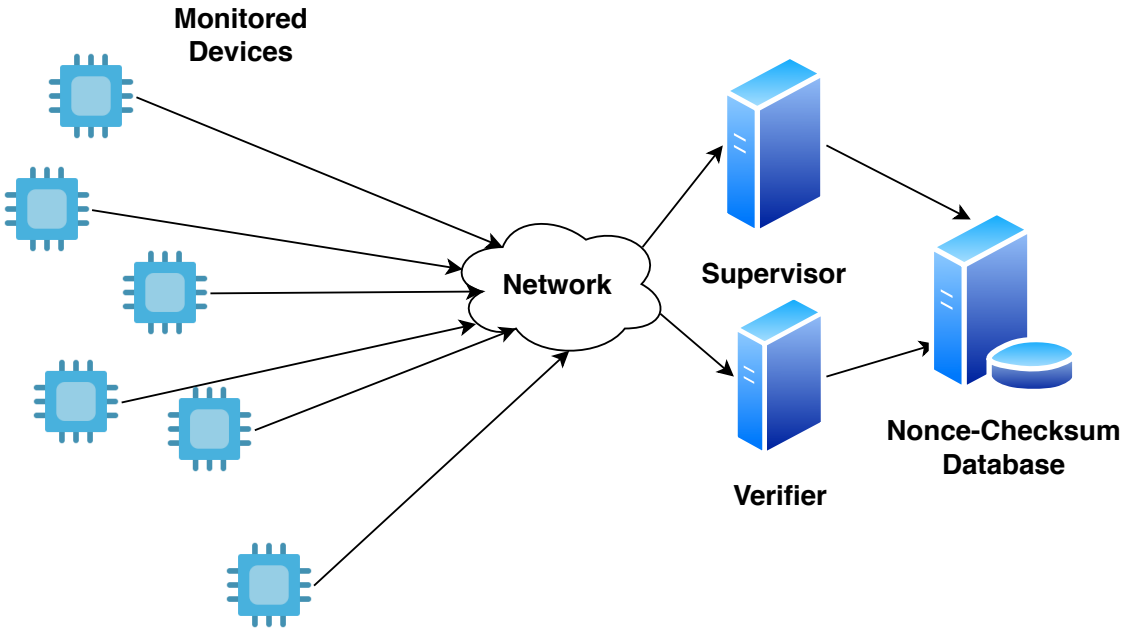

Figure 2.1: Remote attestator structure

- **The verifier**

  The verifier is in charge of receiving the computed checksum from the devices. To check if the hash is correct the verifier queries the backend database. If the checksum is incorrect, the verifier will mark the device as tampered. If the device is marked as compromised, its data should be treated as not trustworthy. In that case, the best option is to stop using it until the problem is resolved. In general, it is better not to stop the application on the device itself. That would be a clear signal to any potential attacker showing the compromise has been detected. Showing a security mechanism is activated might be beneficial for an attacker to understand the purpose of the attestation code.

The client component

The client component is the attestator code in execution on the device to monitor. The client receives a nonce from the supervisor. Based on the nonce, it computes a checksum of the memory areas it has to protect. Finally, it sends it to the verifier part of the server.

Since the client component is computing a checksum, the only controllable areas are the ones remaining constant. In particular, the attestator is designed to include areas inside the text segment, which is read-only.

The Figure 2.2 is a graph showing an overview of the whole system functionality:

Communication

For simplicity, the communication part of the attestator is not described. There are two reasons. First, not every device might offer the same protocols for communication. Second, the attestator uses a protocol called WebSocket Protocol. It is designed just for the project itself. If the protocol needs to be changed, it needs to be changed at both the endpoints, which is not a trivial task. Since the objective is an analysis of the attestator portability, the communication part is not going to be considered. It is assumed that it might be different based on the requisites and the support the used system offers.

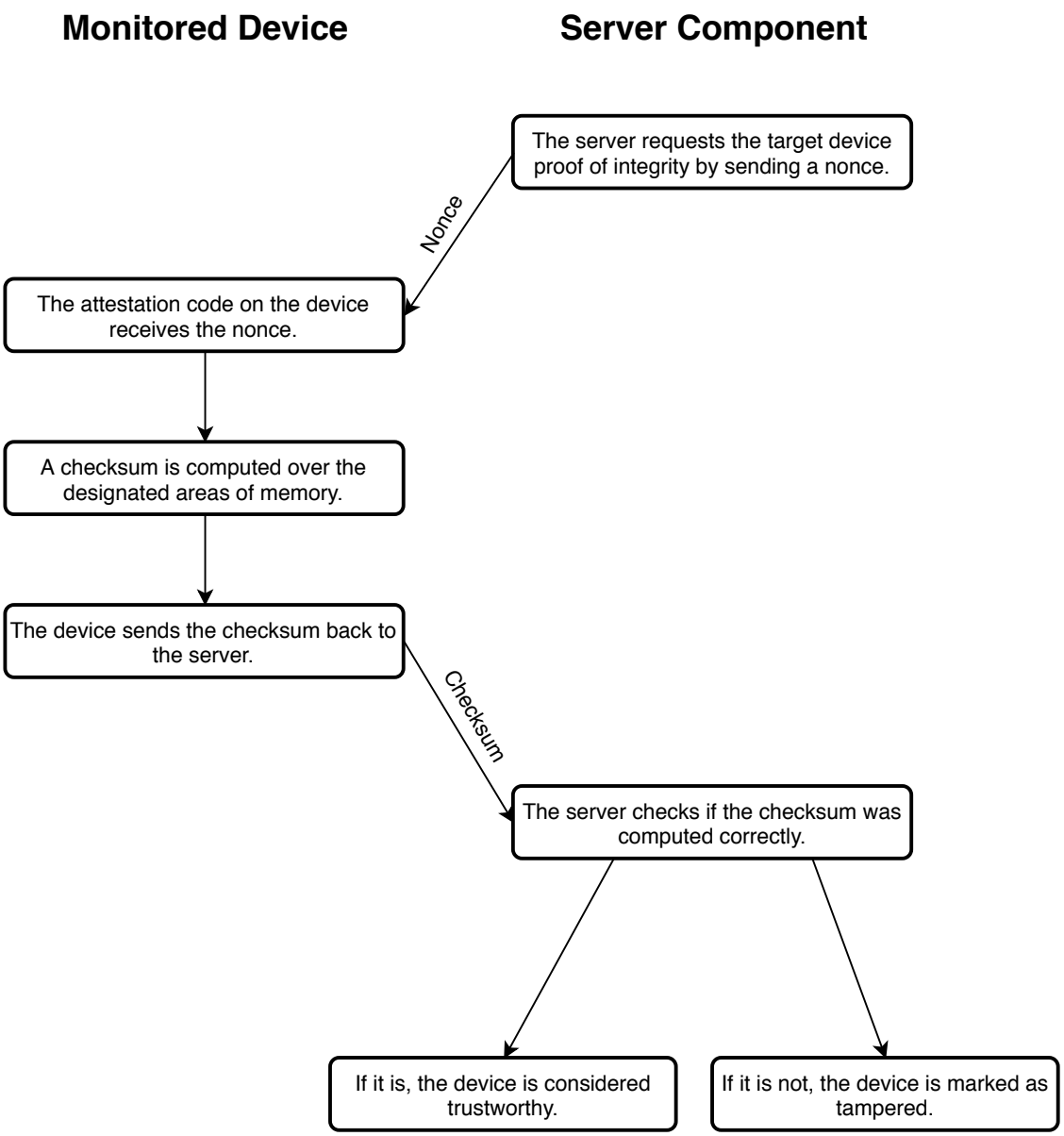

Figure 2.2: Remote attestator graph overview.

Creating a different hash at every request

The purpose behind the attestator is to verify if certain areas of memory have been tampered with. To achieve this task, the attestator on the device itself computes a hash over these areas of memory.

Using a checksum to ensure an area of memory is unmodified leads to a problem. If the application is always performing the identical computation on a certain area of memory, the checksum is always going to be the same. This exposes the attestator to replay attacks.

To overcome this weakness, the data taken from the monitored areas of memory is not extracted always in the same order. The order is decided based on the nonce the attestator is retrieving from the server. At each nonce corresponds a different order. For each different nonce, the checksum is going to be different and not trivial to compute, even if the areas of memory being checked are the same.

The method for getting a different order of bytes extraction from each different nonce is called *random walk*. The source code for this method can be found in the ra_data_preparation.c file (the

full source of the project can be found on GitHub[2]).

The attestator is going to read the data based on the random walk order. Then it is going to compute a hash of it using a method specified in the configuration. The supported methods in the ASPIRE source are blake2, md5, ripemd160, sha1, and sha256. The objective is not to use the hash with most strength but to use an algorithm that can be supported even by not so powerful devices.

Memory areas

The nonce is provided by the server at each computation. The client component also needs information about the areas of memory it has to monitor. This information is memorized inside the data segment in a structure called blob. At startup, the attestator will parse and load the information it needs from the blob and it will wait for the server's nonce.

This data is stored in a table of memory regions. This table is composed of a linked list of struct `memory_area_t`.

Each memory area contains a linked list of an arbitrary number of memory blocks (`memory_block_t`). A memory block is a single continuous portion of memory.

Snippet 2.3 is taken from *ra_memory.c* source file. It shows the definition of memory areas and memory blocks.

Every memory block has an offset and a size, designating where the area of memory is located. The address of the single memory block is computed summing the offset of the block with a fixed value. This value matches to the text segment offset and it is contained in the `base_address_NAYjDD3l2s` symbol, as defined in *ra_memory.c*.

$$b_{\text{address}} = b_{\text{offset}} + T_{\text{offset}}$$

Where b_{address} is the address of the memory block, b_{offset} is the offset of the memory block and T_{offset} is the text segment offset.

In pseudocode:

```
memory_block_address = block->offset + base_address_NAYjDD3l2s
```

The reason for doing this computation instead of having the address of the block directly memorized is ASLR. If ASLR is enabled and the executable supports it, the text segment's address will be different at each run. This would make impossible for the attestator to locate the correct regions of memory to protect. An explanation of how ASLR functions can be found in Section 5.6.

2.2 Related work on software attestation

Several articles propose a mechanism of attestation based purely on software. This section is an analysis of the most important ones.

Seshadri *et al.* proposed a technique called SWATT that performs software attestation in embedded devices [4]. It uses a pseudo-random memory traversal to compute an ad-hoc checksum function. It is designed for 8-bit architectures and it integrates time control to ensure the checksum function has not been tampered with. Some flaws in SWATT were pointed out in an article from Castelluccia *et al.* [5]. This article proposes two methods to bypass the attestator security mechanism. The first one relies on moving malicious code between executable and not-executable

[2] https://github.com/aspire-fp7/framework

```
struct memory_area_t {
    uint16_t label;
    /* Total number of blocks */
    uint32_t total_blocks;
    /* Total memory area size */
    uint32_t total_size;
    /* Pointers to the head and the tail of the memory blocks' linked list */
    blocks_list_item_ptr blocks_head;
    blocks_list_item_ptr blocks_tail;
};

struct memory_block_t {
    /* Memory offset of the block from the text segment offset */
    uint64_t offset;
    /* Size of the block */
    uint32_t size;
};

typedef struct bai *blocks_list_item_ptr;
typedef struct bai {
    RA_memory_block block;
    /* Pointers to the previous and the next item of the linked list */
    blocks_list_item_ptr previous_item;
    blocks_list_item_ptr next_item;
} blocks_list_item;

/* Uninitialized base address */
uint64_t base_address_NAYjDD312s = 1;
```

Figure 2.3: Listing of memory areas and memory blocks structures from *ra_memory.c.*

memory during the attestation procedure. To achieve such a goal, a technique based on *Return-Oriented Programming* is used. The second one uses code compression to free enough space to hide malicious code. This paper was then contraditicted by the creators of SWATT in another article [6] by pointing out some wrong assumption in Castelluccia's research.

Pioneer is another implementation of attestator [7] by Seshadri *et al.* As SWATT, it also performs a time-based check over the code memory. It was implemented for the x86 architecture and it is aimed to guarantee untampered code execution.

Mobile Guards is a proposition by Grimen *et al.* [8]. A mobile guard is an attestator downloaded directly into the program and with a restricted lifetime. Understanding the attestation procedure is made difficult by creating each guard differently.

SBAP is a software-based attestation protocol designed for devices with limited resources [9]. It fills all the available unused memory with pseudo-random values and then it computes a checksum over the whole data and code memory.

SCUBA is a protocol based on ICE (Indisputable Code Execution) [10]. It is aimed at providing a guarantee of untampered code execution and to provide stronger protection than SWATT. The idea behind it is to create an *untampered execution environment* in which the protected code can run without the possibility of interruption. The SAKE protocol is also designed based on ICE [11]. It uses the ICE primitive, to ensure the execution of the SAKE protocol itself is untampered.

An interesting literature review about software tampering detection can be found in an article from Abdo Ali Abdullah Al-Wosabi [12]. Other articles of particular interest are "Principles of Remote Attestation" [13], "A Large-Scale Analysis of the Security of Embedded Firmwares" [14],

"Attacking and Defending Networked Embedded Devices" [15] and "Remote Software-Based Attestation for Wireless Sensors" [16].

Information regarding the ASPIRE project can be found in an article about reactive attestation from Alessio Viticchié *et al.* [17].

2.3 Open issues

From the literature, it is possible to draw some interesting points concerning some open issues.

The first point regards the hash function. Some papers propose the use of not-standard hashing algorithms. The main reason is that some devices might not have the computational power required to perform complicated hashing algorithms. In general, it is not important to use the most collision-resistant hash. In case a collision is found for a certain nonce, the next nonce will lead to a different computation and will not match.

If the used hash algorithm is not strong enough though, an attacker might be able to find a collision for every nonce the attestator is receiving. In that case, the attestator will fail to detect the tampering. It is important to note that the attacker might have more computational power than the one on the device itself. For this reason, the chosen hash algorithm should be selected carefully to avoid this type of attack.

Another interesting point is the use of time-based checks. This seems like a solid technique but it has some vulnerabilities. Computational time critically depends on the device performing the calculation. As already mentioned, an attacker does not have this limitation. By using a more powerful device, it is possible to execute tampered code in the same timelapse as the original. The attestator might be modified to always perform the correct checksum, even if that alteration slows the procedure down.

Multiple articles try to address the problem of integrity regarding the attestator itself. There are different potential attack vectors against a memory attestator. A simple attack consists in preventing the attestator from starting completely. If that is the case, the server attestation component will not be able to communicate with the device. This type of tampering is easily detectable since the server will not receive any checksum from the device.

A more stealthy attack is accomplished by creating a copy of the protected memory. The attestator checks a confined area of memory inside the program it is protecting. This area of memory is defined inside the attestator data. By tampering this data is possible to redirect the attestation procedure to an uncorrupted copy of the real code. The checksum will then be always computed on an untampered area of memory. Once the attestator is redirected to a copy, the real code can be modified without occurring in detection. There are different ways to prevent this attack. If the data regarding the verified memory areas is obfuscated, tampering it might be impossible. Another prevention technique, as proposed in SBAP [9], is to fill the entire memory with known values. If the check is performed on the entire memory, any modification will be detected. It might still be possible to hide the copy inside a segment that is not controlled, as the data segment.

The article by Castelluccia *et al.* [5] proposes some interesting attack vectors. Even if the whole text segment is controlled by the attestator, it might still be possible to hide the attestator inside the not-executables segments during the attestation and then re-insert it again once the procedure is finished. To move the attestator the authors suggest a *ROP* based technique. There are two ways to detect this type of attack. The simplest way is to ensure the memory at disposal is filled by the protected program. In that scenario, the malicious code will not have enough space to hide inside the memory. Another solution is to check control-flow integrity. This means that also the return addresses on the stack have to be verified.

Some of these problems do not have a solution yet. Most of the protection offered by software attestation is strictly dependent on the concept of security through obscurity. As long as the attestation functionality is hidden and difficult to reverse, the time needed to perform a successful attack might cause a potential adversary to give up. Obfuscation has an important role in this regard. If the attestator is obfuscated, most of the attacks discussed above will require a more significant effort.

Chapter 3

Problem Statement

This thesis aims at analyzing the applicability of the ASPIRE remote attestator's procedure to embedded devices. In particular, there are two topics of interest. The first one concerns about extending the portability of the mentioned attestator by removing some of its dependencies. The second one is about to what extent the attestator can be ported to embedded platforms.

The ASPIRE attestator is a module inside the ACTC (ASPIRE Compiler Tool Chain). It is strictly correlated to the framework itself and it is dependent upon some of its components. This correlation creates a problem in the analysis. If the attestator is part of a bigger project, the portability of the entire project will affect the one of the module itself. Chapter 4 shows the requirements the framework poses over the attestator. The most important ones are the following:

- The framework is really extensive and it might not fit into low-end embedded devices with limited resources. This problem is mitigated by the fact that the applied protection mechanisms are highly customizable. Thus, it is possible to reduce the footprint of other protection methods.
- The Diablo toolchain[1] is used by the ACTC as a linker and it is also responsible for the obfuscation of the binary itself. The framework needs a linker because it takes the source code of the program as input and it outputs a fully functional and protected binary. The Diablo toolchain forces the attestator to be strictly integrated with the framework. In fact, the attestator requires the insertion of some information after the linking process. This insertion is also performed by Diablo.
- The OpenSSL library[2] is used for cryptographic tasks, such as performing encrypted communication and computing the checksum over the protected memory areas. This library is not supported by every embedded device.

Extracting the attestator from the entire ASPIRE toolchain is the best option to correctly analyze its potential applicability. Also, since extending the portability of the attestator is highly desirable, this process entails some advantages. This extraction is not a trivial task since the attestator is highly integrated within the project itself. Some modifications are needed to construct a working attestator without the use of the framework. The attestator relies on the Diablo toolchain to insert information about the attestation procedure in the binary. This information is inserted into a portion of the data segment after the linking process. An analysis of the inner functionality of the executable format is required to create a substitute. The ACTC uses the standard executable format for Linux-like platforms: ELF. Chapter 5 contains a study of this format. It is aimed at understanding how ELF files are executed and how to correctly insert the information needed by the attestator. In particular, the most critical part of this patching

[1] https://diablo.elis.ugent.be

[2] https://www.openssl.org

procedure is the mechanism of ASLR (Address Space Layout Randomization) and the relocation process correlated to it. Also, a good understanding of the ELF's structure, both in memory and on disk is necessary. Even if this chapter does not concern with the attestation process, it is required to complete the examination on the attestator's portability.

After this analysis, creating a custom script to substitute the Diablo toolchain is attainable. Chapter 6 describes the process used to extract the attestator from the framework. With a standalone attestator, it is possible to perform further examinations. Two problems still need some research.

The first one is about the OpenSSL library and its portability implication. As already mentioned, the OpenSSL library is in charge of cryptographic operations. There are other options to perform the same tasks. Two potential substitutes were found during this study: Mbed TLS[3] and WolfSSL[4];. These two libraries both offer cryptographic APIs. Chapter 8 contains an analysis of the support that the most used operating systems for embedded devices have in regards to these libraries. This examination deals with the question posed at the beginning of the study, about the applicability of the standalone ASPIRE attestator to embedded platforms.

The second problem left undiscussed is the use of obfuscation techniques to protect the program. The attestator was isolated from the framework, thus the component responsible for this protection mechanism was removed. Since the attestator was left unprotected from obfuscation, some alternatives are needed to replace the Diablo toolchain's task. These options are discussed in Chapter 7. This analysis concerns the effectiveness of the obfuscation techniques offered by some open-source tools. An in-depth examination was performed to present the best alternatives.

[3] https://tls.mbed.org/

[4] https://www.wolfssl.com/

Chapter 4

Examination of the attestator requirements

This chapter focuses on the ASPIRE attestator's requirements. It outlines the most significant dependencies the ASPIRE framework poses on the attestator module. This examination is crucial to extend the attestator's portability. Once the critical requirements are analyzed, it is possible to make an effort to remove them.

There are two factors this examination has to consider:

- The hardware requirements.

  The attestator cannot run on any device. There are some basic hardware specifications that, if unsatisfied, will make portability impossible.

- The software requirements.

  The attestator also depends on some third-party software. If that software is not portable to a specific device or operating system, adapting the attestator will require some changes.

The following analysis concentrates mainly on software requirements. Hardware requisites are discussed briefly, but, in general, the attestator should be adaptable to almost every not-low-end embedded platform.

4.1 Hardware

The following is a list of the key hardware elements that have to be considered for the attestator functionality.

- Processor architecture.

  Some architectures might have only an 8-bit or 16-bit processor. In that case, adapting such a complicate attestator can be a problem. On 32-bit processors, the attestator should be easily portable.

- Computing power.

  The attestator requires a certain amount of computing power, based on the frequency of the check and on the hash algorithm it uses. If the device is already under considerable load, adding the attestation code can cause serious performance degradation.

  It is important to notice that, since the attestator does not need to run continuously, its computational requirement can be adapted by changing the frequency of the attestation procedure.

- Memory size.

  The memory usage can be an issue. The attestator requires some memory inside the application it is protecting. The company using it has to consider this fact in the choice of the hardware, especially on low-end devices.

- Networking functionality

  The attestator needs to connect with the trusted server to function. This means support for some kind of communication is required. Only a few embedded systems do not offer this functionality.

- Multithreading support

  Since the attestator is launched as a new thread by a program constructor, if threading is not supported, the attestator will not work.

4.2 Software

The software dependencies of the attestator contribute more to create portability issues. As already mentioned, the attestator has some third-party software dependency. Some components are portable to almost any platform, some others are not. The study conducted in this thesis attempts to remove some of those dependencies. The following sections outline the most important ones.

Programming language

Since the ACTC is written in C, the attestator supports only programs developed in this language. Porting it to different languages will require a complete re-writing of its source code.

Diablo toolchain

Diablo (Diablo Is A Better Link-time Optimizer) is a "retargetable link-time binary rewriting framework"[1]. Diablo takes the part of the linker in the compilation process. It takes object files and libraries as input and it outputs a statically linked program. It is necessary for the ASPIRE project to easily customize the output binary. It is in charge of two important tasks concerning the attestator. It is used by the framework to patch the blob (see Subsection 2.1) with the correct information for the attestation component. It is also responsible for the obfuscation of the program itself. To release the attestator from this constraint a tool that substitutes Diablo's job is required.

The blob patching is dealt with in Chapter 6. By using a custom script it is possible to insert data into the blob after compilation.

Some alternatives regarding obfuscation are examined in Chapter 7.

Libwebsocket and Curl

Libwebsocket[2] and Curl[3] are two libraries used within the attestator for the communication with the trusted server. The WebSocket Protocol, already mentioned in Section 2.1, is developed using these two libraries. Some embedded devices support these two libraries, others offer their APIs for the IP stack functionality and web communication.

[1] https://diablo.elis.ugent.be

[2] https://libwebsockets.org/

[3] https://curl.haxx.se/libcurl/

In this study, the communication part is not taken into consideration. Potentially any device with network capabilities is adaptable to the remote attestation scheme. If those two libraries are not supported, some major changes will be required for the connection procedure.

OpenSSL library

From the OpenSSL homepage[4]:

> OpenSSL is a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library.

This library is comprehensive and it might not fit into some low-end devices with limited memory resources. It is usable on some devices, but there are some alternatives.

The attestator uses OpenSSL to perform the cryptographic operations it needs. In particular:

- The cryptographic computation of the memory hash;
- The random walk logic.

Conclusions

This analysis shows that the most crucial dependencies of the ASPIRE attestator are related to software. In particular, the most important ones are the Diablo toolchain and the OpenSSL library. In the following chapters, an effort towards removing these two requirements is made.

[4] https://www.openssl.org

Chapter 5

Analysis of the ELF format

This chapter is an analysis of the ELF executable format's internal functionalities. The purpose of this study is to understand the inner-working of this file format and of its major structures and mechanisms. This whole chapter is based on a study that has been made to learn how executables are being run in a Unix-like environment. This information will then be employed to extract the attestator from the ASPIRE framework. To do so, a script that substitutes the Diablo toolchain has to be constructed. This script will have to analyze the executable and modify its relevant values on disk. When the program executes, these modifications will have their effect on volatile memory as well. Thus, knowing the ELF file format is essential.

5.1 The ELF file format

ELF stands for Executable and Linkable Format. Thanks to its design and portability, it is a common standard file format in Unix environments. The ELF standard defines a set of "binary interface definitions" that works across different operating environments [18].

The ELF format supports different address sizes and endianness. In this way, it can be easily adapted to different architectures and operating systems. It is so used in Linuxes and BSD variants, that the kernel boot image itself is of ELF type.

In the following paragraphs, the ELF file format specifications are examined. The book "Learning Linux Binary Analysis" from Ryan O'Neil [19] gives a really useful insight into the ELF format. The Linux ELF manual page will be used as a reference[1].

Every ELF file is made up by the following structures:

1. The ELF header;
2. Program header table, which is a table of the program headers;
3. The ELF file data itself;
4. Section header table, which is a table of the section headers.

5.2 ELF headers

ELF header

The ELF header contains general information about the file itself. It can be in 2 forms: `Elf32_Ehdr` or `Elf64_Ehdr`. Figure 5.1 shows the ELF header definition.

[1] https://linux.die.net/man/5/elf

```
#define EI_NIDENT 16
typedef struct {
        unsigned char e_ident[EI_NIDENT];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    Elf[32/64]_Addr e_entry;
    Elf[32/64]_Off e_phoff;
    Elf[32/64]_Off e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} Elf[32/64]_Ehdr;
```

Figure 5.1: ELF header.

From the man elf(5) page[2]:

e_indent This field contains the ELF header magic, which provides some information about the file. It always starts with 0x7f, 0x45, 0x4c, 0x46 (0x7f, E, L, F) which identifies the file as of ELF type. It also contains some other information such as the architecture (32 or 64 bits), the endiannes, the version, the operating system and the ABI version.

e_type This field describes what type of ELF the current file is. There are different possible ELF file type:

ET_NONE Type not known or not defined yet.

ET_EXEC This is an executable file. This kind of files can be loaded into memory and executed.

ET_REL This is a relocatable file or generally called an object file. Relocatable files contain portions of code. Multiple object files can be linked together to generate a complete executable file.

ET_CORE This is a core dump file. These files are generated when a program crashed or when a SIGSEGV is delivered. They can be used to reconstruct the crash and determine what went wrong during the program execution.

ET_DYN This is a shared object file. These are libraries that can be loaded and linked to a program at runtime.

This thesis mainly concentrates on files with type ET_DYN or ET_EXEC. Standalone executables can be of one of these two types. If the program is marked ET_EXEC, it is not compatible with ASLR. If it is marked as ET_DYN, it supports ASLR. This difference will be significant during the creation of the patching script that substitutes the Diablo toolchain.

e_machine This field specifies the supported architecture for this file.

e_version This field specifies the object file version. For now, only the value 1 is accepted.

e_entry This field gives the virtual address of the entry point of the program. If the file is not an executable it holds a value of 0.

[2] https://linux.die.net/man/5/elf

e_phoff This field holds the program headers table offset in the file.

e_shoff This field holds the section headers table offset in the file.

e_flags This field holds some flags associated with the object file.

e_ehsize This field specifies the ELF header's size (in bytes).

e_phentsize This field contains the size of one program header entry. All the program headers are of the same size.

e_phnum This field contains the number of program header entries.

e_shentsize This field contains the size of one section header entry. All the section headers are of the same size.

e_shnum This filed contains the number of section header entries.

The readelf[3] tool can be used to analyze the header of an ELF file. An example usage of readelf to analyze the ELF header of an executable compiled as not position-independent can be found in the Listing B.1 in the Appendix B. The values of the various fields analyzed in this section can be seen in that snippet.

As the name itself says the ELF format provides two interfaces through which the file can be looked at: the executable interface and the linking interface. The first is defined in the program headers and the second one is defined in the section headers. This means the ELF executable can be inspected through two different views. These two views describe the same data, but they provide different pieces of information. In general, program headers offer information about how the file should be loaded in memory at runtime. Section headers contain data for linking and relocation purposes.

The following two sections present the structure of both program and section headers.

Program header table

The program headers table contains a list of program headers. Every program header describes a piece of data called segment. Each segment is a block of data that might be loaded in memory at runtime. Program headers describe how the raw executable information on disk should be ported onto memory [19]. They also contain information about what permissions every segment should be loaded with and at which address. The kernel parses the program header table when a program is launched. To locate the program headers table, it uses the information contained in the ELF header.

Listing 5.2 shows the program headers for a 32 or 64 bit ELF file. The two variants are very similar. The description of each field is extracted from the man elf(5) page[4]

p_type This field contains the segment type.

p_offset This field contains the segment offset in bytes inside the file.

p_vaddr This field contains the segment virtual address at runtime.

p_paddr This field contains the segment physical address.

p_filesz This field contains the size of the segment in the file.

p_memsz This field contains the size of the segment when loaded in memory.

[3] https://linux.die.net/man/1/readelf

[4] https://linux.die.net/man/5/elf

```
typedef struct {
    uint32_t p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    uint32_t p_filesz;
    uint32_t p_memsz;
    uint32_t p_flags;
    uint32_t p_align;
} Elf32_Phdr;

typedef struct {
    uint32_t p_type;
    uint32_t p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    uint64_t p_filesz;
    uint64_t p_memsz;
    uint64_t p_align;
} Elf64_Phdr;
```

Figure 5.2: ELF program header.

p_flags This field contains the segments flags such as executable, readable and writable.

p_align This field contains the segment alignment in memory.

The `p_type` field can be of various different types, but the most commons are:

PT_DYNAMIC Dynamic segments contain information regarding dynamic linking. The dynamic linker is responsible for the load and the binds of all the shared libraries needed for the program to execute. This type of segment holds information as a series of structures of type `Elf[32/64]_Dyn`.

PT_LOAD This means that the segment is loadable and it is going to be mapped into memory at runtime. Generally an executable will contain two PT_LOAD segments: the text segment and the data segment. The text segment contains the program code and it is usually mapped in memory with permission read + execute (PF_X | PF_R). The reason is that typically code is not modified at runtime, so it is better to protect this segment from writing. The data segment contains the program data and the dynamic linking information (if the program is dynamically linked). Normally the program data has to change during execution, so this segment is marked as read + write (PF_R | PF_W). In general, if an executable has a segment with unusual permission such as read + write + execute, it might have been tampered with, except for particular cases.

This type of segment will be significant in the following chapters. The data segment and the text segment can be distinguished by their flags. The patching script will use this method to find the position of each of these segments.

PT_NOTE The note segment may contain information about the system the program can be executed on or the vendor of the program itself. This segment is not useful for the executable to run and it can be removed without affecting the program behavior.

PT_INTERP This type of segment contains a pointer to a null-terminated pathname to be invoked as an interpreter. This segment type is useful only for executable files.

PT_PHDR The program header segment contains information about the program header table itself.

GNU_STACK This header is used to store informations about the stack.

GNU_EH_FRAME This header is used to store the exception handlers.

GNU_RELRO This segment is part of an "exploit mitigation technique called Relocation Read-Only (RELRO)" [5]. It maps the Global Offset Table as read-only, preventing a GOT overwrite at runtime. An explanation about the functionality of the GOT can be found in Section 5.5.

To take a look at the segments of an ELF file, the readelf tool with the -l flag can be used. Listing 5.3 shows the readelf output (see Listing B.2 in the Appendix B for the full output).

```
$ readelf -l hello_world

Elf file type is EXEC (Executable file)
Entry point 0x401040
There are 11 program headers, starting at offset 64

Program Headers:
  Type Offset VirtAddr PhysAddr
               FileSiz MemSiz Flags Align
  INTERP 0x00000000000002a8 0x00000000004002a8 0x00000000004002a8
               0x000000000000001c 0x000000000000001c R 0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD 0x0000000000001000 0x0000000000401000 0x0000000000401000
               0x00000000000001ad 0x00000000000001ad R E 0x1000
  LOAD 0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
               0x0000000000000220 0x0000000000000228 RW 0x1000

[...]
```

Figure 5.3: ELF segments listing with the readelf tool.

In the example, it is possible to observe the INTERP segment containing the name of the standard interpreter */lib64/ld-linux-x86-64.so.2*. There are two loadable segments as already explained. The first one (RE) is the text segment and the second one (RW) is the data segment. Another thing to note is the alignment of the two loadable segments, which corresponds to a memory page size. In 32 bit architectures, the page size is typically 4096 bytes (0x1000), whereas in 64 bits architectures it is 2 MB (0x200000 bytes).

The section header table

The section headers table contains an array of section headers. Every section header represents a section in the file. The section headers offer a view of the file used for linking and relocation. Since it is not necessary at runtime, it can be "stripped" away from the binary and the program will still work. A section header can have two formats, based on the system architecture. The Listing 5.4 shows the section headers for 32 and 64 bits systems.

Taken from the man elf(5) page[6] and from "Executable and Linkable Format (ELF)"[7]:

[5] https://medium.com/@HockeyInJune/relro-relocation-read-only-c8d0933faef3

[6] https://linux.die.net/man/5/elf

[7] http://www.skyfree.org/linux/references/ELF_Format.pdf

```
typedef struct {
    uint32_t sh_name;
    uint32_t sh_type;
    uint32_t sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    uint32_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint32_t sh_addralign;
    uint32_t sh_entsize;
} Elf32_Shdr;

typedef struct {
    uint32_t sh_name;
    uint32_t sh_type;
    uint64_t sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    uint64_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint64_t sh_addralign;
    uint64_t sh_entsize;
} Elf64_Shdr;
```

Figure 5.4: ELF section header.

sh_name This field holds the name of the section itself as an index into the string table section.

sh_type This field specifies the section's type.

sh_flags This field specifies flags related to the section.

sh_addr This field contains the address in memory of the section. If the section is not loaded in memory at runtime this field holds 0.

sh_offset This field holds the offset in bytes from the beginning of the file of where the section is.

sh_size This field specifies the section's size (in bytes).

sh_link This field holds a section header table index link which is correlated to this section.

sh_info This field contains additional information about the section. The interpretation depends on the section type

A list of the sections present in an ELF file using can be retrieved using the readelf tool with the -S flag. An example can be found in Listing B.3 in the Appendix B.

Section header types

There are a lot of different section types. The following is an explanation of the most important ones extracted from the man elf(5) page[8].

[8] https://linux.die.net/man/5/elf

.text This is the code section. It holds the instructions of a program.

.data This section contains writable data, such as initialized global variables.

.rodata This section contains read-only data. String literals are stored in this area.

.bss This section contains uninitialized data of the program memory image. Since this data is not initialized, only information about how much space this data takes is needed. When the program is loaded, such space is reserved for these variables and it is initialized to zeros.

.plt This section is the procedure linkage table (PLT). The PLT is described in Section 5.5.

.got This section contains the global offset table (GOT). The GOT works together with the PLT to provide dynamic resolution of libraries at runtime.

.dynsym This section contains a table of dynamic symbols.

.dynstr This section contains the strings needed the for dynamic linking process, typically associated with symbol table entries. All the sections marked with 'str' hold a series of null-terminated strings.

.rel.* These sections contain information on how to perform relocations. They are used at runtime to fix or modify the program in memory based on its memory position.

.hash These section contains a symbol hash table.

.symtab This section contains a symbol table.

.strtab This section contains the string table (as the .dynstr section) referred by the .symtab.

.shstrtab This section, like all the other sections marked with "str", contains a list of strings. In this case, those strings are the name of each section (for example .text, .data...).

.ctors and .dtors These sections contain function pointers to the initialization and the finalization code respectively.

The execution and the linking views

Having seen both the program header table and the section header table, it is time to look at the two views they offer of the ELF file itself. Figure 5.5[9] shows the program as seen by the linking view (through the section header table) and by the execution view (through the program header table). Sections and segments can clearly overlap one another. Typically a segment can contain one or more sections and not vice-versa. It is possible to view this overlap on any ELF file by using the readelf tool with the -l option. An example output can be found in Listing B.4, Appendix B.

5.3 Symbols and symbol tables

A symbol is a reference to an object present inside the code. An object, for instance, can be a function or a variable.

There are two symbol tables: the .symtab and the .dynsym. The .symtab is the main symbol table, while the .dynsym contains only a smaller set of symbols needed for the dynamic linking process. Thus, the .dynsym table will contain copies of information already present in the .symtab table.

[9] https://wiki.aalto.fi/download/attachments/55374575/elf.jpg?version=1&modificationDate=1296509659000&api=v2

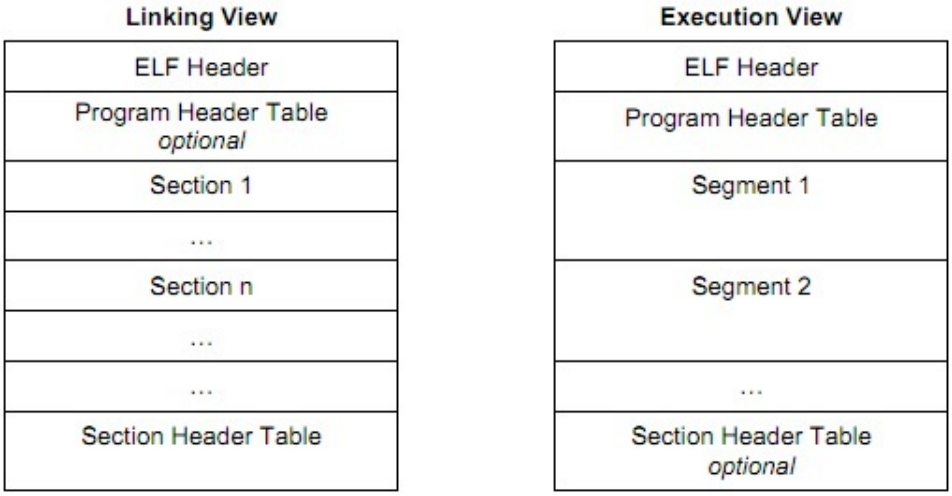

Figure 5.5: Linking and Execution views.

This seems redundant, but only the .dynsym table is loaded in memory at runtime. The .symtab can be removed from the binary without affecting its functionality. To distinguish between what sections are going to be loaded in memory at runtime, it is possible to check the presence of the ALLOC (A) flag.

Both those tables contain a series of entries of type `Elf32_Sym` or `Elf64_Sym`, as shown in Listing 5.6. By using symbol entries, information about a particular symbol can be extracted starting from its name.

```
typedef struct {
    uint32_t st_name;
    Elf32_Addr st_value;
    uint32_t st_size;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t st_shndx;
} Elf32_Sym;

typedef struct {
    uint32_t st_name;
    unsigned char st_info;
    unsigned char st_other;
    uint16_t st_shndx;
    Elf64_Addr st_value;
    uint64_t st_size;
} Elf64_Sym;
```

Figure 5.6: ELF symbol entries.

5.4 Relocations and the relocation process

"Relocations are entries in binaries that are left to be filled later" [20], either at linking time by the linker or at runtime by the dynamic linker. A relocation entry looks like the Listing 5.7 or it can also require an addend, as in Listing 5.8.

```
typedef struct {
    Elf[32-64]_Addr r_offset;
    uint[32-64]_t r_info;
} Elf[32-64]_Rel;
```

Figure 5.7: ELF relocation entry without addend.

```
typedef struct {
    Elf[32-64]_Addr r_offset;
    uint[32-64]_t r_info;
    int[32-64]_t r_addend;
} Elf[32-64]_Rela;
```

Figure 5.8: ELF relocation entry with addend.

The elf(5) man page[10] provides a precise definition of relocation:

> Relocation is the process of connecting symbolic references with symbolic definitions.

A relocation entry contains the information required by the linker to modify the binary content in memory. A good example of relocation is contained in the book by Ryan O'Neil [19]. Snippet 5.9 shows the example in action.

In the example, the program listed at line 2 is compiled. The call to the function `foo` is not resolved yet, as the linker does not know at what address this function is going to be inside the program. The objdump output at line 18 shows indeed that the call to `foo` is filled with zeros. In the linking process, the call is patched by GCC, based on the relocation entry seen in the output from readelf on line 27. Listing 5.10 presents the disassembled program after the linking process.

The linker patched the relocation at offset 0x40110b to point to the correct location of the function `foo`. To compute the value of 0x00000003 (displayed at line 8 in little-endian), the following formula is applied:

$$s_{\text{value}} + s_{\text{addend}} - S_{\text{offset}} = s_{\text{offset}} \tag{5.1}$$

Where s_{value} is the symbol value, s_{addend} is the addend field of the symbol, S_{offset} is the section offset and s_{offset} is the computed offset value.

For example in this case:

$$0x401113 - 0x4 - 0x40110c = 0x3 \tag{5.2}$$

5.5 Dynamically linked binaries

By default, GCC compiles dynamically linked binaries, unless otherwise specified. Dynamically linked means that the binary itself does not contain all the functions it needs, but those functions have to be loaded and resolved at runtime. For example, if the program calls a function like `printf`, this function does not reside inside the program itself, but it is provided by some system library. Typically in GNU/Linux systems, this library is libc.so.6.

[10] https://linux.die.net/man/5/elf

```
1  $ cat main.c
2  #include "foo.h"
3
4  void main()
5  {
6      foo();
7  }
8
9  $ gcc -c main.   # Compile the program
10 $ objdump -d main.o # Disassemble the object file
11 main.o: file format elf64-x86-64
12 Disassembly of section .text:
13
14 0000000000000000 <main>:
15    0: 55 push %rbp
16    1: 48 89 e5 mov %rsp,%rbp
17    4: b8 00 00 00 00 mov $0x0,%eax
18    9: e8 00 00 00 00 callq e <main+0xe>
19    e: 90 nop
20    f: 5d pop %rbp
21   10: c3 retq
22
23 $ readelf -r main.o  # Show information about relocations
24
25 Relocation section '.rela.text' at offset 0x1d8 contains 1 entry:
26   Offset Info Type Sym. Value Sym. Name + Addend
27 00000000000a 000a00000004 R_X86_64_PLT32 0000000000000000 foo - 4
28
29 Relocation section '.rela.eh_frame' at offset 0x1f0 contains 1 entry:
30   Offset Info Type Sym. Value Sym. Name + Addend
31 000000000020 000200000002 R_X86_64_PC32 0000000000000000 .text + 0
```

Figure 5.9: ELF relocation entry example.

```
1  $ gcc main.o foo.o -o a.out -fno-pic   # Link the program with the object
       file containing foo()
2  $ objdump -d a.out      # Disassemble the executable
3  [...]
4  0000000000401102 <main>:
5    401102: 55 push %rbp
6    401103: 48 89 e5 mov %rsp,%rbp
7    401106: b8 00 00 00 00 mov $0x0,%eax
8    40110b: e8 03 00 00 00 callq 401113 <foo>
9    401110: 90 nop
10   401111: 5d pop %rbp
11   401112: c3 retq
12
13 0000000000401113 <foo>:
14   401113: 55 push %rbp
15 [...]
```

Figure 5.10: ELF patched relocation entry example.

Libraries are loaded in memory only once if required and are used by every program which needs them. If two programs have to call a function such as `printf`, they will both refer to the same memory where libc.so.6 is loaded. There are two advantages to this approach:

1. The library is only loaded in memory once. This means that once the library is loaded, every program can use its functionality without having to load the whole library again, thus reducing memory usage.
2. The library is one for all the systems, so if for example, a new patch is released for a particular library, it is substituted without having to recompile every binary in the system.

When a program is loaded into memory, it does not know the location of the libraries it needs. To resolve these addresses a method for dynamic relocation is needed.

One possible solution could be to have each library loaded at a known memory offset every time, just like ET_EXEC binaries. This might be fine for an executable because when a new executable is loaded in memory, it gets its own address space. For shared libraries, this approach might be not suitable. If every library has to be loaded in a certain address space, in the event this space is already taken, the library would not be able to be loaded at all. This would make the creation of new libraries "very difficult to maintain and prone to errors" [20] (every library would need its different address space assigned).

The other solution, which is the one typically used, is to have shared libraries compiled as position independent code (ET_DYN). Position independent code (PIC) is code that can be loaded anywhere in memory and it will still work. To achieve such thing, the offsets in the assembly are calculated relative to the instruction pointer, not to the instruction addresses. In this way, every shared library can be loaded at an arbitrary address. If shared libraries are loaded at unknown address each time, binaries need a way to resolve their positions at runtime.

GOT/PLT is the method used to resolve these types of relocations.

The GOT and the PLT

The global offset table is used to resolve data offset not known until runtime. The procedure linkage table is used to resolve external functions. An example found in an article from Ian Wienand [20] can be used to explain how the global offset table (GOT) and the procedure linkage table (PLT) work. Listing 5.11 shows the GOT functionality. The GOT is basically a table containing a series of addresses.

In this example, the program listed at line 2 is compiled as a shared library (line 8). The variable `foo` is supposed to be external to the program. The objdump output of the compiled executable shows that the returned value is taken from the location pointed by the content of 0x3fe8 (line 18). This is an offset in the .got section (line 29). When the program is loaded in memory, the dynamic linker will patch this GOT offset, to make it pointed to the correct location of the integer foo. To do so, the linker will extract the information it needs from the relocation entry in the '.rela.dyn' section (line 37)

The GOT is used to resolve variables. To resolve functions binaries use the PLT indirection mechanism. The example in Listing 5.12 is very similar to the last one, except the external symbol is a function and not an integer.

In this example, the `function` is not calling `foo` directly, because it does not know where it is. Instead, the call will jump to an address in the .plt section. From there a jump to an address contained in the .rela.dyn is taken, as it is possible to observe at line 27. The analysis with radare2 in Listing 5.13 shows that the jump at 0x1030 points to 0x1036, which is the next instruction. The next instruction will push the value of 0 on the stack and it will jump to another location in the .plt section (line 20 of Listing 5.12). This process is called lazy binding.

When the function `foo` is called for the first time, its relocation entry will point to the next instruction. The next instruction will set up a call to the dynamic linker. The dynamic linker will resolve the location of that function searching in the libraries the program requires. If it finds it,

```
1  $ cat shared.c
2  extern int foo;
3
4  int function() {
5      return foo;
6  }
7
8  $ gcc -shared -fPIC -o libtest.so shared.   # Compile the program as a shared
       library
9
10 $ file libtest.so
11 libtest.so: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
       dynamically linked,
       BuildID[sha1]=9d5c5b313f399173b4774353edbe8af02cefec60, not stripped
12
13 $ objdump -d libtest.so      # Disassemble the executable
14 [...]
15 00000000000010f5 <function>:
16     10f5: 55 push %rbp
17     10f6: 48 89 e5 mov %rsp,%rbp
18     10f9: 48 8b 05 e8 2e 00 00 mov 0x2ee8(%rip),%rax # 3fe8 <foo>
19     1100: 8b 00 mov (%rax),%eax
20     1102: 5d pop %rbp
21     1103: c3 retq
22 [...]
23
24 $ readelf --sections libtest.so --wide    # Show information about sections
25 [...]
26 Section Headers:
27   [Nr] Name Type Address Off Size ES Flg Lk Inf Al
28 [...]
29   [18] .got PROGBITS 0000000000003fd8 002fd8 000028 08 WA 0 0 8
30 [...]
31
32 $ readelf --relocs libtest.so    # Show information about relocations
33
34 Relocation section '.rela.dyn' at offset 0x3d8 contains 8 entries:
35   Offset Info Type Sym. Value Sym. Name + Addend
36 [...]
37 000000003fe8 000300000006 R_X86_64_GLOB_DAT 0000000000000000 foo + 0
```

Figure 5.11: GOT resolution for variables.

it will patch the relocation entry to point to the function `foo` itself, not to the next instruction. In this way, only the first call to `foo` is going to require the dynamic linker resolution. The next call will have the address already resolved.

5.6 Address space layout randomization

Address Space Layout Randomization or ASLR is a method used to limit the exploitability of memory vulnerabilities. ASLR randomly assigns the memory address space of every running process, by randomizing the base address of the executable and the positions of the stack and the heap. This can help preventing a vulnerability to be exploitable. Even if an attacker can

```
1  $ cat shared.c
2  int foo();
3
4  int function() {
5      return foo();
6  }
7
8  $ gcc -shared -fPIC -o libtest.so shared.c    # Compile as a share library
9
10 $ objdump -d libtest.so      # Disassemble the library
11 [...]
12 0000000000001020 <.plt>:
13     1020: ff 35 e2 2f 00 00 pushq 0x2fe2(%rip) # 4008
           <_GLOBAL_OFFSET_TABLE_+0x8>
14     1026: ff 25 e4 2f 00 00 jmpq *0x2fe4(%rip) # 4010
           <_GLOBAL_OFFSET_TABLE_+0x10>
15     102c: 0f 1f 40 00 nopl 0x0(%rax)
16
17 0000000000001030 <foo@plt>:
18     1030: ff 25 e2 2f 00 00 jmpq *0x2fe2(%rip) # 4018 <foo>
19     1036: 68 00 00 00 00 pushq $0x0
20     103b: e9 e0 ff ff ff jmpq 1020 <.plt>
21 [...]
22
23 0000000000001105 <function>:
24     1105: 55 push %rbp
25     1106: 48 89 e5 mov %rsp,%rbp
26     1109: b8 00 00 00 00 mov $0x0,%eax
27     110e: e8 1d ff ff ff callq 1030 <foo@plt>
28     1113: 5d pop %rbp
29     1114: c3 retq
30 [...]
31
32 $ readelf --relocs libtest.so     # Show information about relocations
33 [...]
34 Relocation section '.rela.plt' at offset 0x480 contains 1 entry:
35   Offset Info Type Sym. Value Sym. Name + Addend
36 000000004018 000300000007 R_X86_64_JUMP_SLO 0000000000000000 foo + 0
```

Figure 5.12: PLT-GOT resolution for functions.

overwrite the instruction pointer, ASLR will make direct jumps unreliable, because the addresses used by the program will be unpredictable.

It is important to note that ASLR does not resolve vulnerabilities. It is just a layer of protection that can be circumvented. An interesting article about ASLR security mechanism and some techniques to bypass it can be found on the Corelan Team website[11]. This article concerns about Windows operating systems, but the methods are applicable to Linux systems too.

To support ASLR the executable has to be compiled as Position Independent Executable (PIE). By default, GCC compiles as ET_DYN which support ASLR. ET_EXEC ELF files do not support it and so they will be loaded in the same address space each time.

[11] Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR, https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

```
[...]
[0x00001030]> pdf  # Prints the disassembly of the function at the current
    offset
/ (fcn) loc.imp.foo 6
| loc.imp.foo ();
| ; CALL XREF from sym.function (0x110e)
\ 0x00001030 ff25e22f0000 jmp qword reloc.foo ; [0x4018:8]=0x1036 ; "6\x10"
[0x00001030]> pxq @reloc.foo   # prints the value at the specified offset
0x00004018 0x0000000000001036 0x0000000000004020 6....... @......
[...]
```

Figure 5.13: Lazy binding.

The following is an example of how ASLR works.

```
#include <stdio.h>

void main() {
    printf("%p\n", main);
}
```

Figure 5.14: Example program that prints the address of the `main` function.

The program in Listing 5.14 prints the address of the `main` function. First, the program is compiled as PIE, which is the default for GCC. Then the program is launched multiple times. The output in Listing 5.15 shows how ASLR changes the function position in memory.

On each different run, the address of the `main` function is different. ASLR randomizes the address space of the whole text segment each time the program is loaded into memory. Using radare2 is possible to analyze the address of the `main` function on disk. This address only resembles the last part of the program's output. Listing 5.16 presents the analysis made with radare2. First the binary is open in line 1. Then it is analyzed in line 2 with the *aaa* command. Finally the functions are listed with the command *afl* at line 4. The offset 0x1135 at line 6 is not the real address at which the `main` function is loaded at runtime. ASLR will add a random value to this offset every time the program is executed.

If the code is compiled without PIE, ASLR does not work anymore. An application compiled without position independent capabilities cannot be loaded at any memory address. Its references are absolute addresses, and not addresses relative to the instruction pointer.

Listing 5.17 shows the analysis done on a program compiled without PIE. The output shows the position of the `main` function does not change anymore. By analyzing the binary with radare2, it's possible to extract information about the real position of that function (line 15). The function will always be loaded in that position.

Understanding relocations and ASLR is crucial to substitute the Diablo toolchain's task. If the binary is compiled as ET_DYN (ASLR enabled), any pointer inside the text segment will have a relocation entry in the .rela.dyn section. It is possible to modify the functions pointers required by the attestator by patching these entries.

5.7 Tools of the trade

This section presents some of the tools that were tested during the entire research. Some of them are widely used in the examples of this thesis. Others were just tested, but are worth mentioning.

```
$ gcc aslr.c -o a.out  # Compile the program
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
    linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
    BuildID[sha1]=f828cc2468c65e0fb799b410cb9b6f0960e767a0, not stripped
$ ./a.out     # Execute the program multiple times
0x55a28b92d135
$ ./a.out
0x5621c1e1f135
$ ./a.out
0x55bb9c8c3135
```

Figure 5.15: Example program output with ASLR.

```
1  $ r2 a.out   # Open the executable with radare2
2  [0x00001050]> aaa # Analyze the binary
3  [...]
4  [0x00001050]> afl  # List all the functions
5  [...]
6  0x00001135 1 31 sym.main
7  [...]
```

Figure 5.16: ASLR example program analysis.

This research aims at finding the most suitable means to replace the Diablo toolchain in the patching process. To properly substitute it, a tool to analyze and modify ELF executables is required.

readelf

readelf[12] is a Linux command-line program. It displays information about ELF format object files. It is capable of extracting almost any information regarding ELF files' structure. In this thesis, readelf was mostly used to list information about program headers (–segments flag), section headers (–sections flag) and relocations (–relocs flag). Symbols can be listed with the –syms flag.

objdump

objdump[13] is a standard Linux command-line program. It is useful for displaying information about generic object-files. In this research, objdump is used mainly as a disassembler (-d option). Since it is designed to analyze object-files it can also be used as readelf. For example the -p option lists information about the program headers and the dynamic sections.

radare2

radare2 is a framework which contains a set of command lines utilities. It can be used to do almost anything with binaries. From the radare2 GitHub page [21]:

[12] https://linux.die.net/man/1/readelf

[13] https://linux.die.net/man/1/objdump

```
1  $ gcc aslr.c -o a.out -fno-pie -no-pie   # Compile the program
2  $ file a.out
3  a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
       linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
       BuildID[sha1]=a57dc058a0d4891856844ad52e958cb06f21f54b, not stripped
4  $ ./a.out       # Execute the program multiple times
5  0x401122
6  $ ./a.out
7  0x401122
8  $ ./a.out
9  0x401122
10 $ r2 a.out       # Open the executable with radare2
11 [0x00401040]> aaa    # Analyze the binary
12 [...]
13 [0x00401040]> afl     # List all the functions
14 [...]
15 0x00401122 1 27 sym.main
16 [...]
```

Figure 5.17: Example program output without ASLR.

> Radare project started as a forensics tool, a scriptable command-line hexadecimal editor able to open disk files, but later added support for analyzing binaries, disassembling code, debugging programs, attaching to remote gdb servers...

The framework is composed of multiple tools. Each tool can be used independently, but combined they provide almost every functionality needed for binary analysis and manipulation. The strength of radare2 is its support for almost any format of binary (not only of ELF type). It is also still under development and it is highly recommended to always download the latest version from GitHub, even if there are binary releases.

A book explaining the basic functionalities of the radare2 framework can be found online [22]. It is also possible to find videos regarding r2con, the annual radare2 congress, and radare2 in general on youtube.com. There are two recordings worth mentioning. "radare demystified (33c3)" explains the basic commands of radare2. "SUE 2017 - Reverse Engineering Embedded ARM Devices - by pancake" is an introduction to firmware reversing.

The project is by default a command-line tool, but it is possible to use it with a graphical user interface by installing Cutter[14].

This tool was used extensively during the research, mainly for reversing purposes. Just the basic functionalities were used, but it has proved itself to be one of the most useful and comprehensive tool.

r2pipe

R2pipe is an API to interact with r2 instances with various methods [22]. It is possible to use radare2 functionality through pipes, HTTP queries and TCP sockets.

This API allows the creation of programs that exploit the framework functionality. It is possible to execute radare2 commands directly from the program. This makes automating tasks easier.

[14] Cutter, a graphical user interface for radare2, https://github.com/radareorg/cutter

The supported languages are Python, NodeJS, Go, Rust, Ruby, Perl, Erlang, Haskell, Dotnet languages, Java, Swift, Vala, NewLisp and Dlang.

Listing 5.18 is an example taken from the radare2 book [22] of a simple analysis of the /bin/ls binary performed in python using r2pipe. The example shows how it is possible to execute r2 command directly from the script.

```
import r2pipe

r2 = r2pipe.open("/bin/ls")
r2.cmd('aa')
print(r2.cmd("afl"))
print(r2.cmdj("aflj")) # evaluates JSONs and returns an object
```

Figure 5.18: r2pipe example.

GDB

GDB, or GNU Project debugger is the default debugger for Linux-like systems[15]. It is a terminal tool and, as radare2, it does not offer an intuitive interface to the user.

During this study, GDB was used mostly in the process of understanding ELF internal functionalities.

There are two interesting plug-ins for GDB focused on software security. They are both python modules loaded into the debugger. The first one is called *pwndbg*[16] and the second one *peda*[17] (Python exploit Assistance for GDB). Both were tested during the initial study on the ELF file format.

Python

Python offers a lot of libraries for binary analysis and manipulation. The following are the ones that were tested during this research.

angr

From the angr website[18]:

> angr is a multi-architecture binary analysis toolkit, with the capability to perform dynamic symbolic execution (like Mayhem, KLEE, etc.) and various static analyses on binaries.

It offers an analysis suite for binaries. The analysis is performed not at the assembly level, but the intermediate representation(IR) is used.

It is possible to use angr to conduct program emulation. The strength of this suite is the possibility to execute an emulation on *symbol variables*. As the angr guide explains, in symbolic emulation, every variable does not hold a certain value, but a range of possible value. Each

[15] https://linux.die.net/man/1/gdb

[16] https://github.com/pwndbg/pwndbg

[17] https://github.com/longld/peda

[18] https://docs.angr.io/

operation the program executes is used to construct an *abstract syntax tree* or AST of the program. The AST can be used to create a set of constraints for every variable. Such constraints can be solved by an SMT solver.

For example, it might be possible to guess a required input password without brute-forcing, by letting angr solve the require input a certain branch of the control flowgraph has. Examples of this functionality can be found in the angr GitHub page[19].

The angr suite was tested but not used during this research. It is worth mentioning, because of its application in binary analysis. An interesting article regarding angr is "The Art of War: Offensive Techniques in Binary Analysis" [23].

pwntools

"pwntools is a CTF framework and exploit development library" written in Python[20]. It is designed for exploit development, so it offers methods to connect remotly to a target, to manipulate assembly, pack integers and in general abuse vulnerabilities.

As for angr, during the course of this research most pwntools functionality were tested for learning purposes.

There is one useful library for manipulating ELF objects called pwnlib.elf. This library was considered during the design of the script needed by the attestator in Chapter 6. It offers a simple API for extracting information from object files and patching values.

pyelftools

"pyelftools is a pure-Python library for parsing and analyzing ELF files and DWARF debugging information." [21]

It is a really useful library in terms of extracting information, but it has a downside. It does not offer writing functionality. If the task requires some manipulation on the file, it has to be implemented by hand or by using another library.

Ruby

Ruby also offers different libraries to analyze binaries. Two libraries, in particular, offer interesting functionality in terms of ELF manipulation.

Metasm

Metasm is an assembly manipulation suite. It can assemble, disassemble, compile and link code. It can also be used as a debugger. It is remarkably comprehensive and it supports various architectures and file formats [24].

The metasm suite GitHub page[22] contains illustrative examples of its functionality and metasm APIs are documented at rubydoc.info[23].

For what concerns binary manipulation the `Metasm::ELF` is the most interesting module. It permits to easily access any information related to an ELF object.

The downside to metasm is the fact that is very extensive. Learning how to use it properly requires time.

---

[19] https://github.com/angr

[20] http://docs.pwntools.com/en/stable/

[21] https://github.com/eliben/pyelftools

[22] https://github.com/jjyg/metasm

[23] https://www.rubydoc.info/github/brainsucker/metasm/Metasm

rbelftools

rbelftools[24] is a more lightweight ELF parser inspired by pyelftools. Unlike its Python variant, it offers a set of more complete functionality, not only for parsing but also for manipulating ELF objects.

The documentation for rbelftools can be found on rubydoc.info[25] as well.

Its functionalities and its simplicity make it a highly compelling option when looking for a library for manipulating ELF files.

[24] https://github.com/david942j/rbelftools

[25] https://www.rubydoc.info/github/david942j/rbelftools

Chapter 6

Creation of a standalone version of the ASPIRE attestator

Having analyzed the ELF file format, it is possible to separate the attestator from the whole aspire framework. This chapter aims at creating a custom build of the ASPIRE attestator. Once the attestator is extracted from the project, its dependencies will be further analyzed.

As explained in Chapter 4 one of the major dependency of the attestator is the Diablo toolchain. Extracting the attestator from the framework implies the creation of a substitute. A custom script was used to replace it. The second strict dependency of the attestator is the ASPIRE communication protocol. As already explained in Chapter 4, the connection part was not considered during the research. To avoid using the protocol already in place, a hard-coded nonce was inserted inside the program. It is important to note that this custom attestator is just a proof of concept. Its purpose is to prove it is possible to run the attestator separated from the entire ASPIRE framework. In Chapter 8, an analysis of its potentially extendible portability is done.

Most of the code used in this chapter was taken from the ASPIRE project GitHub page [2].

6.1 Design and architecture

Extracting the attestator from the ASPIRE framework and removing its network functionality is a trivial process. Replacing the Diablo functionality is a more difficult task. As described in Section 2.1, the attestator extracts the information it needs from a structure called blob. The blob is not initialized in the source code. This means it has to be filled after compilation. In general, the process that has to happen is the following:

1. The user specifies the functions the attestator has to protect. This is possible by using a configuration file.
2. The user compiles together the program he wants to protect and the attestator. The attestator's blob is still empty, so another step is required to have a completely functional binary.
3. The user executes the script specifying the target binary and the configuration file. The script reads the configuration file and patches the blob with the correct information required by the attestator.
4. The user can now execute the protected program.

Figure 6.1 shows a diagram of the building process.

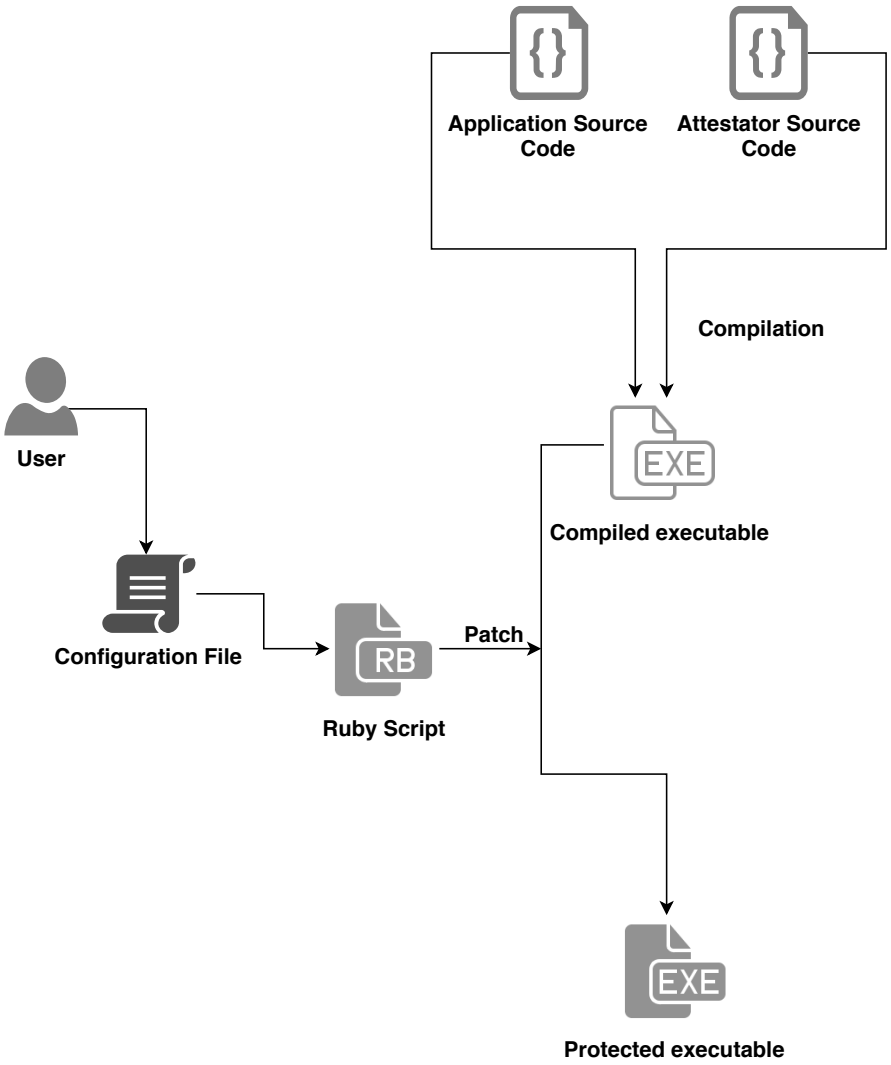

Figure 6.1: Attestator building process.

Configuration file

For what concerns the configuration, the user has to be able to specify a list of functions he wants to protect. Only the specified function are secured by the attestator. To quickly parse this information, a JSON format custom configuration file is used. There are good JSON parsers for almost every scripting language. The functions need to be specified by name. Listing 6.2 is an example of configuration file. The attestator number is customizable. The other information will be needed by the script to locate the symbols it has to patch. The *protect* field is an array of function names. It contains the names of all the target functions of the attestation.

Preparing the binary

The compilation is a trivial step, although there is one important thing to note. The binary has to be prepared for the adjustments of the patching script. This means the script is strictly correlated to the attestator source code. Some options have to be evaluated before writing the script.

Patching the blob with offsets relative to the text segment might not be the simpler solution. Potentially a list of pointers to the target functions can be inserted in the attestator's source code. The program itself will compute the correct offsets in each run, even with ASLR enabled. In that scenario, the script will only be in charge of patching the functions' sizes. This approach was discarded because it has an important security downside. The blob would contain exactly

```
{
  "attestator_number" : 255,
  "aid_size" : 16,
  "blob_structure_name" : "ra_data_structure_blob",
  "base_sym" : "text_seg_offset",
  "protect": [
    "function-name-1",
    "function-name-2",
    ...
  ]
}
```

Figure 6.2: Example of JSON configuration.

the pointers to the functions it is protecting. It would be simple for an average reverse engineer to understand what their purpose is. Chapter 7 analyzes different obfuscation options, but it is better to consider this issue already.

Having discarded the pointers' solution, there are also different choices regarding the blob declaration. Declaring the blob inside the source code is a valuable option, but there is one thing to take into consideration. If the blob structure is initialized with zeroes, the compilation process will insert it in the .bss section (as explained in Section 5.2). In that case, only the information about its memory size will be present in the binary. To prevent this, the blob has to be initialized with some bogus values in the attestator source code. The script will overwrite this portion of the binary on disk. The following is a snippet taken from the blob declaration source.

```
uint8_t ra_data_structure_blob[RA_DATA_STRUCTURE_BLOB_SIZE] = { 0x90, 0x90,
    0x90, 0x90 };
```

Another option is to not declare the blob in the program source code and inject it entirely after the compilation step. Since the injection process would add some complications, the easier solution was adopted. Injection in the data segment is not a minor task because it might require to fix data references in the code.

The last thing to examine is the base address. Using offsets relative to the text segment implies the need for a pointer to the text segment itself. In C, it is possible to construct a pointer to a function, but the function's position in memory is determined by the linker. To solve this issue, a custom pointer has to be created after the linking process, by modifying the binary. The easiest way to construct it is by altering an already existing function pointer. To do so, the base address has to be declared as a pointer to any function present in the text segment. The script will be in charge of patching it. There are two possible scenarios. In case ASLR is enabled, a dynamic relocation entry will be created inside the .rela.dyn section. Its functionality is similar to the one described in Section 5.4, with the exception it has to be resolved by the dynamic linker at runtime. The addend field of the relocation has to be corrected with the text segment offset value. If ASLR is not enabled, the text segment is always loaded at a constant memory address. In that case, the base symbol is a constant value the script has to patch.

ASLR analysis

This section shows an example of the relocation entry for the base symbol, both for ASLR enabled and disabled. In this example, the pointer to the text segment is initialized as the following.

```
uint64_t text_seg_offset = (uint64_t) dummy_function;
```

If the binary is of type ET_EXEC (no ASLR), the pointer to the text segment is a constant value. As shown in Listing 6.3, by analyzing the binary with radare2, it is possible to check that the `text_seg_offset` (line 9) variables holds indeed the address of the `dummy_function` (line 11).

After the blob is patched it will hold the address of the text segment, instead of the one pointing to `dummy_function`. *is* is the command for listing symbols and ~ is the internal grep command. *pxw* prints the hexadecimal word dump at the current address.

```
1  $ gcc src/*.c -o a.out -lpthread -lcrypto -DDEBUG -DDEBUG_ADS_PARSE
       -DHARD_DEBUG -no-pie -fno-pie   # Compile the program
2  $ r2 a.out   # Open the binary with radare2
3  [0x004012c0]> aaa # Analyze the binary
4  [...]
5  [0x004012c0]> is~text_seg_offset  # Search for the text_seg_offset symbol
6  097 0x000071c0 0x004081c0 GLOBAL OBJ 8 text_seg_offset
7  [0x004012c0]> s 0x004081c0   # Seek to the text_seg_offset location
8  [0x004081c0]> pxw    # Print the value
9  0x004081c0 0x00401f62
10 [0x004081c0]> afl~dummy_function  # Print information about the
       dummy_function function
11 0x00401f62 1 11 sym.dummy_function
```

Figure 6.3: Text segment offset without ASLR.

Listing 6.4 shows the same example but with ASLR enabled. If ASLR is enabled, the address of the `dummy_function` is going to be different every run. The loader will be in charge of patching the `text_seg_offset` variable with the correct value pointing to that function. The loader extracts the information it needs from the relocation entry in the .rela.dyn section. Relocations can be listed using the readelf tool (line 3). The output from radare2 at line 13 shows that the function at offset 0x2fe3 (the value of the relocation entry) is indeed `dummy_function`.

```
1  $ gcc src/*.c -o a.out -lpthread -lcrypto -DDEBUG -DDEBUG_ADS_PARSE
       -DHARD_DEBUG   # Compile the program
2  $ readelf --symbols a.out | grep text_seg_offset # Print information about
       the text_seg_offset symbol
3     100: 000000000000a1c0 8 OBJECT GLOBAL DEFAULT 24 text_seg_offset
4  $ readelf --relocs a.out | grep a1c0   # Print information about its
       relocation
5  00000000a1c0 000000000008 R_X86_64_RELATIVE 2fe3
6  $ r2 a.out   # Open the binary with radare2
7  [0x000022d0]> aaa # Analyze the binary
8  [...]
9  [0x000022d0]> is~text_seg_offset  # Search for the text_seg_offset symbol
10 100 0x000091c0 0x0000a1c0 GLOBAL OBJ 8 text_seg_offset
11 [0x000022d0]> s 0x2fe3    # Seek to the text_seg_offset value
12 [0x00002fe3]> pdf     # Print what is present at that value
13 / (fcn) sym.dummy_function 11
14 | sym.dummy_function ();
15 | 0x00002fe3 55 push rbp
16 | 0x00002fe4 4889e5 mov rbp, rsp
17 | 0x00002fe7 b800000000 mov eax, 0
18 | 0x00002fec 5d pop rbp
19 \ 0x00002fed c3 ret
```

Figure 6.4: Text segment offset with ASLR.

The dynamic loader will patch the content of the text segment offset variable with the address at which the offset 0x2fe3 is loaded. The script will have to modify this value to the offset of the

text segment itself. This offset can be listed with the readelf tool as well.

```
$ readelf --segments a.out --wide
[...]
 LOAD 0x002000 0x0000000000002000 0x0000000000002000 0x003e99 0x003e99 R E
     0x1000
[...]
```

Based on the above output, the patching script will have to overwrite the relocation entry value from 0x2fe3 to 0x2000.

Patching the blob

Since the blob is already present in the binary, the script task is to patch it. The following is the procedure the script has to execute.

1. Open the configuration file;
2. Parse the information it contains. A JSON parser is required for this operation;
3. Open the target executable file;
4. Find the position of the blob in the data segment;
5. Find the position of the base symbol;
6. Find the position of every function the attestator has to protect and retrieve information about their sizes;
7. If ASLR is enabled, patch the base symbol present in the .rela.dyn section so that it points at the beginning of the text segment. If ASLR is disabled, patch the base symbol in the data segment with the constant memory address of the text segment;
8. Patch the blob with the information retrieved in step 5.

Considering the script has to retrieve information from an ELF executable file, some libraries for parsing and manipulating executable files have been evaluated. Two scripting languages, in particular, best fit these needs: Python and Ruby. They both offer some libraries for manipulating ELF files. In Section 5.7 is possible to find a description of some of the libraries. After some tests, Ruby was chosen as the designated language. Porting it to Python should not be too difficult.

With the script written and executed on the target, it is possible to run the protected program. The attestator will be spawned in a thread and it will start computing checksums of the functions it is configured to protect.

6.2 Testing

Listing 6.5 is a test of the custom attestator on a simple program. The attestator target is the `main` function. The program is compiled together with the attestator on line 1. In this example, the application is compiled using GCC with ASLR enabled. To test it without ASLR the '`-no-pie -fno-pie`' flags have to be specified.

Then the Ruby script is launched with the compiled binary as the target and with the file config.json as the configuration file (line 2). From the output it is possible to see the script in action on the executables file. Once the binary has been patched, it is ready to run.

The program is launched at line 19.

The output shows the attestator performing a security check by computing the checksum over the `main` function. Since the nonce is constant, the calculated hash is always the same, because

```
1  $ gcc src/*.c -o a.out -lpthread -lcrypto -DDEBUG -DDEBUG_ADS_PARSE
       -DHARD_DEBUG   # Compile the program
2  $ ./patch_ra_data_blob.rb -e a.out -c config.json    # Execute the patching
       script
3  Reading the config file...
4  Check if the binary is of type ET_EXEC or ET_DYN...
5  Check Address Space Layout Randomization...
6  ASLR => true
7  Check if the binary has a symtab...
8  Symbol table found
9  Searching for text_seg_offset...
10 Searching for ra_data_structure_blob...
11 Base symbol found at offset 0x91c0
12 Blob found at offset 0x9180
13 Searching for symbol main...
14 Symbol main found: {:memory_offset=>3974, :size=>43}
15 Patching the base symbol...
16 Aslr is enabled => Searching for the relocation entry...
17 Patching the blob...
18 Blob patched correctly!
19 $ ./a.out    # Run the protected binary
20 (Attestator) An attestator is being started
21 [...]
22 (Attestator 255) Prepared data size 43
23 550000C00048FF00E8000000FF89E0C300003DB800F25A00000AB8F[...]
24 (Attestator 255) Attestation data hashed, 16B buffer
25 (Attestator 255) Hashed data size 16
26 5D920360A17BFCD40F1D0EE6B87FA5EF
27 [...]
28 (Attestator 255) Request processed in = 0.001681 s
29 [...]
30 (Attestator 255) Prepared data size 43
31 550000C00048FF00E8000000FF89E0C300003DB800F25A00000AB8F[...]
32 (Attestator 255) Attestation data hashed, 16B buffer
33 (Attestator 255) Hashed data size 16
34 5D920360A17BFCD40F1D0EE6B87FA5EF
35 [...]
36 (Attestator 255) Request processed in = 0.002100 s
37 [...]
38 (Attestator 255) Prepared data size 43
39 550000C00048FF00E8000000FF89E0C300003DB800F25A00000AB8F[...]
40 (Attestator 255) Attestation data hashed, 16B buffer
41 (Attestator 255) Hashed data size 16
42 5D920360A17BFCD40F1D0EE6B87FA5EF
43 [...]
44 (Attestator 255) Request processed in = 0.001629 s
```

Figure 6.5: Attestator patching process and test.

the data is extracted from the protected function in the same order every time. With a different nonce, the checksum would be different.

In Section 6.1 the relocation's patch was discussed. It is possible to check that indeed the relocation was modified by the script as predicted.

```
$ readelf --relocs a.out | grep a1c0
```

```
00000000a1c0 000000000008 R_X86_64_RELATIVE 2000
```

An interesting test is to modify the `main` function disassembly after compilation. This resembles a potential adversary trying to modify bytecodes of the program to gain an advantage. It is possible to patch the code both at runtime or on disk. Listing 6.6 is an example of how to use radare2 [21] to patch the binary on disk.

```
1  $ r2 -w a.out
2  [0x000022d0]> aaa
3  [...]
4  [0x000022d0]> s main
5  [0x00002f86]> pdf
6  [...]
7  | 0x00002f94 488d3dc03700. lea rdi, qword str.Hello_World ; 0x675b ; "Hello
       World!" ; const char *s
8  | 0x00002f9b e8e0f0ffff call sym.imp.puts ; int puts(const char *s)
9  | 0x00002fa0 bf0a000000 mov edi, 0xa ; int s
10 | 0x00002fa5 e806f2ffff call sym.imp.sleep ; int sleep(int s)
11 [...]
12 [0x00002f86]> s 0x00002fa0
13 [0x00002fa0]> wa mov edi,0xf
14 [0x00002fa0]> pdf
15 [...]
16 | 0x00002f94 488d3dc03700. lea rdi, qword str.Hello_World ; 0x675b ; "Hello
       World!" ; const char *s
17 | 0x00002f9b e8e0f0ffff call sym.imp.puts ; int puts(const char *s)
18 | 0x00002fa0 bf0f000000 mov edi, 0xf ; int s
19 | 0x00002fa5 e806f2ffff call sym.imp.sleep ; int sleep(int s)
20 [...]
21 [0x00002f86]> exit
```

Figure 6.6: Binary patched with radare2.

The binary is open with radare2 in writing mode (*-w* option). From the disassembled output at line 7 it is possible to understand that the main function prints "Hello World!" and then calls `sleep` with a parameter of 10 (lines 9-10). The number 10 is patched to be 16 (0xf hexadecimal). The instruction is modified by using the *wa* command (line 13).

After the modification, the program is launched again. The attestator should compute a different checksum, as the `main` function has been tampered with.

This is just an illustrative modification, an attacker is likely to attempt modifying important calls such as a check for a license key, or a logic functionality of a game. Ideally, vulnerable code should be obfuscated. This is just a test to show the attestator can detect single-byte modification.

The output of the modified program is shown in Listing 6.7. The computed hash is indeed different. This shows the attestator was able to detect the change.

6.3 Conclusions

In this chapter, a successful attempt at creating a custom separate version of the ASPIRE attestator was presented. The communication part was not taken into consideration in this build. To have a properly functional attestator, the networking component should be developed, based on the platform and the libraries at disposal.

The ASPIRE project provides a complete set of protections for binaries, which comprehends more than just the attestator. In particular, the obfuscation part was not examined in this section.

```
$/a.out
(Attestator) An attestator is being started
[...]
Hello World!
[...]
(Attestator 255) Prepared data size 43
550000C00048FF00E8000000FF89E0C300003DB800F25A00000FB8F[...]
(Attestator 255) Attestation data hashed, 16B buffer
(Attestator 255) Hashed data size 16
DFAAD6E421E4FE8C8347D13E43ADCF3C
[...]
(Attestator 255) Prepared data size 43
550000C00048FF00E8000000FF89E0C300003DB800F25A00000FB8F[...]
(Attestator 255) Attestation data hashed, 16B buffer
(Attestator 255) Hashed data size 16
DFAAD6E421E4FE8C8347D13E43ADCF3C
[...]
(Attestator 255) Prepared data size 43
550000C00048FF00E8000000FF89E0C300003DB800F25A00000FB8F[...]
(Attestator 255) Attestation data hashed, 16B buffer
(Attestator 255) Hashed data size 16
DFAAD6E421E4FE8C8347D13E43ADCF3C
[...]
```

Figure 6.7: Attestator tampering detection.

Obfuscation is not only helpful in protecting the attestator itself from reversing (the source code of the ACTC is readable online), but it can protect parts of the software which are proprietary and which should not be revealed. Obfuscation is essentially security through obscurity, so to say, but the time and the resources it can take might slow and deter an attacker enough to make it give up.

The next chapter contains an evaluation of some obfuscation methods and their strengths and weaknesses.

Chapter 7

Obfuscation: analysis of the tools and the techniques to protect the attestator

This chapter contains an analysis of some tools that can be used as a replacement for the obfuscation protection offered by the ASPIRE framework. The tools are evaluated based on the protection mechanisms they employ. The chapter includes a lot of examples and it can be used as a reference for the most common obfuscation techniques.

7.1 Obfuscation purposes and tecniques

Obfuscation, in regard to software, is a mechanism through which the original functionality of the program is hidden. To accomplish this goal various techniques can be used. By definition, the protection offered by obfuscation is "security through obscurity". Obfuscation can only slow a persistent attacker because the program functionality has to be contained in the binary in some form. In general, obfuscation is used:

- To protect intellectual property.

  If the program contains proprietary algorithms, disclosure of the procedures can cause financial damage to their owner. By obfuscating the code, the company that owns the algorithm makes harder for any competitor to reverse it.

- To protect a crucial piece of code.

  For example, some programs require a serial key to function. If the code in charge of checking the serial key can be easily reversed, a potential attacker might be able to bypass the check and obtain a free working product.

- To make the general functionality of the program difficult to understand.

  There are some software products that, if modified, can cause financial damage to the company that had developed them. The more the code is understandable, the easier it is to recognize what to modify to get an advantage

  Cheats for games typically require a basic understanding of the program functionality. A program difficult to understand can prevent the creation of cheats based on the modification of the software.

Obfuscating the code has lots of advantages, but it also has a downside. Typically obfuscation techniques are based on inserting additional code or by modifying the control flow graph of the

program. In general, the more the code is obfuscated, the more the execution time increases. To avoid performance issues obfuscation should be used just on the parts of code crucial to protect.

Typically the strength of obfuscation is given by the sum of multiple strategies combined. If just one obfuscation technique is used, a good reverse engineer might be able to automate the process of de-obfuscating the code, which makes this security measure pointless.

There are different techniques through which the code can be obfuscated. In the following sections, some of the tools to obfuscate a program and their mechanisms are tested and analyzed.

7.2 Obfuscator-LLVM

LLVM is a project containing a "collection of modular and reusable compiler and toolchain technologies" [25]. Thanks to a subproject of LLVM called Clang, it is possible to compile C code, as with GCC.

An obfuscator for the LLVM suite can be found on GitHub[1]. There is also a commercial version with enhanced capabilities, but only the free version is analyzed in this section. Since this obfuscator works for the LLVM compilation suite, it offers a version of the Clang compiler capable of obfuscating the generated code. This compiler takes the source code as input and it produces an obfuscated ELF binary as output.

There are 3 main features it offers in terms of obfuscation. In the following paragraphs, they are analyzed one by one.

- Instructions substitution;
- Bogus control flow;
- Control flow flattening.

Instruction substitution

This technique replaces basic assembly operations with functionally equivalent but more complicated ones, the program will have the same functionality, but it will be more difficult to reverse.

Since it is possible to remove this kind of obfuscation by re-optimizing the code, this mechanism does not add a lot of security. Combined with other strategies it renders the code complicated and hard to read. An example of this technique at source code level can be found in the Obfuscator-LLVM GitHub page wiki [2].

Listing 7.1 contains the example program source code. Listing 7.2 and Listing 7.3 show the assembly code of a compiled example program, with and without this obfuscation technique applied.

Even if the disassembly output is more verbose than the C source, it is possible to clearly distinguish what the code is doing in both snippets by analyzing the assembly code.

In the obfuscated disassembly there are two more instructions at lines 12 and 14. In this particular case, they are extremely easy to spot and ignore in an attempt to reverse the program.

The level at which this instruction insertion is done can be selected in the configuration. Having more and more of these useless instructions can make enough noise to cover what the program is doing. As already said though, since it is possible to revert the process, this mechanism is not really that effective and it should be combined with other strategies.

[1] https://github.com/obfuscator-llvm/obfuscator/wiki

[2] https://github.com/obfuscator-llvm/obfuscator/wiki/Instructions-Substitution

```
#include <stdlib.h>

int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  a = a * 2;
  return a;
}
```

Figure 7.1: Instruction substitution example program.

```
1  push rbp
2  mov rbp, rsp
3  sub rsp, 0x20
4  mov dword [local_4h], 0
5  mov dword [local_8h], edi ; argc
6  mov qword [local_10h], rsi ; argv
7  mov rsi, qword [local_10h]
8  mov rdi, qword [rsi + 8] ; [0x8:8]=-1 ; 8 ; const char *str
9  call sym.imp.atoi ; int atoi(const char *str)
10 mov dword [local_14h], eax
11 mov eax, dword [local_14h]
12 add eax, 2
13 mov dword [local_14h], eax
14 mov eax, dword [local_14h]
15 add rsp, 0x20
16 pop rbp
17 ret
```

Figure 7.2: Disassembled program without instructions substitution protection.

Bogus control flow

The control flow graph is a useful way to determine the logic of a program. Modifying this graph can make a lot harder, for a potential reverse engineer, to understand what the logic flow of the program is.

This technique creates a new block inside the functions call graph. From this block the program jumps back to the original block through an opaque predicate. Opaque predicates are conditional jumps whose result is known in advance, but cannot be evaluated statically. The result of an opaque predicate is always going to be the same, but it might be difficult to guess what it is without dynamic analysis.

Listing 7.4 provides an example source for the bogus control flow protection mechanism. The control flow graph of this program compiled without obfuscation looks like Figure 7.5. It clearly resembles the flow graph of the C source. Compiling it with the bogus control flow protection leads to a different result, as observable in Figure 7.6. The flowgraph looks considerably more complicated, even though the functionality of the program is preserved.

The weakness of this type of protection is that some tools might be able to spot opaque predicates by using statical or even dynamical analysis. If an opaque predicate is found, its whole branch in the flow graph can be cut out and considered as not reachable code, thus simplifying the reversing process

```
1  push rbp
2  mov rbp, rsp
3  sub rsp, 0x20
4  mov dword [local_4h], 0
5  mov dword [local_8h], edi ; argc
6  mov qword [local_10h], rsi ; argv
7  mov rsi, qword [local_10h]
8  mov rdi, qword [rsi + 8] ; [0x8:8]=-1 ; 8 ; const char *str
9  call sym.imp.atoi ; int atoi(const char *str)
10 mov dword [local_14h], eax
11 mov eax, dword [local_14h]
12 add eax, 0x6abc334a
13 add eax, 2
14 sub eax, 0x6abc334a
15 mov dword [local_14h], eax
16 mov eax, dword [local_14h]
17 add rsp, 0x20
18 pop rbp
19 ret
```

Figure 7.3: Disassembled program with instructions substitution protection.

```
#include <stdlib.h>

int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  if(a == 0)
    return 1;
  else
    return 10;
  return 0;
}
```

Figure 7.4: Bogus control flow example program.

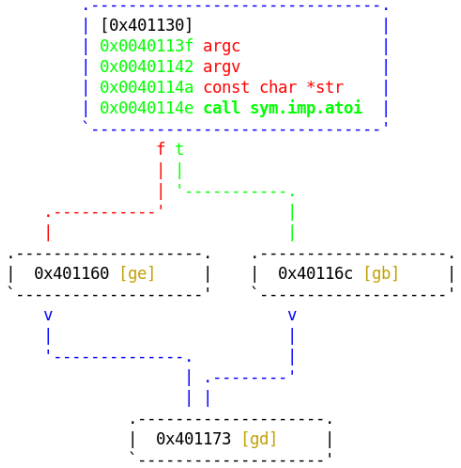

Figure 7.5: Program without bogus control flow protection.

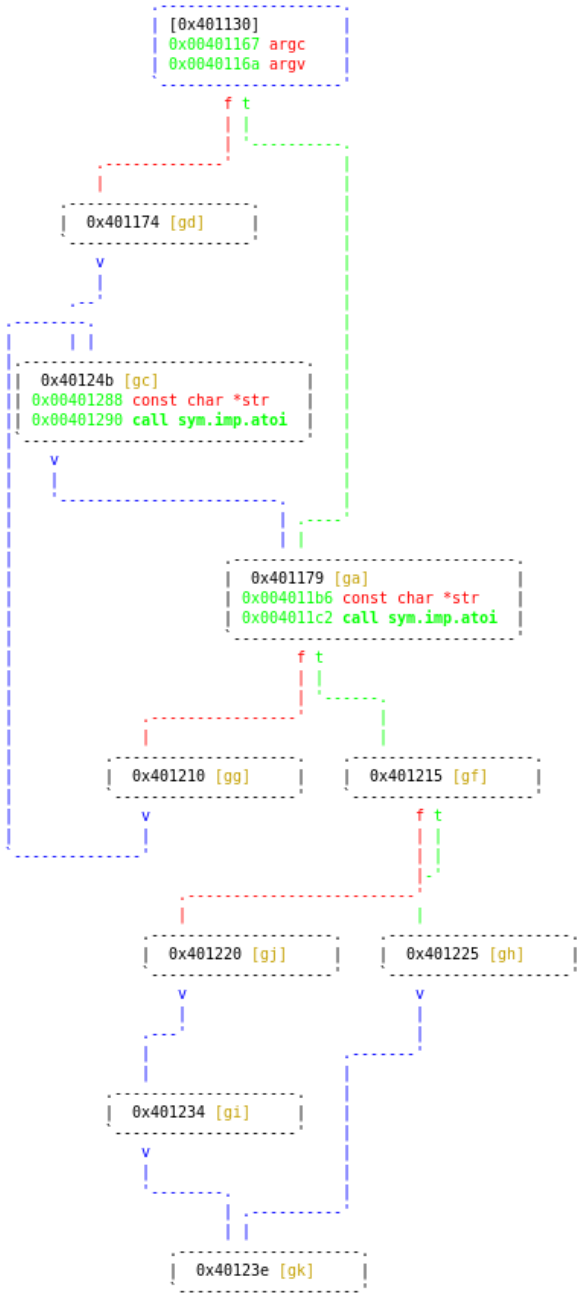

Figure 7.6: Program with bogus control flow protection.

Control-flow flattening

As the name itself says, this technique causes the control flow graph to flatten completely. There is an interesting article about this tecnique by László *et al.* [26]. As this article explains:

> In the case of most real life programs, branches and their targets are easily identifiable due to high level programming language constructs and coding guidelines. [...] The idea behind control flow flattening is to transform the structure of the source code in such a way that the targets of branches cannot be easily determined by static analysis, thus hindering the comprehension of the program.

The GitHub page of the LLVM-Obfuscator[3] shows an example of what this process is doing at the code.

```
#include <stdlib.h>
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  if(a == 0)
    return 1;
  else
    return 10;
  return 0;
}
```

Figure 7.7: Program without control flow flattening protection.

The applied protection will convert the code in Listing 7.7 into something like Listing 7.8:

```
#include <stdlib.h>
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  int b = 0;
  while(1) {
    switch(b) {
      case 0:
        if(a == 0)
          b = 1;
        else
          b = 2;
        break;
      case 1:
        return 1;
      case 2:
        return 10;
      default:
        break;
    }
  }
  return 0;
}
```

Figure 7.8: Program with control flow flattening protection.

The branches in the protected program are converted into a while statement with a switch selector inside. The result is a flow graph that looks like a complex switch construct. Figure 7.9 from the Tigress website[4] shows how this transformation looks like in the flow graph. After that transformation, it is complicated to understand what the execution order of all the blocks is since they are all on the same level.

[3] https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-Flattening

[4] http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html

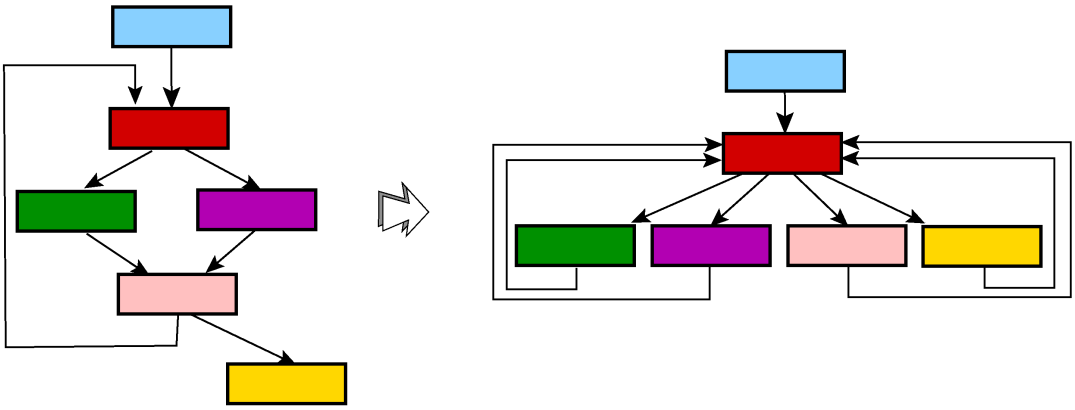

Figure 7.9: Control flow flattening

Review

Combining all of the above techniques, a decent protection level in terms of obfuscation can be achieved. As the FAQ page of the project on GitHub says though[5], this obfuscator should not be used in production. It is still being tested and therefore it is potentially easily reversible.

7.3 Tigress

Tigress is another interesting obfuscation tool. From the Tigress home page [27]:

> Tigress is a diversifying virtualizer/obfuscator for the C language that supports many novel defenses against both static and dynamic reverse engineering and de-virtualization attacks.

Tigress works directly on the C source code and it offers more protection techniques than the one described in the previous section. The output of this tool is still in the form of C source files, so a compiler is needed to get a fully functional executable. It is possible to analyze the transformation Tigress is doing on the source code itself.

The following is an evaluation of the most promising transformations Tigress offers.

Function virtualization

This transformation is based on the concept of virtualized code. The idea is to transform the original bytecode into a custom instruction set. This instruction set can be emulated inside a custom interpreter. When this transformation is activated on a function, Tigress transforms the function itself in an interpreter whose purpose is to execute the bytecode which corresponds to the original function.

Virtualization is regarded as a strong defense against reverse engineering. There is a paper by Jonathan Salwan *et al.* about the security of this protection technique [28], which also talks about the Tigress obfuscator. This article analyzes ways through which the process of virtualization can be reverted almost back to the original source code. The recovery process has some limits and the article even discussed techniques thanks to which this analysis could be made impossible.

[5] https://github.com/obfuscator-llvm/obfuscator/wiki/FAQi

Function jitting

Jit stands for just-in-time compilation. When using this protection on a function, Tigress will convert the function itself into another function which, when executed, will dynamically compile the original function and run it. The jitting mechanism makes static analysis extremely difficult because the code being run is not present in the binary on disk. Instead, it is created at runtime and it is only present in memory.

Dynamic function jitting

This protection is very similar to the function jitting transformation. The difference is that "the jitted code is continuously modified and updated at runtime" [27]. This makes even more difficult to dump the original code from the binary, since even during dynamic analysis, the code is continuously modified.

Control-flow flattening and opaque predicates

Both of these techniques were already discussed in Section 7.2.

Function merging and splitting

These two mechanisms are complementary to each other. The first one consists in merging multiple functions into a single one. The second splits a single function into multiple ones. Based on the original function, both methods can have their benefits and downsides. Usually, functions are used to separate tasks related to different areas of concern inside the program. Having functions which do not follow this logic will make the process of understanding the purpose of the analyzed software harder.

Both of these techniques were tested. Listing 7.10 shows the unprotected source code. In Listing 7.11 the functions `f1` and `f2` are merged into a single function called `_1_f1_f2`.

```
void f1() { /* Execute f1 */... }
void f2() { /* Execute f2 */... }

int main() {
    f1();
    f2();
}
```

Figure 7.10: Program without function merging protection.

The splitting technique is similar to the one above and can be found in Listing B.5 in the Appendix B.

Both of these approaches offer a good increase in terms of security protection, without deteriorating the performance as other mentioned mechanisms.

Function argument randomization

This transformation changes the order of function arguments randomly and it can add bogus arguments. Arguments passed to a function can be used to understand its functionality. The advantage of doing such a thing is the enhanced difficulty in understanding what the purpose of a function is, based on the argument passed to it.

```
int main(...)
{
  [...]
  _1_f1_f2(0, 0);
  _1_f1_f2(0, 1);
}
void _1_f1_f2(void *tigressRetVal , int whichBlock__0 )
{
  if (whichBlock__0 == 0) {
    /* Execute f1 */
    [...]
  } else
  if (whichBlock__0 == 1) {
    /* Execute f2 */
    [...]
  } else {

  }
}
```

Figure 7.11: Program with functions merging protection.

Figure 7.12[6] shows what this transformation looks like.

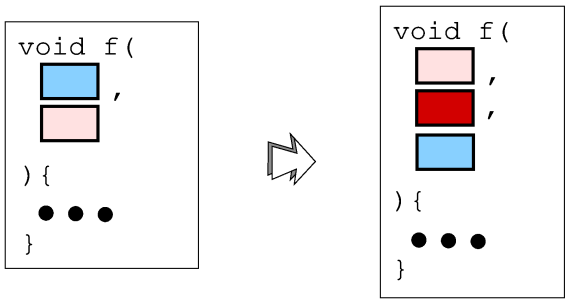

Figure 7.12: Function argument randomization

As the previous protection, applying this technique does not add a lot of performance issues. Combined with functions merging or splitting, the output will be a code designed in an extremely counter-intuitive way and very difficult to reverse.

Encoded literals, data and arithmetic

Encoding is the process of hiding what the real value of an expression is until it is necessary. Literals and initialized variables can be a goldmine for a reverse engineer. If this type of data is contained inside the program as plaintext, a potential reverser can use it to trace back what the various parts of the program are used for. This is why encoding literals and data, in general, is extremely important.

Literals are constant values. To encode a literal Tigress either replace it with an opaque expression (if it is an integer for example) or creates a function that generates it at runtime (if it is a string).

[6] http://tigress.cs.arizona.edu/transformPage/docs/randomizeArguments/index.html

To encode variable data Tigress can use a non-standard representation. Listing 7.13, taken from the Tigress website[7], is an example showing how this functionality can encode variable data.

```
int main () {
  int arg1 = ...
  int arg2 = ...
  int a = arg1;
  int b = arg2;
  int x = a*b;
  printf("x=%i\n",x);
  }
```

Figure 7.13: Encoded literals example program.

The mentioned code can be translated into the following:

```
a = 1789355803 * arg1 + 1391591831;
b = 1789355803 * arg2 + 1391591831;
x = ((3537017619 * (a * b) - 3670706997 * a) - 3670706997 * b) + 3171898074;
printf("x=%i\n", -757949677 * x - 3670706997);
```

The example shows it becomes not so trivial to understand what the value of x is.

Arithmetic procedures can also give away information about what the program is supposed to do. Even more, if the program contains a proprietary algorithm, it is better to encode it in a way difficult to reverse.

To encode arithmetic Tigress uses an approach similar to the one in section 7.2. A simple arithmetic expression can be translated into a more complicated one. For example:

```
z = x + y + w
```

Can be converted into:

```
z = (((x ^ y) + ((x & y) << 1)) | w) +
     (((x ^ y) + ((x & y) << 1)) & w);
```

ADVobfuscator

ADVobfuscator is another promising project in regarding obfuscations[8]. This obfuscator works only for C++11/14 and it is based on metaprogramming. This analysis mainly focuses on the C language, so the functionality of the ADVobfuscator is not described here. There is an article from Black Hat Europe 2014 that explains how it works [29]. This method is interesting enough to mention and it might be an option if using C++ is a possibility.

7.4 Conclusions

Tigress can potentially offer a more complete set of security protections than the Obfuscator-LLVM analyzed in the preceding section. Out of the two, it is the most valuable choice to add this protection layer to the unprotected attestator. There is something to consider though. If the

[7] http://tigress.cs.arizona.edu/transformPage/docs/encodeData/index.html

[8] https://github.com/andrivet/ADVobfuscator

software is running on a less powerful device in terms of computation, having so many protection mechanisms stacked one upon another might cause serious performance problems. The attestator itself is run as a thread inside the program and it will continuously consume computation resources.

Using Tigress is a choice that has to be made based on the computational power at disposal and based on the resources the program alone requires. Since it can be highly customizable, it is possible to select the protection strength that best suits both the security and the performance level required.

Chapter 8

Analysis on the portability of the attestator

This chapter is an analysis of the portability of the attestator for the most common architectures and operating systems. In Chapter 6, a proof of concept working on a 64 bit x86 architecture and a standard Linux box (Debian) was built. This paragraphs will mainly focus on embedded platforms for the Internet of Things.

There are two topics this chapter will address regarding the custom attestator's build. In the first part, a list of potential changes for the requirements analyzed in Chapter 4 is presented. These changes are aimed at extending the portability of the attestator. The second part is a study concerning the support offered by the most used embedded operating systems.

8.1 Extend the portability

This section proposes, for each requirement analyzed in Chapter 4, some potential solution to extend the attestator portability. Similarly to the requirements analysis, this section is divided into two parts.

Hardware

Most of the hardware requisites analyzed in Chapter 4 are not circumventable.

The portability to 8-bit and 16-bit processors might require some complicate modifications. The computing power and the memory size requisites suffer from this same problem. It is definitely possible to create an attestator that works on less powerful platforms, as already seen in Section 2.2. The ASPIRE attestator is not portable to these devices.

The networking functionality is a prerequisite of the attestation component. Without it, remote attestation is not possible. The communication method can be decided based on the platform at disposal, but there must be a means to interact with the trusted server.

Regarding the multi-threading support, there is no bypass to this problem, except implementing some sort of multi-threading logic by hand.

It might be possible to alternate code executed by the program itself and code executed by the attestator, even if implementing this kind of solution would lead to a serious performance worsening. It is worth considering that the attestator does not need to run continuously. The frequency at which the memory integrity is checked can be adjusted to the device capabilities. This holds even when considering a device that supports multi-threading. If the device is under a heavy workload, it is possible to reduce the number of checks considerably, leaving more computational resources to the main program.

Most embedded devices support multiple threads and this problem should be easily solvable. The API offered to code the logic of the thread might differ, but in general, it should be possible to port that part of the attestator to almost any device.

Software

Diablo

In Chapter 6, a custom build of the ASPIRE framework's attestator was created as a proof of concept. The custom attestator is not dependent on the Diablo toolchain. This proves Diablo dependencies can be resolved by using other tools for performing its task. One of the tools discussed in Chapter 7 can be used to substitute the obfuscation functionality

Libwebsocket and Curl

Since the communication protocols are highly dependent on what the platforms offer, the communication part of the attestation protocol was overlooked in this section as well.

Different devices can have various methods to link with each other. A remote attestation can happen only on devices able to communicate over a network. The module needed for that functionality should be constructed and adapted based on the protocols and the APIs the device supports.

OpenSSL

As already introduced in Chapter 4 the downside about using OpenSSL library is its size. There are various alternatives to this library.

One option to replace it is to write a custom implementation of the hash and the random walk function. This is the solution adopted by some of the proposed applications cited in Section 2.2. Writing a custom solution ensures the code is designed at its best for the hardware it runs on. The problem with that is that custom solutions are typically not portable. With the target of extending the portability of the attestator, it is better to employ something more flexible. The best way to guarantee portability is to use a library.

In this study, two libraries were considered. Both of them support embedded systems.

- wolfSSL[1];
- Mbed TLS[2].

These two libraries were chosen based on the support they offer and the fact they are used extensively. Both of them contain APIs for cryptographic operation and hashing. Those can be used to substitute OpenSSL in the attestator. They also offer SSL/TLS capabilities which can be utilized for securing the communications. As already discussed, the communication part is not examined in this study, but their support should be considered as a valuable option in the implementation of the networking aspect.

[1] https://www.wolfssl.com/

[2] https://tls.mbed.org/

8.2 Embedded OSes cryptographic support

These two libraries are not supported by every operating system. The following paragraphs are an analysis of their portability across the most used ones. Apart from hardware issues, if the OS supports the use of one of these libraries, the attestator should be portable to that system. Without the support for any cryptographic library, the hashing and the random walk algorithms will have to be implemented by hand. This would be inconvenient. To analyze the problem, a list of potential operating systems is needed.

A list for the most used operating system can be found in an Embedded Markets Study from April 2017 [30]. This study is not up to date, but no more recent versions were found. Figure 8.1 and Figure 8.2 are extracted from the study's presentation. Figure 8.1 shows the usage of embedded operating systems in April 2017. Figure 8.2 shows the usage predicted in the following 12 months. Clearly, the percentages might be outdated, but the list can be used as a guideline for the analysis.

With a list of the most used operating systems, it is possible to check their support for any of the three cryptographic libraries mentioned before. Table 8.1 shows what libraries those OSes offer. An empty field means the library is not supported. The question mark means that no reference was found to the specific operating system, but the similarity with others suggests the library might be usable.

The results of the analysis are the following:

- 25 operating systems were researched;
- Only 5 of them support the OpenSSL library.
- 15 OSes support the WolfSSL libraries;
- 9 OSes support Mbed TLS library;
- 2 OSes support all the libraries taken into consideration;

The collected data can be represented as in Figure 8.3. With these results at hand, it is possible to draw some conclusions.

For the OSes with OpenSSL support, the attestator should be easily adaptable since all of them are Linux-based systems.

Of the three libraries, WolfSSL is the one which gives more support. An interesting thing to notice is that every platform supporting Mbed TLS also supports the WolfSSL library and not vice-versa. This is another point in favor of WolfSSL.

This leads to the conclusion that WolfSSL is a better choice since it offers more extensive support, covering also the systems Mbed TLS can be run on.

It is also important to note that only 8 out of 25 systems do not have support for any of these libraries, or at least the compatibility is unknown. This means the attestator should be extensively adaptable to all sorts of different operating systems, as long as the hardware requirements are satisfied.

8.3 Executable file format

One more aspect has to be taken into consideration. The study and the modification made on the attestator were done with the ELF file format in mind. The ELF format is the standard for almost any Unix-like operating system, but other embedded OSes do not utilize it. It would be easier to port the attestator to a platform where the environment is the most similar to a Linux, but it is possible, with extensive modification, to adapt the attestator even to a different platform. To achieve such a goal, the inner functionality of the different file format should be studied and analyzed, to understand how the attestator requires to be adjusted.

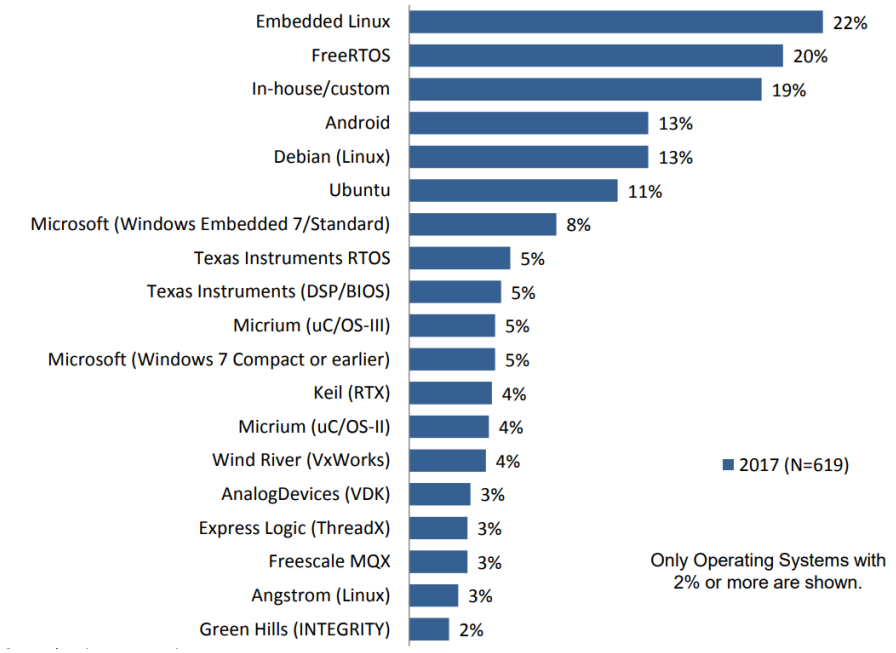

Figure 8.1: Embedded operating systems usage in April 2017

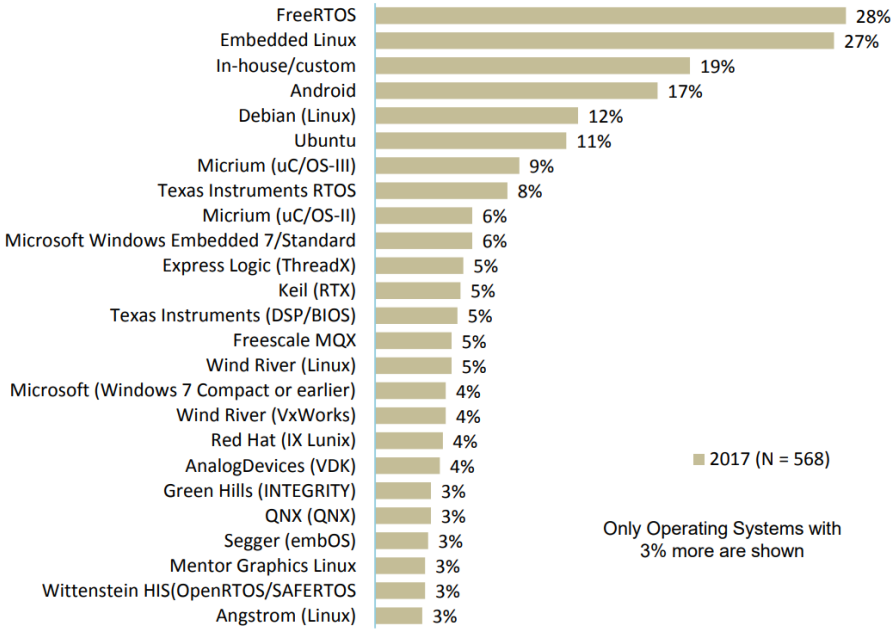

Figure 8.2: Predicted embedded operating systems usage in April 2017

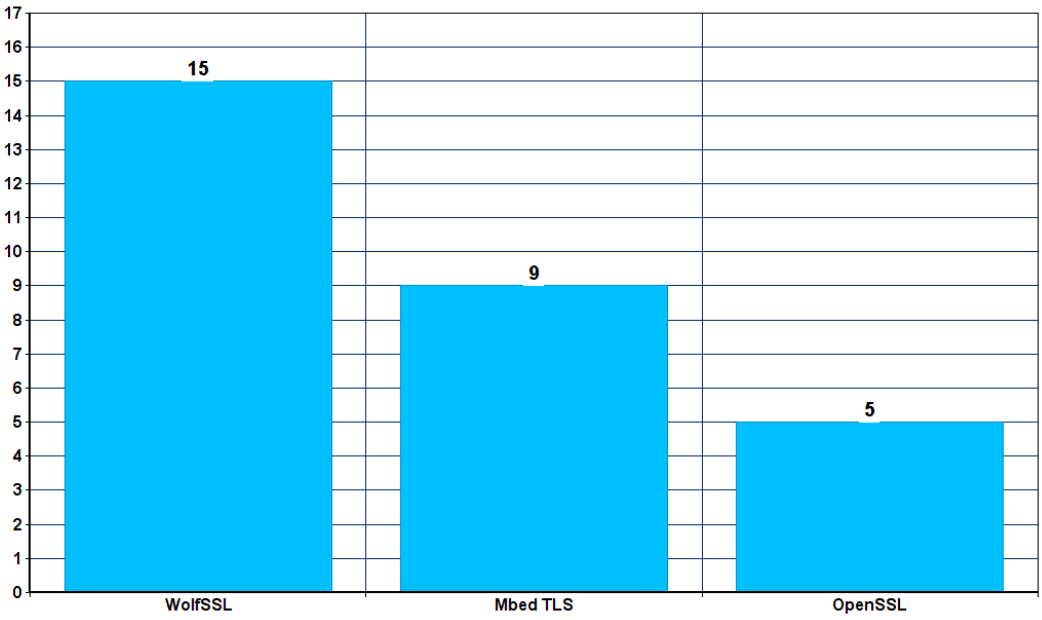

Figure 8.3: Cryptographic libraries support

Taking into consideration only the top ten used operating systems, 8 out of 10 use the ELF file format as the executable standard. Custom operating systems might use it as well. This means that potentially only 1 out of 10 top operating systems do not use the ELF file format (Microsoft Windows). This data is shown in Table 8.2.

Considering the broad usage of this file format, any attestation solution for embedded systems should be focused on it. This is also the reason behind the study in Chapter 5. It is important to note that, even if the executable format is the same, the portability is not guaranteed. Different operating systems might offer diverse APIs for low-level logic, like threads and network functionalities. Linux OSes like Debian or Ubuntu should provide the same system calls. This makes adapting the attestator source to these systems more straightforward.

8.4 Conclusions

In this chapter, the attestator portability was discussed. Based on the consideration in Section 8.1, the hardware requisites should not be a problem, except for particularly low-end devices.

The more significant obstacle regards the software requirements. Section 8.2 shows the extent to which it might be possible to adapt the attestator's cryptographic functionality. Even if the study considers outdated information it is possible to conclude that most operating systems support at least one library with cryptographic APIs. What this analysis does not show is the required effort to adapt the attestator. The more similar the system calls offered by the system are to Linux-like environment, the simpler it should be to readjust it. This also holds for network functionalities. This part was ignored during the study, but its implementation should be considered in the portability.

Ideally, the used OSes should be selected based upon the portability issue, but that is not always possible. Interesting to note, the software running on the device might also have this

Operating system	WolfSSL	Mbed TLS	OpenSSL
FreeRTOS	✓	✓	
Embedded Linux	✓	✓	
In-house/custom	?	?	?
Android	✓	✓	✓
Debian (Linux)			✓
Ubuntu	✓	✓	✓
Micrium (uC/OS-III)	✓		
Texas Instruments RTOS	✓		
Micrium (uC/OS-II)			
Microsoft Windows Embedded 7/Standard	✓	✓	
Express Logic (ThreadX)	✓	✓	
Keil (RTX)	✓	✓	
Texas Instruments (DSP/BIOS)			
Freescale MQX	✓		
Wind River (Linux)			✓
Microsoft (Windows 7 Compact or earlier)	?	?	
Wind River (VxWorks)	✓	✓	
Red Hat (IX Lunix)			
AnalogDevices (VDK)	?	?	
Green Hills (INTEGRITY)	✓		
QNX (QNX)	✓		
Segger (embOS)	✓	✓	
Mentor Graphics Linux	?		
Wittenstein HIS(OpenRTOS/SAFERTOS	✓		
Angstrom (Linux)	?	?	✓
Overall	15	9	4

Table 8.1: Cryptographic library support for embedded operating systems

Operating system	ELF as standard for executable
FreeRTOS	✓
Embedded Linux	✓
In-house/custom	?
Android	✓
Debian (Linux)	✓
Ubuntu	✓
Micrium (uC/OS-III)	✓
Texas Instruments RTOS	✓
Micrium (uC/OS-II)	✓
Microsoft Windows Embedded 7/Standard	

Table 8.2: ELF file format support for the top 10 embedded operating systems

problem. Selecting the platform based on the portability of the software will result in an advantage in the portability of the attestator itself.

If the supported executable format is not of ELF type, it is reasonable to think that also the supported system calls will be different from a standard Linux environment. The script developed in Chapter 6 is also targeted to the ELF format. This means adapting the attestator for not-ELF executables file format will require a lot more effort.

By intersecting the data about the cryptographic libraries' support and the executable file format, it is possible to extract some information.

If the operating systems not supporting ELF executable are excluded from the potential portability, about 8 out of the top 10 operating systems use that format. Out of these 8:

- 6 of them support WolfSSL;
- 4 support Mbed TLS;
- 3 support OpenSSL.

Overall, only 1 out of 8 does not support any cryptographic library. So, out of the top 10 operating systems, 7 of them offer both cryptographic support and use ELF as the standard executable format. The attestator should be reasonably portable to those systems.

It is also possible to extract some percentages, but the data is outdated and it might not be reliable.

Chapter 9

Conclusions

This chapter describes the conclusions of the analysis that has been made in this thesis. As described in Chapter 1, this analysis was aimed at answering two questions.

The first one regards the ASPIRE attestator's portability problems. Chapter 6 shows creating a standalone version of the attestator is possible. Separating the attestator from the entire ASPIRE project resolves some critical dependencies, such as Diablo[1]. Some dependencies, such as the OpenSSL[2] library, are not solved in that build. Chapter 8 proposes some alternatives to the significant requisites the attestator has. In general, relaxing most of the attestator's requirements is achievable by replacing its major dependencies.

The second point regards to what extend the attestator is portable, with a particular focus on embedded platforms. Chapter 8 analyzes this issue and shows the potential portability is extensive. As explained in Section 8.4, out of the top ten operating systems, 7 of them offer the needed functionalities. This means the attestator's requirements are solvable for most of the environments and the attestator should be extensively portable.

Two problems were not considered during this analysis. Ignoring them might lead to the wrong conclusion.

The first one is the network functionality. It was mentioned multiple times during this thesis but it was never addressed. Remote attestation is by definition a remote procedure. Without network communication, verification by a trusted server cannot occur. Communication is a critical part of the attestator's security scheme. Even if the attestation procedure is protected from tampering, the network element might become the weak spot. If the communication is done in plaintext, an attacker can intercept and alter the communication to always result in a positive verification. For this reason, the connection side of the attestator must be secured as well. It is important to note that this might also restrict some requirements and it might limit the portability to some systems.

Also, a security flaw in the communication component of the attestator has to be considered. The attestation procedure is completely separate from the functionality of the software it is protecting. An attacker will be able to distinguish between the traffic sent by the device for its normal functionality and the one related to the attestator's procedure. There is no trivial solution to this problem. An option is to merge the attestator's communication with the one of the protected software. The problem with this solution is its implementation. Mixing the two parts has the downside of strictly coupling the attestator and the protected software's code. This holds both for the client and the server endpoint. If the attestator has to be strictly incorporate inside the device's code, inserting it as protection will require more effort and modifications of the protected software. It will not be possible to have a separate component easily configurable and deployable, like the one offered by the ASPIRE framework. It is also necessary to consider the fact

[1] https://diablo.elis.ugent.be

[2] https://www.openssl.org

that the server component of the application will have to be combined with the attestator as well. Ideally a component server-side will be in charge of identifying application and attestation portions of traffic. Once the two are separated, it will deliver them to the correct service. This component, which is basically a reverse-proxy, is not trivial to develop, but would partially separate the security mechanism and the application logic on the server endpoint. On the client side doing this de-coupling might be more complicated. The difficulty in implementing integrated communication is the main reason why that component is kept separated in the ASPIRE project.

The other problem which was not addressed is the complete obfuscation of the attestator component. Chapter 7 shows some potential tools for obfuscating the software. Tigress is probably the best option, as it offers more functionalities and it has more configuration capabilities. It is possible to obfuscate most of the attestator's code by using this tool. The problem regards the blob used by the attestator to retrieve information about the protected memory areas. It is not possible to obfuscate the blob itself with standard Tigress techniques. Only the variables holding the information extracted from the blob can be protected with the encoded data and arithmetic methods mentioned in Section 7.3 Chapter 7.

The blob itself cannot be obfuscated for two reasons.

- The blob is inserted after the executable is compiled, while Tigress works at the source-code level.
- The blob is declared as a single chunk of memory. Only the attestator's code can extract the information contained in it. The Tigress tool does not know how to encode the information contained in the blob, since only the attestator knows the meaning of it.

This might lead to a serious security issue. If an attacker can figure out the meaning of the information inside the blob, a memory-copy attack to the attestator becomes trivial. The attacker can create a valid copy of the protected code somewhere in memory. By modifying the blob, it is possible to redirect the attestator to that area. At this point, the attacker can modify the real code without occurring in detection. There are two potential fixes to this problem. The first one is implementing some variety of obfuscation mechanism inside the attestator itself. The patching script should be modified to insert the properly obfuscated information inside the blob. The information present in the blob will then be de-obfuscated during the extraction. The other solution is to declare the blob data directly inside the source code. Tigress will then obfuscate this data with its mechanisms. As explained in Section 6.1 Chapter 6, this solution was discarded for anti-reverse reasons, but it might be valuable if the blob is obfuscated. In general, the solution to this problem should not affect the attestator's portability in any way.

Overall, considering these two issues, the portability of the attestator itself should not be affected critically. The attestator still has some security problems, but it is important to note that such security method cannot be unbreakable. Since the software running on a third-party device can always be altered, particularly without using dedicated hardware, tampering attacks are always possible.

Appendix A

Manuals

A.1 User manual

This section contains the user manual of the software used during this thesis. It contains the description of the project's structure and the purpose of each file. It also outlines the configuration and the usage of the script that can be utilized to patch the protected binary.

It is important to note that this version of the attestator is a proof of concept. The protection mechanism is incomplete without a check performed server-side. Since the communication part is not included in this version, this attestator is usable for nothing but tests.

A.1.1 Requirements

Operating Systems

The attestator can be tested on a Linux-like system with x86-64 bits platform. It was tested on the following operating systems:

- Ubuntu server 18.04.3 LTS Bionic Beaver with gcc version 7.4.0;
- Kali GNU/Linux Rolling 2019.1 with gcc version 8.3.0.

In general, the build should work on any Linux-like platform with 64 bits Intel architecture.

Other dependencies

There are also some mandatory dependencies:

- OpenSSL library versions 1.1.1 or 1.1.1b;
- ruby version 2.5.5p157.

Some ruby gems are required for the patching script to execute. Those are the following:

- metasm;
- elftools;
- optparse;
- json.

GCC and OpenSSL are required to successfully compile the POC of the attestator. Ruby and its libraries are used by the patching the script.

A.1.2 Creation Process

Compiling the binary

There are two phases in creating a binary with a functional test attestator, as outlined in Section 6.1.

The first step is compiling a binary by merging the project source code and the attestator source code. GCC was used to compile the code. It is important to avoid the use of `-s` flag when compiling with GCC. The patching script requires relocation information to execute. If the binary is stripped, this information is unavailable and the script will fail.

To compile the project with GCC:

```
$ gcc $source_directory/*.c -o $test_filename -lpthread -lcrypto -DDEBUG
    -DDEBUG_ADS_PARSE -DHARD_DEBUG
```

`$source_directory` is the directory containing all the C source code files and `$test_filename` is the output filename.

The above command compiles the program with ASLR enabled. Two more flags are needed to test it without layout randomization: `-no-pie` and `-fno-pie`.

Configuring the attestator

The next step is the configuration. An example configuration can be found in the *config.json* file (see Listing A.1). Table A.1 shows the usage of each configuration file's field. In general, only the `protect` field should be modified. The other parameters require modifications in the source code of the attestator itself.

```
{
  "attestator_number" : 255,
  "aid_size" : 16,
  "blob_structure_name" : "ra_data_structure_blob",
  "protect": [
    "main"
  ],
  "base_sym" : "text_seg_offset"
}
```

Figure A.1: Example configuration file.

Patching the binary

Once the configuration file is filled, it is possible to run the patching script. The script can be executed with the commands shown in Listing A.2. The first command ensures the patching script has executable privileges. To list how to use the script, just run it without any arguments, as shown at line 2. The script requires two parameters:

- `-e` or `--elf` parameter specifies the target elf file to patch.
- `-c` or `--config-file` parameter specifies the configuration file name.

Line 6 of Listing A.2 shows an example of how to run it on an executable file. If there is an error in the configuration file or the binary does not contain the required information, the script will output an error to the console. A list of common errors is explained in Table A.2.

Field	Description
attestator_number	The attestator identifier. It might be possible to have multiple attestators in future versions.
aid_size	The attestator id size in bytes. Default should be 16 bytes.
blob_structure_name	The name of the blob structure inside the executable. Change this parameter only to match the one in the source code files. The name has to be enclosed in double quotes.
protect	This field contains the list of functions the attestator has to protect. Functions should be specified by name, enclosed in double quotes and separated by comma.
base_sym	This field contains the name of the text segment offset. It should be changed only to match the one declared in the source code files. The name has to be enclosed in double quotes.

Table A.1: Configuration file fields' description

```
1  $ chmod +x patch_ra_data_blob.rb
2  $ ./patch_ra_data_blob.rb
3  Usage: patch_ra_data_blob [options]
4      -e, --elf ELF_FILE The elf file to patch
5      -c, --config-file CONFIG_FILE The json configuration file
6  $ ./patch_ra_data_blob.rb -e $test_filename -c config.json
7  Reading the config file...
8  Check if the binary is of type ET_EXEC or ET_DYN...
9  Check Address Space Layout Randomization...
10 ASLR => true
11 Check if the binary has a symtab...
12 Symbol table found
13 Searching for text_seg_offset...
14 Searching for ra_data_structure_blob...
15 Base symbol found at offset 0x91c0
16 Blob found at offset 0x9180
17 Searching for symbol main...
18 Symbol main found: {:memory_offset=>3974, :size=>43}
19 Patching the base symbol...
20 Aslr is enabled => Searching for the relocation entry...
21 Patching the blob...
22 Blob patched correctly!
```

Figure A.2: Example running the patching script

Automating the creation process

A script performing the above operations is included in the project. Instead of compiling the binary and executing the script manually, it is possible to perform these two operations by running the *test-build.sh* script.

```
$ chmod +x test-build.sh
$ ./test-build.sh
[*] Compiling the test file...
[*] Test file compiled
[*] Patching the binary...
[...]
[*] ./a.out to run the test
```

Error	Description
Unable to parse the config file	The script was unable to parse the configuration file.
aid_size field not set	The aid_size field is not correctly set in the configuration file.
blob_structure_name field not set	The blob_structure_name field is not correctly set in the configuration file.
protect field not set	The protect field is not correctly set in the configuration file.
protect field empty	The protect field is empty in the configuration file.
attestator_number not set	The attestator_number field is not correctly set in the configuration file.
Function names must be of type String	The protect field is not declared as an array of strings. Each string should be declared within double quotes.
Duplicate function names	There are duplicates in the protect field list.
base_sym field not set	The base_sym field is not correctly set in the configuration file.
There are too many memory areas to fit in 2 bytes	The maximum number of protected functions has been reached.
File is not of type ET_EXEC or ET_DYN	The file specified in the `--elf` parameter is not an executable.
Looks like the binary is stripped!	The file specified in the `--elf` parameter appears to be stripped. The script does not work on stripped executables.
Cannot find the symbol for the base address	The script could not find the base address' symbol in the executable. The base_sym field should be set accordingly to the project source code.
Cannot find the symbol for the blob data structure	The script could not find the blob structure's symbol in the executable. The blob_structure_name field should be set accordingly to the project source code.
Cannot find the segment where the blob and the base symbol are	The script was unable to locate the segment containing the blob and the base address symbols.
Cannot find the symbol for the [function-name] function	The script was unable to locate the symbol for the [function-name] function. That function might not exist.
Cannot find the relocation entry	The script was unable to find the relocation entry in the executable.
Error patching the target file!	There was an error during the patching of the executable. The user's permission might be insufficient.

Table A.2: Errors description

Running the test executable

Once the script successfully patched the binary, it is possible to test it by running the executable itself, as shown in the test (Section 6.2).

```
$ ./a.out
[...]
```

The output will show the attestation procedure and the computed hash for the hardcoded

nonce. The *attestator.c* file has to be modified to test the attestator with a different nonce. The default nonce is set as the following. Both the value and the size of the nonce can be modified.

```
uint8_t nonce[8] = {0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8};
```

A.2 Developer manual

This section contains the developer manual for the software developed during this thesis. It describes the structure of the project and it outlines the open issues not resolved in this test case.

A.2.1 Test attestator's structure

Since the separation of the attestator from the ASPIRE project was discussed in Chapter 6, reading it is highly recommended.

The attestator's source can be found in the *src* folder. Table A.3 shows a brief content's description of each file. Most of the source code of the attestator is extracted from the ASPIRE project [3]. It was developed fully by Ph.D. Alessio Viticchiè. To get a full reference, consult the GitHub page of the project[1].

The source code of the attestator has to be compiled within the source code of the project it has to protect. In this example, the *hello_world.c* file represents the source of the unprotected program. It is essential to understand the procedure the attestator uses to retrieve the information it requires. Once the program is compiled with the attestator, some memory space is reserved for a chunk of memory called blob. The blob is filled with junk values after the compilation process. The attestator has to extract the correct information from the blob at runtime to properly operate. This information includes the position and the size of every function it has to protect. At program startup, the attestator retrieves the information from the blob and fills the structures that control the attestation process. These structures are outlined in Chapter 2 Section 2.1.

A script is needed to insert the proper data inside the blob. *patch_ra_data_blob.rb* is responsible for this task. It contains a simple ruby script that parses the directives from the configuration file and patches the binary accordingly. The configuration is in the *config.json* file.

To fix the blob's content, the script has to fill it with the correct information. It is important to note that the script has to insert the data not in the form of structs, but as raw bytes. To correctly inject these bytes, the attestator's pre-configuration function was analyzed. To be compliant with the attestator, the script has to insert bytes in the following format:

- Attestator ID (16 Bytes);
- Attestator number (8 Bytes);
- Memory areas count (4 Bytes);
- List of memory areas with the following structure:
  - Memory area label (2 Bytes);
  - Memory blocks count (4 Bytes);
  - List of memory blocks with the following structure:
    - Memory block offset (8 Bytes);
    - Memory block size (4 Bytes).

Figure A.3 shows the role the script and the executable have in regards to the blob. The script inserts the correct bytes into it, based upon the configuration file and the binary itself. The executable extracts this information and fills the structures needed by the attestator. The blob is effectively responsible for the transmission of data between the script and the attestator itself.

[1] https://github.com/aspire-fp7/remote-attestation/

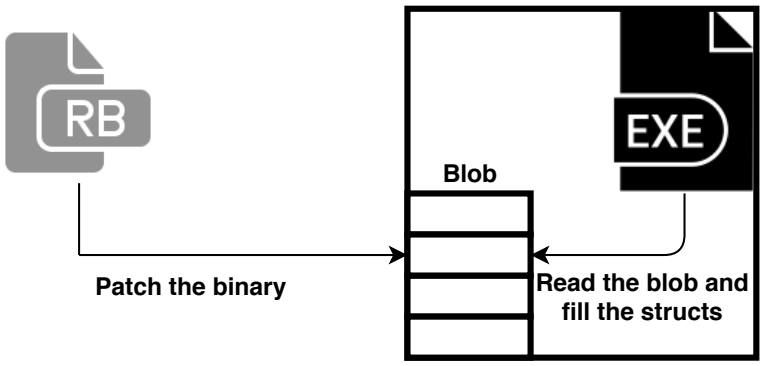

Figure A.3: Information exchange overview.

File	Content description
attestator.c and attestator.h	This is the main source for the attestator.
generic_function.c and generic_function.h	These files are needed by the random walk logic.
hello_world.c and hello_world.h	This is the main function of the program itself. This should be changed to match the program to protect.
ra_blob.c and ra_blob.h	This is the declaration of the blob containing the attestator's information.
ra_data_preparation.c and ra_data_preparation.h	These files contain the random walk logic.
ra_data_table.c and ra_data_table.h	These files contain the logic for extracting the information from the blob.
ra_defines.h	This file contains constants' definitions.
ra_do_hash_md5.c and ra_do_hash.h	These files contain the procedure for computing the hash over memory. In this case, an md5 hash is performed, but it is possible to use different hashes.
ra_memory.c and ra_memory.h	These files contain the functions needed for memory extraction.
ra_nonce_interpretation_1.c and ra_nonce_interpretation.h	These files contain the logic for the interpretation of the nonce. Different possible interpretations are usable.
ra_print_wrapping.h	This file is a wrapper for print functions.
ra_results.h	This file contains the definitions for the attestator's results.

Table A.3: Source files' description.

A.2.2 The patching script

The patching script is a simple ruby script. It follows the sequence outlined in Chapter 6 Section 6.1. This sequence boils down to three steps.

The first part is the parsing of the configuration file. It specifies the list of functions the

attestator has to check. Since the configuration file is in JSON format, the script uses a JSON parsing library called *json*. The script parses the configuration and checks its correctness. If an error is found in the configuration file, an error is displayed to the user.

The second step is the parsing of the binary information. The script uses two libraries to parse the executable: *metasm* and *eltools*. Since this data is extracted from the symbols' information, the binary must not be stripped. If it is, the script will display an error and exit. The name convention used within the attestator's source has to match with the configuration file. If the used names are different, the script will not be able to find the corresponding symbols inside the binary and will throw an exception.

The third and final step is the patching of the binary itself. The script uses the information retrieved in the previous steps to fill the blob with the correct values. It also fixes the base symbol by pointing it to the beginning of the text segment. In this phase, the library *elftools* is used.

The patching of the base symbol is particularly interesting because it is different with and without address space layout randomization. If ASLR is enabled, the script will look for the relocation entry in the .rela.dyn section. It will then patches the offset of that entry to point to the text segment offset. Listing A.4 shows the core lines of this procedure.

```
[...]
# Find the relocation entry in the .rela.dyn section
base_reloc = e.section_by_name('.rela.dyn').relocations.find { |s|
    s.header.r_offset == @base_sym_vaddr }
[...]
# Patch the offset
base_reloc.header.r_addend = @text_seg.offset
[...]
```

Figure A.4: Patching procedure with ASLR enabled.

If the program is compiled as ET_EXEC (ASLR disabled), the procedure is more straightforward. The script has to find the base symbol in the binary and modify its content. Listing A.5 shows the sequence the script follows in this case.

```
[...]
# Find the base symbol position in the binary
base_sym_offset = base_sym.value - data_seg.vaddr
@base_sym_pos = data_seg.offset + base_sym_offset
[...]
# Patch the offset
File.write(@filename, [@text_seg.vaddr].pack('Q'), @base_sym_pos)
[...]
```

Figure A.5: Patching procedure without ASLR enabled.

A.2.3 Open issues

Communications

As already mentioned multiple times during this paper, the standalone attestator is not complete. The communication and the server-side functionalities were not taken into consideration. These

components are included in the original ASPIRE attestator[2]. Since the implementation is peculiar to the ASPIRE project, they were not inserted in the test.

An overview of how ASPIRE communication operates can be found in Chapter 2 Section 2.1. Any implementation should match this general structure. There are two critical factors to consider, in the development of such functionality.

- The communication has to be compliant with the device. Low-end devices might support only a few communication protocols. Ideally, the server should be capable of handling different types of devices and communications.
- The communication has to be protected from MITM (Man-In-The-Middle) attacks. If an attacker can tamper with the messages, he might bypass the attestation procedure. Thus, encryption is essential.

Since the attestator relies on a trusted server-side check, without communication it can only be used for testing purposes. So, the development of this component is a requisite for the attestation scheme.

There is another significant factor to examine in the development of the communication element. As already mentioned, the ASPIRE project's attestator uses its distinct channel to interact with the server. To increase the security of the attestation, the communication channel used by the attestator should be the same used by the application itself. This is not a trivial solution, as it requires a re-design of the application's structure, but the advantages it offers are worth examining the development options.

Support for multiple attestators

The patching script was designed to be as simple as possible. For this reason, every function was assigned an entire memory area inside the blob. This limits the maximum number of functions the attestator can protect. As seen in Section A.2.1, the attestator is capable of handling more than one block for each memory area. This functionality should be introduced in the script to maximize the ASPIRE attestation component's potential.

The user should be able to specify which memory areas every function refers to. The reading of the JSON configuration file has to be modified to develop this functionality. The script will also have to patch the blob accordingly to this modification.

The attestator can be fully functional without this feature, but it is a good plus.

[2] https://github.com/aspire-fp7/remote-attestation/

Appendix B

Examples

```
$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
    linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
    BuildID[sha1]=adde3daa1937aa688874049761d709961b005a1a, not stripped
$ readelf -h hello_world
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x401040
  Start of program headers: 64 (bytes into file)
  Start of section headers: 14560 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 11
  Size of section headers: 64 (bytes)
  Number of section headers: 29
  Section header string table index: 28
```

Figure B.1: Example of reading an ELF header using the readelf tool.

```
$ readelf -l hello_world

Elf file type is EXEC (Executable file)
Entry point 0x401040
There are 11 program headers, starting at offset 64

Program Headers:
  Type Offset VirtAddr PhysAddr
               FileSiz MemSiz Flags Align
  PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x0000000000000268 0x0000000000000268 R 0x8
  INTERP 0x00000000000002a8 0x00000000004002a8 0x00000000004002a8
               0x000000000000001c 0x000000000000001c R 0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x0000000000000438 0x0000000000000438 R 0x1000
  LOAD 0x0000000000001000 0x0000000000401000 0x0000000000401000
               0x00000000000001ad 0x00000000000001ad R E 0x1000
  LOAD 0x0000000000002000 0x0000000000402000 0x0000000000402000
               0x0000000000000150 0x0000000000000150 R 0x1000
  LOAD 0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
               0x0000000000000220 0x0000000000000228 RW 0x1000
  DYNAMIC 0x0000000000002e20 0x0000000000403e20 0x0000000000403e20
               0x00000000000001d0 0x00000000000001d0 RW 0x8
  NOTE 0x00000000000002c4 0x00000000004002c4 0x00000000004002c4
               0x0000000000000044 0x0000000000000044 R 0x4
  GNU_EH_FRAME 0x0000000000002014 0x0000000000402014 0x0000000000402014
               0x000000000000003c 0x000000000000003c R 0x4
  GNU_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000 RW 0x10
  GNU_RELRO 0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
               0x00000000000001f0 0x00000000000001f0 R 0x1

[...]
```

Figure B.2: Example of reading ELF segments using the readelf tool.

```
There are 29 section headers, starting at offset 0x38e0:

Section Headers:
  [Nr] Name Type Address Off Size ES Flg Lk Inf Al
  [ 0] NULL 0000000000000000 000000 000000 00 0 0 0
  [ 1] .interp PROGBITS 00000000004002a8 0002a8 00001c 00 A 0 0 1
  [ 2] .note.ABI-tag NOTE 00000000004002c4 0002c4 000020 00 A 0 0 4
  [ 3] .note.gnu.build-id NOTE 00000000004002e4 0002e4 000024 00 A 0 0 4
  [ 4] .gnu.hash GNU_HASH 0000000000400308 000308 00001c 00 A 5 0 8
  [ 5] .dynsym DYNSYM 0000000000400328 000328 000060 18 A 6 1 8
  [ 6] .dynstr STRTAB 0000000000400388 000388 00003d 00 A 0 0 1
  [ 7] .gnu.version VERSYM 00000000004003c6 0003c6 000008 02 A 5 0 2
  [ 8] .gnu.version_r VERNEED 00000000004003d0 0003d0 000020 00 A 6 1 8
  [ 9] .rela.dyn RELA 00000000004003f0 0003f0 000030 18 A 5 0 8
  [10] .rela.plt RELA 0000000000400420 000420 000018 18 AI 5 22 8
  [11] .init PROGBITS 0000000000401000 001000 000017 00 AX 0 0 4
  [12] .plt PROGBITS 0000000000401020 001020 000020 10 AX 0 0 16
  [13] .text PROGBITS 0000000000401040 001040 000161 00 AX 0 0 16
  [14] .fini PROGBITS 00000000004011a4 0011a4 000009 00 AX 0 0 4
  [15] .rodata PROGBITS 0000000000402000 002000 000011 00 A 0 0 4
  [16] .eh_frame_hdr PROGBITS 0000000000402014 002014 00003c 00 A 0 0 4
  [17] .eh_frame PROGBITS 0000000000402050 002050 000100 00 A 0 0 8
  [18] .init_array INIT_ARRAY 0000000000403e10 002e10 000008 08 WA 0 0 8
  [19] .fini_array FINI_ARRAY 0000000000403e18 002e18 000008 08 WA 0 0 8
  [20] .dynamic DYNAMIC 0000000000403e20 002e20 0001d0 10 WA 6 0 8
  [21] .got PROGBITS 0000000000403ff0 002ff0 000010 08 WA 0 0 8
  [22] .got.plt PROGBITS 0000000000404000 003000 000020 08 WA 0 0 8
  [23] .data PROGBITS 0000000000404020 003020 000010 00 WA 0 0 8
  [24] .bss NOBITS 0000000000404030 003030 000008 00 WA 0 0 1
  [25] .comment PROGBITS 0000000000000000 003030 00001c 01 MS 0 0 1
  [26] .symtab SYMTAB 0000000000000000 003050 0005b8 18 27 43 8
  [27] .strtab STRTAB 0000000000000000 003608 0001cf 00 0 0 1
  [28] .shstrtab STRTAB 0000000000000000 0037d7 000103 00 0 0 1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figure B.3: Example of reading ELF sections using the readelf tool.

```
$ readelf -l hello_world --wide

[...]

 Section to Segment mapping:
  Segment Sections...
   00
   01 .interp
   02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
      .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03 .init .plt .text .fini
   04 .rodata .eh_frame_hdr .eh_frame
   05 .init_array .fini_array .dynamic .got .got.plt .data .bss
   06 .dynamic
   07 .note.ABI-tag .note.gnu.build-id
   08 .eh_frame_hdr
   09
   10 .init_array .fini_array .dynamic .got
```

Figure B.4: Example of ELF sections-segments overlapping using the readelf tool.

```
int f1() {
    int a = 0;
    int b = 3;
    a += 4 * 4 * b;
    a -= b / 3;
    return a;
}

int main() {
    f1();
    return 0;
}
```

The above code is transformed as the following:

```
int main(int _formal_argc , char **_formal_argv , char **_formal_envp )
{
 [...]
  f1();
}
void _1_f1_f1_split_2(int *a , int *b )
{
  *a -= *b / 3;
}

void _1_f1_f1_split_1(int *a , int *b )
{
  *a = 0;
  *b = 3;
  *a += 16 * *b;
}

int f1(void)
{
  int a ;
  int b ;
  _1_f1_f1_split_1(& a, & b);
  _1_f1_f1_split_2(& a, & b);
  return (a);
}
```

Figure B.5: Example of the Tigress function splitting functionality.

Bibliography

[1] Trusted Computing Group, "Trusted Computing Group Homepage." https://trustedcomputinggroup.org/, Accessed: 2019-07-13

[2] Aspire-fp7 Project, "The central ASPIRE framework repository." https://github.com/aspire-fp7/framework, Accessed: 2019-08-17

[3] Aspire-fp7 Project, "ASPIRE Reference Architecture v2.1." https://aspire-fp7.eu/sites/default/files/D1.04-ASPIRE-Reference-Architecture-v2.1.pdf, Accessed: 2019-08-17

[4] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices", May 2004, pp. 272–282, DOI 10.1109/SECPRI.2004.1301329

[5] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the Difficulty of Software-based Attestation of Embedded Devices", 2009, pp. 400–409, DOI 10.1145/1653662.1653711

[6] L. van Doorn, "Refutation of "On the Difficulty of Software-Based Attestation of Embedded Devices" Adrian Perrig CyLab / CMU", 2010

[7] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems", ACM SIGOPS Operating Systems Review, vol. 39, 2005 2005, pp. 1 – 16

[8] G. Grimen, C. Mönch, and R. Midtstraum, "Tamper Protection of Online Clients through Random Checksum Algorithms.", 01 2006, pp. 67–79

[9] Y. Li, J. M. McCune, and A. Perrig, "SBAP: Software-Based Attestation for Peripherals", vol. 6101, 07 2010, pp. 16–29, DOI 10.1007/978-3-642-13869-0_2

[10] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "SCUBA: Secure Code Update by Attestation in sensor networks", WiSE 2006 - Proceedings of the 5th ACM Workshop on Wireless Security, vol. 2006, 01 2006, pp. 85–94, DOI 10.1145/1161289.1161306

[11] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software Attestation for Key Establishment in Sensor Networks", Ad Hoc Networks, vol. 9, 06 2008, pp. 372–385, DOI 10.1007/978-3-540-69170-9_25

[12] A. Al-Wosabi and Z. Shukur, "Software tampering detection in embedded systems - a systematic literature review", Journal of Theoretical and Applied Information Technology, vol. 76, no. 2, 2015, pp. 211–221

[13] G. Coker, J. Guttman, P. Loscocco, A. L. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. T. Sniffen, "Principles of remote attestation", Int. J. Inf. Sec., vol. 10, 06 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[14] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares", 08 2014

[15] K.-H. Baek, S. Bratus, S. Sinclair, and S. Smith, "Attacking and Defending Networked Embedded Devices", 01 2007

[16] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote Software-Based Attestation for Wireless Sensors", 07 2005, pp. 27–41, DOI 10.1007/11601494_3

[17] A. Viticchié, C. Basile, A. Avancini, M. Ceccato, B. Abrath, and B. Coppens, "Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks", 2016, pp. 73–84, DOI 10.1145/2995306.2995315

[18] Tool Interface Standard (TIS), "Executable and Linkable Format (ELF)." http://www.skyfree.org/linux/references/ELF_Format.pdf, Accessed: 2019-07-12

[19] R. E. O'Neill, "Learning linux binary analysis", Packt Publishing, 2016, ISBN: 1782167102, 9781782167105

[20] Ian Wienand, "PLT and GOT: the key to code sharing and dynamic libraries." https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html, 2014, Accessed: 2019-08-14
[21] Radare2 Team, "Radare2 github repository." https://github.com/radare/radare2, 2017
[22] R. Team, "Radare2 book." https://radare.gitbooks.io/radare2book/content/, 2017
[23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis", 2016
[24] Metasm, "A cross-architecture assembler, disassembler, compiler, linker and debugger." https://www.cr0.org/progs/metasm/, Accessed: 2019-08-17
[25] The LLVM Compiler Infrastructure, "LLVM Project Homepage." https://llvm.org/, Accessed: 2019-08-20
[26] T. László and k. Kiss, "Obfuscating C++ programs via control flow flattening", Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica, vol. 30, 08 2009, pp. 3–19
[27] The Tigress C Diversifier/Obfuscator, "Tigress, a diversifying virtualizer/obfuscator for the C language." http://tigress.cs.arizona.edu/index.html, Accessed: 2019-08-20
[28] M.-L. P. Jonathan Salwan, Sébastien Bardin, "Symbolic Deobfuscation: From Virtualized Code Back to the Original", 06 2018, pp. 372–392, DOI 10.1007/978-3-319-93411-2_17
[29] S. Andrivet, "C++11 metaprogramming applied to software obfuscation", 2014
[30] AspenCore Media Group, "Embedded Market Study 2017." https://m.eet.com/media/1246048/2017-embedded-market-study.pdfl, 2017, Accessed: 2019-08-23