FELADATKIÍRÁS

Leitold Márton Ferenc

mérnök informatikus hallgató részére

Memória snapshot alapú rootkit detekció beágyazott Linux környezetben

A rootkitek olyan kártékony szoftverek, melyek fejlett rejtőzködési technikákat alkalmaznak annak érdekében, hogy nehéz legyen jelenlétüket kimutatni egy fertőzött számítógépen. Értelemszerűen, fontos feladat a rootkitek detektálására alkalmas technikák, megoldások fejlesztése és vizsgálata. Rootkit detekcióra több hozzáállás is létezik, ezek közül jelen feladat fókuszában azok a megoldások állnak, melyek kizárólag az OS kernelének egy adott pillanatban rögzített memória képében, egy ún. memória snapshotban próbálnak rootkit jelenlétére utaló nyomokat, anomáliákat azonosítani. Ezek a megoldások jól alkalmazhatóak olyan beágyazott eszközökön, ahol egy megbízható végrehajtási környezetben futó megbízható alkalmazás bizonyos időközönként megkapja a vezérlést, és hozzáfér a nem megbízható környezetben futó OS kernel memóriájának pillanatnyi képéhez.

A hallgató feladat a szakirodalom áttekintése, egy memória snapshot alapú rootkit detekciós módszer kiválasztása, megértése, és fontosabb részeinek implementálása proof-of-concept jelleggel beágyazott Linux környezetben. Nem elvárás az OS kernel integritásának folyamatos monitorozása, de szükséges azon OS kernel struktúrák feltárása, melyeket egy rootkit potenciálisan módosíthat, és ezen struktúrák helyének meghatározása a kernel memória képében. Ilyen struktúrák lehetnek például a kernelben implementált függvények címeit tartalmazó struktúrák, melyeket egy rootkit átírhat a vezérlési folyam megváltoztatása céljából. A feladat része továbbá az elkészült implementáció működésének szemléltetése.

Tanszéki konzulens: Dr. Buttyán Levente, docens

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Leitold Márton Ferenc

Memory snapshot based rootkit detection in an embedded Linux environment

SUPERVISORS

Dr. Buttyán Levente

Futóné Papp Dorottya

BUDAPEST, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott **Leitold Márton Ferenc**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2019. 12. 13.

...…….

Leitold Márton Ferenc

Összefoglaló

A dolgok internete (angolul IoT) olyan eszközöket jelent, amelyek interneten vannak összekapcsolva egymással. Ezek az eszközök ma már egyre elterjedtebbé válnak. Sajnos az IoT eszközöket nem kellő mennyiségű biztonsággal látták el. Ennek okai lehetnek például a korlátolt hardveres erőforrás, túl gyors ütemű fejlesztés vagy kevés pénz. Így viszont fennáll annak a veszélye, hogy ezeket az eszközöket megtámadják és átveszik felettük az uralmat.

Egy lehetséges megoldás, hogy észleljük, ha a támadó átvette az irányítást az eszköz felett, a Kernel Integritásának Monitorozása. Nagyon sokféle módszer létezik ennek a megvalósítására, én Nick L. Petroni, Jr. és Michael Hicks [1] által javasolt megoldást választottam. Ebben a cikkben részletesen le van írva, hogy hogyan monitorozzák a kernelt és ezt hogyan kell megvalósítani. Az ötlet az, hogy a függvényekre mutató pointereket figyeljük, és ha ezeknek az értéke olyan memória címre mutat, amit nem ismerünk, akkor tudjuk, hogy a kernelt megtámadták. A kernel monitorozását nem lehet magán a monitorozott kernelen végezni, mert ha a kernel kompromittálódott, akkor a monitorozás eredménye már nem lehet megbízható. Egy megoldás erre, ha egy olyan rendszerben monitorozunk, aminek van elválasztott biztonságos világa, és ebből monitorozzuk a normális világot. A biztonságos világban tudunk egy memóriaképet készíteni a normális világ kernelének memóriájáról és ezt bejárva meg tudjuk állapítani, hogy kompromittálódott-e. Egy ilyen megoldást nyújt az OP-TEE, ami egy nyílt szoftver és egy megbízható, futtatható környezetet valósít meg.

Ebben a szakdolgozatban bemutatom, hogy hogyan kell megszerezni a szükséges bemeneteket a kernel integritását monitorozó programhoz, ami később tud az OP-TEE-n futni. A szakdolgozat nagy részében a GCC-vel fogok foglalkozni, ami egy fordító program. A GCC-vel fordítom le a kernelt és a fordítás közben ki tudom nyerni a belső struktúrákat és függvény pointereket. Ezekkel, és még néhány bemutatott adattal később megvalósítható a Kernel Integritásának Monitorozása egy programmal.

Abstract

Internet of Things, physical devices that are connected to the internet, are getting more widespread than ever before. Unfortunately, these devices are developed with insufficient security features because of limited time, allocated funds or limited hardware resource. This can result in these devices being compromised.

One way to harden the security and to defend against attacks is called Kernel Integrity Monitoring. This way, we can make sure that the kernel is secure and not taken over. There are many approaches that exist, and I chose Nick L. Petroni, Jr. and Michael Hicks paper [1], because there is sufficient documentation in the article on how they implemented it, and it can serve well to protect the kernel. The idea in that paper is that we can monitor function pointers, and if those point to unauthorized code region, then we can tell that the kernel cannot be trusted. Kernel Monitoring cannot be implemented on the kernel that we are monitoring because the result is not credible. A solution to this can be with a system that runs a normal world and a secure world, and from the secure world we can take a snapshot of the normal world kernel's memory and check the integrity of it. OP-TEE, an open source trusted execution environment implements the secure and the normal world.

In this thesis I will describe how to create the necessary inputs for the kernel monitoring algorithm that will later can run on OP-TEE. For the main part, I needed to dive into GCC which is a compiler. For GCC, I wrote a plugin that can extract the function pointers and structures that contain function pointers from the normal world kernel. With these and some additional info from the kernel a Kernel Integrity Monitor can later be created.

1 Introduction

Background

The Internet of Things (IoT) are physical devices which are connected to the internet. These can be smart fridges, thermostats, smart lights, alarm clocks, sensors, etc. IoT are opening up new possibilities in our world. IoT devices are developing rapidly, right now there are about 8.3 billion IoT devices worldwide, but that number is expected to grow in the coming years[1]. By 2025 it is predicted that there will be more than 20 billion devices, as shown in Figure 1.

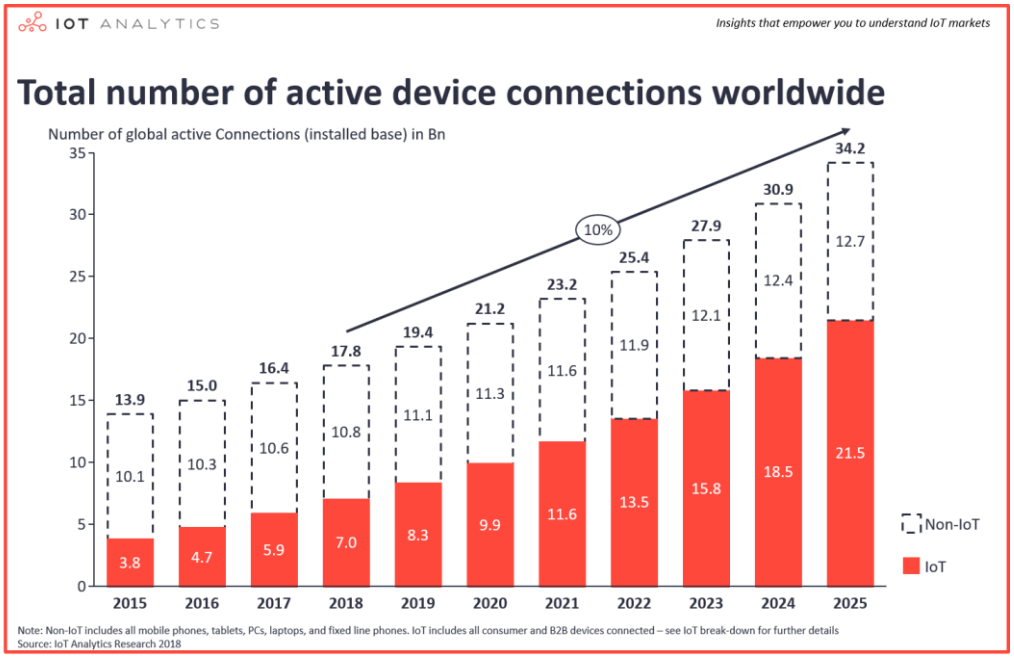

Figure 1 - Total number of active device connections worldwide

Therefore, it is important, that IoT devices have sufficient security, but that is often not the case. IoT devices are developed without many regulations about security and this creates some burdens. This can be because of the cost of implementing security features on these devices or hardware restriction. These devices are susceptible to several vulnerabilities including weak authentication, SQL injection, lack of updates, etc. These issues make IoT devices adapt slower, as businesses are worried about the

[1] https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ last visited on 2019.12.12

security concerns. Thus, Integrity Monitoring can help with this, by periodically monitoring the state of the system, and alerting if suspicious activity is found.

Kernel Rootkit

One way to monitor a system is to look for Kernel rootkits. Kernel rootkits are a possible vulnerability that this system faces. Kernel rootkits are a malicious software which can exploit an operating system. Rootkits can enable privileged control on the system and open a backdoor for later access. These types of malicious codes are hard to detect as they run in the kernel in the highest privilege and can hide themselves quite well. One of the various defense mechanisms against rootkits is Kernel Integrity Monitoring. One way to the monitoring is checking the control flow to discover any malicious changes.

There are many approaches to choose from to do Kernel Integrity Monitoring. My approach to Kernel Integrity Monitoring is based on the Nick L. Petroni, Jr. and Michael Hicks paper [1]. In this thesis I present how I adapted the article. I will present how I created a GCC Plugin that can extract necessary data from a Linux kernel. This information is matched with the exported symbols and can be used to be an input to an Integrity Monitor.

2 Related work

Integrity monitoring could only be considered using techniques that are isolated from the kernel which is being monitored. The reason is that monitoring done on the same device could be tampered and therefore, it cannot be trusted. Because of that I only considered solutions that are able to work with snapshots.

2.1 Kernel

Kernel is the main software component of the operating system. It interacts with low-level hardware and user-space software. It schedules the tasks that the operation systems runs are running and manages memory resources. It contains many device drivers, which are used to control some kind of hardware device, for example network cards or USB. It is the center of operations, new processes start here, and system calls are handled here as well. If the kernel stops working because it experiences a kernel panic, it can stop the whole operating system. Kernels run at the highest privilege, therefore, if a rootkit gains control, it will gain control of the whole system. In such scenario, no part of the system can be trusted, as user space programs can be easily modified as well. Therefore, it is very important that a kernel stays unharmed. A high level overview of the kernel can be seen on Figure 2.

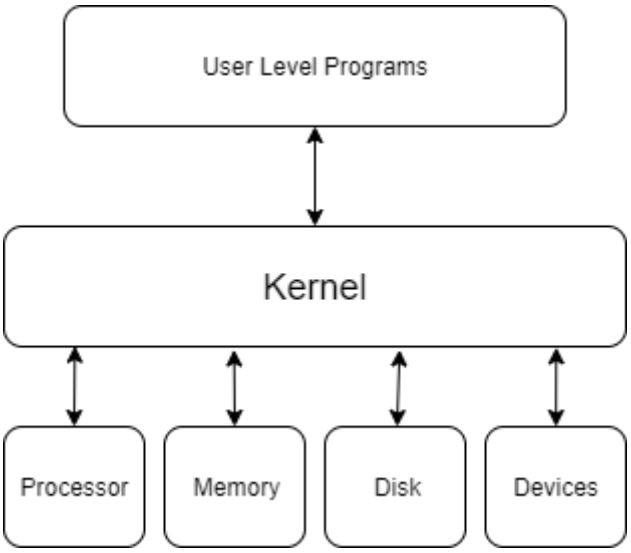

Figure 2 – High level overview of the kernel

2.2 Snapshot

A snapshot is a state of the system at a given time. It is usually used in high-availability systems to save the disk. Here we can use the snapshot to save the memory of the system for analyzing it later. This is, because when we need to access the kernel data from the normal world kernel, we are using the secure world kernel. The normal and secure world are a separation where the secure world is isolated from the normal world. If we would process the data in the secure world kernel and some error would happen that could cause the kernel to panic and the whole system would come to a stop. Thus, we are only using the secure world kernel to grab the data and process it in the secure world user space.

2.3 Copilot and Gibraltar

The first article I considered for solving this problem was Copilot [2]. Copilot is a Coprocessor-based Kernel Runtime Integrity Monitor. It uses a PCI card Monitor to connect to the Admin station and send the data found in the memory for the Admin station. Then the Admin station performs the check and can alert if the system is compromised. The goal is not to remove the rootkit, just to detect it. Another advantage is that it can still work well even if the system is heavily compromised. A disadvantage for me is that they use already existing tools to check for a rootkit presence while my task was to write my own program.

Gibraltar is a program which Detects Kernel-level Rootkits using Data Structure Invariants [3]. It implements an approach to satisfy non-control data structure integrity. It first fetches the kernel memory pages from the memory, then analyzes the raw memory pages and extracts data structures. It outputs a partial kernel snapshot. Later on, an Invariant generator processes these snapshots and infers the likely data structure invariants. But to discover which variables can change and what data they can hold, Gibraltar first needs to run in detection mode and perform some normal workload example, such as editing text files or running some codes. For this reason I found this method difficult and not well applicable in real life.

2.4 State based control flow integrity

State based control flow integrity was first proposed in [1]. The paper presents a method to monitor kernel integrity based on the control-flow integrity. Control-flow

integrity is a property that in case it fails to hold, then there is a violation in the kernel. A violation in a kernel means that it is likely that it got attacked and is not running as expected. A program satisfies control-flow integrity as long as its execution only follows paths according to the control-flow graph, determined in advance. Objects that can be modified during an attack:

- Kernel text
- Processor registers
- Function pointers
- Return addresses

The program extracts variables that point to functions in the kernel and in the snapshot. It traverses these function pointers and checks if these values point to valid functions and not to a malicious code regions. Function pointers are like normal pointers, but instead of some a data they point to a code region. Therefore, modifying a function pointer to point to arbitrary code can be very dangerous. An example function pointer is:

```
int add(int a, int b)
{
    return a+b;
}
int (*fun_ptr)(int, int) = add;
```

Where the `fun_ptr` points to a function called add, which takes integers as parameters and returns the sum of them as an integer. Validating a function pointer can be done in different ways:

- Validating a code region – where the pointed code is validated
- Validating a function start address
- Validating the function type

Detection can still miss some attacks if between two validations, the function pointer of some variable is modified, executed and changed, back because it searches for persistent attacks. Still, for me it seemed to be as an effective way to detect rootkits that create persistent changes in the kernel. I also opt for do the validation with validating function pointers.

2.5 TEE and Global Platform Technology

Trusted Execution Environment or TEE for short is a processor's secure area where code and data are protected and handled with confidentiality. A TEE is an isolated environment which runs simultaneously with the operating system. It also intends to provide a higher level of security, than a rich environment. Global Platform Technology specifies TEE [4] whose primary purpose is to protect its assets from other environments.

Use cases of a TEE include, but are not limited to:

- Authentication
- Digital Rights Management
- Mobile Financial services
- Cryptographic operations
- Secure key storage

Applications inside the TEE are called Trusted Applications. These TAs run in sandbox mode and are protected from each other with software and hardware isolation.

2.5.1 Arm's TrustZone Technology

Arm's TrustZone Technology [5] is a security extension which implements the TEE standard. It works with two virtual processors which are backed by hardware based access control seen on Figure 3. On these two virtual processors run two operating systems with each of their own kernel and software. The secure environment usually has less capabilities and also, it has less attack surface. From the rich environment the software can communicate with the secure environment with a Secure Monitor Call. The call triggers a context switch and execution will begin in the secure environment. Therefore, TrustZone makes hacking more difficult, but it adds complexity to the development.

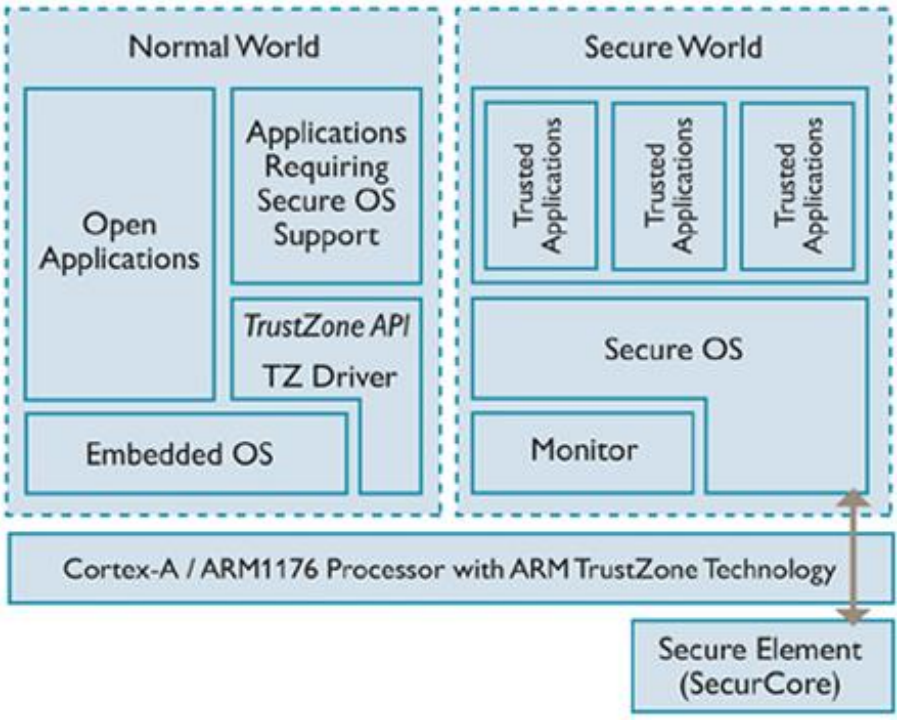

Figure 3 – The Architecture of Arm TrustZone Technology[2]

[2] https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html last visited on 2019.12.03

3 Design

In this chapter, I present my design for implementing kernel integrity monitoring based on [1]. I introduce the technologies I used and with them how I designed an implementation.

3.1 Technologies

In this part I would like to describe the technologies that are useful for the Integrity monitor. Understanding them helped me design and create the program to get the inputs.

3.1.1 Linux kernel

To construct the integrity monitoring, I need to be able to see the kernel structures and the source code. Linux kernel is one of the best solution for that, as it is an open-source kernel. There are many distributions that use Linux kernel, and also, it has been ported to different hardware. The Linux kernel is a Monolithic kernel [6], meaning it runs all the device drivers inside the kernel and not as an isolated user program like done in a Microkernel. This way, the Kernel space is bigger, and shown in Figure 4. There are also about 20 million lines of code in the Linux kernel maintained by thousands of developers. Most of these lines are code for various drivers.[3]

[3] https://www.phoronix.com/scan.php?page=news_item&px=Linux-19.5M-Stats last visited on 2019.12.08

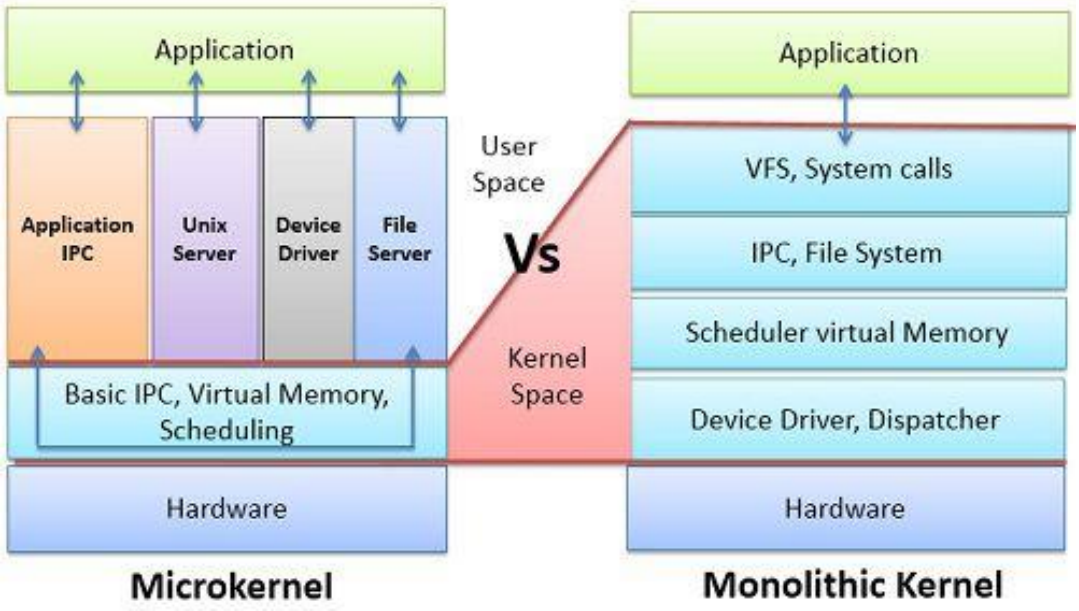

Figure 4 – Microkernel vs Monolithic Kernel[4]

3.1.2 OP-TEE

OP-TEE stands for Open Portable Trusted Execution Environment and it has been developed by ST-Ericsson. In 2013, Linaro[5] started working with ST-Ericsson, and in 2014 they made it an open source project. OP-TEE is being constantly updated and maintained by a group of contributors on GitHub[6]. OP-TEE uses TrustZone technology to implement the TEE standard. In the normal world and in the secure world it uses the Linux kernel, however, in the secure world it uses a modified Linux kernel. OP-TEE provides isolation among the TAs and from the normal world. It aims to be portable and to have a small footprint.

In OP-TEE, TAs cannot see the normal world kernel, that can only be done by the Pseudo Trusted Applications. PTAs are kernel modules in the secure kernel and they have the privilege to access the normal world kernel, shown in Figure 5.

---

[4] https://techdifferences.com/difference-between-microkernel-and-monolithic-kernel.html last visited on 2019.12.03

[5] Linaro is also the maintainer of GCC and the Linux kernel

[6] https://github.com/OP-TEE/ last visited on 2019.12.12

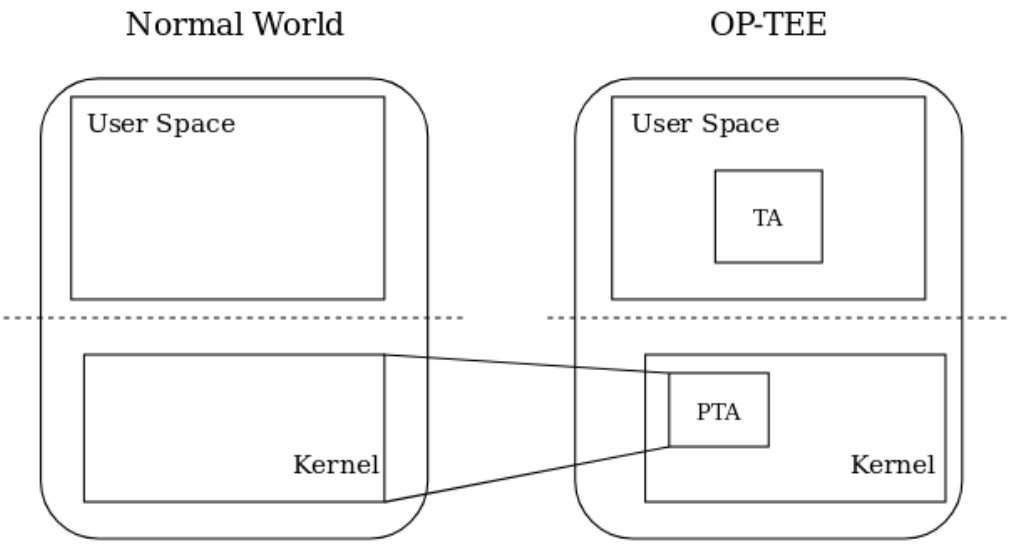

Figure 5 – TA and PTA in OP-TEE architecture

3.1.3 GCC

I used GCC to extract data that will be vital to generate the Integrity Monitor. GCC's can be extended with a plugin which can help to get the necessary data out at the compile time. The GNU Compiler Collection, also known as GCC [7] is a compiler system created by the GNU Project. The GCC was originally founded by Richard Stallman and released in 1987. GCC is a default compiler to Linux and the Linux kernel. Also, it has many front ends for languages, like C, C++, Java, Fortran, Go, Ada etc. GCC has been ported to several architectures, ARM based, Intel based and AMCC based chips. GCC is distributed under the GNU General Public license. GCC is being developed continuously by a community. GCC has numerous benefits for me, including that it is free to use, there are plugins available and I found informative documentations online. Since GCC has been made, it has been using internal representations seen in Figure 6.

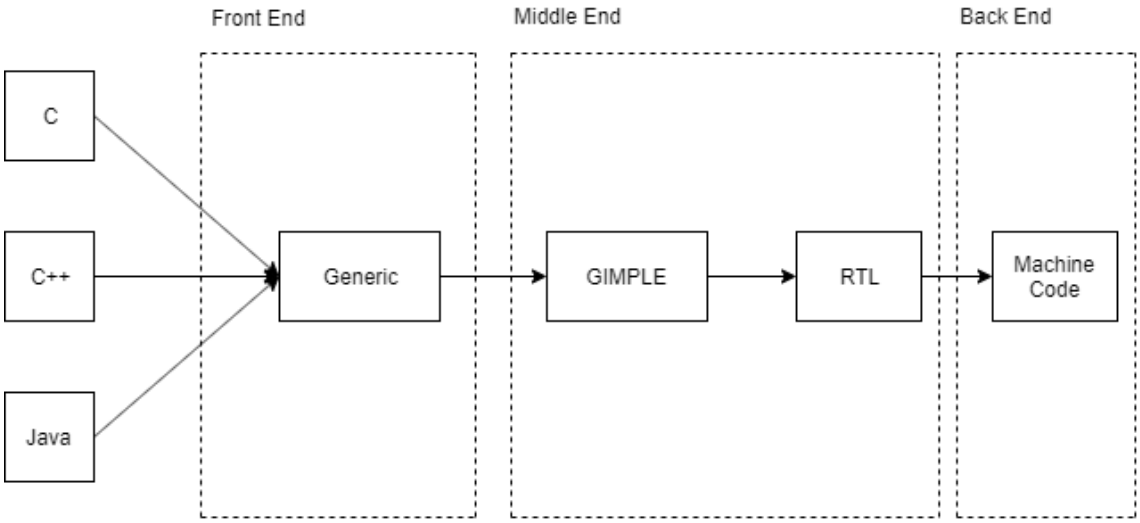

Figure 6 - GCC Compilation Stages

In Figure 6, there is a representation on how GCC creates machine code from source code. First, it parses the code and creates GENERIC. It used to be that each language had their own tree code, but with the introduction of GENERIC and GIMPLE that changed. C, C++ and Java all create GENERIC. GENERIC is a tree format and only the important part of the code is saved here, so, spaces and semicolons are omitted. After that, GCC lowers the code to GIMPLE. GIMPLE is a three-address code, and it uses temporary variables to help the optimization. Before the machine code comes RTL, called the Register Transfer Language. RTL is an assembly like language with infinite registers. After that the machine code is produced for the target architecture.

3.1.4 GCC Plugin

GCC introduced a plugin in version 4.5.0. It is useful if one want to modify or analyze information while GCC is compiling and do not want to dig into GCC source code too much. Therefore, plugins provide an easy way to extend GCC and add extra features. It allows one to hook up ones code at different part of compiling stages. Mainly it is useful when GCC is at GENERIC and GIMPLE stages as these are the most documented. At stage RTL, the documentation [7] advises one not to use it.

3.2 System.map

System.map is a table which contains symbols of the Linux kernel. Symbols can be global variables or functions that are exported. The System.map table contains these exported symbols and their address in the kernel. The table can be used to debug kernel

oopses, but in our case it can help find function pointers. System.map file can change between rebuilds, therefore, the safest is to get it when a system is running at /proc/kallsyms which is generated when the kernel boots up. An example for a System.map content:

```
ffffff800811a5b0 t posix_cpu_timer_del
ffffff800811a6a8 t posix_cpu_timer_create
ffffff800811a7b0 t process_cpu_timer_create
ffffff800811a7e8 t thread_cpu_timer_create
```

The first column is the address inside the kernel where the symbol is located, the second is the symbol type[7] and third is the symbol name.

3.3 Program design

A basic design can be seen on Figure 8. It represents how to create the monitor algorithm which is created in accordance with the paper [1]. Some modifications were necessary, because the article was written in 2007 and technology and some programs have changed since then. Their idea used a different monitoring technique seen on Figure 7.

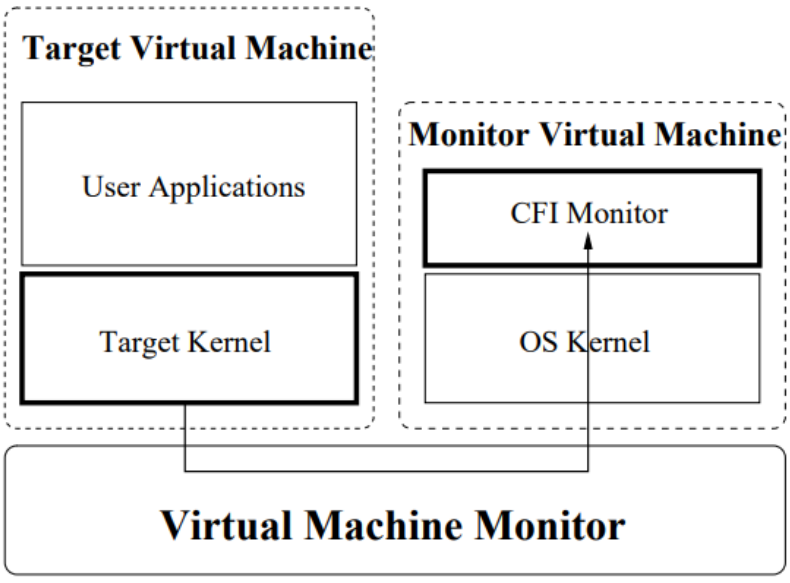

Figure 7 – Monitoring solution used in SBCFI Monitor

[7] Here the t stands for the text section type

In my implementation, I do not use the virtual monitor, I use the OP-TEE's Secure World for separation of the normal world kernel. The Integrity Monitor will be placed in this secure world and use a Pseudo Trusted Application to take the snapshot of the Normal World kernel. Then, it will use the Trusted Application to verify the integrity of the kernel. It will traverse the kernel starting from a number of global addresses and check if a function pointer has a correct value. If not, it can mean that the kernel might have been compromised. There are function pointers whose values should not change [8], so we can monitor those.

To create the input files for the monitor generator, we need the following components. First, from the kernel source code, we need to extract the structure and their members. In these structures, we need to find which members point to a structure or to a function therefore, they are function pointers. This was done in the paper with CIL [9]. CIL is a C intermediate language compiler. CIL is used for easy analysis and source-to-source transformation of C programs. While I was looking for description of the software online and trying to find some documentation for it, I realized it would be hard to understand and modify, as it was written in OCaml. Objective Caml is a Caml language which is not easy to understand and used mostly in academic projects. Also, it had been updated years ago, and it does not look like it is being used by many people, so asking for help would be a bit difficult. Instead of CIL, I decided to use GCC and a plugin written for GCC to get the data I needed. As mentioned before, GCC is updated, maintained regularly and I found a few tutorials on the internals of GCC. This way I was able to ask for help whenever I got stuck.

With the GCC Plugin I was also able to extract the function names that are used in the kernel. In the System.map file there are symbols with addresses which come useful as a starting point of the monitoring algorithm. The problem was that there was no clear way to tell which symbol is a variable or a function. Because I extracted the function names I could tell that the symbols found in the System.map and not found in the function names list are probably variables. I also needed the global variables and their types to check which one is in the System.map file, and if they are then what type they are. I only considered structures or function pointers because other types are not useful at this time. Then, after in Match, I found which are the variables from System.map, and also had a structure or function pointer type.

In the Type Mapping I checked which structures contain function pointers or reachable function pointers. After I found that, I also found a way to reach these function pointers. With these information and the Registers which I explain more in 4.5, the Monitor Generator can be created. The design of the implementation can be seen on Figure 8.

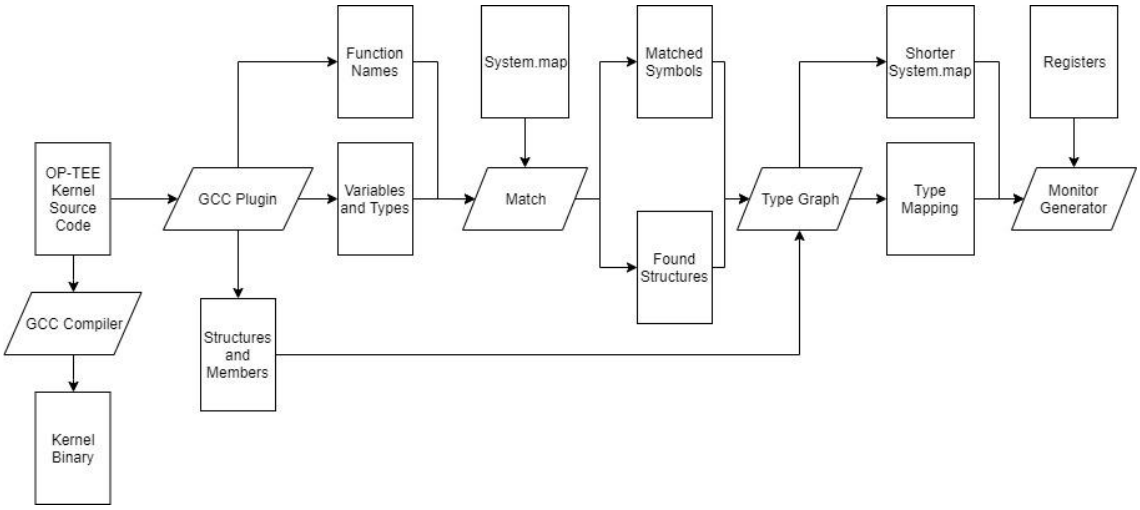

Figure 8 – Design of the program

4 Implementation

Here, I describe the implementation of parts from Figure 8 in detail. I was working on my laptop which had an Ubuntu 18.04 OS. For code editing I used Sublime Text, which is a simpler editor, not an IDE. I used OP-TEE version 3.4.0 and it contained Linux kernel version 4.14.56. For GCC I was using version 7.4.0.

4.1 OP-TEE Kernel Source Code

OP-TEE, as mentioned earlier, is an open source project that is being developed on GitHub.[8] Because the end result is intended to run on OP-TEE, and monitor the normal world kernel, OP-TEE's normal world kernel is necessary. When compiling this kernel the GCC Plugin is included. OP-TEE uses the Linux kernel in the normal world.

4.2 GCC Plugin

There are a few articles online which helps one to get started with writing a GCC Plugin [10], but this documentation has more information on it [7]. To get started, one need to write a function called `plugin_init`. `Plugin_init` will take two parameters, the first one is:

```
struct plugin_name_args *plugin_info
```

This structure contains basic information such as:

- The name of the plugin
- Relative file path to the plugin
- Arguments given to the plugin
- GCC version number
- Help[9]

[8] https://github.com/OP-TEE last visited on 2019.12.02

[9] If these last two are defined inside ones plugin

The second parameter is:

```
struct plugin_gcc_version *version
```

This parameter contains a detailed info about versions: base version, date stamp, development phase, configuration arguments and revision so one can check if ones plugin is compatible with the compiler which uses it. This will be the first function that runs before GCC starts compiling the code. Here, one should assert an integer type variable called: `plugin_is_GPL_compatible`. This is because GCC requires that one's plugin is licensed under General Public License and GCC is licensed under GPL as well.

GCC Plugin functions can be run at specific events and these are defined in gcc-plugin.h. To run the desired functions one need to register these functions to callbacks in the `plugin_init` function. The callbacks that I used in my code are the following:

- `PLUGIN_FINISH_TYPE`
- `PLUGIN_FINISH_DECL`
- `PLUGIN_START_PARSE_FUNCTION`
- `PLUGIN_FINISH`

For registering a callback I used the following instruction:

```
register_callback (plugin_info->base_name, PLUGIN_FINISH_TYPE,
                   plugin_finish_type, NULL);
```

The first parameter of the function is the plugin's name, the second is the event code which is actually an integer value, but one can use the callback's name. The third is ones function name which will run when the callback is triggered and the last one is just a pointer to a plugin specific data that I left as NULL. Before I describe more about what I do at each of these callbacks I introduce the internal structure that GCC uses in the front-end, GENERIC or sometimes referred to as Trees.

4.2.1 Trees

GCC uses GENERIC in the front end, which is a tree representation. Entire functions are represented as trees and GCC defines GENERIC that "if you can express it with the codes in gcc/tree.def, it's GENERIC"[10].

The base for it is the tree pointer type. Accessing the trees are done via macros. Macros usually check that the actual node type is compatible with the request and, if so, they return the requested data. There are a several macros that are predicates which means they check for some conditions and return true or false. Such Macros usually end with `_P`, but according to the documentation [7], this is not always the case. Macros are also all uppercase characters. One should only check if a Macro returns a zero or not, because they do not specify any other return code. Each tree node in the GENERIC has a tree code of type integer which identifies what kind of node it is. To get the code of the tree one can use the `TREE_CODE` macro. Nodes sometimes can be the same types. Usually, different kinds of types have different tree codes, but GCC documentation [7] says that is not always the case.

4.2.1.1 Debug tree

The debug tree function helped me a lot during development, it allowed me see how a piece of code is represented inside GCC. The function can take any tree type as input parameter and it writes out to the standard error output. Using `debug_tree` on the following code:

```
struct my_struct
{
    int my_int;
    char* my_character;
    float my_float;
    void* random_pointer;
    int* int_pointer;
    int* max_segment;
};
```

gives me this output:

```
context     <record_type     0x7fce5b0719d8>     attributes     <tree_list
0x7fce5b070c80>>>
    chain <type_decl 0x7fce5b056b48 D.2762>>
 <record_type 0x7fce5b074000 my_struct type_0 BLK
```

[10] https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html last visited on 2019.12.03

```
    size <integer_cst 0x7fce5affd210 type <integer_type 0x7fce5aeb10a8
bitsizetype> constant 384>
    unit size <integer_cst 0x7fce5affd0c0 type <integer_type
0x7fce5aeb1000 sizetype> constant 48>
    align 64 symtab 0 alias set -1 canonical type 0x7fce5b074000
    fields <field_decl 0x7fce5b056e40 my_int
        type <integer_type 0x7fce5aeb15e8 int public SI
            size <integer_cst 0x7fce5ae99f18 constant 32>
            unit size <integer_cst 0x7fce5ae99f30 constant 4>
```

Here we can see that on the 4th line there is my structure name, `my_struct` which is a structure, therefore, it has a `RECORD_TYPE`. Late in the line that begins with fields, one can see that my first field is called `my_int` and it has an `integer_type` and a size of 32 bits.

4.2.2 Finish Type

The first function that I used is the `plugin_finish_type` function which uses the callback `PLUGIN_FINISH_TYPE`. In this function, my goal is to find all of the structures that are inside the Linux kernel and the members of these structure. For the monitoring algorithm, it is also important to know which offset these structure members have inside the structure. Usually they are continuously put into the memory, but to optimize computations with these members, it is much easier and faster if GCC puts them a location that is 32-bit or 64-bit aligned based on the architecture of the CPU. This is achieved by adding paddings. For example here, in this structure:

```
struct my_struct {
    char a;
    int x;
}
```

The padding will be 3 bytes after char a, because a character is 1 byte as shown in Figure 9.

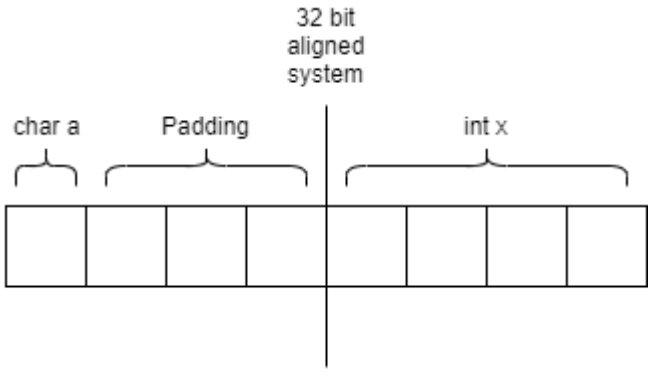

Figure 9 – Padding inside a structure

Some structures contain members that are structures as well and it is easier to get the offset of the next member at compile time than to calculate it manually. Therefore, the goal in this function is to get the structure's name, members, members' type, and their offsets inside the structure.

Functions that are registered to the callback take two parameters:

```
void *event_data
void *user_data
```

The `event_data` parameter is where one get the information that GCC is providing at that specific pass, in tree format and `user_data` is just ones pointer to plugin specific data that one could give at the `register_callback` function.

To convert the `event_data` to a tree structure one just need to use the following code:

```
tree type = (tree) event_data;
```

Since I was only interested in the structures that GCC encounters during at the compile time, I had to search for in tree.def what GCC uses to represent structures and look into the output of `debug_tree`. Here I found that it uses the `RECORD_TYPE`. Therefore, I needed to check if the tree that I'm working with is a `RECORD_TYPE`, so, I used this code:

```
if (TREE_CODE (type) == RECORD_TYPE) {…}
```

Where `TREE_CODE` is a simple macro and it actually gives back an integer which later helped later in the debugging process when I looked at other tree variables. Finding out the name of the structure is done in two steps. First, one have to use the `DECl_NAME` macro on the tree which gives one back an `IDENTIFIER_NODE`. An `IDENTIFIER_NODE` is similar to the C or C++ identifier, but here it also helps if one are using overloaded operators. Nevertheless, the important thing is to get the name of the structure to use the `IDENTIFIER_POINTER` macro on the `IDENTIFIER_NODE` which returns a `NULL` terminated `char*`. See Figure 10 for a visual representation.

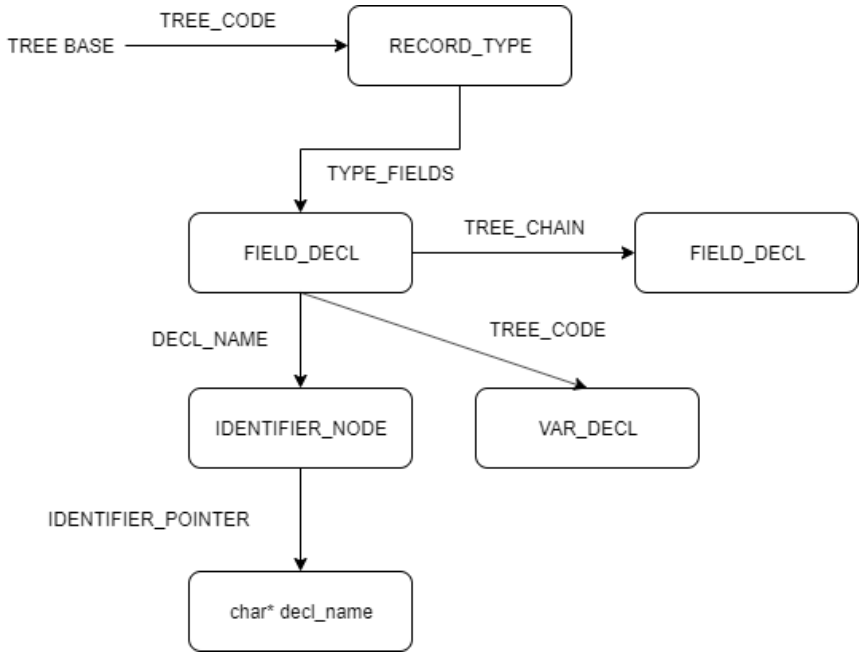

Figure 10 – Structure in a tree format

Getting the first field of the structure can be done with the following macro:

```
tree field = TYPE_FIELDS(type);
```

Tree fields are linked into a linked list, they are chained together with a pointer, so after the first one one can use `TREE_CHAIN` to get the next one, and at the end of the list it returns `NULL`, see Figure 10. After getting the field I checked with the `TREE_CODE` macro that it is, in fact a field, just to make sure. First, while testing, I used the `FIELD_DECL` code for checking, but while compiling the kernel and including my plugin I found that it was false and in the kernel it used a different type. I looked deeper in the manual and found that GCC uses 4 different tree codes for the declaration inside a structure:

- `FIELD_DECL`
- `VAR_DECL`
- `CONST_DECL`
- `TYPE_DECL`

Because `TREE_CODE` returns an integer, and the definition order of the `TREE_CODE` in the tree.def file actually matched the integer they use, I was able to conclude that at kernel compiling time, GCC uses the `VAR_DECL` tree code.

In tree.h, GCC defines what kind of different type it uses to represent internally the variables. It can be `float16_type_node`, `float32_type_node`, etc. depending on how many bits are used to represent a float. But there are also `integer_zero_node`, `integer_one_node`, `integer_three_node`[11]. I found around 159 different types. To get what type is a variable I used the `TREE_TYPE` macro. It returns an integer, but in this case I could not find an easy way to see which integer maps to which node, so a better idea was to just use an if for each of them:

```
if(type == ptr_type_node){
    return "ptr_type_node";
  }
```

As one can see there are nodes that contain pointers, in fact, there are 12 of them. I thought GCC would use one like this for the function pointers, but it turned out that that is not right. For the function pointers I could not find the tree type, but I found a macro, `FUNCTION_POINTER_TYPE_P` which equals to nonzero, when I used the `FIELD_DECL`'s type, like so:

```
if(FUNCTION_POINTER_TYPE_P(TREE_TYPE(field))){
          //this is a function pointer.
      }
```

This function pointer predicate can be found in the source code in tree.h but it is not documented in the manual. It works that if the given type's tree code is `POINTER_TYPE` or `REFERENCE_TYPE` and the `TREE_CODE` of that type's `TREE_TYPE` is a `FUNCTION_TYPE`. See Figure 11 for visualization.

[11] But I did not find `integer_two_node`

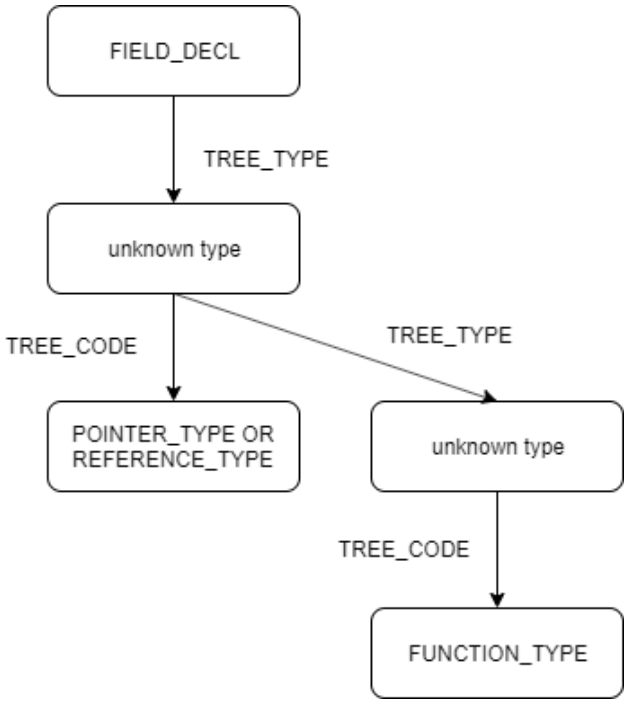

Figure 11 – Determining if a variable is a function pointer

So, I have found the structure's name, its member types, and now I need to find the offset. Finding the member's name is analogous as finding the structure's name. I looked into the source code and saw that I can use `byte_position` to get the offset which gave back an `INTEGER_CST`, which is an integer constant expression. To get the number, I just used the `TREE_INT_CST_LOW` macro:

```
int offset = TREE_INT_CST_LOW(byte_position(field));
```

After finding a structure, I saved it and its members, their types and offsets into a txt. An example of a structure that I found is the:

```
struct class_attribute {
     struct attribute attr;
     ssize_t (*show)(struct class *class, struct class_attribute *attr,
               char *buf);
     ssize_t (*store)(struct class *class, struct class_attribute *attr,
               const char *buf, size_t count);
};
```

Which in my output file looked like this:

```
class_attribute 3
attr attribute 0
show function_ptr_type 16
store function_ptr_type 24
```

Where `class_attribute` is the structure name, and 3 represents how many members it has. After that, comes the first member which is `attr` with an attribute type which is, in fact a structure and it has an offset of 0. The following two members are show and store, and they both have a function pointer type which is correct, as it can be seen on the code. Here is Figure 12 for high level view for the callback

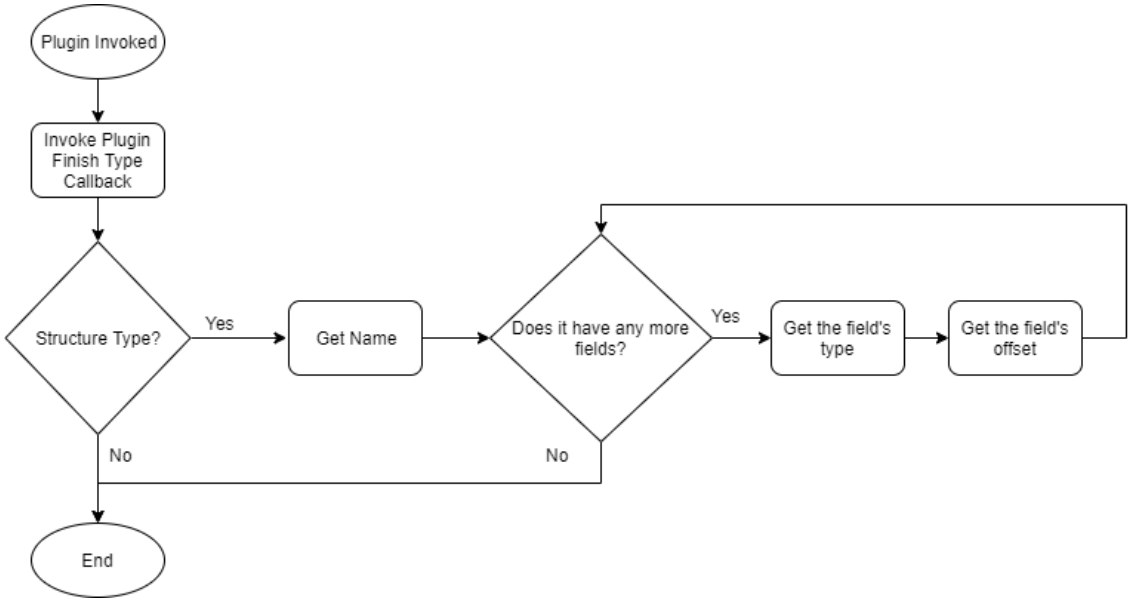

Figure 12 – Finish Type high level overview

4.2.3 Finish Declaration

The second function that I used is the `plugin_finish_decl` function which uses the `PLUGIN_FINISH_DECL` callback. The idea here is to extract all the variables which have a structure type or they are a function pointers themselves. These will be useful in the Match, where I will see if there is any variable which is exported that matches to the symbols of System.map.

First, I check if the tree I am processing is not a structure, because I only care about such variables. Then I check if I can find the variable's type. If I cannot find the type, then I can check if the variable's type tree code is `RECORD_TYPE`, which means it's a structure. After that, I can use the `TYPE_NAME` macro to get the structure's name. Checking if the variable is a function pointer is the same as in chapter 4.2.2 and shown in Figure 11. Getting the name of the variable is also analogous as in chapter 4.2.2 or in Figure 10. An example output is the following:

```
function_ptr timer_try_to_cancel
function_ptr timer_arm
```

```
k_clock clock_posix_cpu
k_clock clock_posix_dynamic
```

Where the first word is either a structure name, or `function_ptr`, which means that the second word, the variable name either represents a structure or a function pointer. Other types are omitted.

4.2.4 Start Parsing a Function

The third function I used is the `plugin_start_parse_function` which uses the `PLUGIN_START_PARSE_FUNCTION` callback. Here I just want to get the function names that appear in the kernel code. In the System.map file, described in chapter 3.2, there are symbols, but it can be either a function or a variable. I did not find any way online to tell which one is which and the symbol's type also does not specify it. So, I only extracted all the function names and if I found a symbol name, which matched the function name, I disregarded that symbol.

Getting the function name is not difficult, but it can be a bit tricky, since the tree code of the function can be:

- `FUNCTION_TYPE`
- `METHOD_TYPE`
- `FUNCTION_DECL`
- `LABEL_DECL`

After that, I just used `DECL_NAME` to get the `IDENTIFIER_NODE` then `IDENTIFIER_POINTER` to get the characters, in the same way as described in chapter 4.2.2 and show in Figure 10.

4.2.5 Plugin Finish

I used the `PLUGIN_FINISH` callback to create a `plugin_finish` function where I saved my gathered data to the text files, and I used it also to close the file handlers while developing to get some statistics.

4.2.6 Using the plugin

GCC can only include the object file while compiling, not the source code, so one need to compile ones plugin. If one have a plugin written called myplugin.cc, here is the code for it:

```
$ g++ -g -I`gcc -print-file-name=plugin`/include -fpic -shared -o
myplugin.so myplugin.cc
```

The `-g` flag will include debugging information of ones plugin. If the compilation of a program crashes when there was an active plugin, GCC advises against to report that issue as most likely the problem was in ones code. Also, it sometimes tells what was the issue and even the line number where ones plugin crashed which can be very helpful. Other times it only says that there was a segmentation fault which is not that useful while debugging. On the above code `-print-file-name` is used to print the library file, `plugin`/include`. `-fpic` and `-shared` are used to create a shared object with position independent code. `-o` names the output file in this case it will be `myplugin.so`.

Including the plugin can be done with flag `-fplugin=./myplugin.so` if the object file is in the same directory where we are doing the compilation. We can also specify arguments to the plugin which can be done with

```
-fplugin-arg-myplugin-log = log.txt
```

One can get this log.txt argument with the following code:

```
if (strcmp (plugin_info->argv[i].key, "log") == 0) {
      logfile = plugin_info->argv[i].value;
    }
```

which was mentioned in [11].

While I was compiling the kernel source code for OP-TEE, I was using the given makefile. Makefiles are basically sets of instructions that tell the compiler and the linker what to do and what dependencies each instruction needs. So, if a large program changes only the needed part will recompile. One can, of course, add options for the compiler in the makefile and specify which compiler to use. There was already a `KBUILD_CFLAGS_KERNEL` flag in there so I just filled it out:

```
KBUILD_CFLAGS_KERNEL := -fplugin=./myplugin.so
```

This way whenever I compiled the OP-TEE kernel my plugin got included only at the kernel compiling time.

4.3 Match

After I found all the function names, and also the variables which type is a structure or a function pointer, I can match those to the System.map file. First, I read in the variable's names and I only consider the ones which are unique, otherwise it would be hard to decide which one of them is exported or not. Then I read in the function names. While reading in the symbol's name from the System.map file, I can first check if it is a function and if it is, then I discard it. If it is not a function then I can search in the variable name list if it matches one of them. If it does I am still not done, because I have to search the existing found symbols whether it matches with them, as System.map has duplicate entries, like zero which appears 10 times. From those zeroes it is challenging to decide which one is which variable (or function) so I only use unique values. Finally, two files are created, one that contains these matched symbols where the symbol is possibly a function pointer or a structure, and the second is just all of the structures that can be found in this matched file. Using the match is simple it is just a simple c program that can be compiled with gcc:

```
$ gcc match.c
```

and can be run with:

```
$ ./a.out
```

The program just needs to be in the same directory where the Function Names, System.map and Variables and Types files are. A high level overview of the workflow is shown in Figure 13.

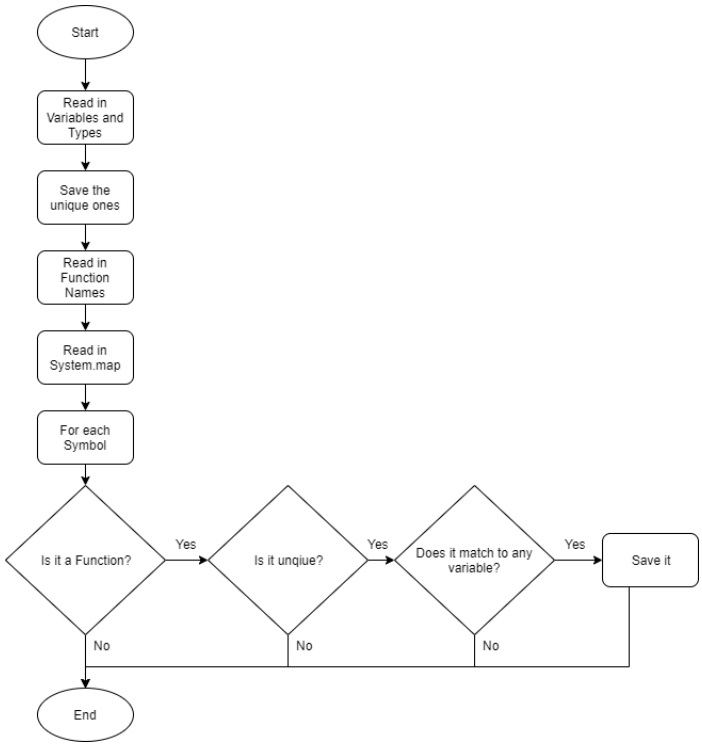

Figure 13 – Flowchart of the Match program

4.4 Type Graph

Now we know the addresses where we can start searching for function pointers. Some of these addresses contain structures that already contain function pointers. There are also some structures which contain other structures which contain function pointers, and the goal is to find these structures and the way to get to them and pass this information to the Monitor Generator.

First, it reads in the Structures and Members then the Found Structures. At each structure it checks if it contains a function pointer: if it does, then it will save how to get to that structure and how many function pointers there are and what offset they have inside the structure. Then it checks if it contains another structure, then it will move to that structure and will check if that one contains a function pointer, and after that, if it contains a structure. A flowchart for the program can be seen on Figure 14.

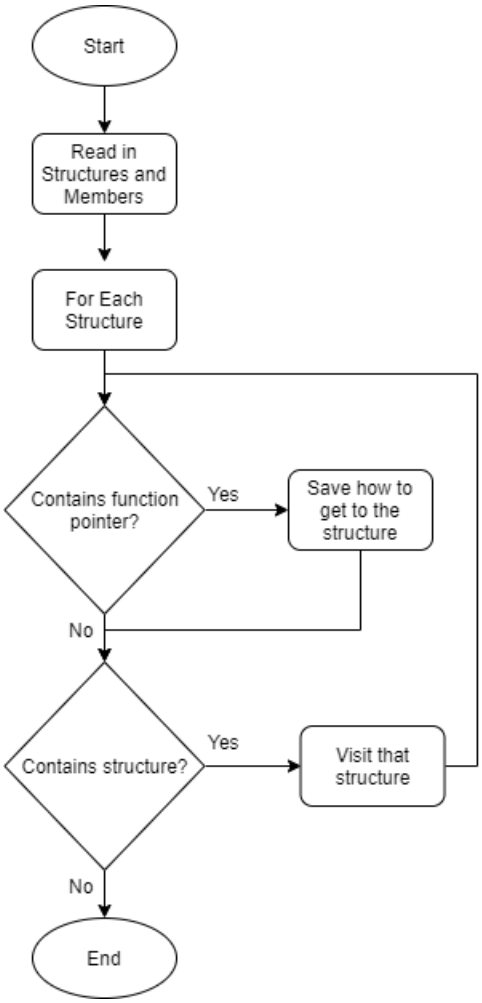

Figure 14 – Flowchart of the Type Graph program

Let's look at an example, here is a structure inside the kernel:

```
struct platform_driver {
     int (*probe)(struct platform_device *);
     int (*remove)(struct platform_device *);
     void (*shutdown)(struct platform_device *);
     int (*suspend)(struct platform_device *, pm_message_t state);
     int (*resume)(struct platform_device *);
     struct device_driver driver;
     const struct platform_device_id *id_table;
     bool prevent_deferred_probe;
};
```

And here is the device_driver structure (6th member of the platform_driver):

```
struct device_driver {
     const char     *name;
     struct bus_type      *bus;
     struct module        *owner;
     const char     *mod_name; /* used for built-in modules */
     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
     enum probe_type probe_type;
```

```
        const struct of_device_id *of_match_table;
        const struct acpi_device_id    *acpi_match_table;
        int (*probe) (struct device *dev);
        int (*remove) (struct device *dev);
        void (*shutdown) (struct device *dev);
        int (*suspend) (struct device *dev, pm_message_t state);
        int (*resume) (struct device *dev);
        const struct attribute_group **groups;
        const struct dev_pm_ops *pm;
        struct driver_private *p;
};
```

As it can see both of these structures contain 5 function pointers and the second structure is reachable from the first structure. For this, my program output is the following:

```
platform_driver 5 1
probe 0
remove 8
shutdown 16
suspend 24
resume 32
device_driver 40 5 0
probe 56
remove 64
shutdown 72
suspend 80
resume 88
```

Here, the first line tells one the structure name, the number of function pointer this structure contains, and the number of structures it contains that contain function pointers. The following 5 lines are the name of the function pointers and their offsets in the structure. The `device_driver` line is the next structure that is in 40 bits offset in the platform driver structure, and contains 5 function pointes and does not contain any structure that contains function pointers.

Based on the structures that contain function pointers or function pointers can be reached from them, the program also outputs a shorted System.map where all the symbols which are variables have a structure type that are like these structures. An example of the output is:

```
rtctimer rtc_timer 60686
bctimer hrtimer 60707
saved_ftrace_func function_ptr 60871
```

Here, `rtctimer` is the symbol name, `rtc_timer` is the structure and 60686 is the line number where this symbol can be found in the System.map file. The `saved_ftrace_func` is a `function_ptr` which can be found in line 60871.

4.5 Registers

On Figure 8, there is a Registers input to the Monitor Generator. Registers can be useful because they can also be a global starting point to find function pointers. Many registers hold different types of data, and determining that type can be difficult. Because of that, I only considered registers that have a fixed type. I used the ARM manual to determine the register [5]. These are the following:

- Stack pointer - `sp`
- Frame Pointer, which is a copy of SP before function stack allocation - `fp`
- Link Register it holds the return address, but return addresses not cause persistent modification so this might not be needed - `lr`
- Translation Table Base Register 0: This register holds the base address of the translation table - `TTBR0_EL1`

Getting the register value can be done in the Pseudo Trusted Application. To interact with a Pseudo Trusted Application, first we need to implement a Trusted Application.

4.5.1 Implementing a Trusted Application

There are several ways to implement a Trusted Application, one of the easiest is to examine the example TAs and implement a similar one [12].

- First, there needs to be a main program that will call our TA for testing purposes. TAs usually only start if they are invoked by a normal world command.
- Invoking the TA can be done with the TA's `UUID` which stands for Universally Unique Identifier. Each TA should define a `UUID` value.
- TAs also need to implement entry points, these are:
    - `TA_CreateEntryPoint`

    - TA_DestroyEntryPoint
    - TA_OpenSessionEntryPoint
    - TA_CloseSessionEntryPoint
    - TA_InvokeCommandEntryPoint

- For implementing a simple TA, we can write our code in the TA_InvokeCommandEntryPoint function which returns a TEE_RESULT which upon succession is TEE_SUCCESS.
- To build the TA into the OP-TEE, we also need to implement Makefiles that sets configuration values, and also a sub.mk that lists the sources to build.
- user_ta_header_defines.h is necessary where TA properties are defined.

Pseudo Trusted Applications are kernel modules that are implemented directly into the OP-TEE core. For some example PTAs one can look into the core/pta and use the core/pta/stats.c as a template. PTAs are also statically built into OP-TEE. They run at the highest privilege level as they run in kernel mode. To get the register values, I implemented an example PTA to see if I can access it. PTAs are implemented in C and one way to access the register is to use inline assembly like this:

```
asm("mov x0, fp");
register unsigned long long reg1 asm("x0");
```

This code moves the fp register value in to the x0 register, and with the second instruction I move the x0 value to a reg1 called register type. To access the TTBR0_EL1 register, one need to change the code a bit:

```
asm("mrs x0, TTBR0_EL1 ");
register unsigned long long reg1 asm("x0");
```

5 Evaluation

During development, I always tested the plugin and experimented with it. Because of the incomplete documentation and a lack of online resources, it was trial and error a lot of time. I tested the plugin on simple files that I wrote myself to see if it works and where it fails.

5.1 Testing with the OP-TEE kernel

To get the structures and variables from the kernel I needed to compile the OP-TEE kernel. This was achieved with the makefile, described in more detail in 4.2.6. Compiling the OP-TEE kernel takes some time, and also I did not find any way to force the kernel to recompile, so I had to compile the kernel without the plugin include to be able to compile it after with the plugin. Also, if something failed during compiling one of the files, it was hard to debug as there was no debugger, and I could not compile the file by itself because it had dependencies to other files.

5.2 Performance

I was developing on a laptop and it had an Intel Core I3-5005U 2GHz 2 Core processor, 8 GB of RAM and an SSD. On that, it took 40 minutes to compile the kernel with the plugin included and about 25 minutes without the plugin. If I was writing a TA or PTA and to recompile the OP-TEE, it took about 2 minutes. The Match runtime is also a few minutes as it does a lot of string comparison. The Type Graph runs under a minute.

It collected around 1700 function pointers from 210 structures. There are about 2000 symbols whose type is one of these structures and also 200 symbols whose type is a function pointer. So, I estimate that with the data I provide there could be 16500 reachable function pointers which I suppose can be a good starting point to see if a rootkit modifies one of them.

5.3 Limitations

The program has a few limitations. One of them is that it cannot identify pointers, which point to structures, or double pointers. An example of this would be the same as described in 4.4. Here, in the device_driver structure:

```
struct device_driver {
     const char     *name;
     struct bus_type      *bus;
     struct module        *owner;
     const char     *mod_name; /* used for built-in modules */
     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
     enum probe_type probe_type;

     const struct of_device_id *of_match_table;
     const struct acpi_device_id    *acpi_match_table;
     int (*probe) (struct device *dev);
     int (*remove) (struct device *dev);
     void (*shutdown) (struct device *dev);
     int (*suspend) (struct device *dev, pm_message_t state);
     int (*resume) (struct device *dev);
     const struct attribute_group **groups;
     const struct dev_pm_ops *pm;
     struct driver_private *p;
};
```

Both the groups and the pm structure contain function pointers, but I could not get to them because I was unable to extract the data. In the program I can verify when I see a pointer, but I was unable to go past that.

```
<field_decl 0x7f7f3435a720 groups
    type <pointer_type 0x7f7f3426c930
        type <pointer_type 0x7f7f3426c738 type <record_type 0x7f7f3426c690
attribute_group> …
```

Here we can see the double pointer to the groups variable that I exctacted with the debug_tree function that I used in the plugin source code.. It says inside that it is an attribute_group type, but I was unable to get that data out. I was trying to use the TREE_TYPE macro on the first pointer I got a VOID_TYPE. After I looked more into the documentation [7] and there I found an other macro called TYPE_PTRMEM_POINTED_TO_TYPE, but this still wasn't helpful. So, this still needs some working, so the program can collect more function pointers and do a better analysis.

Another limitation is that there might not be enough global addresses to find sufficient number of function pointers. But getting more global addresses, not just the System.map, became too complicated for me. I found that in the RTL stage GCC gives

the symbols an address, but it is not a visible type, and also working with RTL is not easy.

6 Conclusion

In my thesis, my goal was to find a method for rootkit detection, understand it and create the input files for it. The method I chose was based on paper [1]. I chose this paper as it described a way to monitor function pointers and it could be implemented on different platforms. The paper also explained how they themselves implemented it and that was a good starting point. I chose OP-TEE, because it already contains a secure world, which I can use to monitor the normal world's kernel.

While implementing the ideas found in the kernel, I found that the software they used is much more difficult to use then I hoped for, therefore, I turned to GCC and used their plugin features. Creating a plugin is also not an easy task, but with help from the manual [7] and online sources I was able to develop a plugin that provided sufficient outputs. Writing other programs to create the input files for the monitor generator was not that difficult, but I still had to work on them a bit.

With these provided input files, even with the limitations they have, I firmly believe an Integrity Monitor can be created and probably detect rootkit modifications. In the future, there are other techniques that could be implemented, for instance rootkit detection in the user mode, or validating the kernel text.

Bibliography

[1] Nick L. Petroni, Jr. and Michael Hicks. 2007. Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). ACM, New York, NY, USA, 103-115. DOI: https://doi.org/10.1145/1315245.1315260

[2] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. 2004. Copilot - a coprocessor-based kernel runtime integrity monitor. In Proceedings of the 13th conference on USENIX Security Symposium - Volume 13 (SSYM'04), Vol. 13. USENIX Association, Berkeley, CA, USA, 13-13.

[3] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2011. Detecting Kernel-Level Rootkits Using Data Structure Invariants. IEEE Trans. Dependable Secur. Comput. 8, 5 (September 2011), 670-684. DOI: https://doi.org/10.1109/TDSC.2010.38

[4] GlobalPlatform Technology TEE System Architecture Version 1.2

[5] "ARM Architecture Reference Manual." ARM Developer, ARM Limited, 2019, developer.arm.com.

[6] Robert Love. 2010. Linux Kernel Development (3rd ed.). Addison-Wesley Professional.

[7] Stallman, Richard M. "GNU Compiler Collection Internals." GCC, the GNU Compiler Collection, GCC Developer Community, 2019, gcc.gnu.org.

[8] Brown, Neil. "Object-Oriented Design Patterns in the Kernel, Part 1." *[LWN.net]*, 2011, lwn.net/Articles/444910.

[9] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proceedings of the International Conference on Compiler Construction (CC), Apr. 2002.

[10] Davis, Matt. "An Introduction to Creating GCC Plugins." *LWN.net*, 27 Sept. 2011, doi:https://lwn.net/Articles/457543/.

[11] Jones, Richard WM. "Playing with GCC Plugins." *Virtualization Tools and Tips*, 24 Feb. 2016, rwmj.wordpress.com/2016/02/24/playing-with-gcc-plugins/.

[12] "OP-TEE Documentation." *OP-TEE Documentation*, 25 Nov. 2019, readthedocs.org/projects/optee/downloads/pdf/latest/.

Acknowledgment

I would like to thank my thesis mentors, Dr. Levente Buttyán and Dorottya Futóné Papp for giving me support and help throughout this project.

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004) [12]

[12] Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.