eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

File integrity monitoring on Linux systems

Author

Aritz Herrero Perez de Albeniz

2021

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

File integrity monitoring on Linux systems

Author

Aritz Herrero Perez de Albeniz

Director(s)

Jose A. Pascual Saiz

Summary

In this new era of technology and interconnected systems, the number of devices connected to the Internet has grown significantly, with more and more businesses and organizations offering services in the *cloud* and more users having all kinds of “smart“ connected IoT devices. With this, the number of attacks has also grown with the same pace if not faster, requiring new systems to ensure data integrity, confidentiality and privacy.

The objective of the project is to ensure that the integrity of these devices is maintained by designing and developing a system capable of monitoring, detecting and notifying changes of files stored in the target filesystem. To accomplish that goal, the system will maintain a hash digest representation of a subset of the filesystem using cryptographic techniques and mechanisms to detect changes in real time powered by the Linux kernel and the operating system.

All this leads to a robust intrusion detection system, capable of ensuring the integrity of a subset of the filesystem containing information considered critical in real time, offering a solution that is both fast and efficient, in terms of memory usage. The solution is designed to work on any kind of devices running Linux, which comprehends high end x86 based devices as well as low power ARM based devices.

Contents

List of Figures

List of Tables

1. CHAPTER

Introduction

In the last few years, technological advances have democratized the access to computers and the internet. Moreover, the expansion of IoT devices due to the popularization of smart lights, home assistants and other "smart" devices has multiplied the number of devices connected. This new paradigm with more than 22 billion connected devices at the end of 2018 and an estimation of 38,6 billions by the end of 2025, not only increases the importance of digital security but also the difficulty of maintaining it correctly.

The project is defined in the context of operating systems and cybersecurity, which are two topics that have gained social and academic relevance in recent times. This end of degree work focuses on one of the roots of cybersecurity, precisely the integrity. Ensuring that critical files and data has not been modified by unauthorized actors is a mandatory part of any system security infrastructure. Furthermore, the trust in the whole system is based on this very principle.

Traditionally, the kind of solutions offering this type of services has been very resource intensive or needed another physical device to handle the overhead. This proposal focuses on making a solution capable of working in any kind of devices, including high end x86 devices as well as low end ARM ones with little overhead.

The base of this intrusion detection system (IDS) will be a Merkle tree that will represent the filesystem. This structure has already been proved secure to detect modifications in the tree for blockchain [1] and some investigation has already been made for the application of it to filesystems such as ZFS [2].

Merkle trees are created using cryptographic hash functions. In this case, mainstream

and state of the art cryptographic hash functions will be used, SHA-256 and Blake3 in particular.

With this, the integrity can be guaranteed by periodically recreating the tree but there is nothing to automatically detect changes to the system. For that task, some utilities offered by the operating system and the kernel will be used to detect these modifications in real time. This tools, namely inotify or fanotify, are used by file explorer indexers and antivirus software and a variety of programs that need to keep track of alterations made to open files in order to update them.

The combination of the two, offers a robust proof of concept (PoC) design and system to offer integrity services on various systems and platforms without too much overhead for them, reducing the performance impact that could be generated.

The scope of the project also contains the integration with another project called “Distributed integrity monitoring system based on Blockchain technology”. The integration completes the IDS with another layer of security that guarantees that the system is not locally compromised or tempered providing a decentralized blockchain to store and manage the root hashes of a network of systems running the system designed and developed in this project.

This report, containing all the work done and all the information used to elaborate the IDS system, is structured in 8 chapters, including this one. First, the general goals of the system will be explained. Then, the planification. After that, the preliminaries to understand and develop the solution followed by the design and implementation and the experiments. Finally, the conclusions and future work will be covered.

2. CHAPTER

The aims of the project

This chapter describes the general objectives of this End Of Degree work as well as the specific goals. The necessary required knowledge to complete this work is also described.

2.1 General objectives of the project

The main aim of this project is to design and develop a system capable of detecting unwanted file modifications, ensuring the integrity of them. In summary, the project will create a software based intrusion detection system (IDS), with sensors to detect changes, alert mechanisms and logging.

To do it, hash functions and efficient cryptographic techniques are going to be used, in combination with operating system implementations that allows real time filesystem monitoring to detect the changes made to it. To accomplish this goal, two daemons will be used. The detailed designed architecture is described in Chapter 6.1.

2.2 Specific objectives of the project

To ensure that the project can be completed and organized correctly, the work was divided in smaller and concrete tasks. These can be interpreted as partial objectives and each one represents an important milestone in the process.

Before beginning with the design and development of the actual solution, it is necessary to research about the different hashing techniques; their security and performance balance, their implementation in the Linux kernel and the hardware requirements, such as hardware implementation of the algorithms to obtain an acceptable performance. Without neglecting the security, the priority of the used hash algorithm must be the performance of software implementations. The solution must be capable of obtaining good performance not only high end x86 devices but also in ARM based low end and low power ones like IoT devices.

The proposed solution is going to implement some kind of Merkle tree structure to achieve its goal, thus the foundations and algorithm design will be analysed.

Since Linux kernel based systems will be used, the variety of available filesystem change detection systems will be analysed. Such as the base API dnotify[3] and the more advanced APIs inotify[4] or fanotify[5].

After that, it is important to study the methodology of daemon development. As a side requirement, it is helpful to be comfortable compiling C and C++ programs and the Linux kernel itself as well as managing different testing environments. For instance, virtual machines, containers and real machines.

Afterwards, with all the gathered information, it is time for design, development and implementation phases. First, the general architecture of the solution will be done. Then, the work will be divided in two software pieces; on the one hand, the daemon containing the detection mechanisms and, on the other, the daemon containing the tree structure.

The final objective of this project is to complement and integrate with the project that implements the distribution of the information between nodes.

The major goals of the project are the following ones:

1. To study and analyze the performance and security of different hash functions.
2. To research the development techniques for Linux daemons.
3. To design a optimal architecture to ensure the goal of the project.
4. To implement the design and evaluate the performance and security.
5. To integrate the solution with the network implementation.

Even if this solution focuses on systems running the Linux kernel, the same approach could be developed for other systems as Microsoft's Windows or Apple's Mac OS.

3. CHAPTER

Project management

As in any project, planning is a necessary step. The definition of different stages and phases along with identifying the risk involved in them will be done in this chapter. All of this will be done with the objective of correctly managing the time, which is one of the most valuable resources on a project of this sort. To avoid potential delays and deviations, a plan with deadlines and milestones is going to be developed with an efficient distribution of the task through time.

Project management also includes all the necessary work to correctly manage all the generated files and source code properly, in order to maintain a proper work dynamic and guaranteeing the proper advancement of the project.

3.1 Description of the phases and their features

The development of the project will be divided in three major categories; management, development and documentation. These cover the main decisions and crucial aspects of the project and will include any difficulties encountered during the development.

3.1.1 Management phase

The management phase group focuses on the tasks necessary to manage the project. Starting with planning and time estimations to complete each activity and following with the

analysis of the deviations that may occur during the execution of the plan. The estimation of the cost of each phase, the possible risks and deviation along with where to place each task in the timeline of the project is key to ensure a successful evolution.

- **Planning**: In this stage, all the specific tasks will be defined and the time needed to complete them will be estimated, selecting the appropriate time windows. Afterwards, a work plan will be defined with the important milestones and key dates.

- **Tracking**: With the aim of checking whether the established plan is being completed as expected, the project will be analyzed during the development process. The necessary actions should be taken to ensure the tasks are being completed in the given time designed for them. The main purpose of this is to find risks and identify critical points and delays so the tasks and milestones can be updated accordingly; modifying, creating or replacing the current objectives if necessary.

3.1.2 Development phase

This phase contains all the tasks related to the actual execution of the project. All the necessary investigation, research and the design of the solution, including all the programming and the analysis of the results.

- **Preliminary study**: All the previously mentioned research part is included in this task; Linux daemon development techniques, hash functions security and performance analysis, kernel and C/C++ code compiling, Linux kernel file-system change monitoring and Merkle tree structures.

- **Design**: This task groups the architecture design and the programming the different aspects of it. Such as the two daemons, the communication between them, the hash algorithms to use and the integration with the network system.

- **Analysis**: After developing the solution, the proposal will be analyzed. The analysis includes not only the performance but also the security of it. The main objective of doing it is to find weak points, errors and possible upgrades to the design and implementation.

3.1.3 Documentation phase

The documentation phase will be carried away in parallel to the other ones, it sums up all the work done to write and prepare the documentation of the project.

- **Memory:** An important part of the project is to keep track of all the research, work and advances made during the time it lasts. This item includes all the related work necessary to write, correct and complete the report.
- **Presentation:** At the end of the timeline, the work must be presented in the defense. Preparing a good presentation needs some time, and this tasks will be used to keep track of it.

3.2 Time estimations

After explaining and defining the different phases of the development of the project, the expected time necessary is going to be estimated as a part of the planning process. The estimations and the real amount of time it was necessary to complete each one of the tasks is detailed in the Table 3.1.

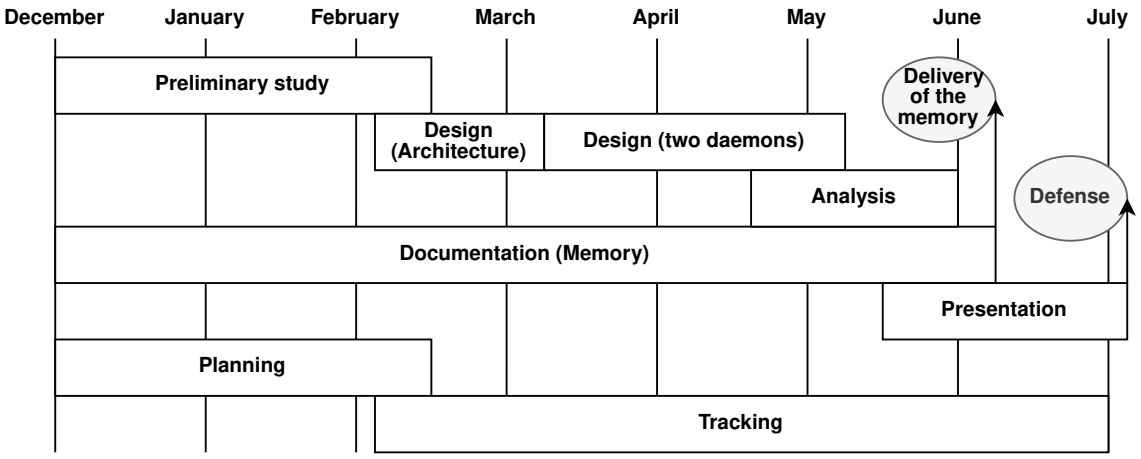

Figure 3.1: Project timeline and milestones.

The project has some important milestones that need to be achieved on time in order to complete the project as represented in the Figure 3.1. The last and most important milestone is the defense, expected to be made in July 2021.

	Estimated time (h)	Final time (h)
Management phase	25	22
Planning	15	13
Tracking	10	9
Development phase	205	191
Preliminary study	50	45
Hash functions	10	9
Development methodology (Daemon)	15	13
File-system notification mechanisms	15	14
Merkle tree	10	9
Design	125	114
Architecture	25	22
Notification daemon	50	45
Merkle tree daemon	50	47
Analysis	30	27
General	15	14
Performance overhead	15	13
Integration	No estimation	5
Documentation phase	75	88
Documentation	55	68
Presentation	20	20
Total amount of time	**305**	**301**

Table 3.1: Time estimation for each tasks and their final required time.

3.3 Risk Management plan

Large projects like this always have risks involved with them. For this reason, it is necessary to prepare a strategy beforehand as well as an actuation plan to minimize any potential risk before they cause an impact on the project.

Some of the risks involve the management of the data, source code and documents generated during the project. To mitigate any data loss that could occur, all the data will not only be backed up in a separate disk but also online backup services will be used.

In large projects, managing the code could generate problems if it is not done correctly. Source code version control, like git, can help to correctly manage different versions of the program. For instance, in addition to the main version, a variation to test the performance

of the solution. It also facilitates keeping track of all the changes made in each step of the development and easily go back to any specific change if it is necessary.

Another major risk could be deciding to use a technology or software that might not meet the requirements to complete the project. Reading the documentation before starting to code helps reduce the risk of this event. The cost related to re-planning of the project is much smaller if it occurs during the documentation and planification phase.

All of this, along with a conscious planning, helps to successfully finalize the project in the estimated time, fulfilling all the specified goals of it.

3.4 Deviations

As of any project, during the progress of the it some deviations occurred. Some of them were design changes that were discovered during the first phases. This kind of early deviations do not usually suppose much time difference. On the other hand, some other deviations occurred which induced time difference.

When the project was first defined, the structure was different. Instead of two daemons, the part responsible for detecting changes on the files was a kernel module. During the project documentation and design phase, some major inconveniences and incompatibilities were found.

The available headers for kernel module development are really small. This means, all the complex functions already implemented in libraries are not available. Moreover, standard and common system calls should not be used, even if technically it is possible to do so [6]. These system calls are designed to offer access to the kernel to userland processes. The recommended approach is to use the subset of kernel calls available but not all the ones that were necessary are implemented on such a low level.

Taking all of that into consideration, it is not surprising that inotify or fanotify are not available to the kernel. So, in order to detect the changes made to the filesystem, something similar to what those APIs do would need to be implemented almost from scratch.

This does not mean at all that this approach is impossible, but the amount of knowledge and time required to do it does not make sense when Linux offers a variety of complete APIs implementing the same thing on a higher level.

After analyzing the situation, the decision of dropping the kernel module was taken in favour of a more complete user-level approach.

Another deviation was the integration with the complementary project. Even if the integration was expected, it was not defined in the tasks and time estimations. This decision was taken because it was not sure how and when it was going to be done or even if it would be possible to do it due to compatibility or time constraints. Finally, it was carried away and the time necessary to do it was small enough not to represent a big deviation in the total hours estimated to the project.

All of all, the total time estimations were meet and the project was finished on time. The development phase was overestimated at the start and in reality, the hours necessary to complete all the tasks were a bit smaller. In contrast, some other task like writing the documentation took more time than planned. In conclusion, the total amount of time was approximately well estimated but the dedication needed for each task was not as good.

4. CHAPTER

Linux: Daemons and D-Bus

This chapter contains explanations and information of the used technologies to complete this end of degree work. More precisely, all the related information of Linux daemons, D-Bus and all the other technologies that go with them.

4.1 Linux Daemons

In Unix-like systems, like Linux, a daemon is a type of program which runs in the background without requiring any user interaction or control. It waits until some specific event occurs that triggers an action or response.

The tasks carried out by daemons are really diverse. For instance, a daemon can be a web server waiting for a petition to serve a website or some process in charge of writing information to a file, like syslog.

But, before digging into the structure of Linux daemons, some introduction Linux processes shall be made.

4.1.1 The Linux process system

A process is a running instance of a program, they are managed by the kernel and they have some distinct properties. Each running process has an unique identifier called PID and a parent PID, which indicates who launched it. The init process is the first process

launched after the kernel is loaded and, for this reason, it always has the PID of one. Init is responsible for starting all the other processes needed by the system to run. This process will run until the system is shut down. In addition to this first process, there are three main types of processes in Linux: interactive, batch and daemon.

Interactive processes are run and controlled through a terminal session by a user. Otherwise stated, such processes are not run automatically by the system rather by someone connected to the system.

Batch processes, in contrast, are not associated with any command line and they are launched from a queue. They usually perform recurring tasks when the system usage is low.

Finally, a daemon is a process whose parent has a PID of 1 and, as it has been mentioned before, runs in the background. Daemons are usually started at boot time by the system, even though they can be launched after that too. Commonly, the daemon version of a program has a "d" at the end. Such as, sshd for the ssh server or httpd for Apache http web server.

4.1.2 The structure of a daemon

A daemon can be structured in two different ways; SysV daemons and new-style daemons, as described in the Linux manual pages [7]. SysV daemons are the traditional way of doing it and they follow this structure:

- Close all the open file descriptors.
- Reset signal handlers and signal mask.
- Sanitize the environment block and reset variables.
- Call fork to create a background processes.
- Detach from terminal and create a independent session with setsid.
- Call fork again from the child process.
- Call exit on the first child to ensure only the second child (the actual daemon) stays around.
- In the daemon process, connect all the standard output to /dev/null.

- In the daemon process, set the umask to 0 and change the working directory to the root directory (/).
- In the daemon process, write the PID (from getpid) to a file.
- In the daemon process, drop privileges if applicable.
- Notify the original process that the initialization is complete.
- Call exit in the original process.

But nowadays, the recommended fashion of developing is the new style daemon structure. For this kind of daemons, none of the previously mentioned initialization steps need to be done and it is even recommended not to execute them.

To create a new-style daemon, the following must be implemented:

- Manage SIGTERM and SIGHUP to shutdown or reload the configuration.
- Provide a correct exit code. Init system will use them to detect errors.
- If possible, expose the daemon's control interface via D-Bus IPC and grab a bus name.
- Systemd integration: provide a .service unit file with the corresponding information for starting, stopping and restarting.
- Rely on the init system resource limits and privilege dropping.
- If applicable, the daemon should notify the startup completion.
- The logging in new-style daemons may be done using fprintf() or printk() which will be forwarded to syslog instead of using the syscall syslog() directly.
- O_NOCTTY must be specified on all open() syscalls so that no controlling TTY is accidentally acquired.
- If the daemon offers services via sockets to local or remote clients, it should be made socket-activatable. This ensures that it can be restarted without losing any request.

Depending on the style, the structure will change but on a higher level, the execution of both daemons can be separated in two main sections: the initialization and the actual daemon code. For this project, the new-style daemon structure will be used, since it is the recommended approximation for all newly developed ones. The following sections will only refer to this type.

4.1.3 Launching a daemon

While old style daemons daemons tend to be only activated at boot time or manually by the administrator, new style ones offer a great variety of activation methods.

This mainly includes socket-based activation, bus-based activation, device-based activation, path-based activation and timer-based activation. Every method has its own specification but that will not be analyzed here. For this project, boot and bus types will be explored, since the two daemons should be started at boot and one of them has a dependency on the other's bus service.

For the activation on boot, in new style daemons a symlink should be made on the `.wants/` directory of one of the following systemd targets: `multi-user.target` or `graphical.target`.

These directories indicate the system what each boot type wants to be running when the boot process is finished. The system will try to launch all the daemons corresponding to the files inside that directory.

The Bus activation is different, the daemon will be activated whenever a client, this is any other process, wants to use any of the D-Bus services the daemon offers. The bus activation can be configured in the D-Bus Service files. These files are similar to the systemd service files that will be explained in section 4.1.4 but for D-Bus specific configurations.

4.1.4 Systemd

Systemd[8] is used as an initialization process for Linux based systems, so when the system starts, systemd is launched as the init process to initialize all the other processes and it always has the PID 1. Systemd is responsible of managing itself and any other daemon running. It can be defined as a daemon managing daemon.

In the architecture of the system, it is located on top of the Linux kernel and provides a structure to manage, initialize and stop all the daemons that work with the kernel and

userspace. It also provides daemons to manage basic system components at low level like journald, networkd, logind and user sessions. Appart of these, it manages the D-Bus IPC daemon and PulseAudio for sound.

To integrate a daemon with systemd, a unit file should be made. This file defines the type of daemon, the name, the files that should be executed, the dependencies and other relevant information related to the daemon execution and configuration.

As described in [9] these files are composed by some sections and directives with a value. Each section is defined as "[Section name]" and describes some metadata and how the daemon should behave. The Unit section is mandatory, it describes the general information , the type and basic requirements and dependencies of the daemon. There is an optional section called Install that defines the behaviour of the daemon when it is enabled or disabled. There are some other use-case specific sections such as Sockets, Automount or Swap that are only used when the daemon makes use of it.

Inside each section, the directives are declared in the form of "DirectiveName=Value".

4.1.5 D-Bus

When considering how to communicate two processes, some techniques were analyzed. Some of them were the classic IPC mechanisms. For instance, shared memory [10, p. 368-421], named pipes [10, p. 252-265] and sockets [10, p. 490-591] among others. But since this project is using daemons, the decision of using D-Bus was taken.

D-Bus[11] is an inter-process communication or IPC mechanism that provides a bus-like messaging interface for processes on the same system. It provides a wire-protocol to accomplish that along with a bus daemon which allows apps to find and control each other. Freedesktop publishes a low-level library and a reference implementation of the protocol but it is not recommended for general use. Instead, there are a lot of more high-level implementations and bindings for different programming languages[12]. Each implementation has its unique things but all of them are compatible.

For the programs, the sd-bus systemd implementation will be used to take advantage of an already used library, as it will be used for the daemon and logging.

Now some of the basic D-Bus nomenclature of the key components will be defined:

- **Bus names:** The bus names are used to identify an application on the bus. There are two types of names: Well-known and unique. Unique names are assigned when

the connection to the bus is established. They begin with ":" and they can not be reused in the lifetime of the bus. Well-known names instead, are requested by the client and are used to easily identify programs. They follow a DNS-like naming scheme; "org.freedesktop.Dbus", for example.

- **System bus:** A bus designed to communicate system applications and user sessions.
- **Session bus:** A bus designed to exchange information between two applications, also known as user bus.
- **Object path:** It is an unique identifier for an object inside of an application. They follow the standard Unix file system path structure.
- **Interfaces:** The interfaces list the methods and signals available to each D-Bus object. Each one defines the arguments and types they accept.
- **Methods:** Methods are functions inside an object that can be called from another application. They accept any number of arguments and they can return a response with any number of values.
- **Signals:** Signals are similar to method responses, they can contain any number of values. In this case, a group of clients will subscribe to the signal and will receive asynchronously the signal whenever the D-bus object sends it.

Using D-Bus: The "server"

The server will be the D-Bus client offering the object to other programs, that is, provides a method than can be called to accomplish some actions. To create it, first the method code should be written. The code will contain a function accepting some arguments, doing some calculations and returning with some values to the bus.

Then, in the main program, these steps need to be followed:

- Open a bus connection, with either the system bus or the user bus.
- Define and add the interface definition to the bus, including the object and method name.
- Request a well-known bus name (if necessary).

- Inside an infinite loop, process and wait new method calls from other programs.

Using D-Bus: The "Client"

In order to use the newly defined object and method, in the client these steps should be made:

- Open a bus connection, with the bus of the object you want to use.
- To send a new message, call the method with the well-known name, the interface name, the object name, the method name along with the arguments.
- Wait for the response.

4.2 File system change detection mechanisms

In Linux operating systems there are needs to monitor and detect events in the filesystem for a variety of causes. For this reason, there are three APIs available in the Linux kernel to monitor changes: dnotify, inotify and fanotify. These tools can be used by search engines, to index changed files without re-scanning the whole filesystem each time; antivirus services, to monitor and block changes to certain files or to detect changes and maintain file system integrity, which is the aim of this project.

4.2.1 Dnotify

Dnotify was the first developed API for this matter, nowadays it is only available for compatibility reasons but it has been replaced by inotify. This first approximation supported a very little set of change types and it could only monitor directories. It was necessary to open a file descriptor for each watched item. Opening file descriptors can be a problem to unmount removable devices as Linux does not allow unmounting devices with open file descriptors in them. The mechanism used to notify the changes was sending signals to the applications, and had issues with hard links. All this made dnotify a heavy and resource intensive system to monitor changes.

4.2.2 Inotify

Inotify was designed as a replacement for dnotify. The major goal was to fix some of the problems that were present. Some of the advantages over the older API are that inotify only requires a single file descriptor. The file descriptor is used to create a watch table that will contain all the "watch descriptors" linked to each watched element. This change alone fixes the problem with removable devices and the bottleneck generated by a lot of open file descriptors.

Moreover, inotify also allows to monitor files, granting developers a more granular control of the monitored system if they need to. Inotify discards the signal notification system used by dnotify in favor of a more robust select and poll mechanism.

It is important to mention that inotify has its drawbacks too. Even if the supported set of event types is bigger, the API is not able to report all the sysfs and procfs events. It also forces the developer to create a watch for every subdirectory since it does not support recursive watching of directories. In addition, it does not work with network file systems because it requires the kernel to be aware of the events to notify them. Another big inconvenience is the rename event. Inotify generates two separate events and detecting them can be a challenge and generate problematic race conditions [4, Dealing with rename() events].

4.2.3 Fanotify

Fanotify is yet another change notification API, but it also provides interception characteristics. Two queues are used for that, one of them will contain the notifications and it works similarly to what inotify does. The other one, the permission queue, in contrast, will notify the event and wait for a response to authorize the use of that element.

When it was implemented, fanotify had a really small subset of event types and it did not support create, delete or move events. Linux 5.1 added the support for them in a revised version of the API[13].

In comparison with inotify, this API supports recursive event monitoring of entire mount points. This can facilitate some development when the whole filesystem needs to be monitored. Like with the others, directory monitoring is not recursive, so newly created subdirectories need to be manually watched.

5. CHAPTER

Integrity: Hash function and Merkle tree

This chapter will focus on hash functions and the Merkle tree structure. First a introduction to hash functions will be made, then the two hash functions that will be used for the end of degree work will be explained. Finally, how Merkle tree structures work will be analyzed.

5.1 Hash Functions

A hash algorithm is a function whose objective is to convert an arbitrary amount of data into a predetermined size output of bytes, normally interpreted as hexadecimal number strings. If these functions are used in the cryptographic security space, they are called cryptographic hash functions and they must provide a great degree of security as they must be attack resistant. The more used and recognizable ones are MD5 or SHA (1, 2, 256, 512).

5.1.1 Properties

Hash functions have some common properties:

- **Efficiency and low computational cost**: Hash functions are designed to be fast and light. In most functions, the calculations are done very efficiently to achieve

this objective. Some specific functions are designed to be intentionally slow to provide more security and prevent brute-force attacks. Such functions are used to store passwords, for example.

- **Fixed output length**: For every possible input, independently of the size of it, the output must always be the same length. Hash(data) => h(2^n) bits. That does not mean that every function has the same length. Moreover, some of the functions can be used to generate different output sizes by changing the configuration parameters.
- **Randomness**: The generated output should seem random.
- **Irreversible**: It should not be possible to infer the input from the output.
- **Deterministic**: Hash functions must always provide the same output for a non-changing input.

5.1.2 Preimage and collision attack resistance

These kinds of attacks pretend to obtain a message which, after being hashed, a specific output is generated. There are two main resistances to them.

- **Preimage resistance**: Knowing the output y, it should not be computationally feasible to find any input x that after being hashed, matches that output. h(x) = y.
- **Second preimage resistance**: It should not be computationally feasible to find any second input that generates the same output string as a predefined input. Given the message x, find a second message x' so that h(x) = h(x'). There is a more general approach to this kind of attack that consists of finding any two inputs that generate the same output.

For an algorithm that generates an output of n bits, the amount of tries necessary to get a collision should be 2^n. If the two properties are correctly fulfilled, the only possible way to obtain a collision is to use brute-force methods.

5.1.3 Uses cases

There are some widespread uses of hash functions that are really important for the way we use technology nowadays.

- **Password storing**: Passwords should not be stored as plain text or even be known by the service. The generated hash will be stored instead of the actual password so only the user knows it.
- **Data integrity and identification**: Hashing the same file two times will give the same output if the contents are not modified. This property is used to identify and guarantee file integrity of distributed software or any kind of file, for example. This characteristic will be used in this project to check file integrity.
- **Blockchain technology**: Another use that has become mainstream lately is blockchain technology. Blockchain uses hashes to verify the integrity of each transaction and each block. Moreover, even the mining part in some currencies like Bitcoin is based on discovering a hash that starts with n number of bits set to 0.

5.1.4 SHA-256

SHA-256 is part of the SHA-2 hash function family along with SHA-224, SHA-384 and SHA-512. These functions use the same algorithm but provide a different length for the output. For example, SHA-256 generates a 256 bit output while SHA-384 generates 384 bits for it. SHA-2 is a more secure version of the SHA-1 algorithm and provides a more robust protection against the attacks mentioned in the previous section as it has no collisions found as of today [14].

SHA-256 is a widely used algorithm fora great variety of applications such as generating digest of files or internet security (IPSec, AES, TLS, SSH, PGP) and provides a good performance overall since it can take advantage of advanced introduction sets of modern processors such as SSE4, AVX1 or AVX2 [15].

For all of this, it is a good starting point for any application that relies on cryptographyc digests.

5.1.5 Blake3

Blake3 is a more state of the art cryptographyc hash algorithm that provides a much faster, secure, highly parallelizable alternative to SHA-1, SHA-2, SHA-3 and Blake2. It is designed with modern architectures in mind, to guarantee good performance in both, x86 and ARM based systems taking advantage of their SIMD instructions [16].

The key features and characteristics of BLAKE3 are:

- Binary tree structure to provide a highly parallelizable design.
- Fewer rounds in the compression functions compared to BLAKE2s.
- Zero-cost keyed hashing.
- Built-in extendable output.

All of these provide an advantage in performance and security against the already mentioned BLAKE2 and SHA functions, making Blake3 a really interesting alternative to test and compare in new applications that rely on fast-computing hash digests.

5.2 Merkle tree

A Merkle tree or hash tree is a data structure that allows an arbitrary set of independent data to be linked with a single hash value. Each layer of the tree will create a hash using the hash values of its immediate child nodes. The leafs will contain the hash of the actual data. These trees can be both, binary or non binary and the structure is highly scalable. R. Merkle proposed this structure along with an algorithm to calculate the hash of all the nodes, leafs and the root hash. As explained in [17] with a binary tree, each node must perform three operations:

- Authenticate the right node (Obtain the hash of the right node).
- Authenticate the left node (Obtain the hash of the left node).
- Sign a single message (Calculate the hash).

That implies that each node will take the hashes of its two children and calculate a single hash with them. That hash will be used by its parent to calculate a new hash. This operation will be repeated until the root node is reached.

The Figure 5.1 shows an example of the calculation of the root hash in a small binary tree. First, the child nodes calculate the hash of the data associated with them, a file in this case. Then, each intermediate node uses the hashes of its children to calculate a new hash. Finally, the root node uses the hashes of the two child intermediate nodes to calculate the root hash.

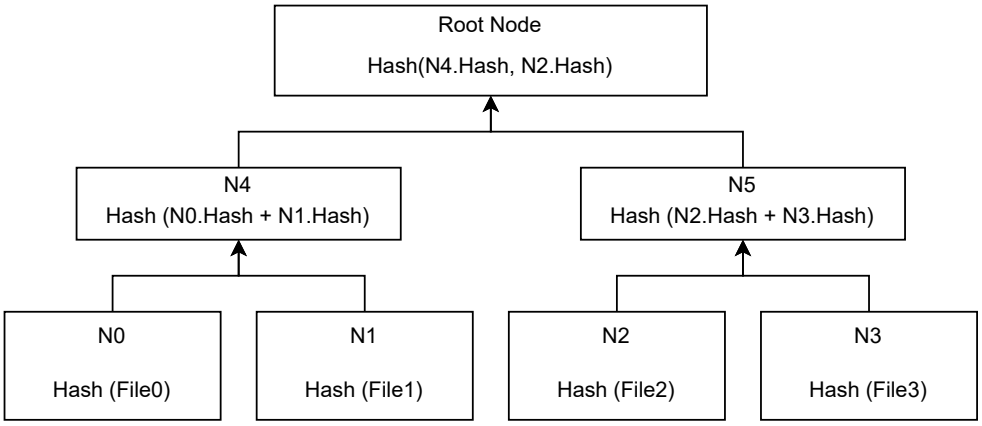

Figure 5.1: Example of a Merkle tree.

As mentioned before, the tree does not have to be a binary tree. In a tree with an arbitrary number of children in each node, the process of calculating the corresponding hash will be nearly the same. Instead of aggregating two hashes on each one, a hash will be aggregated for each child. This approximation can be really useful in some scenarios such as blockchain technologies like Bitcoin [18] or when dealing with complex structures such as filesystems.

The tree structure implies that if a node changes and consequently, the hash associated to that node changes, all the nodes starting from it to the root will have incorrect hash values. An example of this is shown in Figure 5.2. Such property can be utilized to detect changes or tampering.

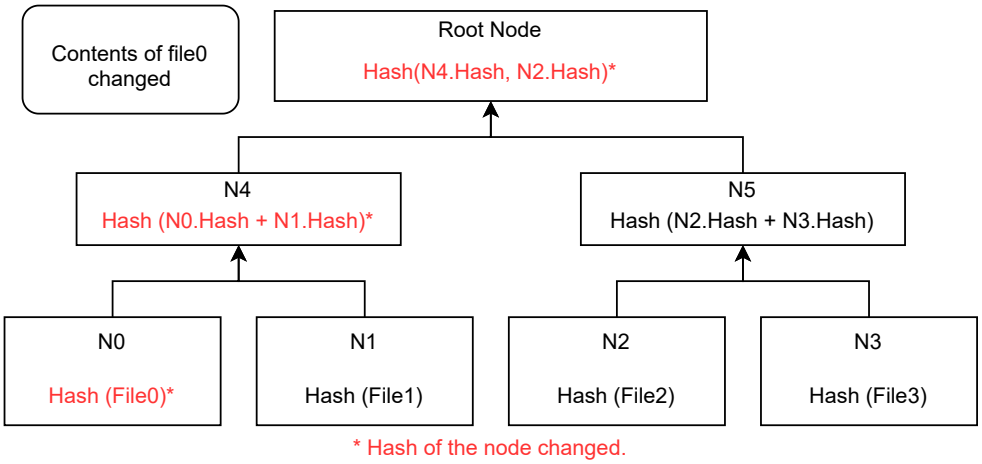

Figure 5.2: Example of a change in Merkle tree.

6. CHAPTER

Design and implementation

In this chapter, the design of the solution and how it has been implemented will be described. First, the general design of the architecture and some decisions that have been taken. Then, each daemon will be explained and finally, the integration with the complementary project will be detailed.

6.1 Architecture and general overview

The objective of this project is to detect changes made to the selected set of files and directories. To do so, it generates a data structure similar to a non-binary Merkle tree of the directory structure with the corresponding hashes. The root hash represents the whole structure and if any of the files or directories is altered, the root hash will also change.

To facilitate the work, the process will be performed by two different programs. The first one, named fsCheck, will be in charge of generating the Merkle tree during the startup. Then, it will update it when it receives a notification indicating that a file or directory has changed. The other daemon, named fsNotifier, will monitor the filesystem to detect when a file or directory is accessed and send a notification whenever it is modified, deleted or created.

Creating the daemons is not enough, they must communicate between them for the system to work. This will be explained in more detail in the Section 6.2 but, in summary, fsNotifier will send a message to fsCheck containing the path of the file or directory that

triggered it and a code to identify the change. In the Figure 6.1 there is a general overview of the system.

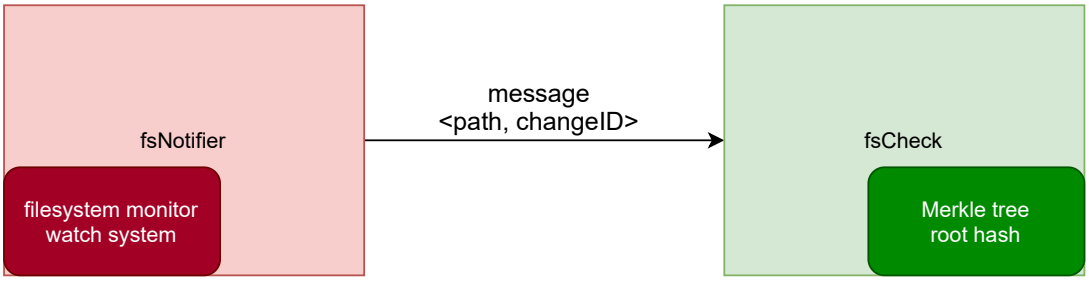

Figure 6.1: Overview of the system and the communication between daemons.

6.2 Communication between the daemons

To make possible the communication between the two daemons and complete the designed functionality, D-Bus is used. During their initialization process each one will request a bus name and in the case of fsCheck, a D-Bus object will be created. This object exposes a public facing method called getUpdate that can be called to send the information. The details about how this is implemented on each daemon will be explained in the Section "How it works" of each daemon.

Each time a change is detected by fsNotifier, it will call the `getUpdate` method with the corresponding information as it has been explained before. D-Bus will then route the message from the sender to the owner of the method to complete the communication.

6.3 fsCheck - The tree

The fsCheck daemon needs to create a tree to store the corresponding hash of each file and directory. The generated structure will replicate the structure of the filesystem starting in the root directory. Each file or empty directory will be a leaf and will not have any child nodes. Directories that have files or other directories in them will be considered intermediate nodes.

6.3.1 Classes and data structures

To replicate the structure, two classes were designed. First of all, a tree class that will contain the generated root hash, the total number of nodes and a pointer to the root node. This class is thought to be used and accessed by external apps that do not need to understand the tree structure at all. For an external application that could use it, the root hash and the number of nodes are the only meaningful data that need to be accessed.

The second class, the node class, will be used to create the actual tree structure. This class has more members; a path to the file or directory, the hash, a boolean to indicate if the node is a leaf, the type (file or directory) and a vector for the child nodes. In addition, it also contains a pointer to the parent node.

6.3.2 How it works

The operation of this program can be divided in various phases. First, when the program is started, the initialization phase is executed. This phase includes the following steps:

- Opening the log file.
- Assigning the possible signals that can receive the program to the signal handler.
- Open a bus connection to D-Bus.
- Assign the D-Bus object to the connection and request a bus name.
- Load the configuration file, get the initial path and the hash algorithm to use.
- Initialize the Merkle tree and generate all the hashes.

Hash algorithms

This daemon can work with two hash algorithms, Blake3 and SHA-256. The algorithm to use will be specified on the configuration file of the daemon and the default one is Blake3. To reduce the overhead that can be created by checking on each hash calculation what algorithm to use, function pointers are used. Each time a node is created, the pointer of the function to generate the hashes will be updated to the one containing the implementation

of the selected algorithm. To do it 6.1, the placeholders for the calls will be defined using the function member of std. The real implementation of the functions for each hash algorithm will be in the functions with the prefix `_blake3_` or `_sha256_`.

```
1  //mnode.hpp
2  /*Blake3 hash functions*/
3  void _blake3_HashFile();
4  void _blake3_HashDir();
5  /*SHA256 hash functions*/
6  void _sha256_HashFile();
7  void _sha256_HashDir();
8
9  /*std functions holders for the selected hash variant*/
10 std::function<void(Mnode *)> HashDir;
11 std::function<void(Mnode *)> HashFile;
12
13 //mnode.cpp
14 Mnode::Mnode(int mode){
15     if (mode == _BLAKE3){
16         HashDir = std::bind(&Mnode::_blake3_HashDir, this);
17         HashFile = std::bind(&Mnode::_blake3_HashFile, this);
18     }
19     if (mode == _SHA256){
20         HashDir = std::bind(&Mnode::_sha256_HashDir, this);
21         HashFile = std::bind(&Mnode::_sha256_HashFile, this);
22     }
23 }
```

Listing 6.1: Implementation of the selection of the appropriate function pointer depending on hash mode selected.

So the call will always be the same 6.2, independently of the algorithm chosen.

```
1  //mnode.cpp
2  void Mnode::genHash(){
3      if(type == MT_FILE)
4          HashFile(this);
5      if(type == MT_DIR)
6          HashDir(this);
7  }
```

Listing 6.2: Generic call to the function in charge of calculating the hash for files and directories.

Daemon

Since it is a daemon, the new-style daemon design fashion has been followed to implement it. That includes all the steps mentioned in the Section 4.1 for them. Not only what it has been mentioned in steps 1 and 2 but also designing a `.service` file with the adequate configuration of it.

D-Bus

For the D-Bus configuration and implementation, the systemd implementation and functions have been utilized since it was a dependency already in use for the logger. In addition, it is included on almost any modern Linux system. Following the order indicated in the initialization steps, first it is necessary to open a connection with the bus. In this case, the user bus is being utilize. This bus is easier to work with at the developing stages as it does not require root access and there is no need to move the binary of the program to system folders. Doing it is easy enough for both, the system bus and the user bus. The Listing 6.3 indicates how it is being done and how it should be done to use the system bus, commented in the line 7.

```
1  static sd_bus *bus = NULL;
2
3  //fsCheckDaemon.cpp
4  //Open a connection to the bus
5  sd_bus_open_user(&bus); //user bus
6
7  //sd_bus_open_system(&bus); system bus
```

Listing 6.3: Opening a bus connection.

After that, it is necessary to declare a data structure called vtable that contains all the references for the function that will be called whenever a new message is received and after that assign it to the bus adding the object. Finally, the program will request a bus name to make it easy to contact from other processes 6.4.

In the vtable, the signature of the incoming data as well as the one of the reply are indicated. For this function, the "is" signature is used and represents that an integer and a string are expected. The signature matches with the change code and path defined in the architecture. The method will return an integer ("i") if the no-reply clause is not provided in the call.

```
/***
* sd_bus_add_object Assigns the previously defined method to the program in the bus;
* sd_bus_request_name is used to request a bus name for the program.
***/
//fsCheckDaemon.hpp
//The method to be called
static int method_getUpdate(sd_bus_message *m,
void *userdata, sd_bus_error *ret_error);

/* Definition of the bus object and method*/
static const sd_bus_vtable fscheck_vtable[] = {
    SD_BUS_VTABLE_START(0), SD_BUS_METHOD("getUpdate", "is", "i", method_getUpdate,
    SD_BUS_VTABLE_UNPRIVILEGED | SD_BUS_VTABLE_METHOD_NO_REPLY), SD_BUS_VTABLE_END
};

//fsCheckDaemon.cpp
sd_bus_add_object_vtable(bus, &slot,
    "/net/aritzherrero/fsCheck",  /* object path */
    "net.aritzherrero.fsCheck",   /* interface name */
    fscheck_vtable, NULL);
//Request the bus name
sd_bus_request_name(bus, "net.aritzherrero.fsCheck", 0);
```

Listing 6.4: Initialization of the bus.

When a message is received, the function `method_getUpdate`, described in 6.5, will be called. Before processing the data, the bus message must be read and after it, the reply must be sent.

```
//fsCheckDaemon.cpp
static int method_getUpdate(sd_bus_message *m, void *userdata, sd_bus_error *ret_error) {
    const char *str; int change;
    sd_bus_message_read(m, "is", &change, &str);
    /*** Process the data ***/
    sd_bus_reply_method_return(m, "i", 1);
```

Listing 6.5: method_getUpdate: Read and reply bus messages.

At this point of the execution, the initialization process of the bus has been completed. After this process is concluded, the program starts listening for changes received as specified in the architecture section. This second phase consists of an infinite loop where the program waits until a message is received.

When that happens, it will process the message and continue waiting indefinitely for the next one. On this loop, resumed in the Listing 6.6, the program needs to process any existing request and wait until another one is received.

```
1  //fsCheckDaemon.cpp
2  while(true){
3      sd_bus_process(bus, NULL);
4      /*Error checking*/
5      sd_bus_wait(bus, (uint64_t) -1);
6  }
```

Listing 6.6: Main loop of the program, receiving and processing the bus messages.

If something is received, `sd_bus_process` will call the previously defined getUpdate method with the message received from the bus.

Processing a request

To process a request, first it will get the change code and depending on it the appropriate function will be called.

- If a file is deleted, the corresponding entry on the Merkle tree will be deleted and the hash of that branch of the tree will be recalculated.
- If a file is edited, the tree will not be changed but the corresponding hashes of that branch will be recalculated.
- If a file is added, a new entry will be created on the corresponding part of the tree. Then, the hash of the new entry will be calculated and finally, the hashes of that branch will be calculated as with the other other changes.

The appropriate changes when a directory is created or removed are performed too. The update includes removing all the child nodes if there is any after deleting a directory and adding a new directory when it is created, calculating the digests of any files created inside it after.

More detailed information about the implementation and the whole code of the program can be found in the GitHub repository [19] of this daemon.

6.4 fsNotifier - Detection and notification

The fsNotifier daemon is responsible for monitoring the filesystem, detecting changes and sending notifications to the other daemon. To do so, the daemon needs to use one

of the available notification mechanisms on the Linux kernel and create all the necessary watches.

6.4.1 Classes and data structures

To wrap all the necessary objects together, the notifier class has 4 members; an integer containing the file descriptor of the watch system, the path of the root and two maps containing the link and reverse link the path of each entry with the corresponding watch descriptor number. Two maps are utilized to allow more efficient reverse look-ups.

At first, instead of the two maps, a vector was used. This approximation was correct if no watches where removed, but after one is removed and a new one is created, the number of the descriptor freed is not utilized anymore. Using a vector results in a inefficient structure since a lot of empty vector entries are generated.

6.4.2 How it works

Like the other daemon, this one is also implemented as a new-style daemon so the structure is similar and a lot of the steps of the initialization are common. For this very reason, all the common steps will only be mentioned, without a detailed explanation.

- Opening the log file.
- Assigning the possible signals that can receive the program to the signal handler.
- Open a bus connection to D-Bus.
- Load the configuration file and get the initial path.
- Initialize inotify and add a watch to each directory.

The notification mechanisms will not monitor all the subdirectories but only the direct files in them. So, to monitor all the entries starting on the root path, it is needed to manually add a watch to each subdirectory in the file system tree to be monitored. It is important to maintain the relationship between the path of each directory and the generated watch descriptor since the system will not keep them automatically. That means that some kind of structure needs to be maintained and updated each time a new entry is added or removed.

Inotify

This time, not only the bus need to be initialized but also the Inotify structure. To do it, first the daemon will create and initialize a inotify instance and save the file descriptor. This file descriptor will be used to add and remove watch descriptors for each directory. The initialization process is represented in the Listing 6.7.

```
1  //notifier.cpp
2  Notifier::Notifier(fs::path path){
3      basepath = path;
4      ino_fd = inotify_init();
5  }
```

Listing 6.7: Initialization of the watch system for inotify.

Next, it is necessary to add a watch for each one of the subdirectories. To create them, the tree structure of the filesystem starting in the root will be traveled. In the process, a watch will be added for each directory encountered as represented in the Listing 6.8. It is important to mention that the system checks if that directory is a symlink to another path in order to detect and avoid infinite loops caused by path that point to a parent directory.

```
1  //notifier.cpp
2  void Notifier::addInotifyWatch(fs::path basepath)
3  {
4      int i = inotify_add_watch(ino_fd,basepath.c_str(), INO_MODE);
5      store[i] = basepath;
6      reverse_store[basepath]=i;
7
8      fs::file_status s;
9
10     for(auto& p: fs::recursive_directory_iterator(basepath)){
11         s = fs::symlink_status(p);
12
13         if(fs::is_directory(s) && !fs::is_symlink(s)){
14             int i = inotify_add_watch(ino_fd,p.path().c_str(), INO_MODE );
15             store[i] =  p.path();
16             reverse_store[p.path()]=i;
17          }
18     }
19 }
```

Listing 6.8: Adding missing watches for the tree.

After the initialization is completed, the next phase is an infinite loop. In this second phase, the program will wait until the change detection mechanism detects a change. After

that, all the alerts that were generated will be processed and for each one a message will be sent to the fsCheck daemon with the path and the code identifying the modification.

To send the message, as it is been already mentioned, D-Bus will be used. In the Listing 6.9 the steps to generate and send a message are presented. First, the message is created with `sd_bus_new_method_call` indicating the path, object and method to be utilized. The names must be the same indicated in the destination application. After that, the arguments are appended. The arguments must match the signature "is" defined in fsCheck, that is a integer that represents the event code and the path of the event as a standard string.

```
1
2       //fsNotifierDaemon.cpp
3       int send_bus_message(int event, fs::path path){
4          sd_bus_message *m = NULL;
5
6          sd_bus_message_new_method_call(bus, &m,
7                     "net.aritzherrero.fsCheck",      /* service to contact */
8                     "/net/aritzherrero/fsCheck",     /* object path */
9                     "net.aritzherrero.fsCheck",      /* interface name */
10                    "getUpdate");                    /* method name */
11
12         sd_bus_message_append(m,"is",               /* input signature */
13                               event,                /* first argument */
14                               path.c_str());        /* second argument */
15
16         sd_bus_message_set_expect_reply(m, 0);      /*Set No-reply*/
17         sd_bus_send(NULL, m, NULL);                 /* Send the message*/
18         sd_bus_message_unref(m);
19         return 0;
20  }
```

Listing 6.9: Generation and sending process for notification messages via D-Bus.

Next, the message is set to not expect a reply in order to avoid blocking the program until it is received. Not waiting for a response helps to minimize the risk of losing events. Finally, the message is sent with `sd_bus_send`.

Processing requests

If the modification adds or removes a directory, the corresponding watch will be added or removed from the system to continue and maintain the correct monitoring of all the files.

The program observes the directories in order to monitor the files in the tree, instead of

creating a watch for each file. This helps to keep the number of watches smaller and prevents the constant update of the system each time a file is created or deleted.

More detailed information about the implementation and the whole code of the program can be found in the GitHub repository [20] of this daemon.

6.5 Memory leaks

After the initial feature development of the program, even if it works without any crashes, it might still have some errors that cannot be detected with standard tests and use cases but that can cause crashes or unpredictable behaviours in edge cases due to the incorrect use of the memory or memory leaks. The most popular tool to check the correctness of C and C++ programs in this aspect is Valgrind, more precisely, Valgrind's memcheck. This tool offers an automatic detection of memory leaks and other errors related to how the memory is allocated and freed during the execution time.

Memcheck differences between 4 types of problems:

- **Definitely lost**: The program has memory leaks. Some allocated blocks where found without any pointer referencing them. It must be fixed.
- **Indirectly lost**: The blocks are lost because the data structure containing the pointer to them is not reachable. For example, if the first node of a linked list is lost, the access to the other nodes is lost. It must be fixed.
- **Possibly lost**: The program may be leaking memory but some intended behaviours can produce this intentionally. It must be fixed if it is not done intentionally.
- **Still reachable**: This refers to the blocks that have not been freed before the program exits. This is common and not really a problem but can be fixed freeing the blocks adequately before the program exit.

When launching a program with Valgrind, it gives a detailed output of each of the errors found during the execution. At the end, a summary of the execution is displayed, indicating the quantity of each error type and the memory quantity involved on each of them.

For each daemon, the program was executed and the pertinent fixes were applied until both daemons could not have any memory leaks since all the allocated heap block during the execution are freed before it finishes [21], [22].

6.6 Integration with the complementary project

After all the base design and implementation was done, the next step to fully finalize the project is to integrate it with the network of other devices running the same system. The detail of this implementation can be checked in the GitHub repository [23]. This step is not really necessary for this project to work, even if the integration is not set-up. It will still be able to detect the changes, update the root hash and log all of it in the system logs. But the integration offers a extra level of security to verify the trust of all the nodes using the complete system as it constantly validates the root checksums against the values other nodes have stored in them.

6.6.1 Architecture, design and implementation

Similar to what it has been done with the communication with the other daemons, D-Bus will be used to share the information with the network daemon. Every time fsCheck updates the root hash, a message will be sent via D-bus to the daemon managing the network. To do so, a call will be issued to a method exported by that daemon, as represented in the Figure 6.2.

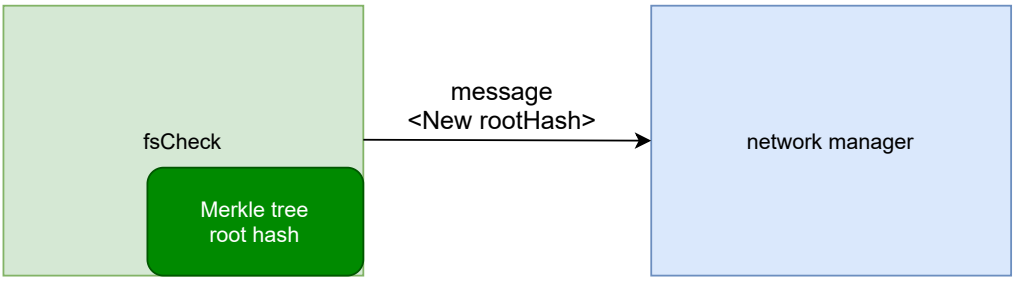

Figure 6.2: Overview of the integration and the communication between daemons.

The implementation is similar to `sd_bus_send` 6.9 used in fsNotifier, but changing the details relative to the service to contact, the object path, method name, etc. In this case, only the hash is sent so the input signature is "s", indicating that a single standard string is sent.

7. CHAPTER

Analysis

After the code base is completed, some analysis should be done to determine how the solution performs on different scenarios and conditions. This chapter will condense all the experimentation that have been done with each of the daemons and the obtained results.

7.1 Experimental setup

Three different directory structures will be used for all the tests. First, a small one with not too much deepness and files not bigger than 10MB. Then, a deeper structure with a similar number of files and directories and similar file sizes. Finally, a worst case scenario, containing the same deeper structure but with larger files up to 200MB. Each experiment will be executed 6 times and the first one will be discarded to reduce any impact that could be generated by a cold run.

The small run has a maximum deepness of 5, with 364 directories and 894 files for a total size of 323MB. For the intermediate one, the deepness increases to 10 with 962 directories and 2088 files with a total size of 906MB. And finally, the worst case scenario, has the same deepness of 10 but contains 832 directories and 1954 files. Even if the total amount of entries is smaller, the total size increases to 9.5GB.

All three structures have been generated using a self-made python tool to randomly create filesystem tree structures. The size of the files is decided with two probability tables, one for the small files and one for the big ones. The number of files and directories in each

layer is also randomly decided based in the maximum deepness, and some parameters that indicate the minimum and maximum number of them. For each step deeper, those values are adjusted to reduce the number of entries in deeper layers.

The experiments will be reproduced using a x86_64 system running inside a virtual machine (VM) and in an ARM based Raspberry PI 3B+ (RPI) in order to test the performance on both architectures. The specification of the systems are the following ones. For the x86 VM:

- CPU: i5-8250U 2 Cores (4 Threads) @ 1.60GHz
- Storage: SATA III 60 GB HDD (5200RPM)
- RAM: 4GB DDR4-2666MHZ
- OS: Ubuntu Server 20.04 (Kernel 5.4.0)

And for the Raspberry:

- CPU: Broadcom BCM2837B0, Cortex-A53 (ARMv8) @ 1.4GHz
- Storage: 32GB Micro SD-Card
- RAM: 1GB LPDDR2-400MHz SDRAM.
- OS: Ubuntu Server 20.04 (Kernel 5.4.0)

7.2 Initialization time

This section groups all the test made to both daemons in order to measure the initialization time required with each of the cases described above.

7.2.1 fsCheck

The daemon fsCheck needs to calculate the hash of the selected part of the filesystem when it initializes. For large sets, this procedure may be slower.

In order to test the initialization time of this specific daemon, the experiment will be repeated with different configurations. This includes using the two different hash algorithms

that have been implemented (SHA-256 and Blake3) and different directory structures. The heavy part of the system is carried away by this, calculating the hashes is the most time consuming action on all the code.

Results

The means of each run for each configuration are represented in the Table 7.1. The elapsed execution time is measured in milliseconds.

	SMALL	INTER	BIG
BLAKE3-RPI	32963.8	46386.6	474170
SHA256-RPI	37598.2	53633.8	492506.4
BLAKE3-VM	327.2	502.4	101200.8
SHA256-VM	3137.2	4442.4	105848.6

Table 7.1: Initialization time (in milliseconds) of fsCheck in the VM and RPI with both algorithms.

To appreciate better the speed difference and results between test sizes and algorithms, the data is represented in the chart of the Figure 7.1.

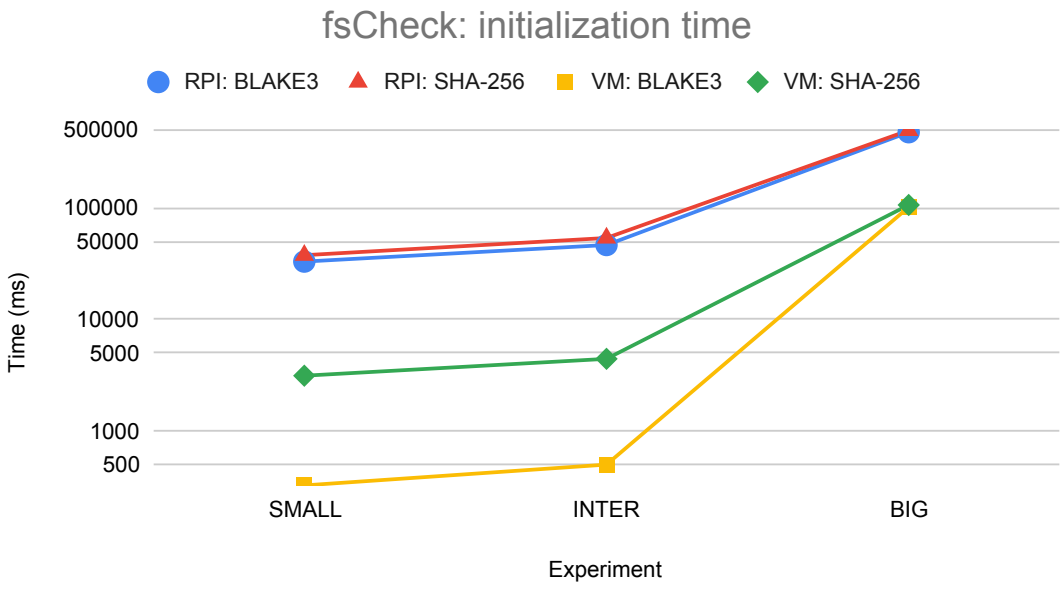

Figure 7.1: Time necessary to initialize fsCheck in various configurations and systems.

Looking to the execution time of both, the VM and the RPI, Blake3 proofs to be faster for all the test cases. But more importantly, the gathered data surfaces a possible bottleneck.

If the storage speed of the system can keep up, Blake3 will be much faster but if the storage struggles to keep supplying enough data, the speed of both algorithms is similar. This occurs faster on the RPI because the raw speed of the SD Card is much smaller so the bottleneck is reached faster. For the VM, this only occurs for the big test case. For the small and intermediate runs, in contrast, the speed gain of Blake3 over SHA-256 is almost 9x.

7.2.2 fsNotifier

As explained earlier, this second daemon generates an inotify watch for each directory in the filesystem section to control. This can implicate that large filesystems can slow down the initialization time or even overflow the limit of watches. For this daemon, the number of files and the size of them is irrelevant since it only watches directories. Therefore, the initialization time of this daemon depends on the number of directories and the deepness of the structure since it needs to travel all of it looking for all the directories.

Results

After testing this daemon on both systems, the results are similar. In both cases, the small configuration is faster and the slowest one is the intermediate one. This result is totally expected, as explained before, the number of directories and the deepness makes the difference and the intermediate test case has more directories than the big one, while keeping the same deepness of 10. In other words, the number of directories will establish the time necessary to generate all the watches. Is important to take into account that, even the slowest run is way faster than fsCheck.

The Table 7.2 contains the results of the experiments of both systems and the Figure 7.2 contains the graphical representation of it, to facilitate the comparison of experiments. The differences in performance between platforms varies the amount of time necessary but the proportions are maintained for the tests.

	SMALL	INTER	BIG
VM	9066.4	21705.2	19701.4
RPI	95681.4	257752.6	216867

Table 7.2: Initialization time (in microseconds) of fsNotifier in the VM and RPI.

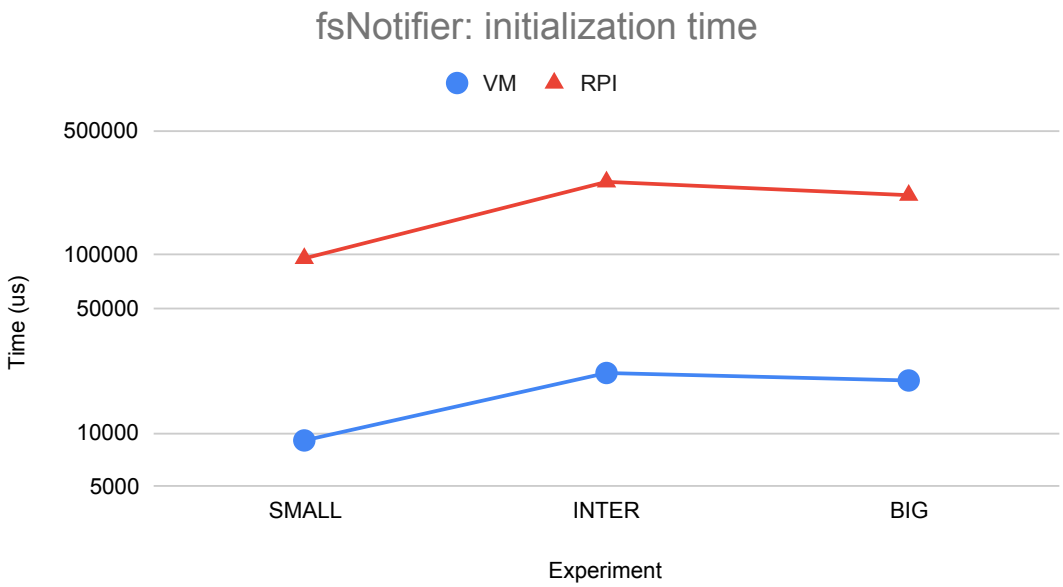

Figure 7.2: Time necessary to initialize fsNotifier in various configurations and systems.

Now, the API used for this has a limit that is set in the kernel to cap out at some number of watches. This limit can be check by looking into the file `/proc/sys/fs/inotify/max_user_watches`. For this tests, the maximum amount of them is not reached. The intermediate test has just slightly over 900 directories and the default limit for these on the utilized Ubuntu systems is 8196. For larger file system, this limit can be increased by changing the configuration in sysctl.

7.3 Memory usage

This section groups all the tests made to both daemons in order to measure the memory usage after the initialization is completed for each of the cases already mentioned and utilized in the initialization time experiment. For this, the memory usage will be obtained from procfs inside of the program itself after the initialization is completed. Although the values of resident, shared and private memory usage will be obtained, only the resident memory values will be analyzed. On Linux, resident memory represents the sum of both, private and shared memory, of the program.

7.3.1 fsCheck

With the same cases, the memory usage of the daemon when the initialization has been completed will be analyzed. For each entry in the file structure, a node is generated. The node contains the relevant information indicating the path, the hash and the children it, has with pointers to those nodes. Since the hash has always the same length, the size of the files is irrelevant and the only important parameter to take into consideration should be the number of entries. The aim of this experiment is to determine whether or not the generated structure occupies too much memory.

Results

The gathered data confirms the hypothesis just explained. The size of the files is irrelevant for the total memory usage of the daemon during the execution. For this very reason, the execution with the most memory usage is the intermediate one. The Table 7.3 contains the results of the test performed for fsCheck that have been graphically represented in the Figure 7.3.

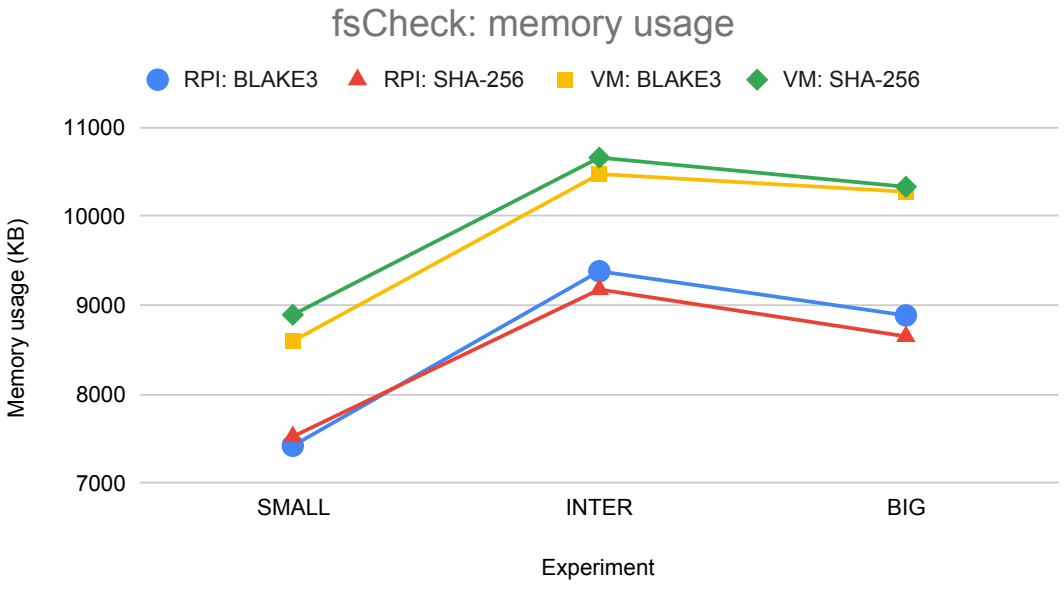

Figure 7.3: Memory usage of fsCheck after the initialization is completed.

	SMALL	INTER	BIG
BLAKE3-VM	8596	10474.4	10275.2
SHA256-VM	8892	10659.2	10331.2
BLAKE3-RPI	7419.2	9381.6	8884.8
SHA256-RPI	7522.4	9176.8	8648.8

Table 7.3: Memory usage (in kilobytes) after initialization of fsCheck on the VM and RPI with both algorithms.

Even if the total size is almost 10 times smaller, the number of entries is bigger on that intermediate scenario. The size of the hash string could be relevant for the total memory usage of the daemon, for instance, if the generated hash output for each entry has more bits, the total amount of memory necessary to store the structure increases. But in this case, both algorithms use 256 bits for it, so the amount of memory taken by the hash string is the same.

It is remarkable that the runs on ARM use significantly less memory – about 10 to 20 percent less – compared to the x86 ones.

7.3.2 fsNotifier

Replicating the test done for fsCheck, the memory usage after the initialization of fsNotifier is completed will be measured. And, in the same way, the only relevant parameter is the number of directories in which the watches should be deployed.

Results

Confirming the hypothesis and similar to what happened with fsCheck, the memory usage depends on the total number of directories in the tree. For that reason, the intermediate run consumes more memory than the other two. Comparing the two systems and architectures, the ARM based one consumes about ten to twenty percent less memory consistently. The Table 7.4 contains the results of the test and have been graphically represented in the Figure 7.4.

	SMALL	INTER	BIG
VM	5306.4	6210.4	5935.2
RPI	4276.8	5272	5026.4

Table 7.4: Memory usage (in kilobytes) after initialization of fsNotifier on the VM and RPI.

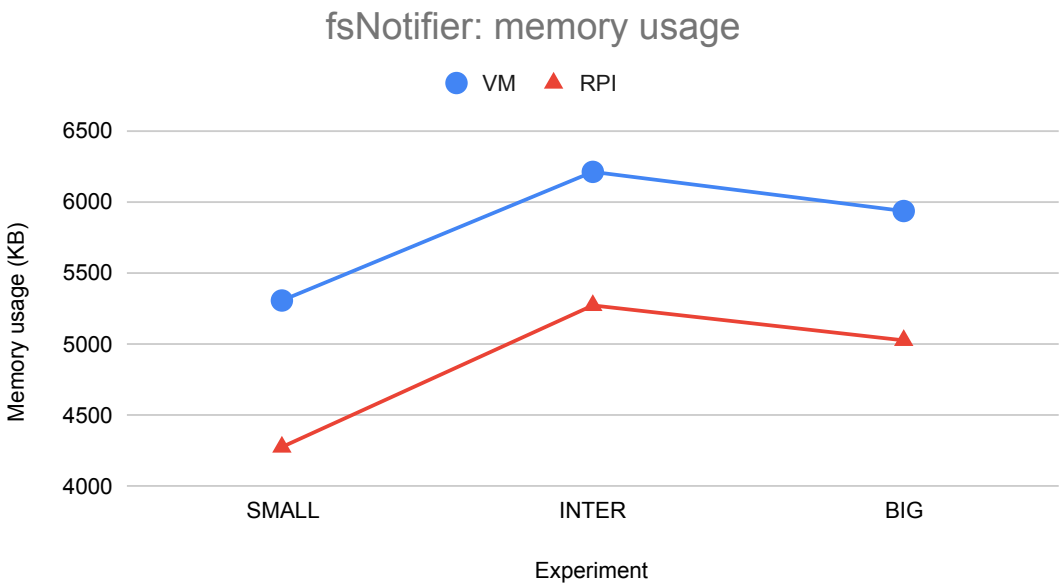

Figure 7.4: Memory usage of fsNotifier after the initialization is completed.

7.4 Response time

The last experiment will analyze the response time of the whole system. In other words, the time required by the system to detect, process and update the tree and submit the new root hash since the change was made. To to it, a third program will generate a change in the selected tree and log the time. Then, it will listen to the responses of fsCheck and will log the time again when the updated hash is received. Using both timestamps, the time required to complete the described process will be measured.

The method just described has some problems to determine the exact start time; since the program itself is doing the modification, the time can only be recorded just before or immediately after the change is done. To compensate for that, both values will be recorded and the mean time will be calculated afterwards.

Following the same structure used for the benchmarks of fsCheck, each system will perform 6 test for each directory tree size and for each hash algorithm. Every test will contain 3 modifications of existing files in different parts of the tree; one in the deepest node, one in an intermediate position and one in the root. With all these values, for each system and configuration the mean response time will be calculated.

Since the mean response time does not represent the whole picture, the maximum and

minimum response times will also be extracted and analysed. In this kind of systems, the worst case scenario is more relevant than the mean time since it represent the real time that should be taken into consideration in order to take design decisions.

Results

The gathered data is represented in the Table 7.5 and contains the results of both systems with both hash algorithms. To represent those results visually, the data has been separated by machine to facilitate the interpretation. The Figure 7.5 contains the results of the VM and the Figure 7.6 contains the results of the test performed on the Raspberry.

		VM			RPI		
		Max	Mean	Min	Max	Mean	Min
SHA-256	Small	2271	1896	1531	9300	7920	6694
	Inter	2820	2093	1380	16914	11344	8236
	Big	3346	2299	1479	23043	12113	8738
Blake3	Small	2282	1881	1460	9256	8026	6531
	Inter	2690	2150	1371	18094	11743	8920
	Big	3521	2201	1368	16532	11505	8347

Table 7.5: Response time (in microseconds) of the system.

The tests show that the response time does not vary too much between runs with different configurations for each system, even if the hash algorithm used is different. There are some differences in the maximum and minimum response times though. This result was expected. If the change is generated near the root, the number of hashes to update is smaller and therefore, the time required to update the tree is smaller too.

With the maximum, minimum and mean values, all the cases are represented. The changes near the root, will obtain results that match with the minimum values. But, in contrast, changes made deeper in the tree will obtain values that will between the mean and maximum response time.

The worst configuration depends on a lot of variables, which includes the size of the structure, the used algorithm and so on. First, the used algorithm does not suppose a major factor on what to expect overall but for the raspberry set-up, using SHA-256 could be a problem if the filesystem is big enough. For the VM, the opposite happens but in this case, the difference between the two algorithms is so small on that single case that it will not represent an appreciable performance hit.

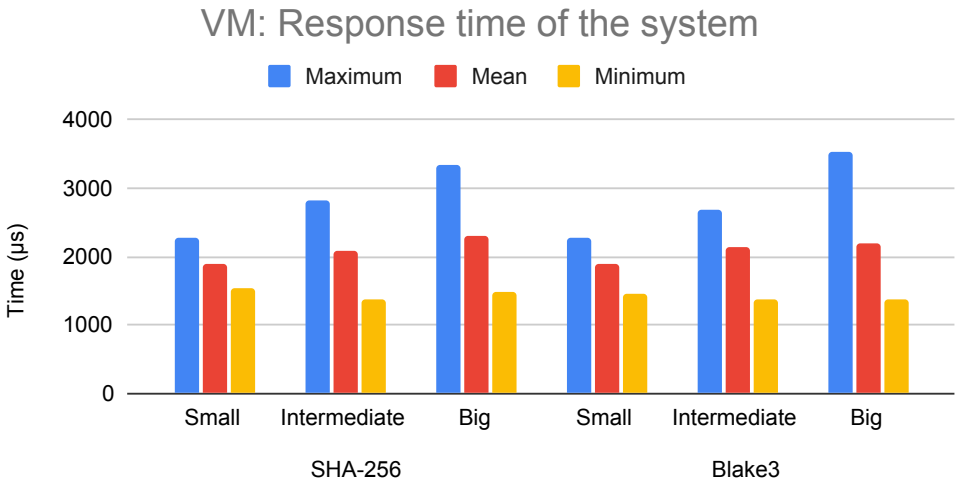

Figure 7.5: Maximum, mean and minimum response time (in microseconds) of the whole system on the VM.

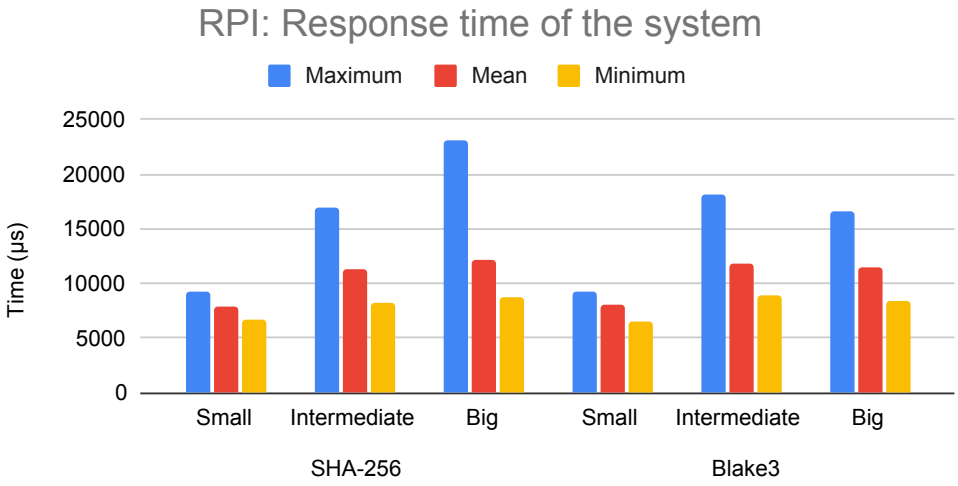

Figure 7.6: Maximum, mean and minimum response time (in microseconds) of the whole system on the RPI.

To sum up, if the system is fast enough, the used algorithm and directory size do not impact in the response time. But on low end devices, such as the raspberry, using smaller directory structures and Blake3 is a better option to keep the response time as low as possible.

8. CHAPTER

Conclusions and future work

After finishing the implementation and analysis of the project, some interesting conclusions can be explained. This chapter will cover them along with the future work that could be done in this research field.

First of all, during the process of development, there have been some problems due to the shortfall in documentation of some key aspects of the solution. For instance, the documentation available of D-Bus is really small, incomplete or too general. As it has already been covered before, there are a lot of implementations of the specification but not all of them fully implement all of it or they do it specifically to meet the needs of certain projects. This generates a lot of sparse documentation that only covers some aspects of the complete specification.

The lack of documentation also expands to the notification systems such as dnotify, inotify and fanotify. Since these APIs are not used by a lot of projects, the available documentation is limited to the Linux Manual Pages and not a lot more.

All of this has held back the development process in some stages of the project but it has been useful to develop some strategies and mechanisms to handle these situations as well as improving the abilities to read, comprehend and extract useful information out of the source code.

Secondly, speaking of the contents of the project, the design and implementation made for this work is only a proof of concept. It demonstrates that Merkle tree data structures can be utilized for file system integrity jobs. Moreover, the efficiency of this structure after the initialization is completed is extremely high since only one branch needs to be

recalculated when a modification occurs, without the necessity of checking and computing all the entries each time a there is one.

It also surfaces the limits of the utilized notification mechanisms. There is no recursive directory watch system in any of the three options available on the Linux kernel for selective directories. In addition, not all the possible events can be handled properly; rename events generate some problems, using certain programs like vim or nano to edit files can be a problem if individual files are watched since it generates a new file instead of actually modifying the existing one, etc.

This work opens some paths to further develop file integrity monitoring systems on Linux systems. The goal of this project is to notify when a change occurs but it does not prevent it from happening. For this matter, fanotify offers some options to block those actions and manually permit the ones that fulfill the defined security policies. Another improvement that could be done is developing a multi threaded version to improve the performance when handling multiple and successive events faster or reducing the initialization time.

Taking it a step further, some work can be done researching about the notification mechanisms of the Linux kernel; the implementation, the limits, how to improve them by fixing the limitations they have right now or proposing a new or complementary API to extend the capabilities available nowadays.

Finally, on another level, the presented solution can be ported or implemented in other systems or architectures, such as Microsoft Windows or Apple Mac OS. Those platforms implement their own notification mechanisms too that could be utilized to monitor the filesystem.

Bibliography

[1] S. Dhumwad, M. Sukhadeve, C. Naik, M. K.N., and S. Prabhu, "A peer to peer money transfer using sha256 and merkle tree," in *2017 23RD Annual International Conference in Advanced Computing and Communications (ADCOM)*, pp. 40–43, 2017.

[2] "Data integrity in zfs using merkle trees." https://blogs.oracle.com/bonwick/zfs-end-to-end-data-integrity.

[3] S. Rothwell, "Linux directory notification - dnotify." https://www.kernel.org/doc/Documentation/filesystems/dnotify.txt.

[4] "Linux manual page - inotify." https://man7.org/linux/man-pages/man7/inotify.7.html.

[5] "Linux manual page - fanotify." https://man7.org/linux/man-pages/man7/fanotify.7.html.

[6] G. Kroah-Hartman, "Driving me nuts - things you never should do in the kernel," *Linux Jounal*, 2005.

[7] "Linux manual page - daemon (7)." https://www.man7.org/linux/man-pages/man7/daemon.7.html.

[8] "Freedesktop software wiki - systemd." https://freedesktop.org/wiki/Software/systemd/.

[9] "Understanding systemd units and unit files." https://www.digitalocean.com/community/tutorials/understanding-systemd-units-and-unit-files.

[10] J. S. Gray, *Interprocess Communications in Linux®: The Nooks & Crannies*. Prentice Hall PTR, 2003.

[11] "Freedesktop software wiki - d-bus." https://www.freedesktop.org/wiki/Software/dbus/.

[12] "Freedesktop software wiki - d-bus bindings." https://www.freedesktop.org/wiki/Software/DBusBindings/.

[13] J. Kara, "Fanotify chages for v5.1-rc1." https://lkml.org/lkml/2019/3/1/400.

[14] Z. E. Rasjid, B. Soewito, G. Witjaksono, and E. Abdurachman, "A review of collisions in cryptographic hash function used in digital forensic tools," *Procedia Computer Science*, vol. 116, pp. 381–392, 2017. Discovery and innovation of computer science technology in artificial intelligence era: The 2nd International Conference on Computer Science and Computational Intelligence (ICCSCI 2017).

[15] J. Guilford, K. Yap, and V. Gopal, "Fast sha-256 implementations on intel ® architecture processors," 2012.

[16] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "Blake3 one function, fast everywhere," 2020.

[17] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87* (C. Pomerance, ed.), (Berlin, Heidelberg), pp. 369–378, Springer Berlin Heidelberg, 1988.

[18] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[19] A. Herrero Perez de Albeniz, "fscheckdaemon." https://github.com/Aritzherrero4/TFG_fsCheckDaemon, 2021.

[20] A. Herrero Perez de Albeniz, "fsnotifierdaemon." https://github.com/Aritzherrero4/TFG_fsNotifierDaemon, 2021.

[21] A. Herrero Perez de Albeniz, "fscheckdaemon - fixing memory leaks." https://github.com/Aritzherrero4/TFG_fsCheckDaemon/commit/0bc09550162a0ac05b99b8295d67c2fa392d20f7, 2021.

[22] A. Herrero Perez de Albeniz, "fsnotifierdaemon - fixing memory leaks." https://github.com/Aritzherrero4/TFG_fsNotifierDaemon/commit/68363efb529fc2c952eb50a8c5551d2f8de2439e, 2021.

[23] A. Belenguer, "Distributed integrity monitoring system based on blockchain technology." https://github.com/AitorB16/Blockchain-based-integrity-monitoring-distributed-net-protocol, 2021.