UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

School of Computer Engineering

Universitat Politècnica de València

N-Modular-Redundant Architecture for Software Applications.

Voter/Monitor implementation.

Bachelor Final Project

Bachelor's Degree in Computer Engineering

Author: Pau Sastre Miguel

Supervisor: Ismael Ripoll

July 2014

Acknowledgments

Thanks to Ismael Ripoll for his valuable lessons and suggestions for improvement. Thanks to Ana Herraiz for proofreading the final project. Thanks to all the people who made it possible.

Abstract

In this project, the novel diversification technique called DRITAE, which stands for Diversified Replication Architecture Through Architecture Emulation, has been implemented.

The technique consists of compiling the source code application using several cross-compilation suites to generate different processor executable binaries (variants). Then, each variant is executed by a fast processor emulator (on the same host) and monitored at the system-call-level interface.

This form of binary diversification preserves the semantic behavior on each variant. There are minor discrepancies in the sequence of system calls generated by each replica (due to emulator or target library implementations), which are filtered out by the monitor.

Paired with this diversification technique, a monitor software with the purpose of simultaneously executing all the generated variants and controlling the processes during its entire life, acting as a N-modular redundant system, has been developed.

The results obtained by this approach to a N-modular redundant system reveals some interesting information:

The results obtained when evaluating this DRITAE architecture and the N-modular-redundant monitor system show a very satisfactory response to failure or misbehavior detection, turning the most common and widespread program vulnerabilities hardly exploitable, if possible at all. Performance wise, the system behaves reasonably well, as there is an overload derived from simultaneously executing multiple variations of the same program, but CPU parallelism can be used to mitigate this overhead.

The validation phase, by having to generate and take advantage of some of the most common vulnerabilities present in software, has proven itself useful in providing the authors with deeper and broader understanding on how the execution flow of a program works, how its memory is structured, which are the mechanisms used by an operating system to try to defend itself from abused processes, and how those mechanisms can be effectively bypassed.

Keywords: redundancy, security, virtualization, failure tolerancy, failure detection, failure recovery, n-variant system, diversification, software vulnerability, UNIX, Linux, process tracing, syscall, operative system, monitorization, testing

Contents

1 Introduction **3**

2 Model / Architecture - Nvariant System **7**

2.1 Architecture Overview 7

2.2 Diversification 7

2.2.1 Granularity 9

2.3 Variants execution 9

2.4 Variants monitoring 9

3 Implementation **11**

3.1 Implementation design analysis 11

3.2 Structure of the monitor 12

3.3 Initialization 13

3.3.1 Monitor-Variant communication setup 14

3.3.2 Launch and initial synchronisation 15

3.4 Step-lock Execution 16

3.4.1 Syscall capture 17

3.4.2 Types of syscalls 17

3.4.3 Comparing Syscalls 24

3.4.4 Moving data 29

3.5 Recovery actions 30

4 Conclusions 33

1 Introduction

Historically, malicious binary code execution has been the most dangerous type of vulnerability due to the large number of occurrences and its high impact. During the last decades, a large number of solutions have been developed and implemented to address the problem. Basically, there are two groups of solutions:

Prevention: help the programmer to write correct code

Mitigation: block or prevent the error to become a security failure on already incorrect applications

New languages such as Ada, Java and C# jointly with robust versions of the libraries and APIS try to solve the root of the problem by helping/forcing the programmer to write correct code. But there are still many applications [TIO, 2012] that are written in C and C++, which have a weak memory model and are recognized error-prone languages.

Buffer overflow, which is the main cause of malicious code execution, has been ranked as the most dangerous software error during several decades. Thanks to the efforts done in many areas to tackle the problem (static analysis tools, robust run-time libraries, buffer overflow detection mechanisms like the "canary", non-executable data areas, etc.) and also due to the rise of other kinds of vulnerabilities related to web programming, buffer overflow is now the third most dangerous kind of error according to CWE/SANS [Mitre, 2011]. It is still an open problem which deserves further research to find better solutions.

Diversification is a widely used strategy to build effective defenses against binary execution injection. The basic idea is to build different versions of the application, known as *variants*, in a way that each instance of the application behaves according to the specification but responds differently to faults. Diversification techniques can be categorized into "manual techniques" and "automated techniques". Manual techniques have a high economic cost, also since humans tend to solve the same problem using similar solutions [Knight and Leveson, 1986], the resulting variants may not be as different as desired. Automatic variant generation is an active research area which has seen many practical advances in recent years.

Diversification is a basic technique that can be used in different ways or combined with others. Laprie et al. [Laprie et al., 1990] defined a *diversified design* as a system (application) with at least two variants plus a decider, which monitors the results of the variant execution, given consistent initial conditions and inputs. The variants are different systems produced from a common service specification. The decider executes the variants concurrently and periodically (decision points) compares the state of variants to determine the safety state of the whole system. Each variant is different with respect to the rest of variant. This form of diversification is the natural extension of the well known TRM (triple redundant modular) mechanism to the software elements. Recently, in order to avoid confusion with other forms of diversification (those without replication and comparison), many authors refer to this technique as: multi-variant execution environment (MVEE) [Salamat et al., 2011]; diversified replication [Huang et al., 2010] or N-Variant [Cox et al., 2006].

Another form of diversification used to prevent malicious attacks is to make the application to appear differently to the attacker at every attack attempt. The variants are build so that its proper execution depends on a property that is randomly selected at start time and is not known by the attacker. The protection mechanism is invalidated if the attacker manages to obtain the secret. The canary (stack protector) and address space layout randomization (ASLR) are examples of this form of diversification which are commonly used in most current systems. This kind of diversity makes more difficult to exploit the existing vulnerabilities and limits the ability to propagate, also the overhead introduced is low or even zero as with the ASLR.

The compiler-generated diversity presented in [Jackson et al., 2011] another form of diversification. The current software market structure (software distributed to end-users through the web using an application store like Android Marketplace or the App Store) allows to build and distribute a different variant to each customer, assuming that the compiler/build system is able to produce a large number of sufficiently different variants. In this case, the number of vulnerable systems, or systems that can be compromised automatically is reduced drastically. This technique is very effective when the goal of the attacker is to maximize the number of compromised systems.

The work exposed in this paper focuses on the automatic N-variant generation using available software tools (COTS) to build a strong multi-variant execution framework.

During the past decade, virtualization has been one of the most active fields, resulting in new technical solutions as well as high quality use products.

Platform virtualization consists of creating a virtual machine (*guest*) that acts as a real computer on top (using the resources) of a real computer (*host*). The virtual machine monitor (VMM) is the software element that recreates the virtual computer by using the

resources and services of the real computer. Typically, the guest system is very close or even the same that the host system, that is, a PC i386 board can be virtualized (be the guest) on an AMD64 host. Most VMMs exploit the fact that most non-privileged instructions of the guest system can be directly executed by the host computer, and only a few privileged or sensible instructions have to be virtualized. This way, it is possible to execute most of the guest code at the same speed as in the host. Modern processors provide some kind of facilities to intercept (and virtualize) those instructions or hardware resources that cannot be directly used by the guest as privileged instructions, virtual memory management and IO access.

A virtualizer that is able to run a guest system with a different architecture from the host computer is called an *emulator*. Many virtualizers use a combination of hardware support and emulation to achieve good performance and compatibility with a broad range of the processor family, implementing in the VMM the extended instructions not provided by the host processor. There are basically two emulation techniques: code interpretation and binary translation. The code interpreting technique is based on a fetch-decode loop. It is simple to implement but not very efficient, since each guest instruction has to be decoded (several jumps/conditions) and perform the specified actions, taking into account the side effects (CPU status bits). Binary translation consists of translating from the guest machine code into host binary code; basically it can be seen as a compiler but rather than the source code, that is a high level language, it is the binary code of some CPU and the result of the compilation is the binary code of the host CPU, which is then directly executed by the host. A well known emulator is the Java virtual machine which interprets the Java byte-code instruction set.

In the recent years, and in particular during the development of the Qemu (Quick EMUlator) by Fabrice Bellard [Bellard, 2005] the binary translation has been largely improved. It implements dynamic binary translation using an internal compiler. The original code blocks (list of instructions terminated by a branch instruction) are dynamically translated into custom intermediate code and then compiled and optimized into the target code using a custom compiler. Since the translation is done at block level many optimization can be done: register allocation or lazy conditional code calculations among others. The use of intermediate code greatly simplifies the addition of new guest architectures while getting the benefit of the optimized host code generation. Qemu is the *de facto* standard for emulation, for example it is used by Google in the development environment for Android applications.

Qemu has two operation modes: system emulation and user-mode emulation. In system mode, Qemu virtualizes the ISA layer (Instruction Set Architecture). In this mode, Qemu is able to execute a complete operating system (BIOS, operating system and applications). In user-mode emulation, Qemu executes a single process that was compiler for a supported

guest CPU, assuming that both guest and host operating systems are the same. For example, it is possible to run an executable compiled for a Linux ARM system in a Linux i386 computer. This mode is only available for Linux and Mac OS X operating systems. An operating system emulator is added to Qemu to support this operation mode, which acts as a proxy to the system calls interface. The solution proposed in this paper employs the user-mode of Qemu as the execution platform for the variants.

This project has been structured in two differentiated parts: the first part, this document which explains the implementation of the N-variant system itself, while the second part, titled "N-Modular-Redundant Architecture for Software Applications - Testing/Validation" and written by Jordi Chulia Benlloch, exposes the validation and testing processes that the N-variant system has gone through. The authors recommend to treat both documents as a whole.

This paper has been organized as follows: Section 2 briefly explains the implementation model of N-variant system. Section 3 explains the design decisions made and goes through a series of details as well as the key aspects of the implementation. The last section sums up the conclusions obtained from this project and offers an insight of possible future lines of work.

2 Model / Architecture - Nvariant System

This section deals with how the N-modular-redundant system works, In which moment in the execution of a process the system starts working; how the diversification is achieved; in which way the concurrency occurs; and by what means are all diversified variants controlled.

- Architecture Overview
- Diversification
- Variants execution
- Variants monitoring

2.1 Architecture Overview

As shown by the Figure 2.1, the architecture of the NVariant system is mainly divided in two parts. The first one is the off-line, where different variants are generated from the source code. The second part is the on-line, as it happens during execution. In this part, each variant is executed by means of different dedicated virtualizers, and a monitor process checks the consistency of the variants execution acting as a middleman between the Operating system and the virtualizers.

2.2 Diversification

Diversification is achieved by taking advantage of the differences in processor architectures: The GCC compiler suite has been used in order to generate a wide range of semantically equivalent variants from a single source code. Moreover, this cross-architecture compilation is set to link statically all libraries needed, which, due to differences in each architecture, may

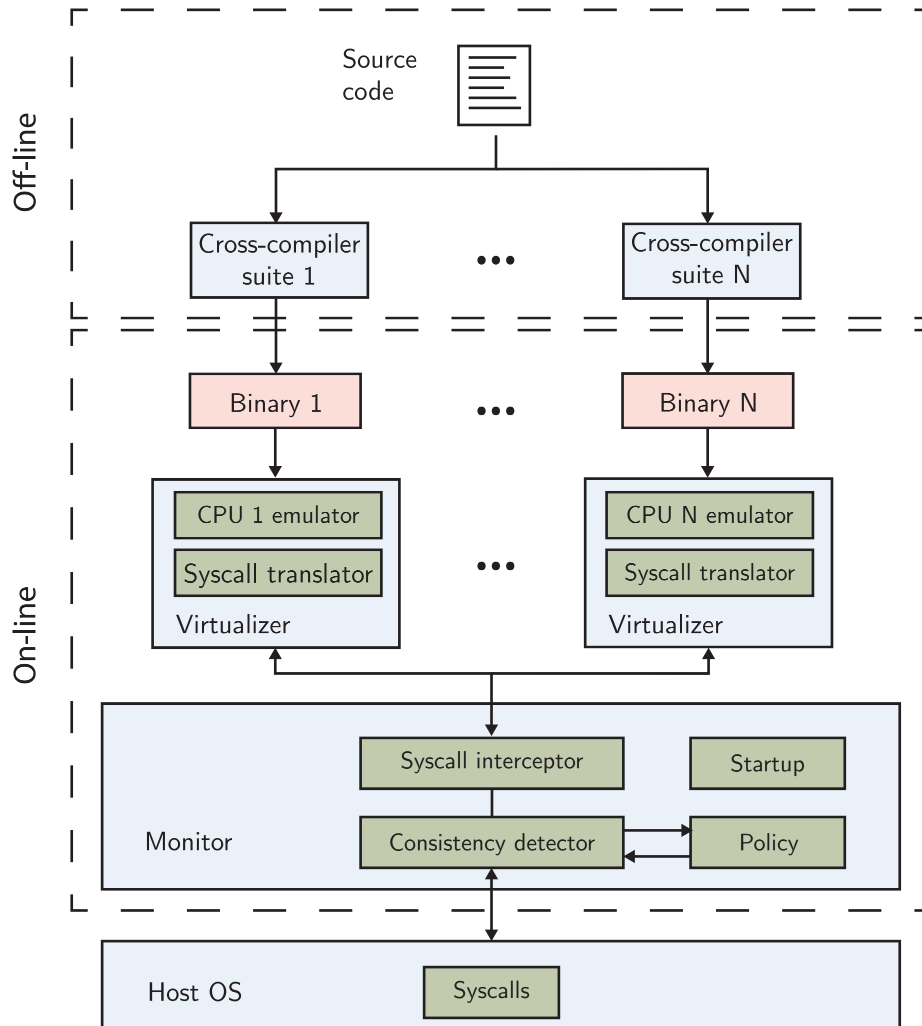

Figure 2.1: N-variant Work flow.

have different implementations. This allows for significant diversification (different stack distributions, endianness, instruction set, etc) while keeping the system simple.

All this takes place off-line. Besides, the ASLR mechanism provided by the operating system introduces a new layer of diversity on-line at the very beginning of all the execution process by simply loading the executables.

2.2.1 Granularity

There are a lot of different ways to obtain diversification. Depending on the amount of divergence introduced in the execution of the different variants, it is more difficult that all the instances fail in the same way, or compromise all the instances at the same time. By compiling the source code, the instructions of each variant are completely different but the semantics stays the same. Since the type and the flow of syscalls are the same in every variant, the monitor can synchronize and check the system each time all the variants try to perform a syscall. By using this approach the monitor checks every variant when a program wants to interact with the operating system. In other words, the monitor will check every compromising action the program wants to perform.

2.3 Variants execution

The diversification mechanism exposed above produces highly differentiated variants with a little effort, but it has a disadvantage. The difficulties arise when the variants cannot be executed directly in the same processor since each variant is an executable for a different architecture.

To solve this without using any kind of special or dedicated hardware, the processor emulator Qemu has been used. Each variant is executed by means of its corresponding Qemu architecture emulator, which emulates the foreign architectures of the single machine where all is executed and translates the different syscalls to the ones of the native architecture.

2.4 Variants monitoring

The N-variant system follows the classic replication architecture: several variants of the same program and one monitor.

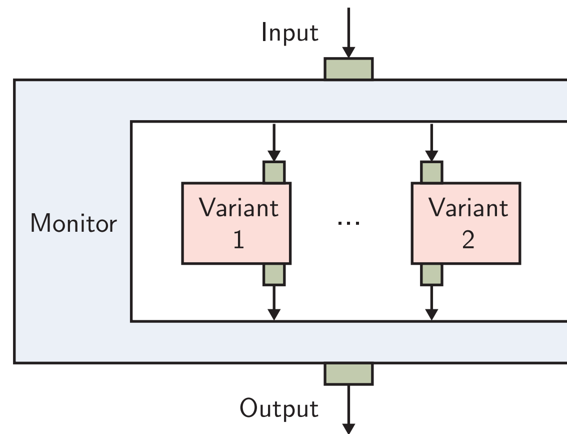

Figure 2.2: The monitor acts as a wrapper to the variants.

The number of variants used is transparent to the system. As it can be observed in the Figure 2.2, the input of the monitor is replicated through all the variants and the output of each variant is then caught by the monitor again. From the point of view of the user, it is also transparent as the user actually interacts with the monitor as if it were the original program.

The monitor emulates different processor architectures in order to obtain the different variants. In order to conduct this emulation, the monitor executes the Qemu emulator for each one of those architectures. All the variants run on the same operating system (Linux in this prototype), and the monitor is in charge of launching all the variants and synchronizing them.

Due to the need of synchronization, an entry point (the first synchronization point the monitor does) is added at the very beginning of each. The the purpose this entry point, is to skip the initialization of each Qemu instance and have an actual starting point common to each variant. After this point, the monitor starts checking each syscall the variants want to perform. In order to conduct the checking, the monitor uses a pipe and the `ptrace()` syscall for communicating with the variants.

The monitor is in charge of detecting when a program is compromised. In order to detect this, every instance of the program is traced and synced with the other instances at some point to compare the state of the program.

3 Implementation

This chapter presents the implementation of the prove of concept of the DRITAE architecture. The chapter is organized as follows: the main design constraints are presented and discussed; those requirements are then used to design a solution to DRITAE architecture; the specific implementation issues are presented later; and finally, the different recovery actions that the system could perform on different situations are discussed.

- Implementation design analysis
- Structure of the monitor
- Initialization
- Step-lock Execution
- Recovery actions

3.1 Implementation design analysis

Several design alternatives have been explored for the implementation of the monitor. The most interesting solutions were:

1. Implementing the monitor in the kernel of the operating system.
2. Implementing the monitor as a standard process, and use the tracing facilities to monitor and control the variants.

The final design solution was chosen according to the following high level requirements:

Portability: Although initially developed for the Linux system, the implementation shall be portable to other UNIX systems.

Development complexity/difficulty: There are many factors that make kernel programming more complex compared with standard application coding. Just to mention a few: non-standard APIs, recompile large amount of code, the code to be modified is spread along multiple files, etc.

Usability: It is simpler and easier to use DRITAE architecture if it can be launched as a simple process (or a set of processes) than if a new kernel has to be installed.

Efficiency: The monitor shall introduce the minimum possible overhead, specially considering the non-negligible overhead introduced by the emulation process.

Obviously, the most efficient implementation is the one done at the kernel level where all information is directly available. In kernel space, it is possible to access to any process memory and intercept the system calls right when they are invoked at almost zero cost. But the rest of the requirements do not match. Specially, the cost of programming this kind of software, which cannot be implemented as a simple kernel loadable module and is far more complex both in time (many kernel re-compilations and reboots) and complexity (changing the implementation of the system calls along many different kernel files).

The authors consider this project as a PoC (Proof of Concept) of the DRITAE architecture. The resulting monitor will serve as the workbench of future developments to experiment with recovery policies and new diversification mechanisms. Therefore, a compact and well-structured implementation is also a key aspect to be taken into account.

The selected implementation design is based on a standard process and the `ptrace()` facility to control the execution of the variants, which has been complemented with specially designed hacks (which do not require kernel modifications) to speed-up time-critical operations on the variants.

The resulting monitor overhead is shown in the Figure 3.1. This overhead is part of an average test, and it will change depending on the type of program executed. On the one hand, if the program interacts a lot of times with the operating system, the monitor overhead will increase. On the other hand, if the process is computationally heavy, the monitor overhead will decrease.

3.2 Structure of the monitor

The monitor has the following logical blocks:

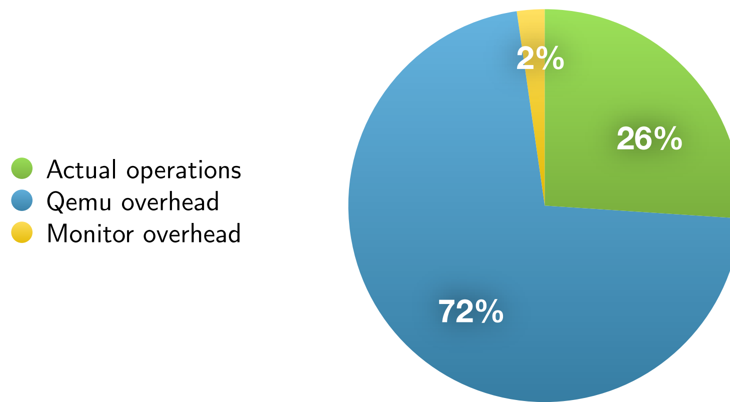

Figure 3.1: Global system overhead.

Initialization: The variants are launched and synchronized by the monitor. That is, the monitor is executing the emulator for every architecture and every emulator is executing the binary compiled for its specific architecture. The monitor also creates the communication channels.

Step-lock execution: Every variant executes the binary code from a synchronization point until reaching another synchronization point. At every synchronization point, the monitor checks the current state of the variants and if the system has been compromised.

Recovery actions: If the system has been compromised, it applies a series of checkings and tries to determine which variant has been compromised and tries to overcome the situation by killing the variant who has been compromised.

Each block is detailed in the following sections.

3.3 Initialization

Before the monitor is able to run all the variants and communicate with them, the monitor has to initialize all the system. In other words, the monitor has to initialize all the communication channels and launch all the Qemu instances as well as reaching a common synchronization point.

3.3.1 Monitor-Variant communication setup

The interaction between the variants and the monitor is one of the key factors for the performance of the system because the monitor is interacting continuously with the variants. All this interaction occurs at two well defined places:

1. System call interception. The monitor is informed by the kernel that an event related to a system call has happened. In this case, the interface is controlled and clearly defined by the Linux kernel.
2. When large amounts of data have to be moved between the monitor and the variant, the pipes are used. In this case, the size of the data or the location of the buffers are given by the variant. Since the safety model used considers that the variants may execute malicious code, this interface must be handled with special care. A malicious attacker may try to access the monitor through this channel.

In the case of the first type of interaction, the monitor uses two key system calls. The first one is the `wait()` syscall, and the monitor uses this syscall as a notification mechanism to detect when a syscall is about to be performed and when is already performed. The second one is the `ptrace()` syscall, it is a very powerful syscall that allows the monitor to be notified by the kernel using the `wait()` syscall, but the capabilities of this syscall are bigger. The reason is that it also allows the monitor to communicate with the process by interacting with processes in different ways, like getting and setting the registers or even memory from the process.

For the second type of interaction, the monitor needs to setup a way to move big amounts of data from the monitor to the variant and vice versa. As the monitor wants to access memory in the variant address space, it cannot copy directly the memory, therefore pipes are employed to make a communication channel. The monitor creates one pipe to communicate in both directions. With this pipe, the monitor is able to communicate with all of its variants. These variants do not know that they have those file descriptors open, but the monitor is in charge of specifying the syscall which the variants have to call to in order to communicate with the monitor by using the `ptrace()` syscall.

Those pipes are created before starting the variants, so when the monitor is forked in order to create the variants, the file descriptors of the pipes are inherited by them.

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

Listing 3.1: Code for allowing the monitor to trace the variant.

```
ptrace(PTRACE_SETOPTIONS, pid, NULL, PTRACE_O_TRACECLONE | PTRACE_O_TRACEFORK );
```

Listing 3.2: Code for set the monitor to trace the forked variants.

3.3.2 Launch and initial synchronisation

After setting up the communication mechanism, the monitor is ready to create and synchronize the variants. In order to perform this task, the monitor goes through the following steps:

1. Forking the monitor.
2. Executing Qemu in the forked process.
3. Executing the variant until the synchronization point is reached.
4. Setting up the variant in the monitor and in the variant itself.

First of all, the monitor is forked. After that, the new processes are enabled to be traced using `ptrace()`, which is set to stop at every syscall. In the Listing 3.1 the `ptrace()` function that perform this task is shown. The monitor also needs to execute `ptrace()` for setting some options, like tracing a forked process. In the Listing 3.2 this `ptrace()` call for setting the options is shown.

After that, Qemu is executed for a given architecture in the forked process.

Before the code of each variant starts being emulated by Qemu, each Qemu instance is initialized, sets its environment up, and in general performs any initialization task it needs, like loading different libraries.

As the Qemu instances are different (the initialization tasks may differ depending on the architecture), the monitor shall not compare the execution of Qemu during initialization, but only once the code of the application has started to execute.

To solve this problem, two different alternatives have been considered and analyzed:

```
struct process_info {
    pid_t pid;
    proc_status_t status;
    arch_t arch;
    syscall_info_t syscall_info;
    pipe_t com;
};
```

Listing 3.3: variant structure.

The first approach considered relies on identifying the syscalls sequence executed by each Qemu instance (which is different depending on the architecture) before launching the real program. Therefore, the number of syscalls to skip would not be the same for all variants, making it impossible unless adding extra logic for each architecture. Also, different Qemu versions could introduce variations in the syscall sequence, therefore making this approach very fragile.

The second approach consists of using a "checkpoint" syscall at the beginning of the main function of the monitored program. When the monitor detects that syscall, it stops the process. This is repeated for every variant launched. By doing this, every variant is stopped at the beginning of the program, being in the same state and allowing the monitor to start comparing each syscall.

The second approach has been selected, using the syscall `utime()` (number 132 in Linux for x86) with the argument set to 0xFFFFFFFF as the "checkpoint". It is an invalid parameter, and that is why it is impossible to appear on real code. This approach makes necessary to alter the original source code of the program, but it is a more generic and safer approach.

The last part of the initialization consists of setting up a mechanism for identifying the variant in the N-variant system. In order to keep track of the variants, the monitor has an array of structures where all the parameters of the variant are stored. As can it be seen in the Listing 3.3, the stored parameters are the PID in order to identify the process; the last status of the process; the architecture; the file descriptors of the pipes to communicate the process with the monitor, and a structure which holds the information from the last syscall.

3.4 Step-lock Execution

The Step-lock Execution phase is in charge of comparing and executing the syscalls. Also, it detects when the system has been compromised, reacting in consequence. The Figure 3.2

shows the main work flow of the Step-lock execution. This work flow is reflected in the main loop of the monitor, which is shown in the listing 3.4.

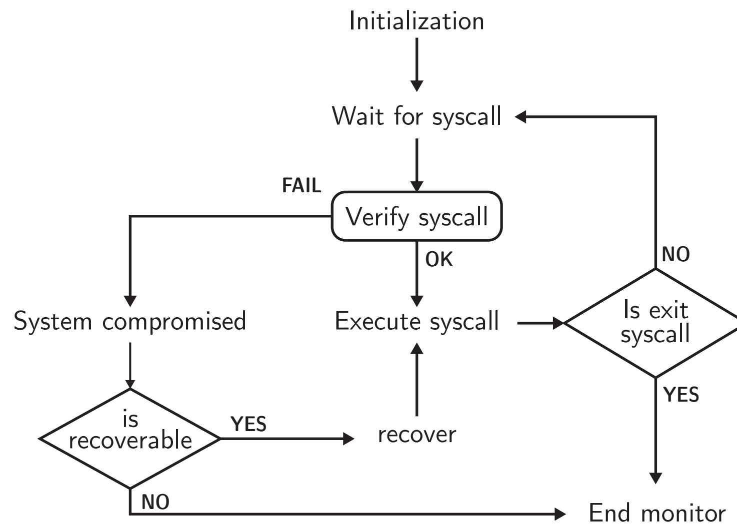

Figure 3.2: Monitor work flow.

3.4.1 Syscall capture

As it can be seen in the Figure 3.2, when all variants are initialized, the monitor resumes all the variants at the same time. Then, the monitor process stops until all processes send the signal "sigtrap" to the monitor notifying that the process is at the entry point of a syscall. At this point, the monitor proceeds to get all the parameters of the syscalls in order to validate the syscall for every process.

All the information of the syscall is stored in the process' info structure using another structure. As it can be observed in the Listing 3.5, the structure contains the syscall number, the six possible arguments a syscall may have and the return value. At this moment, the parameters of the syscall are being captured, so just the syscall number and the syscall arguments should be written. Regarding the reading of the process' information, `ptrace()` is used for getting all the registers involved in the syscall. The Listing 3.6 shows the function that is in charge of getting all the registers and storing them in the syscall_info structure.

3.4.2 Types of syscalls

In order to simplify and unify syscall comparison, the syscalls have been grouped and classified depending on the number and type of arguments. After organizing the syscalls, the monitor is able to get all the different arguments and compare them using generic function

```
while(1){
    wait_for_all_processes();

    /* enter syscall PROCESS */
    status = check_all_processes_status();
    check_and_get_all_processes_syscalls();

    /*  Perform the work before the syscall continue and wait until the syscall and perform
        the work after the syscall */
    perform_syscall_on_all_processes();

    if (get_last_syscall_num() == 231) /* Exit syscall executed. */
        break;

    /* Exiting from syscall */
    status = check_all_processes_status();

    /* Work done */
    prepare_all_processes_for_syscalls();
}
```

Listing 3.4: Main loop code.

```
struct syscall_info {
    unsigned long syscall_num;
    unsigned long args[6];
    unsigned long retval;
};
```

Listing 3.5: syscall structure.

```
void get_process_syscall_info (process_info_t *process) {
    struct user_regs_struct registers;
    ptrace(PTRACE_GETREGS, process->pid, NULL, &registers);

    copy_registers(process,&registers);
}
```

Listing 3.6: Function for getting the syscall info.

handlers for most of them. Unfortunately, not all system calls can be handled using a generic handler, but a dedicated function has to be specially written taking into account the particularities.

Each system call is described by the following parameters:

1. name: Used for debugging purposes, when there is a conflict with a syscall this is the name which is shown in the log file.
2. unify: This parameter is set to “true” when the syscall must be executed by the monitor.
3. ignorable: This parameter is set to “true” when this syscall should be executed only by one of the variations while the other ones do not execute this syscall.
4. args: This is an array of syscall parameters. It stores the information of every argument of the syscall.
   1. name: Used for debugging purposes, when there is a conflict with this argument of the syscall this is the name which is shown in the log file.
   2. pointer: This parameter is set to “true” when the argument is a memory address of the type memory map.
   3. buffer_type: This parameter indicates the type of data structure a pointer points to. It is only set when the pointer parameter is set to “true”.
      i. string: When the buffer is a string the monitor tries to read the buffer until a 0 char is read.
      ii. malloc: When the buffer is a malloc the monitor reads the number of bytes defined in the argument which number is defined in arg_size.
      iii. struct: When the buffer is a struct the monitor reads the number of bytes defined in the the size parameter.
      iv. struct_array: When the buffer is a struct_array the monitor reads the number of bytes defined the argument which number is defined in arg_size, multiplied by the size parameter.
   4. size: This parameter defines the size of the structure when the buffer_type parameter is of type struct or struct_array.
   5. arg_size: This parameter defines which is the syscall argument number where the size of the buffer is stored. It is used when the buffer_type is malloc or struct_array.
   6. type: This parameter defines if the buffer is read only, write only or read and write during the syscall. This parameter is used to know if the buffer must be compared or not. The parameter is only used when the pointer parameter is set to “true”.

```
time_t time(time_t *t);
```

Listing 3.7: Example of safe syscall (time syscall).

```
ssize_t write(int fd, const void *buf, size_t count);
```

Listing 3.8: Example of risky syscall (write syscall).

7. compare: When this parameter is "true", the monitor will compare this argument of the syscall with every other variations.

First of all, the syscalls that are more likely to represent a threat for our system when misused are identified. There are syscalls like `time()` that do not represent a big risk for the system (Listing 3.7), while there are other syscalls like `write()` which can cause a lot of damage (Listing 3.8).

Secondly, it is necessary to differentiate whether the syscall should be "*unified*" or not. A system call has to be **unified** when all the variants shall receive a copy of the syscall output. That is, only one system call shall be performed, and the variants shall receive a copy of its output. As the monitor has all the open resources like socket, files, etc... unifying is the normal behavior, but another reason of unify the syscalls is that the syscall is only executed by one process and not by all the variants.

One syscall that needs to be unified would be the `read()` syscall (Listing 3.9), with this syscall, all the variant must have the same result, so the monitor is in charge of performing the syscall and return the result to all the variants. As opposite, the syscalls that do not need or cannot to be unified: an example of this syscall would be `mmap()` (Listing 3.10). In this case it is only necessary to check the parameters of the syscall, but every variant is in charge of performing the syscall.

Finally, the last parameter used for organizing the syscalls is the argument type. Regarding the kind of arguments that could appear, two main basic data types have been identified:

```
ssize_t read(int fd, void *buf, size_t count);
```

Listing 3.9: Example of unified syscall (read syscall).

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Listing 3.10: Example of spread syscall (mmap syscall).

```
void exit(int status);
```

Listing 3.11: Example of word type (exit syscall).

1. Word type: It is only necessary to get the register from the processor to copy this argument. In the Listing 3.11, the first argument is a word type.

2. Buffer type: This is a more complex type than the word type, as there are different types of buffer. While the word type is usually 32 bits long in a 32 bits architecture and 64 bits in a 64 bits architecture, the buffer type can be of any length, and there are different ways of specifying the size of those buffers. To read a buffer from the variant, the communication mechanism described in Section 3.3.1 is used. Then, in order to read a buffer from the variant, the size of the buffer and the memory position in which the buffer starts are parameters that must be known. The buffers have been differentiated depending on the way the size is specified:

   1. Struct: This type of buffer has a static size, so when it is detected in a syscall, it is not necessary to get the size from another parameter. The syscall shown in the Listing 3.12 is a good example of this type because the second argument is a struct, and the size of this struct is 144, so the monitor knows how big the structure is.

   2. Array Struct: This type of buffer is an array of structs, so in this case it is necessary to define the size of the struct like in the struct type, but it is also necessary to define in which argument the size of the array can be found. An example of syscall that uses this type of buffer can be seen in the Listing 3.13. In this function the first parameter is an array of structures, each structure is 8 bytes and the second argument indicates how many structures are in the first argument.

   3. string type: The string type is an array of characters and in this case the size is defined by the buffer itself. In order to calculate the size, the whole string is traversed until the character 0x00 is found, which means the end of the string.

```
int fstat(int filedesc, struct stat *buf);
```

Listing 3.12: Example of struct buffer type (fstat syscall).

```
int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout_ts, const
    sigset_t *sigmask);
```

Listing 3.13: Example of array of structs buffer type (ppoll syscall).

```
int open(const char *pathname, int flags);
```

Listing 3.14: Example of string type (open syscall).

An example of a syscall that uses this type of buffer is the open syscall shown in the Listing 3.14: in this case the first argument is an string type.

4. malloc type: The malloc type has been defined as a type in which the size is entirely defined by another argument, it means that when reading this type of data, it is necessary to know how many bytes have to be read. Firstly, the register in which the size of the buffer is defined is read, and later on the buffer from the variant is read. One syscall that uses this kind of data is the `write()` syscall shown in the Listing 3.15: This syscall uses a buffer argument as a second argument and defines the size of that buffer in the third argument.

Taking as example the `write()` syscall (3.15), it can be stated that this syscalls writes a number of bytes in a file descriptor. It is a potentially dangerous syscall, because it writes data in the system and interacts with the hardware. This syscall would also need to be unified because all the opened file descriptors are in the monitor. This syscall has three arguments:

1. fd: Word type argument and contains the number of the file descriptor in which the data have to be written.
2. buf: Buffer type argument being a memory pointer, pointing at the memory address in which the data to be written is located.
3. count: This is the complementary information for the “buf” argument, and it defines how many bytes the buffer has.

```
ssize_t write(int fd, void *buf, size_t count);
```

Listing 3.15: Example of buffer type (write syscall).

```
[
    {
        "name" : String ,
        "unify" : Boolean ,
        "ignorable" : Boolean ,
        "args" :[
            {
                "name": string ,
                "pointer" : Boolean ,
                "buffer_type" : [string , malloc , struct , struct_array ] ,
                "size" : int ,
                "arg_size" : int ,
                "type": [in , out , inout ] ,
                "compare" : Boolean
            }
        ]
    }
]
```

Listing 3.16: JSON syscall description file.

Syscall description table

In order to automatize syscall description, a JSON file defining the syscall options and parameters have been created. The structure of this syscall table can be seen in the Listing 3.16. Using this structure all the syscall can be defined using the structure explained before.

The monitor needs to use this syscall information, yet using JSON is quite slow, another solution has been found. This JSON file is transformed to a binary file. In the Figure 3.3 you can see the work-flow for generating this syscall description table. Firstly, a small Python script parses this description file and generates a C file. Secondly, this C file is compiled using GCC, and finally the binary file generated by GCC is directly loaded by the monitor.

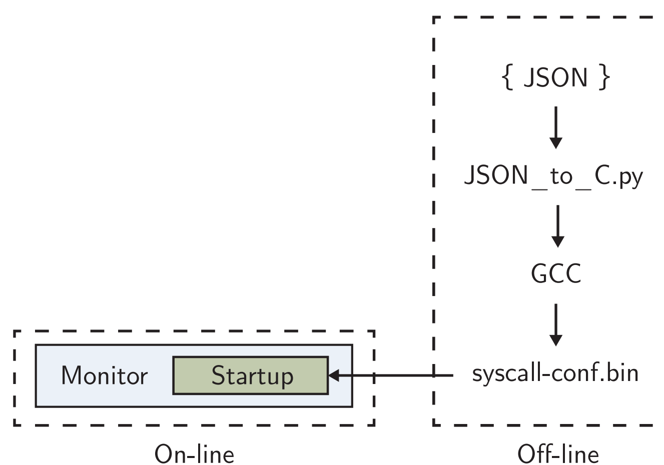

Figure 3.3: Syscall table workflow.

```
{
    "name":"write"
    "unify":true,
    "ignorable" : false,
    "args":[
        {
            "name":"fd",
            "pointer":false,
            "compare":true
        },
        {
            "name":"buf",
            "pointer":true,
            "buffer_type":"malloc",
            "arg_size":2,
            "type":"out",
            "compare":true,
        },
        {
            "name":"count",
            "pointer":false,
            "compare":true
        }
    ],
}
```

Listing 3.17: Definition of the write syscall in the JSON file.

In the Listing 3.17, you can see the definition of the `write()` syscall using the JSON structure.

3.4.3 Comparing Syscalls

Is in this part where the monitor notices if the system has been compromised. As shown in the figure 3.4 there are two main steps in the comparison process.

The first step consists of comparing the syscall number. If the syscall number does not match, it is necessary to check if any of the syscalls are considered secure (that is, the mismatching syscall situation is produced by the differentiation mechanisms used when generating the variants, not because a system failure or misbehavior), meaning that the execution of this syscall in only one variant is not considered a threat to the system. In such situation, the secure syscall shall be skipped until the two syscalls match (with a specified and configurable maximum skipped syscalls limit).

The fact of getting different syscalls from different variants is usually due to the libraries used on each architecture variant. The libraries normally use a lot of syscalls, and the same library compiled for different architectures may use different syscalls for performing the same

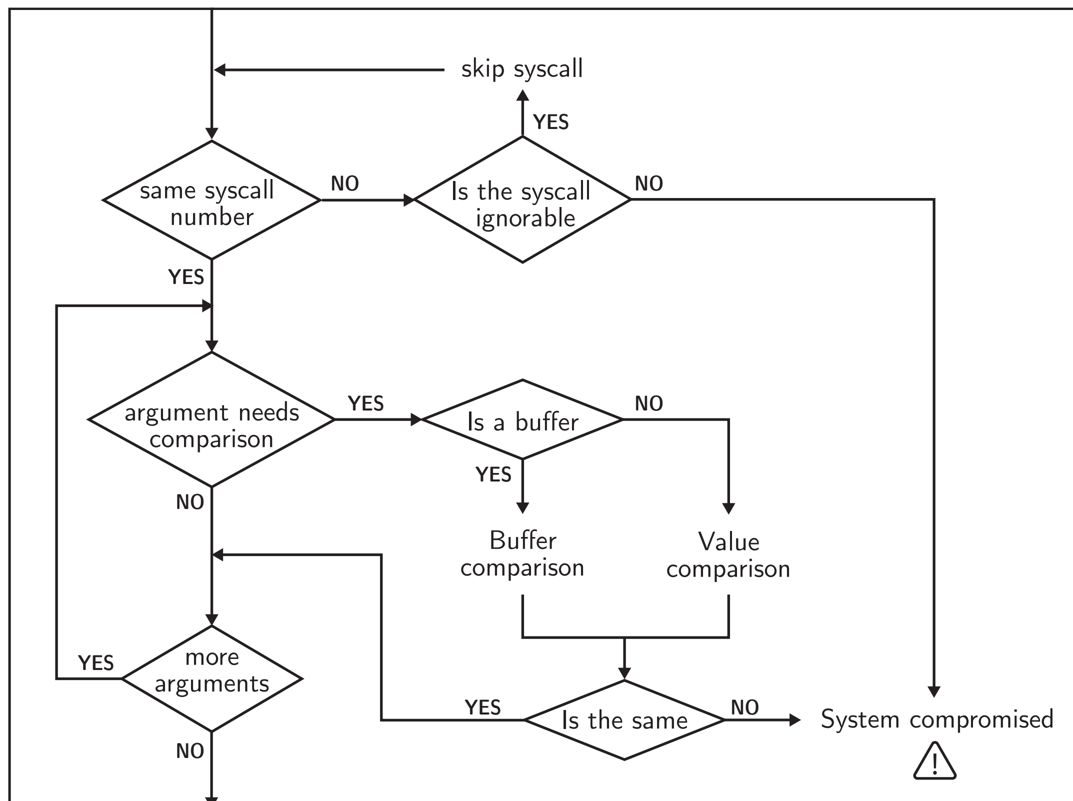

Figure 3.4: Comparison workflow.

function. This would be fixed by checking which the sequence of syscalls matching in different library architectures are, but for the approach taken, it has been considered unnecessary. One example of this problem appears in the glibc library: When using the `printf()` function, glibc perform a malloc for expanding the string but not flushing the string during the function, which leads to two variants performing the flush at different moments.

The second step consists of comparing the arguments of the syscall for which two different comparisons can be made depending on whether the data is a word or a buffer.

If the data is of type word, the values from all the variants are compared directly by reading the registers. Those registers are stored in the process_info struct. But if the type of the data is a buffer, the whole buffer is not compared, as it might be very costly to compare every byte depending on the size of the buffer. Instead, only a percentage of random words of the buffer are compared using `ptrace()` to get the words from the process.

The Listing 3.18 shows the two functions involved in the process of comparison. The `compare_processes_syscalls()` returns 1 if the syscall number and all the arguments are the same, or 0 if the syscall number or some argument is different. The function `compare_arg()` is used to compare every argument in the syscall. The function

```
char compare_processes_syscalls (process_info_t *p1, process_info_t *p2, syscall_t *
    syscall_info) {
    return p1->syscall_info.syscall_num == p2->syscall_info.syscall_num
            && compare_arg(p1, p2, 0, syscall_info)
            && compare_arg(p1, p2, 1, syscall_info)
            && compare_arg(p1, p2, 2, syscall_info)
            && compare_arg(p1, p2, 3, syscall_info)
            && compare_arg(p1, p2, 4, syscall_info)
            && compare_arg(p1, p2, 5, syscall_info);
}

char compare_arg(process_info_t *p1, process_info_t *p2, int arg , syscall_t *syscall_info) {
    return !syscall_info->args[arg].compare ||
        (!IS_POINTER(syscall_info->args[arg].type) && compare_arg_value(p1, p2, arg)) ||
        ( IS_POINTER(syscall_info->args[arg].type) && ( syscall_info->args[arg].type &
            arg_t_buff_out ) && compare_arg_buffer(p1, p2, arg, syscall_info) );
}
```

Listing 3.18: Code used for comparing syscalls.

`compare_arg()` checks if the argument needs to be compared and if the argument is a pointer, it compares the argument as a buffer. Otherwise, it compares the argument as a value.

Generic syscall handling

This part of the monitor is in charge of performing the syscall. At this point, it has been checked that every process tries to perform the same syscall with the same arguments. In order to perform the syscall some things have to be taken into account: Firstly, there is a need to differentiate between the generic logic syscalls and the complex ones. The generic logic syscalls are those from which the logic can be generalized in a way that only with the information defined in the JSON file, the syscall can be correctly performed. On the other hand, if the syscall is a complex one, the logic is defined in the program requiring the execution of special logic in order to be performed by the monitor.

The figure 3.5 shows the work flow when executing a generic syscall like `read()` for instance.

The first step is to check if the syscall needs to be executed by the monitor or it must be executed by the variant itself, for example the `read()` syscall must be executed by the monitor because all the file descriptors belong to the monitor. In this case, if this syscall is executed in the variant, it would trigger an error since this file descriptor does not exist for the variant. An example of a syscall that needs to be executed on the variant would be the `malloc()` syscall because the kernel has to allocated space to the variant and if this syscall

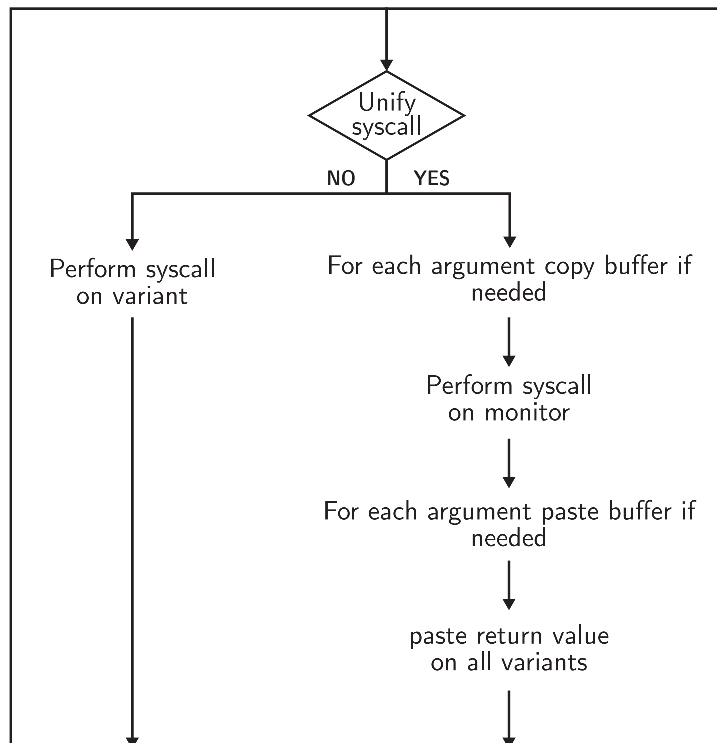

Figure 3.5: Execution work flow.

is executed in the monitor this memory would not be available for the variant.

If the sycall is executed by the variants, the handling is done, but if is executed in the monitor there are a few steps more to be performed. After checking if the syscall needs to be unified, the monitor has to perform the syscall. In order to do it, the number of the syscall and all the data of every argument must be available. The number of the syscall number and all the arguments that are of type word are already available, but if the argument is of type buffer and the syscall reads the content of the buffer, this content must be copied before the syscall is performed.

After getting all the information required, the monitor has to perform the syscall. After performing the syscall in the monitor, all the buffers that the monitor has written need to be copied to every variant. The last thing for completing the syscall handling is to write the return value of the syscall in all the variants. Before writing the return value the syscall has to be canceled in all the variants. In order to do this, the value of the RAX register, which is the one storing syscall number, is changed and set to 0xFFFFFFFF. After setting the register, the variant executes the syscall, the kernel notices that this syscall number does not exist and gives an error. At this point, the RAX register value is set again but with the value the syscall gave to the monitor (return value).

As it can be seen, the monitor performs the syscall that were going to be executed by the variants. The variants execute an invalid syscall, but from the point of view of the

variants, the syscall has been successfully performed.

The Figure 3.6 shows the execution diagram for the `read()` syscall. The monitor is only running when it is handling a syscall, so while the program is running, the monitor is waiting for all the variants to perform a syscall. When all the variants try to perform a syscall, the monitor stops the variants and starts handling the syscall. After checking the parameters of the syscall, the monitor will perform the syscall. Because of the fact that this syscall has an input buffer, it transfers the buffer to all the variants after the syscall is performed by the monitor. That is why the monitor performs a write syscall to write the buffer into a pipe. After all the variants have read the buffer from the pipe, the monitor will continue the execution of all the variants.

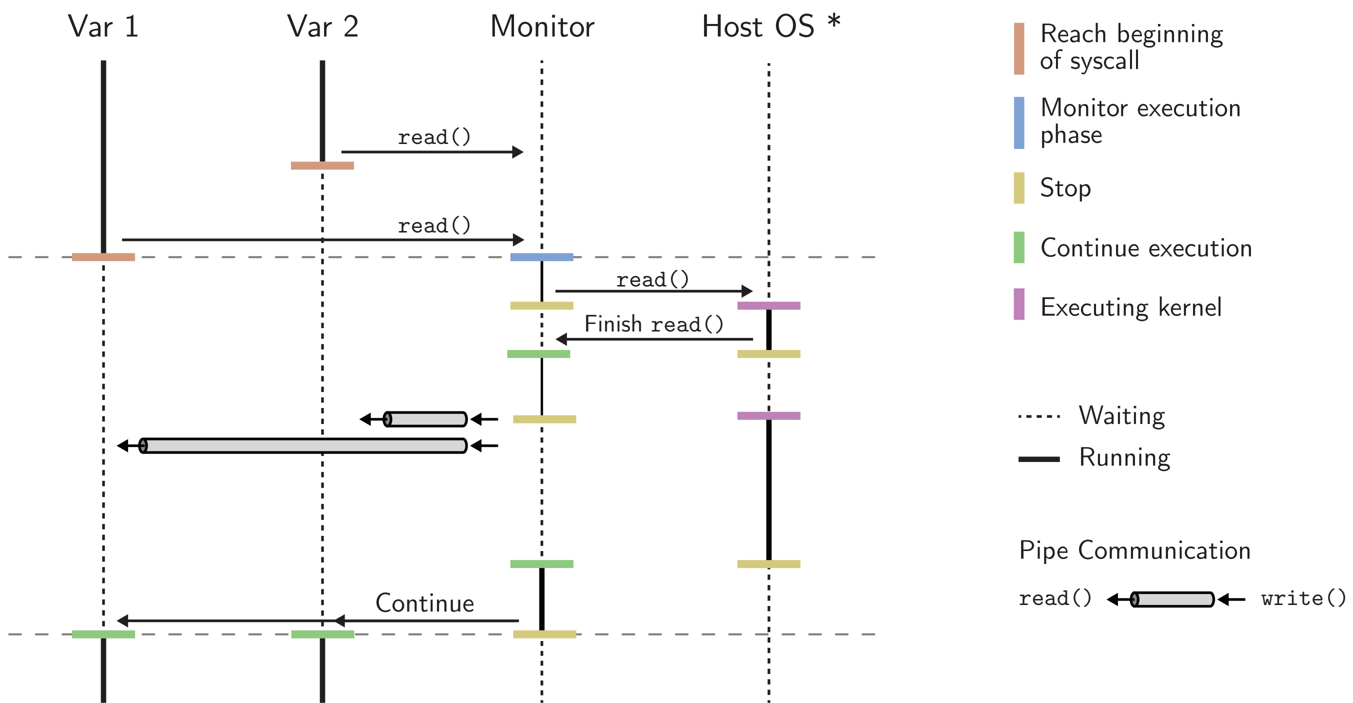

Figure 3.6: `read()` execution digram.

Fork syscall handling

In the case of the `fork()` and `clone()` syscall, a special logic has been implemented in order to set up the monitor to be able to handle the verification of the syscalls of the original and newly-created processes. When a `fork()` or `clone()` syscall is detected, the syscall is executed in the child, and then both father and child processes are stopped at the end of the syscall. Then the monitor is forked, and new pipes are created to communicate the father monitor with the new variants. The final process schema is shown in the Figure 3.7. If only

one communication channel for all the variants were used, race conditions would occur between the old monitor and the new one.

After creating those pipes, the new variants (the child processes after the forks) are assigned to the new monitor (the child monitor after the fork) and the new pipes are opened in those variants in order to communicate with their assigned monitor process. At this point, there are two monitor processes, each one with its own variants, with the ability to communicate independently from the other monitor. After this initialization of the new monitor, the system is ready to resume the execution of both the father and the child variants.

The monitor is the interface to communicate the variant with the system. From the variant's point of view the process tree has been changed, but from the monitor point of view the process tree is the correct one. It means that when the variants want to communicate with its child, the monitor will be communicating with its child monitor, and syscalls like `wait()` will work as expected.

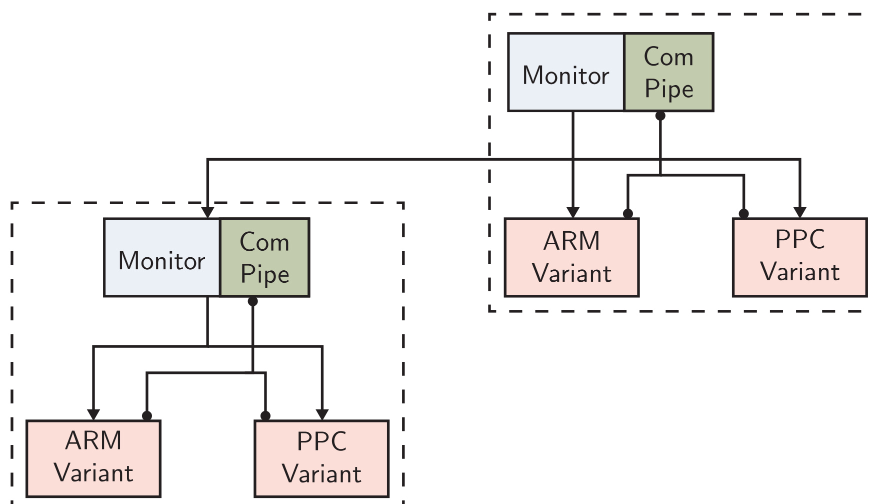

Figure 3.7: Process Tree after fork.

3.4.4 Moving data

There are some syscalls in which moving data from the variant to the monitor or vice versa is required. In order to move that data, the monitor and the variant use a pipe system. Those pipes are initialized in the communication set up process.

This data movement is required sometimes before or after the syscall. For example, the syscall `write()` requires to copy data from the variant before the syscall is executed. But with the syscall `read()` is the other way around since it needs to write data into the variant

```
ptrace(PTRACE_GETREGS, process->pid, NULL, &syscall);
syscall.rip -= SYSCALL_INSTRUCTION_SIZE;
ptrace(PTRACE_SETREGS, process->pid, NULL, &syscall);
```

Listing 3.19: Change instruction pointer in the variant.

after the syscall is performed.

There are two kind of actions, depending on the operation the monitor needs to perform:

1. Read data from the variant: When the monitor needs to read data from the variant, it will perform a `read()` syscall, and the variant will perform a `write()` syscall.
2. Write data to the variant: When the monitor needs to write data to the variant, it will perform a `write()` syscall, and the variant will perform a `read()` syscall.

As explained above, when the monitor needs to move data a syscall needs to be performed in addition to the one that the variant wants to perform. It means that the variant needs to perform more than one syscall. In the monitor side as many syscalls as needed can be performed, but as the monitor cannot execute code in the variant, some tricks need to be done in order to execute more than one syscall in the variant.

When a monitor needs to execute another syscall in the variant, it changes the instruction pointer using `ptrace()`, so it can execute as many syscall in the variant as it wants. The Listing 3.19 shows how the monitor changes the instruction pointer in the variant. It is decreased by two (syscall instruction in x86 is 2 bytes) the current instruction pointer, so the pointer is again pointing to the syscall. By using this technique, the monitor can execute as many syscalls as it needs when the variant is about to execute only one syscall. Moreover, the monitor changes the RAX register so the variant will execute the syscall that the monitor wants. In the case of `read()` syscall, the monitor sets the RAX to 0 and in the case of a `write()` syscall, the monitor sets it to 1.

3.5 Recovery actions

This part of the monitor is one of the most important since it is in charge of recovering the system when it has been compromised, or something has just gone wrong. Nowadays, it is crucial to maintain a system or service, up and running smoothly as much time as possible. Therefore, not only detection but also reaction to failures are necessary.

There are different situations in which the system can fail or be compromised. If the problem is small enough, the measure taken by the monitor will not be dramatical. However, if the system has been fully compromised, the monitor will kill every variant. After killing all the variants, the monitor can relaunch them from the beginning, but it is very costly since it has to synchronize every variant again.

In the situation of the monitor having no variants left (worst case), two measures can be taken:

Resume from checkpoint: If a checkpoint has been defined, the monitor will relaunch the variants from this checkpoint.

The variants can define a checkpoint arbitrarily at any point in execution. In order to define a checkpoint, the monitor uses the same syscall like at the beginning. However, it uses a different value for the first argument this time, 0xFFFFFFFE. If the variant sets another checkpoint, the monitor will remove the old one and save the new one instead.

When the monitor detects that the variant wants to define a checkpoint, it forks every variant and stops the execution of this forked variant. This fork-and-stop action is equivalent to saving the current state the variant processes, so when the monitor needs to restore the variant from a safe position, it only needs to fork the checkpoint again and run this new fork.

By using this checkpoint mechanism, the monitor can mitigate the time used by the system to launch all the variation and initialize Qemu.

Full relaunch: If no checkpoint has been defined, the monitor will relaunch all the variants from the beginning of the variant program.

In the case of the system not being fully compromised, different approaches could have been taken:

1. The first approach would be when the monitor detects which are the variants that have the same syscall number and the same arguments, and it kills the variants that diverge from the majority. If all the syscalls diverge from each other, the monitor aborts the execution of all the variants. In this case, the system would abort when the last two running variants disagree.
2. The second approach would be when the monitor detects and kills the divergent variant, and then launches another variant and executes this variant until reaching the state of the other variants. In order to do this, the monitor needs to keep all the data from all the syscalls and provide all that information to the new variant.

3. The third approach would be to abort the execution of all the variants immediately after any discrepancy. This would be like the worst case explained before.
4. The last approach would be to pause the program and send an alarm to the system administrator, delegating the decision of which recovery action to choose to a human.

The proof-of-concept N-variant monitor implemented has different available recovery actions. In this system, the first and third measures have been implemented. In addition to those recovery actions, the checkpoint mechanism and the full recovery have also been implemented. Those measures can be chosen at the moment of launching the monitor. Those policies have been matched to different ratios of security/availability, being, for instance, the action of aborting execution of all variants when detecting any discrepancy the most secure action but also the one that punishes the availability of the service the most.

4 Conclusions

As explained in the introduction section, this document exposes one of the two parts of a whole project, hence the concluding remarks presented in both documents are the same.

The initial goals planned for this project have been successfully met: a working N-variant system has been implemented and tested against a wide set of common software vulnerabilities. Both, implementation and validation processes, have allowed the authors to gain a much deeper understanding of the internal mechanisms by which a process interacts with an operative system, how the operative system itself manages the requests for system resources of a process, and how a process itself behaves internally and its organization and structure.

In this project, a novel N-variant system based on an automatic processor architecture diversification with monitorization at the system call level has been implemented and validated. This project is part of a research project, where the fundamental ideas of the diversification architecture were developed.

The diversification method is based on the differences of each architecture (endianess, instruction set, register set, address layout, compiler optimizations, etc.) to detect misbehaviors that would remain undetected otherwise, as for each architecture the same fault is manifested in a different manner. Therefore, the more variants running simultaneously, the more likely to detect errors. Variants are created automatically by compiling the source code of the application with several cross-compilers. Therefore, each variant is a GNU/Linux ELF image but with different instructions sets (i.e. x86, x86_64, ARM, MIPS, etc.).

The main contribution of this project has been the design and implementation of the monitor, which acts as the "voter" of a NMR (N-Modular-Redundant) system. It controls the synchronous execution of multiple variants and applies a recovery action when there is a discrepancy between them. Since all the executable images are obtained from the same source code, their execution shall produce a similar sequence of system calls. Each variant is executed on the same host thanks to fast processor emulation and they are controlled by the monitor process that checks every system call to ensure semantic equivalence for all variants.

The monitor has been implemented as a regular Linux process using the `ptrace()` system facility to control the variants. `ptrace()` is a kernel facility to observe and control the execution of processes from user space. It is mainly used by debuggers to trace and inspect debugged programs. It has been used `ptrace()` to automatically track the execution of multiple variants.

Since the `ptrace()` facility was designed to be used as a debugging mechanism and debugging is done off-line, it is very powerful but not as efficient as desired when used intensively. Specifically, moving data between the monitored process and the monitor is very slow. To overcome this major problem, as well as other problems, several advanced solutions were designed and implemented without changing the kernel of the operating system. That is to say that no new system calls or devices are added to Linux.

Thanks to the tricks used in the implementation of the monitor, the variants do not need to contain any kind of "helper" code to interact with the monitor. Both the variants and the operating system kernel are completely transparent.

The tests driven to enclose the capabilities of this N-variant system have proven it to be effective when detecting undesired behavior produced by exploit attempts of some of the most common program vulnerabilities derived from bad programming habits.

With just 3 different architectures (the bare minimum, should some kind of post-detection policy other than aborting execution were implemented alongside the monitor), the system has been able to virtually disable the exploitation of vulnerabilities such as stack buffer overflow, string format vulnerability and any sort of code injection and execution; in addition to that, this systems greatly increases the complexity for successfully exploiting a heap-overflow based vulnerability.

Furthermore, bugs and malware with such a high-profile as the so called "Blaster worm", or the "Heartbleed bug", with worldwide impact and repercussion in terms of privacy and security could have been promptly detected and fixed at origin should a N-variant system similar to the one proposed in this paper were in use, having all subsequent damage avoided.

As it is to be expected, there has to be a tradeoff between the extra security provided by this N-variant system and the performance of the processes or services it protects. There is a considerable performance impact on cpu-intensive applications directly proportional to the number of variants simultaneously running. However, taking advantage of the parallelism offered by multicore processors, this performance penalty can be considerably reduced.

Further research

Further work in the line of what has been done with this N-variant system could explore the possibility of implementing recovery policies in the N-variant monitor based on treating each system call as a checkpoint. Apart from this, being able to rollback a program execution to any past checkpoint, or to launch a new variant at any point in time and using the checkpoints to allow that new variant to eventually catch the state of the other variants up.

Another possibility offered by the N-variant system would be to set an operating system init process to be monitored, therefore being able to effectively monitor and protect each and every process run on that operating system.

Yet another derivation from this work would imply to modify and extend the monitor system to also provide the capabilities of a sandboxing system. In addition to checking the integrity of the execution, the monitor can be aware of what actions interact with the filesystem, or with the network interfaces and depending on some specified policies, preventing or reacting to certain actions that the monitored process would perform.

As a way of mitigating the severe overhead introduced by Qemu, introducing a multiprocessor machine would be possible with each processor being a different architecture and running each variant natively. In this way, the performance would be maximum, and the only overhead introduced would be the one of the monitor.

Derived from the previous possibility, in a system where it were desirable to have a pool of variant architectures but running only simultaneously a subset of all those architectures and rotating the active ones, a way to avoid having idle hardware waiting would be to use FPGAs (Field Programmable Gate Arrays) to reconfigure themselves as the current architecture to be run.

The N-variant monitor can also be extended to log at which point has a program failed or been compromised, as the execution context (last syscalls, arguments, etc) is known at every moment. Furthermore, knowing the context when a variant has failed could be used to build statistics upon which an heuristics system can be build to try to anticipate certain situations by preemptively launching, killing, pausing variants or creating checkpoints in some situations.

Yet another derivation from the monitor could be using the ability of detecting code injection and execution to, instead of aborting execution when detected, sandbox and isolate the target variant of the code injection, giving an attacker the impression of a successful attack, while giving some time to properly react and protect the system.

List of Figures

2.1 N-variant Work flow. 8

2.2 The monitor acts as a wrapper to the variants. 10

3.1 Global system overhead. 13

3.2 Monitor work flow. 17

3.3 Syscall table workflow. 23

3.4 Comparison workflow. 25

3.5 Execution work flow. 27

3.6 `read()` execution digram. 28

3.7 Process Tree after fork. 29

Listings

3.1 Code for allowing the monitor to trace the variant. 15

3.2 Code for set the monitor to trace the forked variants. 15

3.3 variant structure. 16

3.4 Main loop code. 18

3.5 syscall structure. 18

3.6 Function for getting the syscall info. 18

3.7 Example of safe syscall (time syscall). 20

3.8 Example of risky syscall (write syscall). 20

3.9 Example of unified syscall (read syscall). 20

3.10 Example of spread syscall (mmap syscall). 21

3.11 Example of word type (exit syscall). 21

3.12 Example of struct buffer type (fstat syscall). 21

3.13 Example of array of structs buffer type (ppoll syscall). 22

3.14 Example of string type (open syscall). 22

3.15 Example of buffer type (write syscall). 22

3.16 JSON syscall description file. 23

3.17 Definition of the write syscall in the JSON file. 24

3.18 Code used for comparing syscalls. 26

3.19 Change instruction pointer in the variant. 30

Bibliography

[TIO, 2012] (2012).
TIOBE Software: Tiobe Index.

[Bellard, 2005] Bellard, F. (2005).
Qemu, a fast and portable dynamic translator.
In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX.

[Cox et al., 2006] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. (2006).
N-variant systems: a secretless framework for security through diversity.
In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA. USENIX Association.

[Huang et al., 2010] Huang, R., Deng, D. Y., and Suh, G. E. (2010).
Orthrus: efficient software integrity protection on multi-cores.
SIGPLAN Not., 45(3):371–384.

[Jackson et al., 2011] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., and Franz, M. (2011).
Compiler-generated software diversity.
In Jajodia, S., Ghosh, A. K., Swarup, V., Wang, C., and Wang, X. S., editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 77–98. Springer.

[Knight and Leveson, 1986] Knight, J. C. and Leveson, N. G. (1986).
An experimental evaluation of the assumption of independence in multiversion programming.
IEEE Trans. Softw. Eng., 12(1):96–109.

[Laprie et al., 1990] Laprie, J.-C., Béounes, C., and Kanoun, K. (1990).
Definition and analysis of hardware- and software-fault-tolerant architectures.
Computer, 23(7):39–51.

[Mitre, 2011] Mitre (2011).
CWE/SANS top 25 most dangerous software errors.

[Salamat et al., 2011] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., and Franz, M. (2011).

Runtime defense against code injection attacks using replicated execution.
IEEE Trans. Dependable Sec. Comput., 8(4):588–601.