

École polytechnique de Louvain

Constructing Formal Models from Binaries

Author: **Antoine CHENOY**

Supervisors: **Axel LEGAY, Fabien DUCHÊNE, Thomas GIVEN-
WILSON**

Readers: **Axel LEGAY, Charles PECHEUR, Fabien DUCHÊNE,
Thomas GIVEN-WILSON**

Academic year 2019–2020

Master [120] in Computer Science and Engineering

Abstract

Software performing in hostile environment may be vulnerable to fault injection. To ensure a computer program has a failure rate below an acceptable risk level, it must be verified formally. The first step consists of reproducing hardware faults by using a software injection fault tool developed in this master's thesis. After that, two ways of verifying properties on the variations of the binaries are explored. The first one uses Lodin and the decompilation of binaries to LLVM bitcode. This way was put aside due to its many drawbacks. The second way uses plugins integrating with the statistical model checking platform Plasma Lab in order to simulate software with a debugger like GDB, check LTL properties on its execution trace and evaluate the correctness of all the possible faulty variants. Finally, this approach is illustrated and evaluated with two examples checked by the complete toolchain as a feasibility proof.

Acknowledgements

This master's thesis is the result of almost a year of work that would not have been possible without the support and precious help of many people.

I would first like to thank Professor Axel Legay, supervisor of this thesis, and the assistants Thomas Given-Wilson and Fabien Duchêne for their availability, the time they devoted to discuss new solutions for the progression of this project, their relevant remarks and their proofreadings. Many thanks to them for always supporting, encouraging and guiding me throughout the year.

I thank the reader of this thesis, Professor Charles Pecheur, as well as all those who will be interested and will use the conclusions and tools developed in this work.

I am also grateful to my family, relatives, friends and colleagues who supported me in my research and helped me give the best of myself.

Contents

1	Introduction	5
2	Background	7
2.1	Fault injection	7
2.1.1	Software Fault Injection	8
2.1.2	SWIFI	8
2.1.3	Fault Model	8
2.2	Formal Verification	9
2.2.1	Model Checking	9
2.2.2	Statistical Model Checking	9
2.2.3	Linear Temporal Logic	10
2.3	Software Execution and Simulation	10
2.3.1	Executable Format	10
2.3.2	Debugging Information	11
2.3.3	Simulating Software	11
3	SWIFI tool	12
3.1	Previous Work	12
3.2	Motivations	13
3.3	Software Architecture	13
3.3.1	Main flow of the program	15
3.3.2	Implemented fault models	16
3.3.3	Graphical User Interface	18
3.3.4	Tests	18
3.4	Utilization and Case Studies	18

3.5	Performance	21
3.6	Discussions	21
4	Simulation through LLVM with Lodin	22
4.1	Previous Work and Background	22
4.1.1	LLVM bitcode	22
4.1.2	Lodin	22
4.1.3	Microsoft llvm-mctoll	23
4.1.4	Avast RetDec	23
4.2	Motivation	23
4.3	Architecture and Limitations	23
4.4	Discussions	24
5	Simulation Through GDB With Plasma	26
5.1	Previous Work And Background	26
5.1.1	GDB	26
5.1.2	Plasma	27
5.1.3	Monte Carlo Algorithm	28
5.1.4	BLTL Properties	29
5.2	Motivations	30
5.3	Software Architecture	31
5.3.1	Software Simulator Plugin	32
5.3.2	Extended BLTL Plugin	34
5.3.3	Putting Everything Together	35
5.3.4	SWIFI Integration Plugin	37
5.3.5	Tests	37
5.4	Utilization and Case Studies	39
5.4.1	Statistical model checking	39
5.4.2	Finding faulted variants candidates	42
5.5	Performance	44
5.6	Discussions	45
6	Related work	46
7	Conclusion And Future Work	47
A	UML diagrams of the executable simulator plugin	50

List of Figures

3.1	UML diagram of the different fault models.	15
3.2	Screenshot of the GUI window comparing the initial and edited files.	18
4.1	Tool-chain using LLVM as an intermediate step.	23
5.1	Plasma architecture with plugins (taken from [29]).	27
5.2	Monte Carlo algorithm plugin workflow.	28
5.3	Tool-chain using Plasma plugins.	31
5.4	Executable simulator plugin workflow.	34
5.5	Extended BLTL plugin workflow.	35
5.6	Complete plugins workflow with Monte Carlo.	36
5.7	Complete plugins workflow with SWIFI.	38
5.8	Buffer-overflow detection using statistical model checking.	41
5.9	Configuration of the SWIFI generator algorithm.	43
5.10	Results on each possible variants.	43
5.11	SWIFI genrator algorithm time distribution analysis.	44
A.1	Simplified UML diagram of the simulator plugin.	51
A.2	UML sequence diagram of the simulation process.	52

List of Tables

3.1	Metrics about the project files of the SWIFI tool.	14
3.2	Statement code coverage of the tests.	19
5.1	Metrics about the project packages of Plasma plugins.	31

CHAPTER 1

Introduction

Computer programs are used in almost every domain nowadays and handle more and more responsibilities. In domains with critical missions, ensuring software will perform correctly is highly desirable as lives or billions of \$ could be lost.

Some systems also have to deal with hostile environments that could induce unwanted errors into the system. This is for example the case in the aerospace domain where a satellite program may be affected by solar radiation [1]. In the military domain, a drone can be under the attack of an inimical electromagnetic pulse [2]. Each case needs to find a way to evaluate the amount of risk encountered by such faults and the potential consequences.

Instead of sending hundreds of the same satellite with different versions of the software into space, it is possible to reproduce the conditions the software will face up there and examine its behavior. This work wish to develop an entire workflow allowing evaluating correctness of software under fault attack.

The first step towards this goal is to reproduce the potential faults occurring in the hostile environment. The assumption made in this work is that the majority of faults happening in the storage system, the memory or the CPU (in the hardware) can be represented by a fault in the executable file (in the software). For example, a CPU decoding an instruction incorrectly can be represented equivalently as an instruction changed in the executable. A new tool is developed for this purpose. It can insert several types of faults into a binary executable file based on fault models going from switching a bit to changing the address of a jump instruction.

Once a faulted variant has been created, its behavior needs to be evaluated. There exists plenty of methods to evaluate code correctness like testing or reviewing but only a formal method based on a mathematical background can bring a strong confidence in the code reliability. The formal method used in this work is called statistical model checking. It allows checking properties by performing simulations of software on the real hardware and doesn't require building a model. The requirements are translated to properties using code expressions in a mathematical form. Two approaches are explored in order to simulate and check properties on a program.

The first idea uses an existing tool called Lodin able to simulate and check properties statistically on LLVM code. The programs are decompiled to LLVM bitcode and then checked.

The second approach uses the power of debuggers like GDB to simulate and retrieve interesting values during the execution. It takes the form of a set of plugins integrated into the platform Plasma which allows performing statistical checks on the gathered data. Plasma uses a bounded linear temporal logic to define properties. The syntax of this logic is also extended to allow the use of code expressions inside them. Additionally, a plugin performing the generation of combination of executable variants with wanted faults allows having a full toolchain for detecting fault injection vulnerabilities.

Finally, two complete examples are performed using the developed toolchains in order to prove the feasibility of these approaches and then evaluate their performances. Some future steps and possible improvements are also exposed.

All the tools and examples developed in this master's thesis are open source and fully available online on the listed platforms.

CHAPTER 2

Background

This chapter recalls important concepts required to fully understand the work detailed in this thesis. Three sections will present the appropriate background on the following concepts : fault injections, model checking and software simulation.

2.1 Fault injection

Fault injection is a technique used to measure the dependability of a system against faults. Faults are introduced deliberately in the system and the consequences on the system behavior are observed and evaluated. There exists different categories of techniques where a fault is injected in different ways into the software [3]. Hardware based fault injections are closer to the reality scenario where a foreign physical event disturbs the correct behavior of the system. Events like radiation [1] and an electromagnetic pulse [2] can be replicated under controlled conditions in order to discover potential vulnerabilities [4][5]. Another category of techniques is the software based fault injection where a modification on the software level tries to have the same effect as a real fault [6]. This is the kind of faults used in this work as it doesn't require technical equipment and can review a broader number of faults faster. A last category consists of injecting faults during the simulation or the emulation of the system [7].

2.1.1 Software Fault Injection

A software fault injection consists of a small mistake or error that is inserted into the software of interest. The fault can be implemented in many ways depending on the evaluated aspect. For example, disk failures can be implemented by inserting errors in the files used by the system.

This work focus on inserting faults in the executable file of a program. This allows reproducing a wide range of the hardware faults like disk corruptions, some memory errors and some CPU miscalculations.

2.1.2 SWIFI

SWIFI stands for software implemented fault injection and it consists in the insertion of faults by a tool inside the software of interest. The tool generates a new file with a modification in its binary content on the disk. Once an edited variant is produced, it can be analyzed by other tools. This method has the advantage of allowing generating many variants manually and quickly instead of waiting for them to appear (or be produced) in a real-life system.

The possible faults inserted are defined by fault models.

2.1.3 Fault Model

In order to evaluate the behavior of software against a set of faults, the kind of faults needs to be formalized. The fault models presented here are taken from the previous works [8] and [5]. Each fault model is linked to a plausible real-life scenario.

- Many kinds of external events like radiation can make a bit in the system switch value.
- Malicious attack can force a system to reset a set of bits, namely a byte.
- In the same idea, a word in the system can be set to zero by an injection attack.
- A processor might dysfunction and decide to skip an entire instruction.
- An attacker may try to disrupt the control flow of a program by using one of the previous faults, this can be modeled by a target of a jump instruction being modified.

- The above model can be extended to conditional jumps.

All these faults models are specified during the implementation of the SWIFI tool in the next chapter.

2.2 Formal Verification

The reliability of various kinds of systems has to be verified to lower the risk of malfunctions which will perhaps lead to a loss of money, reputation and even human life in some cases. The verification of a system has to be done in a formal way able to give strong mathematical confidence in the results.

2.2.1 Model Checking

Model checking [9] is one of the formal verification techniques. It consists of the definition of a formal model representing the actual system. The model can have several levels of details that allows abstracting some insignificant aspects. To get consistent results, the model has to be faithful to the real system.

Once the model is defined, properties can be checked on it. The properties are checked against the model and evaluated to true or false (or unknown depending on the logic used). The requirements needed to ensure the reliability of the system needs to be defined and then formalized as properties on the model.

2.2.2 Statistical Model Checking

Classical model checking [9] (iterating over each path) is costly and the verification of the properties can sometimes take an unbounded time duration to finish. Statistical model checking [10] is a recent approach based on probabilistic models. Instead of deciding completely if a property holds on every state, it measures statistically its truthfulness. This is done by performing a fixed number of simulations of the system and using a hypothesis testing to decide whether the property can be granted with a given confidence level or not.

This approaches allows checking properties on a system without the need to model it formally. Statistical model checking only needs to be able to simulate the system and be able to retrieve the values needed to check the properties from it. Another advantage is that the simulation time can be estimated and bounded. Stopping the simulations will give a result in all cases (with a lower confidence level of course). The main drawback is that the assertions are probabilistic and

never determined with 100% confidence. This becomes less concerning when we think that software runs on hardware that doesn't have 0% risk of failure either. This probabilistic approach is also well suited when working with fault injection as it allows combining the risk of a fault occurring and the risk of a fault making the software behave badly.

2.2.3 Linear Temporal Logic

Expressing properties formally is a key point in the software verification process. A software execution can be viewed as a series of states. Each state contains all the values of the elements used by the software. A transition from one state to the other is performed by stepping through a portion of the software code. The properties about the software can be defined in terms of these states content. A propositional logic can define properties about each state individually. To be able to consider all the path of the execution (the series of states), this logic needs to be extended. One popular extension is called LTL [11] for linear temporal logic and is a logic describing properties on a path. It's therefore more adapted to describing properties on software execution path. The syntax and semantic of the LTL version used in this work is defined later (5.1.4).

2.3 Software Execution and Simulation

This subsection present how a software is packed into an executable and what information this package can contain. It also reviews how a software can be simulated.

2.3.1 Executable Format

The majority of today's processors executes binary instructions. These instructions are assembled from a human readable representation called an assembly language. These instructions are put inside a binary file called an executable aside with other information. Two existing executable formats are ELF, mostly used on Unix systems, and PE on Windows. The tools developed here are based on native executable format (like ELF and PE) but could be extended to support other kinds of software formats like for example JAR applications, Python script files, ...

2.3.2 Debugging Information

The executable format can also contains debugging information. For example, when compiling a program with GCC and the debug option enabled, it will produce a binary containing DWARF information [12] that allows linking the values in memory during the execution of this program with the original source code variables, functions, etc. This debug information can be used during the simulation to retrieve values for properties based on the software source code.

2.3.3 Simulating Software

Understanding the behavior of software can be performed programmatically in two ways : with a static or a dynamic analysis [13].

The static way is performed by reading the source code or the executable and building a symbolic representation. This often takes the form of control flow graph. As each step is defined in a symbolic way, the model can quickly become oversized and checking properties will need to evaluate all these symbolic expressions.

The dynamic way runs or emulates the software and require a way to access the system internal elements to evaluate properties. Emulating the software can be done by building an interpreter that applies each operation of the program on a virtual state. On the other end, running the program doesn't require maintaining a virtual state but needs to access the actual state at each step of the execution to retrieve the useful values. This last one can be performed by a tool called debugger which can pause program execution to get internal values. Both dynamic ways are explored in this work.

Other background concepts are also presented below in the specific sections as they are more linked to a particular subject or to a technical choice.

CHAPTER 3

SWIFI tool

The first step toward the verification of programs against fault injection is being able to create variants of the original program containing software implemented faults reproducing the hardware ones.

This chapter details the construction of a tool performing insertions of specified faults in a binary file. The tool is called SWIFI for SoftWare Implemented Fault Injection. The available faults are based on the faults models described in 2.1.3.

3.1 Previous Work

The tool described here is based on the work of Nisrine Jafri and Thomas Given-Wilson [14] which consists of a single Python file that can be used through the command line to inject one fault into one binary file. Their implementation supports the following fault models :

- **zerobyte** : set one specific byte to zero,
- **zeroword** : set one specific word to zero,
- **nop** : replace one byte by an x86 **NOP** instruction,
- **armnop** : replace two bytes by an ARM **NOP** instruction,
- **flip** : flip one specific bit,

- **tamper** : set one byte to a specific value,
- **jump** : change the target of an x86 jump instruction.
The supported jump instructions are `JMP rel8`, `JMP rel16`, `JBE rel8` and `JBE rel16`.

A NOP instruction will perform no operation when executed by the processor. A JMP instruction will change the flow of execution and executes the instruction at the given target (based on a condition for JBE).

3.2 Motivations

The existing tool was effective and mostly correct but it was missing some features. The challenge was to rewrite that tool in order to improve its structure and functionalities. First, the structure can be improved to allow a more generic use of the faults models and make it easier to integrate new ones. Second, the new intended features consist of :

- Applying multiple faults at once on a single file,
- Repeating a fault on a specified range of bytes,
- Warning the user if he tries to apply several faults at the same place,
- Adding new supported instructions,
- Optionally reading the faults from a configuration file,
- Creating a graphical tool to visualize the applied faults on the binary.

3.3 Software Architecture

The software sources, tests and documentation are available on a GitHub repository¹. The source files are in the `swifitool` folder while the `examples` folder and `README.md` contain examples and documentation that will be presented in the next section. We will analyze the implementation choices in this chapter.

The implementation was done in Python for several reasons. Python allows developing small applications relatively quickly and is a popular programming language [15]. This should allow anybody to add new features or new fault models in the future. Python also runs on many different platforms and is easy to integrate in a workflow with the command line.

¹<https://github.com/chenoya/swifi-tool>

Some quantitative metrics about the project are shown in the table 3.3.

Path	Length	Lines	LOC	Functions	Classes	Methods
.idea/*	-	-	-	-	-	-
examples/gcd	8192	-	-	-	-	-
examples/gcd.c	150	12	11	2	0	0
examples/gcd_flip	8192	-	-	-	-	-
examples/gcd_jump	8192	-	-	-	-	-
examples/gcd_nop	8192	-	-	-	-	-
examples/gcd_zero	8192	-	-	-	-	-
examples/Graphical_tool.png	74938	-	-	-	-	-
examples/README.md	6969	167	-	-	-	-
swifitool/swifitool/___init___.py	119	3	0	0	0	0
swifitool/swifitool/faultmodel.py	464	16	10	0	1	3
swifitool/swifitool/flp.py	1037	28	22	0	1	3
swifitool/swifitool/jcc.py	4874	95	85	0	1	3
swifitool/swifitool/jmp.py	4765	94	84	0	1	3
swifitool/swifitool/nop.py	1483	36	27	0	1	3
swifitool/swifitool/z1b.py	488	18	13	0	1	3
swifitool/swifitool/z1w.py	1037	27	19	0	1	3
swifitool/___init___.py	102	3	0	0	0	0
swifitool/config.py	282	8	7	0	1	1
swifitool/diff_ui.py	3638	101	70	2	0	0
swifitool/faults_inject.py	4157	86	59	1	0	0
swifitool/utils.py	2273	68	25	4	0	0
tests/___init___.py	68	2	0	0	0	0
tests/test_flp.py	3070	71	46	0	1	9
tests/test_jcc.py	8916	187	86	0	1	14
tests/test_jump.py	8926	187	86	0	1	14
tests/test_nop.py	4607	100	63	0	1	14
tests/test_other.py	1695	44	27	0	1	4
tests/test_z1b.py	1944	50	34	0	1	6
tests/test_z1w.py	4513	97	61	0	1	13
README.md	2219	43	-	-	-	-
setup.py	521	17	15	0	0	0
	184215	1560	850	9	15	96

Table 3.1: Metrics about the project files of the SWIFI tool.

3.3.1 Main flow of the program

The main file of the project is `fault_inject.py`. This file is responsible for understanding the user parameters and producing a new file with the right modifications. First, it parses the user input arguments using the `argparse` module. Then it makes some sanity checks about those and stores them in a configuration object. It also fills a list with the fault models (and their parameters) asked by the user. After that, it computes all the file sections that will be edited by calling the `edited_file_locations` on each asked model. At this time, it also checks that no fault models write at the same position and that the positions are inside the file. The next step consists of copying the original file and then applying each fault models on the new copy. This is done by calling the `apply` method of each fault model. The last step consists of displaying a GUI window comparing the original file and the edited one side by side.

The main function of the program uses fault models in a generic way. Consequently, each fault model needs to implement the same interface (`FaultModel`). This interface gives access to the two methods discussed above as well as a constructor receiving the fault model parameters. There are also three fixed variables that contains respectively the name of the fault model, its documentation (shown when calling the help) and its expected number of arguments. This organization is represented in the figure 3.1.

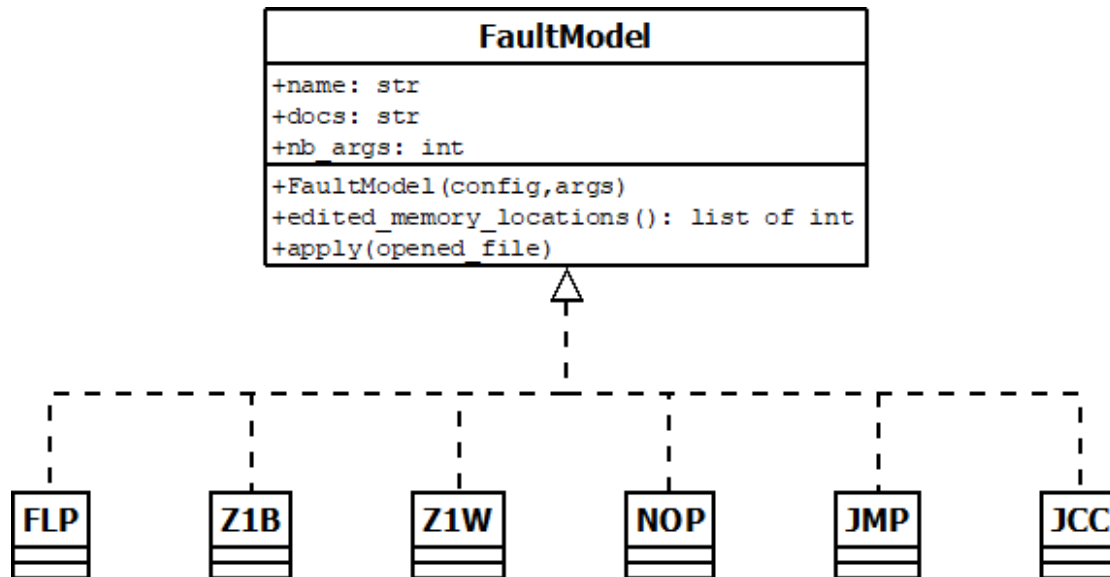


Figure 3.1: UML diagram of the different fault models.

3.3.2 Implemented fault models

The available fault models are FLP, Z1B, Z1W, NOP, JMP and JCC.

FLP

The "flip one byte" fault model edits one bit of the file. It takes as parameters the byte offset and the bit offset inside that byte. If this bit was 0 in the original file, it is replaced it by 1 and vice versa.

Z1B

The "zero one byte" fault model is quite straightforward, it simply replaces by 0 all the bytes given as parameters in the file (one byte or a range of continuous bytes).

Z1W

The "zero one word" fault model works the same way as the Z1B fault model but performs operations on words instead of bytes. The word size in terms of bytes has to be defined by the user with the parameter `-w`. If a range of bytes is given as parameter, the range must contain an integer number of words.

NOP

The "no-operation fault model" will put an instruction that does nothing at the desired offset in the file. On the x86 architecture [16], the smallest NOP instruction is encoded on one byte (0x90) while on ARM [17], it is encoded on two bytes (0x00BF). It is also possible to insert in integer number of NOP on a specific range. The user must be aware of the instructions alignment in the executable section of the file. If the specified range of NOP does not cover entirely a previous instruction, this will break the way the processor decode the next instructions and execute completely different instructions. This behavior might be desired and is left to the user's attention.

JMP

This fault model allows the user to change the destination (i.e., target) of a jump instruction. The user has to specify the jump instruction location in the file and the target pointing to a new location. Generally, this target address lies inside the file but in some cases with libraries, it could point to somewhere else. Anyway, a warning is printed if the user tries to set a target outside the file range. This

instruction is architecture dependent so it handles several jump variants for x86 and some other for ARM.

On x86, the Intel reference manual [16] defines eleven variants of the **JMP** instruction. Five of them are "far" jump rarely used in software and not supported in a first time. Three other are jumps based on the content of registers or memory so not suitable for this type of fault model. The last three are "short" jumps available with an operand of 8, 16 or 32 bits. This operand is an offset in bytes from the end of the jump instruction. This is the kind of instruction the fault model wants to edit. By checking the opcodes at the given address, the tool can determine which variant of **JMP** he has to face. Depending on this, it will compute the right operand : $\text{offset} = \text{target} - \text{addr of end of jmp instruction}$. It then checks that this offset fits in the number of bits of the operand and if yes, edits the file accordingly.

On ARM [17], the "branch" instruction is targeted. The tool checks that the **B** or **BL** instructions have no condition (on ARM, each instruction is conditional and we only want the unconditional ones here). Then it computes the offset the same way as with x86 except ARM works with a range of 26 bits rounded by 4 (so 24 bits in the end). The Thumb instruction versions are currently not supported.

JCC

This fault model is similar to the **JMP** one except it deals with conditional jump instructions.

On x86 [16] this corresponds to the instructions : **JA**, **JAE**, **JB**, **JBE**, **JC**, **JCXZ**, **JECXZ**, **JRCXZ**, **JE**, **JG**, **JGE**, **JL**, **JLE**, **JNA**, **JNAE**, **JNB**, **JNBE**, **JNC**, **JNE**, **JNG**, **JNGE**, **JNL**, **JNLE**, **JNO**, **JNP**, **JNS**, **JNZ**, **JO**, **JP**, **JPE**, **JPO**, **JS** and **JZ**. Each in their 8, 16 and 32 bits operand versions (if they exist). The tool covers every conditional jump of the x86 instruction set. The computation of the offset follows the same principle and is applied in the same way.

On ARM [17], the focus is still on the **B** and **BL** instruction but with a condition this time (other than "Always"). The possible conditions are : **eq**, **ne**, **cs**, **hs**, **cc**, **lo**, **mi**, **pl**, **vs**, **vc**, **hi**, **ls**, **ge**, **lt**, **gt**, and **le** forming **BEQ**, **BNE**, ... and so on. The offset also works the same way, as well as the instruction edition.

3.3.3 Graphical User Interface

A new feature consists of a graphical window comparing the initial file and the edited one. It uses the module `tkinter` to draw a window containing tree columns. These are the byte offsets, the contents of the initial and edited file in hexadecimal. The differences are highlighted using a different color for each type of fault.

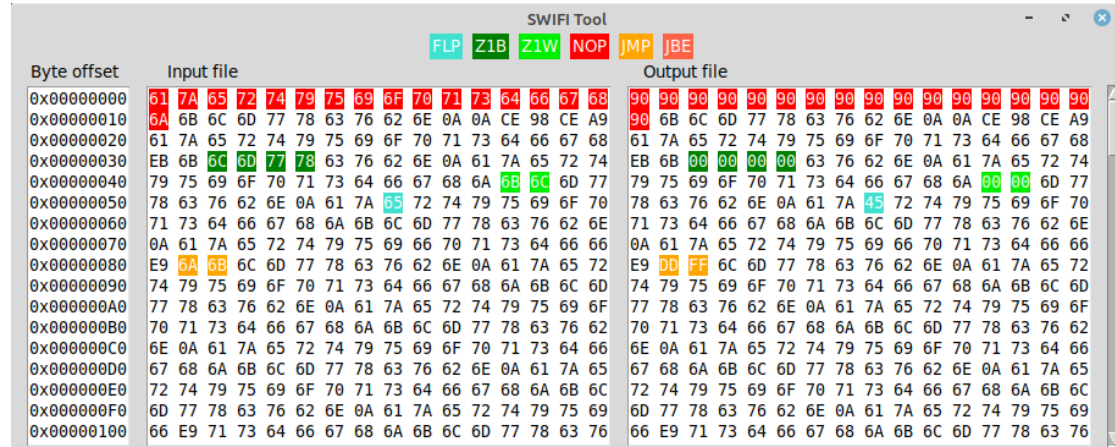


Figure 3.2: Screenshot of the GUI window comparing the initial and edited files.

3.3.4 Tests

Unit tests are available in the `test` folder. These tests were created to increase our confidence about the correctness of the program. They were designed to cover most of the lines of the program code (usually called code coverage tests). The following table 3.2 reports the percentage of covered statements per source file.

Each fault model is tested with 100% code coverage. The missing percentages come from less important code parts or from the GUI. Ideally, testing the GUI part could be done by simulating the user interaction.

3.4 Utilization and Case Studies

The project requires Python 3 and the file to start is `fault_inject.py`. You can get some help on the parameters by running `python3 fault_inject.py -h`. This help and some additional information are included in the README file at the root of the project on GitHub.

File	Statements	Missing	Coverage
swiftool/___init___.py	0	0	100%
swiftool/config.py	7	0	100%
swiftool/diff_ui.py	70	70	0%
swiftool/faults_inject.py	59	4	93%
swiftool/utils.py	25	0	100%
swiftool/faults/___init___.py	0	0	100%
swiftool/faults/faultmodel.py	10	0	100%
swiftool/faults/flp.py	22	0	100%
swiftool/faults/jcc.py	85	0	100%
swiftool/faults/jmp.py	84	0	100%
swiftool/faults/nop.py	27	0	100%
swiftool/faults/z1b.py	13	0	100%
swiftool/faults/z1w.py	19	0	100%
	421	74	82%

Table 3.2: Statement code coverage of the tests.

The project contains an **examples** folder with a **README** that describes a practical case for each fault model currently implemented. The case of study is a small C program where some instructions are edited with different fault models. This explanation file contains for each fault model, the command used to perform the right operation as well as the result in term of decompiled assembly code of the operation on the executable file. One of the example is presented hereafter.

Example editing a **JMP** instruction

The code to create the example executable is the following :

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

It is compiled with `gcc -o gcd gcd.c` to an executable. The disassembly of this executable is :

```
...
000000000000005fa <gcd>:
```

```

5fa: 55                push    %rbp
5fb: 48 89 e5          mov     %rsp,%rbp
5fe: 48 83 ec 10       sub     $0x10,%rsp
602: 89 7d fc          mov     %edi,-0x4(%rbp)
605: 89 75 f8          mov     %esi,-0x8(%rbp)
608: 83 7d f8 00       cmpl    $0x0,-0x8(%rbp)
60c: 75 05            jne     613 <gcd+0x19>
60e: 8b 45 fc          mov     -0x4(%rbp),%eax
611: eb 13            jmp     626 <gcd+0x2c>
613: 8b 45 fc          mov     -0x4(%rbp),%eax
616: 99               cltd
617: f7 7d f8          idivl   -0x8(%rbp)
61a: 8b 45 f8          mov     -0x8(%rbp),%eax
61d: 89 d6            mov     %edx,%esi
61f: 89 c7            mov     %eax,%edi
621: e8 d4 ff ff ff   callq   5fa <gcd>
626: c9               leaveq  %eax
627: c3               retq
...

```

To apply a JMP fault model on the instruction at the offset 611, the tool is used with :

```
python3 ../swifitool/faults_inject.py -i gcd -o gcd_jmp -a x86 JMP 0x611 0x600
```

This produce an new edited executable file. Disassembling the results gives :

```

...
00000000000005fa <gcd>:
5fa: 55                push    %rbp
5fb: 48 89 e5          mov     %rsp,%rbp
5fe: 48 83 ec 10       sub     $0x10,%rsp
602: 89 7d fc          mov     %edi,-0x4(%rbp)
605: 89 75 f8          mov     %esi,-0x8(%rbp)
608: 83 7d f8 00       cmpl    $0x0,-0x8(%rbp)
60c: 75 05            jne     613 <gcd+0x19>
60e: 8b 45 fc          mov     -0x4(%rbp),%eax
611: eb ed            jmp     600 <gcd+0x6>
613: 8b 45 fc          mov     -0x4(%rbp),%eax
616: 99               cltd
617: f7 7d f8          idivl   -0x8(%rbp)
61a: 8b 45 f8          mov     -0x8(%rbp),%eax

```

```

61d:  89 d6                mov    %edx,%esi
61f:  89 c7                mov    %eax,%edi
621:  e8 d4 ff ff ff      callq  5fa <gcd>
626:  c9                  leaveq
627:  c3                  retq
...

```

The jump instruction at 0x611 has been successfully edited to point to the wanted target (0x600). Note that this new target is not aligned with the existing instructions. This will force the processor to decode the instructions differently and execute a completely different program.

3.5 Performance

A benchmark applying a JMP fault on an executable file of 825 kB was done. It was repeated 1000 times on a system running on a SSD. The entire benchmark took 2.967 seconds which means about 3 ms per fault injection. Among these 3 ms, it was also measured that 53% of this time was devoted to copying the original file. 29% of the time was spent to parse and check the arguments and 13% to read and edit parts of the new file.

This small tool was not built for performances (which does not prevent it to be efficient anyway) and it is more focused on usability. Writing it in a lower-level programming language could speed it off a little bit even if the main bottleneck will always be the system I/O speed since we copy files.

3.6 Discussions

This tool is a first step in the toolchain of this work. It adds interesting features compared to the existing tool. While it has been created for a specific use in this work, it can be used within lots of other contexts. Its ease of use and genericity allow it to be integrated and extended quickly. The unit tests also ensure a good confidence level of correctness in the tool and the implemented models.

A logical evolution of the tool would be the addition of new fault models as well as extending the existing ones. The GUI could also be extended to, for example, allow the user to directly perform faults on the binary without the need to use the command line and parameters.

CHAPTER 4

Simulation through LLVM with Lodin

Now that we have several variants of the binary executable with some faults in them, we need to verify properties on each variant. To perform this step, the first idea was to use an existing tool called Lodin that could check properties on LLVM bytecode. To be able to feed our variants to Lodin, we have to decompile them into LLVM bytecode first.

4.1 Previous Work and Background

This section presents some backgrounds and the tools used in the chapter.

4.1.1 LLVM bytecode

LLVM bytecode [18] is an intermediate representation format used by the LLVM compiler. It is produced from source code and then translated into binaries for the corresponding platform.

4.1.2 Lodin

Lodin [19] [20] is an under construction software that allows simulating LLVM code. It can evaluate, relatively to a specified confidence level, properties on some characteristics of the code. Typically on the temporary LLVM registers values. It emulates the behavior of the code dynamically by understanding its semantics and produces traces on which properties can be checked afterwards.

4.1.3 Microsoft llvm-mctoll

Llvm-mctoll [21] is a recently created tool able to transform binaries into LLVM IR (or bytecode). It currently supports x86-64 and arm32 Linux/ELF. Also only a subset of the possible binary executables can be translated. For example, C++ binaries or SIMD instructions still need to be supported.

4.1.4 Avast RetDec

RetDec [22] also allow decompiling binaries into LLVM bytecode. It supports a greater range of platforms and has more features. The tool is intended to decompile to C code but since an intermediate step is LLVM bytecode, it can be used in our context.

4.2 Motivation

The objective is to analyze the feasibility of this approach using LLVM as an intermediate step. A toolchain has to be defined and the right tool for each step needs to be selected. The properties also have to be integrated in the process at one or several steps.

4.3 Architecture and Limitations

The planned toolchain is presented in the figure 4.1.

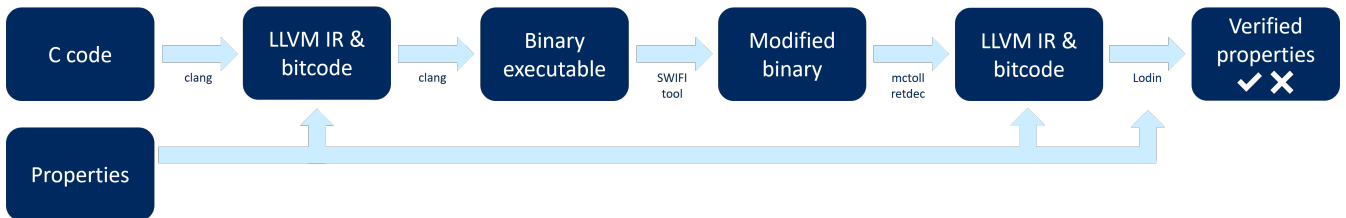


Figure 4.1: Tool-chain using LLVM as an intermediate step.

Going from the C code to LLVM bytecode to the binary executable is a standard flow when compiling the program with the `clang` compiler [23]. After that, we use the SWIFI tool presented in the previous chapter to insert faults in this executable. Then, we try to go back to LLVM bytecode. This transition is done using two promising tools : `llvm-mctoll` and `retdec`. The last step consists of finding a way to combine the properties and the new LLVM code and let Lodin performs the

verification. Since Lodin needs to receive the properties in function of the LLVM identifiers (except for some predefined ones), we have to get the mapping between properties given in function of C variables to properties given in function of LLVM identifiers. This is not a trivial task and several tricks can be found to overcome it. An idea was to put the variable of interest inside a global volatile memory location and then locate its identifier in LLVM more easily.

However before developing an entirely automated pipeline, we performed some tests with each tool, these tests showed the many drawbacks of this approach.

Decompilation Limitations The two decompilers often could not produce the equivalent LLVM code for some binary files due to their own limitations. The generated LLVM is also not easy to read by a human and is sometimes much longer than if it was generated from the C code. This makes the manual matching of the C variable to the LLVM identifiers almost impossible in many cases.

Properties Translation To translate properties based on C to ones based on LLVM, we must use some tricks due to the fact that there exists no surjective relation (no straightforward way to go from one to the other). For example, when a C program is compiled to assembly, a variable might be placed at a defined offset in the stack or only stay in registers. Without the help of the compiler, it becomes really hard to tell what has been placed where. Coming back from assembly to LLVM keeps the issue (or make it worse) because LLVM also works directly with memory or temporary registers. The trick of using a recognizable memory location to store the used variable is not foolproof and is not really scalable. It also requires editing the original program which cannot always be possible.

Lodin Limitations Lodin doesn't already support all the LLVM instructions which again limit the set of programs we can check with this kind of tool-chain. Lodin also has a really strict way of defining properties which makes it harder to port C properties. However thanks to some modification by its author, Lodin could be able to check properties defined inside the C file before compilation and decompilation like the ones used in LLBMC [24]. The assertion added in the C code would then have to be carried correctly through all the toolchain and decompiled correctly.

4.4 Discussions

All these limitations make us realize that this way of working was feasible but not the most efficient and the easiest way to proceed. Taking into account that the decompilers and Lodin cannot currently work on the complete instructions set

and that the properties translations are not straightforward, we decided not to go deeper with this method and looked for a nicer approach to the problem.

CHAPTER 5

Simulation Through GDB With Plasma

The goal of this chapter is to describe the conception of a tool able to simulate an executable and check properties on it. The system used to check properties will be Plasma and the executable simulation will be done with the GDB debugger.

5.1 Previous Work And Background

This section presents the tools and the related background used in this chapter.

5.1.1 GDB

The GNU Debugger [25] [26] is a tool able to analyze the behavior of a running program. It can stop the execution of this one and retrieve information on the current state. It works on many platforms and with many programming languages like C, C++, Go, Rust, ... If the running program has been compiled with debugging information, GDB can link elements in the code (like variables) with the memory location during the execution.

As quick taste of the possibilities, one can use GDB to follow the evolution a variable or evaluate more complex expressions.

```
(gdb) print x
$1 = 5
(gdb) next
```

```

(gdb) print x
$2 = 10
(gdb) print x*x+sizeof(int)
$3 = 104

```

This kind information and the relatively easy way of accessing it can be useful in the construction of a simulation tool.

5.1.2 Plasma

Plasma Lab [27] [28] [29] is a statistical model checking platform. It is based on the notion of plugins. Some plugins are responsible of the model and perform the simulations. Other ones manage the requirements and check them on traces produced previously. And the last ones manage the entire process and aggregate the results statistically. The three types of plugins (simulators, checkers and algorithms shown in 5.1) can all be integrated into Plasma independently by implementing a specific JAVA interface.

Plasma orchestrates all the plugins so the user can perform model checking either via a graphical interface or via the command line.

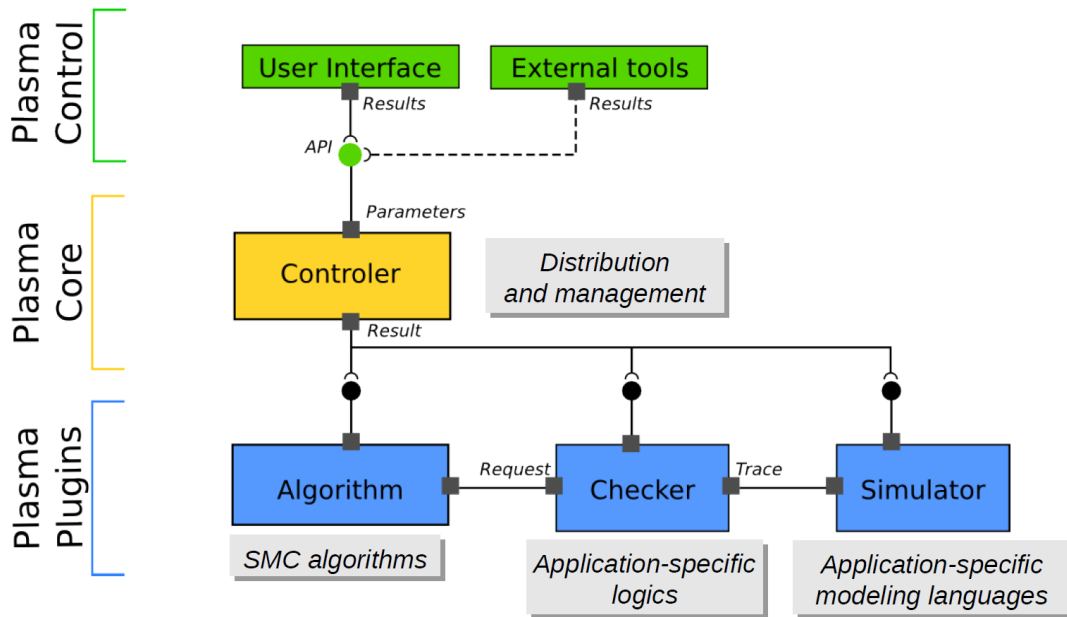


Figure 5.1: Plasma architecture with plugins (taken from [29]).

A set of plugins of all sorts are already present in Plasma. Two of them used in this work are the Monte Carlo algorithm and the Bounded linear temporal logic checker.

5.1.3 Monte Carlo Algorithm

The Monte Carlo Algorithm [29] is one of the core concepts behind statistical model checking. It can evaluate the probability of a property being satisfied knowing the results of the evaluation of this property on a set of independent simulations. Each simulation produces a trace w_i with i going from 1 to N , the number of simulations. A property φ satisfiability on a path is written $w_i \models \varphi$. The estimation of the property's probability of being satisfied on the system is :

$$\bar{\gamma} = \frac{\sum_{i=1}^N \mathbf{1}(w_i \models \varphi)}{N} \text{ where } \mathbf{1}(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{otherwise.} \end{cases}$$

This can be used to get an idea of the property holding or not. To get a result with a fixed confidence level, the number of simulation N must be chosen accordingly. Instead of giving a number of simulation to the algorithm, one can give the precision ϵ and the confidence level δ such that $P(|\bar{\gamma} - \gamma| \geq \epsilon) \leq \delta$. The number of simulation required is computed by the algorithm using the Chernoff-Hoeffding bound [30] :

$$N = \lceil \frac{\ln 2 - \ln \delta}{2\epsilon^2} \rceil$$

This algorithm is already implemented as a plugin in Plasma and its workflow can be represented as in 5.2.

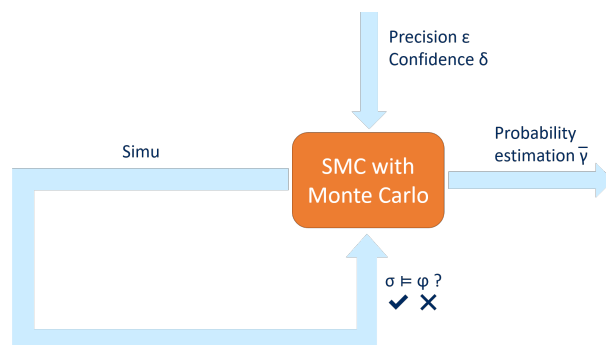


Figure 5.2: Monte Carlo algorithm plugin workflow.

5.1.4 BLTL Properties

Properties can be defined using the bounded linear temporal logic. A plugin allowing defining and checking formulas on the traces generated by a model is already available in Plasma [29]. The "bound" characteristic is due to the fact that we work with traces extracted from a simulation. If a property is defined on an infinite path length, the simulation would never terminates and keeps generating states to feed the checker. Only a symbolic approach could check properties on an infinite trace. To counter this problem, a maximum number of states on which the property is checked is added to the logic.

The syntax and semantics used in this chapter are detailed hereafter.

Syntax

```
Formula ::= 'F <=' Bound Formula
          | 'G <=' Bound Formula
          | Formula 'U <=' Bound Formula
          | Formula 'W <=' Bound Formula
          | 'X' ('<=' Bound)? Formula
          | Formula '&' Formula
          | Formula '|' Formula
          | Formula '=>' Formula
          | '!' Formula
          | '(' Formula ')'
          | 'true'
          | 'false'
          | Numerical (Operator Numerical)?
```

```
Bound ::= Numerical
```

```
Numerical ::= ident | number
```

```
Operator ::= '<' | '<=' | '!=' | '=' | '>=' | '>' | '+' | '-' | '*' | '/'
```

Semantic A BLTL property is checked on a trace written as $w = (s_0, t_0), (s_1, t_1), \dots, (s_L, t_L)$. s_j is the state containing all the identifiers and their numerical values at this state. t_j is the time ($\in R_{\geq 0}$) associated to this state (if the system does not have the notion of time, a number showing the order of the states is used). $w^j = (s_j, t_j), \dots, (s_L, t_L)$ denotes the portion of the trace starting from j (included).

The truthfulness of the formulas can be decided using the following rules.

$w \models F \leq B\varphi$	iff $w \models \text{true } U \leq B\varphi$
$w \models G \leq B\varphi$	iff $w \models \neg(F \leq B\neg\varphi)$
$w \models \varphi_1 U \leq B\varphi_2$	iff $\exists i, t_0 \leq t_i \leq t_0 + B$ and $w^i \models \varphi_2$ and $\forall j, 0 \leq j < i, w^j \models \varphi_1$
$w \models \varphi_1 W \leq B\varphi_2$	iff $w \models (\varphi_1 U \leq B\varphi_2) \mid (G \leq B\varphi_1)$
$w \models X \leq B\varphi$	iff $\exists i, i = \max(j \mid t_0 \leq t_j \leq t_0 + B)$ and $w^i \models \varphi$
$w \models X \leq \varphi$	iff $w^1 \models \varphi$
$w \models \varphi_1 \& \varphi_2$	iff $w \models \varphi_1$ and $w \models \varphi_2$
$w \models \varphi_1 \mid \varphi_2$	iff $w \models \varphi_1$ or $w \models \varphi_2$
$w \models \varphi_1 \Rightarrow \varphi_2$	iff $w \models \neg \varphi_1 \mid \varphi_2$
$w \models \neg \varphi$	iff $w \not\models \varphi$
$w \models \text{true}$	always
$w \models \text{false}$	never
$w \models N_1 Op N_2$	when the arithmetic operation evaluates to true or $\neq 0$

identifiers are set to their current value in the state

5.2 Motivations

The previous try using LLVM showed its limitations quickly. This part tries a more straightforward way. One way to simulate an executable is to simply run it but this doesn't allow to retrieve properties easily at a precise moment. As presented previously, there exists tools that are able to run programs, stop them when wanted and retrieve values on the current state. These tools are commonly called debuggers and a popular one [31] : GDB, has been used by many developers across the world since years and contains a rich and useful set of features. The main idea of this section is to use this power of debuggers to run and retrieve data about the execution and then perform statistical checking on this data.

On the other side, the statistical checking part can be performed by the modular tool Plasma. Integrating a software simulation to it would allow using the existing algorithms and checking tools already available. We have at our disposal two really powerful tools that are maybe just waiting for one thing, collaborate with each other to make more powerful things possible. Using all the power of GDB as a simulator integrated into the Plasma ecosystem opens the door to a quantity of new possibilities.

An extension to the existing BLTL logic plugin to define properties on programs is also created.

This chapter also defines a new plugin to combine and automate the fault injection process to the verification process.

The earlier toolchain is therefore simplified using the Plasma platform and new plugins.

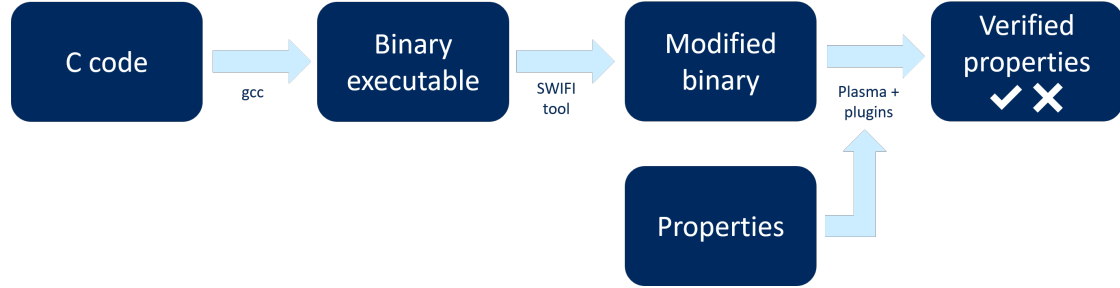


Figure 5.3: Tool-chain using Plasma plugins.

5.3 Software Architecture

The plugins implement the requirements to be integrated in Plasma. They are developed in Java and separated in several packages. All the plugins can be compiled, tested and integrated using the Maven compilation system. There are three plugins each implementing a different module type and able to communicate with each other. The source code of the plugin as well as some examples and documentation are available on a GitHub repository ¹. Some metrics highlights are shown in the table 5.1. Each line contains the sum of the files inside the packages. The length is in bytes.

Package	Length	Lines	LOC	Classes	Methods
gdbmiapi.*	32143	744	644	9	48
softwarebltlchecker	14715	404	352	2	21
softwaresimulator	21878	652	539	5	58
swifigenerator	19962	618	532	6	48
	88698	2418	2067	22	175

Table 5.1: Metrics about the project packages of Plasma plugins.

¹<https://github.com/chenoja/software-analysis-plasma-plugins>

The developed plugins are :

A Simulator Plugin Also called a model plugin. It allows to simulate the execution of binaries using GDB and retrieve values in order to build an execution trace. The simulator is composed of two packages, one is the actual plugin and can work with any kind of program simulator provided that it implements a small interface. The other part is the GDB debugger API that implements this interface and links the simulator with the GDB process.

A Requirement Checker Plugin This one extends the existing BLTL plugin by allowing using expressions on the program names (variable, functions, ...) directly inside a property.

An Algorithm Plugin A new plugin that can perform insertions of faults on the binary, simulate the variants and check properties on them. It uses the previous SWIFI tool to generate all the possibilities of variants using the desired combination of fault models.

5.3.1 Software Simulator Plugin

The structure of the plugin is represented in the figure A.1. The plugin implements all the necessary classes to be usable from Plasma as a plugin. The interfaces required by Plasma are shown in the blue package. The red package implements all the Plasma classes needed for a simulator plugin. The green package implements everything related to GDB interfacing.

Plasma Plugin Implementation

The `softwaresimulation` package implements a `SoftwareSimulatorFactory` responsible for creating an instance of a `SoftwareSimulator` according to the configuration. The `SoftwareSimulator` has all the methods to be able to perform a Plasma simulation and produce a trace of `SoftwareState` containing some `VariableIdentifier`. The plugin uses a generic interface to the debugger. In this work, the interface is implemented to use GDB but it could be done for other debugger like the Java Debug Interface for Java software.

The `softwaresimulation` also handles the user configuration and is responsible for watching the values of interest during the execution of the simulation.

GDB Application Programming Interface

GDB does not provide an easy accessible API with a module or anything that can be integrated quickly. However, it does provide a structured language usable for input and output communication with a GDB instance. This interface is called GDB/MI [32]. Since no existing library was viable to use through JAVA, the plugin implements its own parser and own generator for MI messages in order to communicate with the GDB process via the standard system I/O streams.

The parsing is performed using the ANTLR library. The GDB/MI syntax is defined in a `g4` file used by ANTLR to generate parser classes. Those classes then allow to retrieve elements of the string MI response pragmatically.

GDB Manipulation

To control GDB from JAVA, the simulator launch it in a separate process with the MI interpreter option. Each time an operation needs to be done by GDB, the translated command in MI is sent to the GDB `stdin`. The simulator then waits for an output readable on the `stdout`. The output is a confirmation with or without data depending on the command. This response is then parsed by the ANTLR generated code and the simulator can assert the response is valid and retrieve the data of interest.

Simulator Interface Implementation

The simple GDB API built here implements the `Simulator` interface to be usable from the plugin. Each method required is implemented using the right GDB command. For example, the `nextInstruction` method will send the next instruction command (`ni` for the classical user, `-exec-next-instruction` here with the MI option) to GDB. The methods to evaluate expression in C (like variable names, ...) use the `-data-evaluate-expression` MI command.

In addition, the GDB simulator implements some specific values. Those are properties available only on some type of debugger. In this case, the properties are the CF and OF flags of the processor and a predicate telling if the stack memory has been edited between two steps of the simulation. The stack is copied from the current stack pointer of the function and the stack end and then compared to the previous copy. The bounds are obtained through the operating system by looking at the process information so this feature is only available on Linux. The defined predicate is useful for detecting vulnerabilities like buffer overflow.

Plugin workflow

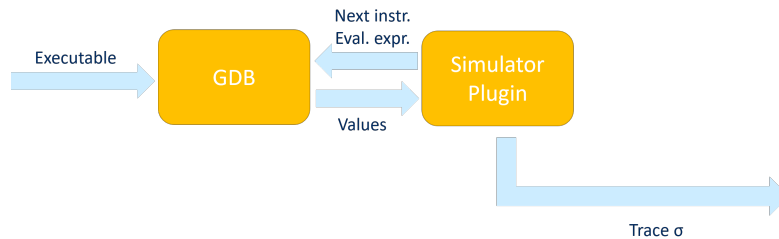


Figure 5.4: Executable simulator plugin workflow.

The figure 5.4 shows the flow of the information for the complete plugin. We receive an executable and using GDB, produce a trace of execution.

The figure A.2 shows a more detailed version of the flow of the operation through a sequence diagram. Plasma uses our simulator plugin and this one uses the GDB simulator which uses GDB in a Java sub-process. The black messages represent the JAVA call and return flows. The red one is a command executed on the system and the yellow messages are MI data sent to GDB.

To sum up the situation, the simulator executes one step of the function of interest, gather all needed values to fill the current state and put that state in a list. The simulator repeats this until the end of the function and exit the program. All the saved states form the trace which is analyzed by the checker.

5.3.2 Extended BLTL Plugin

The second plugin extends the existing Plasma BLTL plugin. The new features needed are the capability to have expressions about program elements inside the properties and a way to extract traces of variants that violated the property.

The existing BLTL logic can already use variables defined in the simulator plugin like the line number and the program counter. The syntax of expression added here are the one used by the debugger expression.

```
G <= 100 line != 10
```

The syntax extension will allow putting expressions that evaluate to a boolean value between two '\$' characters such that property like this becomes possible.

```
G <= 100 $table[5] == 15$
```

The extension brings one new definition to the syntax :

```
Numerical ::= '$' debugger_expression '$'
```

If the debugger expression contains '\$', they can be escaped by writing two of them. The debugger expression has the same semantic as an identifier. It is evaluated by the debugger and the result (1.0/0.0/NaN) is placed in a variable of which the identifier will replace the expression in the BLTL.

When a property is violated, the plugin will display the trace of execution that led to the invalid state. The traces contain all the watched variables and expressions and is structured in an XML format. These traces can also be exported into files to be used later by someone or by another tool.

The plugin flow shown in figure 5.5 takes as input a trace and a property and tell if this last one holds on the trace. If its not the case, it displays or saves the trace.

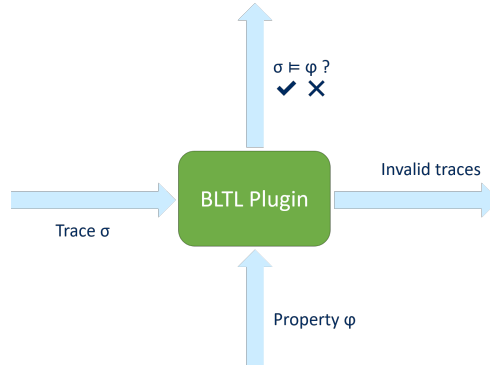


Figure 5.5: Extended BLTL plugin workflow.

5.3.3 Putting Everything Together

When putting all the plugins together, we get a fully working workflow managed by Plasma (in figure 5.6). This global process takes as input an executable, a desired precision and confidence and a set of properties. Several simulations are run and the properties are evaluated on the produced traces. The results are aggregated by Monte Carlo to produce a probability estimation of the property satisfiability. The invalid traces encountered are also exported.

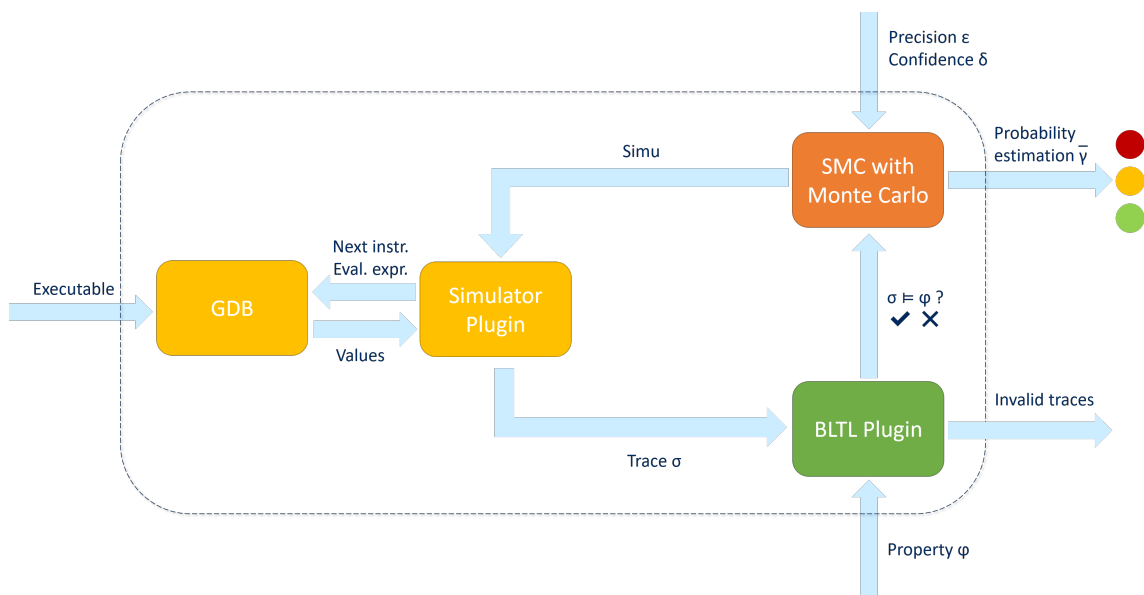


Figure 5.6: Complete plugins workflow with Monte Carlo.

5.3.4 SWIFI Integration Plugin

The third plugin is an algorithm implementation launching the simulations and gathering the checker results. It works with both of the previous plugins. Instead of performing multiple simulations of the same software, it performs a simulation on every generated variant of the software of interest. It uses the SWIFI tool presented earlier to generate all possible combination of faults given a configuration. On each of these variants, it uses the simulator to produce a trace and then the checker to validate or not this trace. It then reports the results in three categories: if the property was successfully verified, if not or if the checker was not able to check the property. The last one can happen when the fault injection makes the software unable to run, makes the software loop infinitely or prevents the debugger from accessing the internal values of the program.

This last plugin was the last link in the chain. It allows to find potential fault injection vulnerabilities in software. Potential because the plugin will only perform one simulation of each candidate. It allows to already perform a selection on the candidates. For example, if the goal of the software analysis is to find whether a bank system will leak important information upon fault attack, all variants that crash can be eliminated and the remaining ones can still be validated later using the Monte Carlo algorithm with a defined precision and confidence level if necessary (i.e., if there is visible non-determinism making the simulation vary). Its complete workflow is shown in figure 5.7.

5.3.5 Tests

Unit tests are not really useful in this case as the plugins implements the Plasma interfaces and are therefor not usable without the orchestration performed by Plasma. The Plasma tool is also mainly based on a graphical interface which is harder to tests automatically.

To ensure to plugin works correctly on interesting situations, a set of examples based concrete cases are present on GitHub. They use the entire workflow and also ensure the plugin integrate correctly with Plasma. Some of them are presented in the next part.

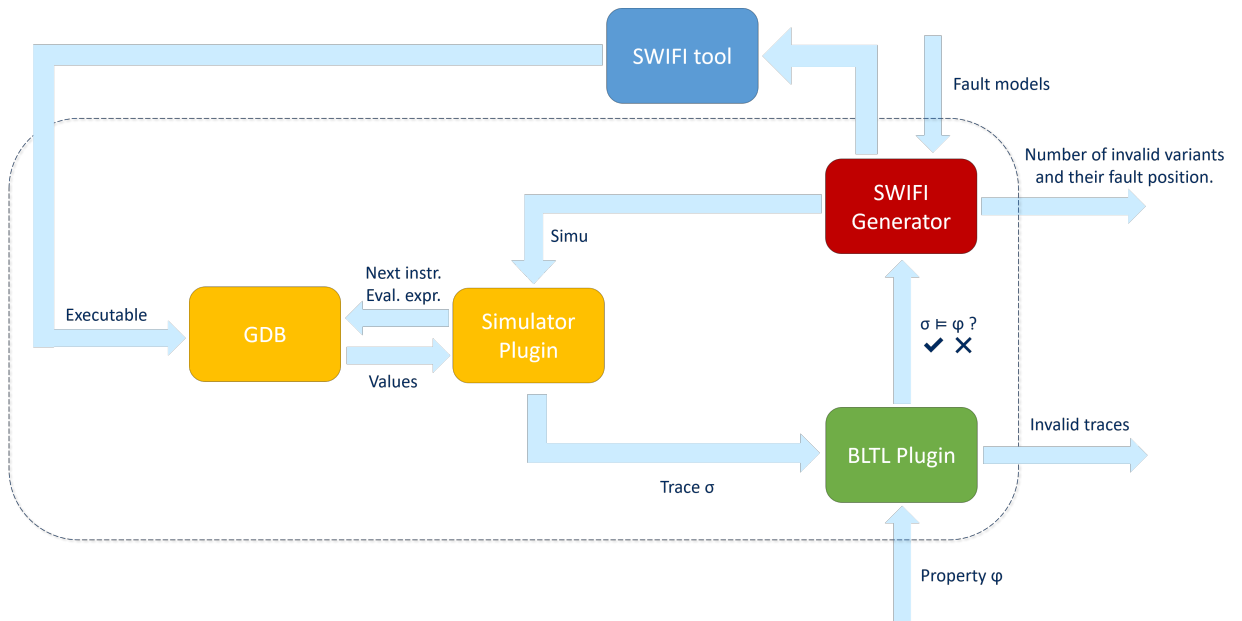


Figure 5.7: Complete plugins workflow with SWIFI.

5.4 Utilization and Case Studies

There is two ways of using the plugins. The first one consists in performing statistical model checking of one executable. This can be done using the simulator plugin, the extended BLTL plugin and the Monte Carlo algorithm. The second one allows checking fault injection vulnerabilities by finding variants of the executable that might not satisfy the specifications. This is done using the simulator plugin, the extended BLTL plugin and the SWIFI algorithm. Some documentation on how to configure each plugin is available on the GitHub repository. All the presented examples are available on GitHub to be fully reproducible.

5.4.1 Statistical model checking

This example is based on a program possibly vulnerable to buffer overflows (reading or writing outside the range of a buffer) [33]. This program also contains some non-determinism which makes the detection harder. The source code is the following :

```
28  ...
29  int main(int argc, char **argv) {
30      int x = 42;
31      char buf[8];
32      memset(buf, '\0', sizeof(buf));
33      srand(time(NULL));
34      int r = rand() % 14; //overflow on x when r = 13
35      printf("%d\n", r);
36      strncpy(buf, "dddddddddddddddddddddddddd", r);
37      if (x == 100) {
38          printf("secret");
39      } else {
40          printf("%s\n", buf);
41      }
42      return 0;
43  }
```

Running this program should at first sight never print "secret" and always print a random number of "d". However a problem occurs when `r` is greater than 8 as the buffer will be filled over its capacity. At one point when `r` is greater than 13, it will write on the memory location of the `x` variable and change its value (the memory structure is more detailed on the GitHub version if needed). When `r` equals 13, the variable `x` will take the value 100 due the buffer overflow and the fact that the "d"

ASCII value is 100. This will allow the program flow to go inside the `if` condition and display the secret.

Regular testing methods might not have the chance to get the bug as the `r` value might never be set to 13 during the tests. Our statistical model checking approach comes to the rescue.

First, we need to create a new simulator (or model) configuration inside Plasma. The configuration contains the path to the executable, the function we want to simulate, the variables we want the debugger to watch for us and some other settings (described in the user manual on GitHub).

```
1 executable = "../software-analysis-plasma-plugins/examples/  
  ↪ buffer_overflow/buffer_overflow"  
2 function = "main"  
3  
4 [simulator]  
5 name = "gdb"  
6  
7 [simulator.options]  
8 CF = true  
9 OF = true  
10 STACK_M = true  
11 gdb_path = "/usr/bin/gdb"  
12  
13 [variables]  
14 x = "x"  
15 r = "r"  
16 b0 = "(int) buf[0]"  
17 b12 = "(int) buf[12]"
```

The `b12` variables defined here with a C expression will in fact have the same value as `x` since they point to the same memory location.

The properties ensuring the desired behavior of the program can be defined in several ways using a BLTL configuration. We can for example ask that the variable `x` never reaches 100.

```
1 [traces]  
2 type="print"  
3
```

```

4 [BLTL]
5 G <= #100 $x!=100$

```

Or ask that the execution flow never reaches the line 38.

```

1 [traces]
2 type="print"
3
4 [BLTL]
5 G <= #100 line!=38

```

We also ask the plugin to display the traces that led to a violation. The LTL bound is set to a sufficient (higher than the execution trace lenght) value : 100.

The last step consists of launching the Monte Carlo Algorithm with the desired precision and confidence :

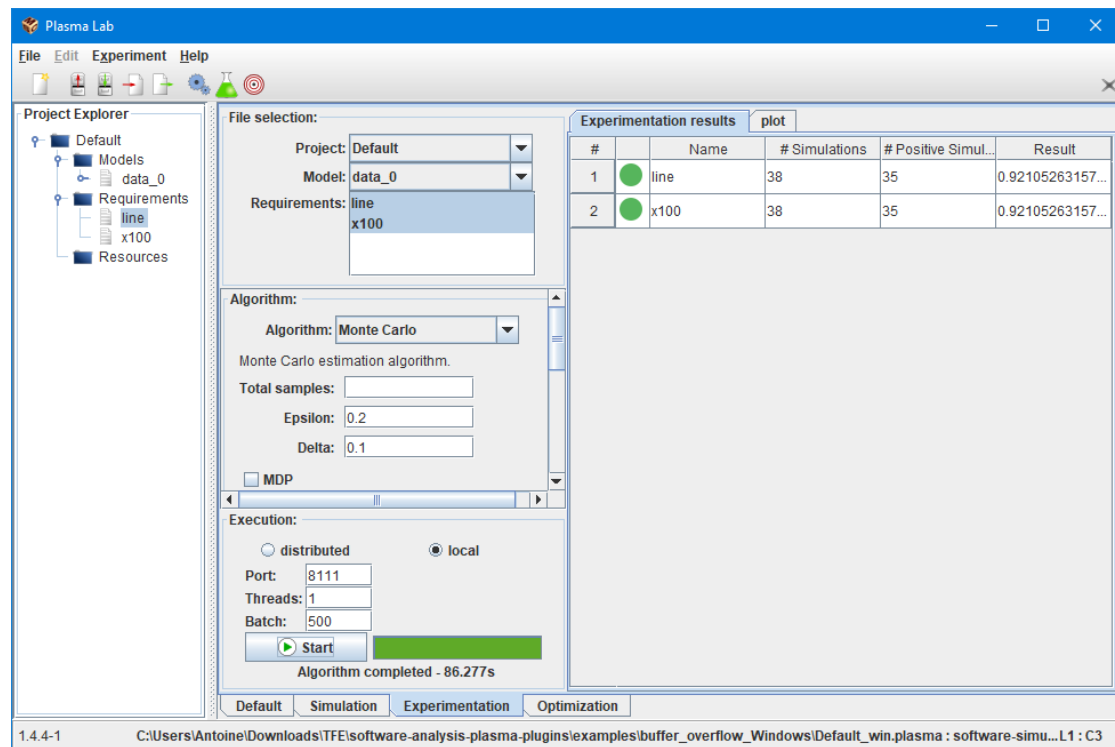


Figure 5.8: Buffer-overflow detection using statistical model checking.

We can see that the properties are violated in some cases. The execution trace is printed in XML format on the console accordingly. The results show us that the properties are satisfied only in 92% of the executions which corresponds to the 13 chances over 14 to not have the bug.

5.4.2 Finding faulted variants candidates

The goal here is to find if some fault injected in the software could lead to a bad behavior. We take a small and simple code example :

```
1  #define SECRET_CODE 4242
2
3  volatile int one = 1;
4
5  int main() {
6      if (one == 0)
7      {
8          return SECRET_CODE;
9      }
10     else
11     {
12         return 0;
13     }
14 }
```

This code return 0 in every situation as the condition always evaluates to false. The volatile variable is there to prevent the compiler from removing the unreachable code part (as it cannot make the assumption that the value always stays at one anymore). The code is compiled with GCC and with the debug option enabled (`gcc -g -o hello hello.c`). This program does not have any non-determinism and no parameters so running it multiple times will always lead to the same behavior. The goal here is not to use statistical model checking on it and check its correctness but we want to determine if the behavior of this code can change when targeted by a fault injection. The fault used in this example is a Z1B which can happen in real conditions if the memory erases a byte for example.

First a software simulation configuration is created :

```
1 executable = "../software-analysis-plasma-plugins/examples/
   ↪ buffer_overflow/buffer_overflow"
2 function = "main"
3
4 [simulator]
5 name = "gdb"
6
7 [simulator.options]
8 CF = false
```

```

9  OF = false
10 STACK_M = false
11 gdb_path = "/usr/bin/gdb"
12
13 [variables]

```

Then the correct behavior is asserted with the following BLTL configuration :

```

1  [traces]
2  type="print"
3
4  [bltl]
5  G <= 100 line != 8

```

Asking the algorithm to apply one Z1B fault, it will test all variants with a byte zeroed at each possible positions (9104 ones since this is the size of the file).

Figure 5.9: Configuration of the SWIFI generator algorithm.

#		Name	#Simulations	#Validated	#Crashed/Could n...	#Invalidated
1		test	9104	7364	1738	2

Figure 5.10: Results on each possible variants.

The results can be either valid meaning the simulation trace of this variant satisfied the property, not validated or unknown as the program might crash or GDB might not be able to read the debug information correctly. In our case, we can ignore the crashed variants as they won't leaks the secret code. Analysing the two variants violating the property :

- The first one is due to the `one` variable being set to zero and making the condition `true`.
- The second one is due to the DWARF information being altered and making GDB display the wrong line number and so violating the property. This is a false alert as it doesn't really leaks the secret code.

Generally speaking, the numbers obtained from the algorithm could not be fully trusted as for example an error in the DWARF section could make a property holds while its not true in reality. A variant can also be dangerous but not tested due to the unreadable DWARF. The goal of this workflow is therefor more to help the user to find candidates that would need to be checked more precisely later.

5.5 Performance

Some measurements have been performed on the `swifi_integer_overflow` example. The run was analysed with Java Flight Recorder to understand where the plugins spend most of their time.

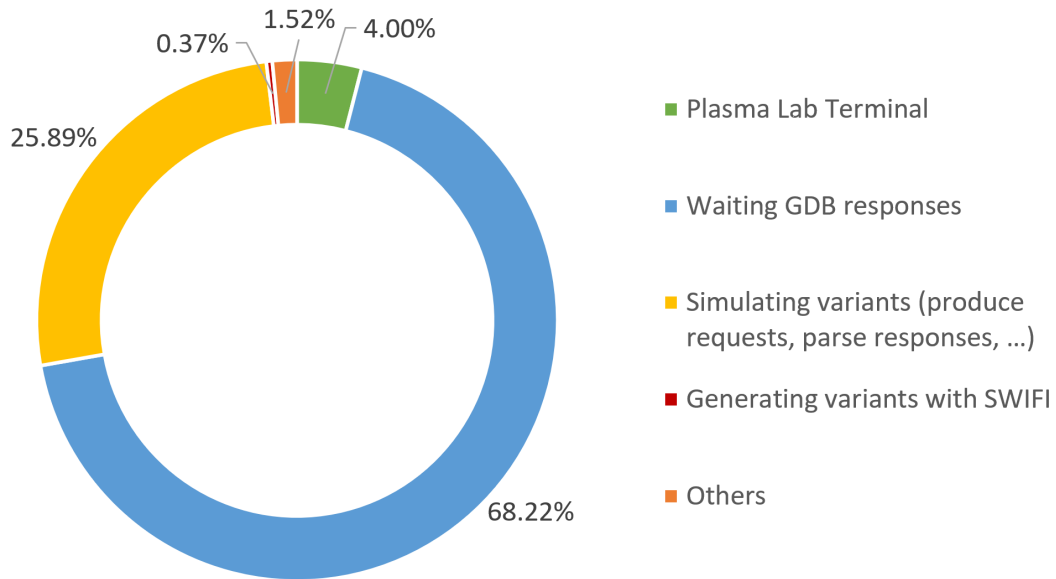


Figure 5.11: SWIFI genrator algorithm time distribution analysis.

The most time consuming part was the execution of GDB where the plugins spend 68% of their time waiting for a result from the debugger. The other big

part is also related to communication and consist of all the the simulator actions like storing traces or parsing responses from GDB. The other parts are almost insignificant and doesn't impact the time consumption that much.

Another interisting experiment that emphasis the cost of GDB in the workflow was performed by replacing the simulation of the executable by simply running it (without retrieving information). The total time to tests all the combinations with GDB is 9m42s and by simply running them is 3m25s. This 2.8 factor is the price to pay to be able to collect all kind of information during the execution and verify interisting properties on them.

5.6 Discussions

Along this chapter, we constructed two complete workflows. The first one intends to validate properties on one program using the Monte Carlo algorithm. The second one helps finding candidates behaving incorrectly when targetted by a fault injection. Two examples showed the feasibility of each approach as well as its limitations.

One of the great advantage of using GDB is the fact that we do not need to edit the original executable file to insert checks or properties. Debuggers exist for a wide range of programming language and so this approach can potentially support a lot more programming languages and systems.

One small disadvantage is the limitation of Plasma to use `Double` values requiring all the variables to fit in a `Double`. Variable containing text strings could be useful for example. Another current limitation is that the plugins can only simulate a function without watching the child or other functions.

CHAPTER 6

Related work

Several SWIFI tool have already been developed. FERRARI, FTAPE, XCEPTION and some other are described in [3]. Most of them performs fault injection at run-time by changing the memory, forcing the CPU to fail or inject congestion on the network.

This previous work [8] by Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet and Axel Legay address the same problem but investigate another process. One main difference is the verification of the requirements. The properties are inserted into the executable as a call to a special library. The binary executable variants are first decompiled to LLVM using MC-Sema and then checked with LLBMC which will simulate the LLVM and intercept the call to the fake library in order to check the property placed in the call.

The idea of simulating a program by using some sort of interface to stop it and retrieve values is not new. One major successful use of this method is the Java Path Finder tool [34]. It is more advanced in terms of simulation as it behaves as a JVM and can save/restore states. It also performs model checking of the software without the need to actually model it in a formal way but its main goal is to find data-races or deadlocks in JAVA applications.

CHAPTER 7

Conclusion And Future Work

The objective of this master's thesis was to verify the reliability of computer systems evolving in hostile environments. To avoid an overly costly evaluation, the goal is to reproduce these conditions locally and to verify the behavior of these software in a monitored environment.

The first step consisted in the creation of a new tool allowing to insert software faults in an executable file in order to reproduce the possible effects of hardware faults. The tool developed is based on the notion of fault models. Each fault model aims to reproduce a set of real cases of failure. This small tool is intended to be relatively simple to use and visual thanks to the graphical interface. It is also intended to be integrated into other workflows and is easy to extend by adding new fault models. A complete example was explained and its performances were measured.

Future work on this tool may for example extend the graphical interface but also add functionalities such as new fault models or more precise targeting of binary instructions. The last one could determine alone the size of an instruction that we want to skip via the help of disassemblers like Capstone [35].

Now that we have candidates with faults, we want to evaluate their behavior relatively to requirements defined for the system. To achieve this goal, a recent approach in the formal verification field called statistical model checking is used.

A first method explored the use of an existing tool called Lodin making it possible to verify statistical models on LLVM code. The variations of the executables should then be decompiled to LLVM bitcode. Several tool performing this steps were tested. Although this approach is possible, it has been put aside because of the many drawbacks such as the limitation of each tool, the complex workflow and the difficulties in using properties based on components of the program code.

Depending on the evolution of each tool, this method could be more feasible in the future. For example, Lodin can be extended to support assertions inserted into the program source code. Another thing that could be done is integrating with an existing compiler in order to, aside from translating the code to LLVM, also translate the properties so they can work with elements available during the LLVM simulation. The decompilation of the variants would need to produce a similar LLVM structure to the original code such that the properties could still work on them.

The second method explored combines the power of debuggers like GDB and the modularity of a statistical model checking tool like Plasma. The statistical model checking being based on a series of simulations, these can be carried out by GDB which turns out to be the perfect tool to produce a trace step by step by evaluating each expression necessary for the properties during execution.

A first plugin based on using GDB to produce a execution trace of an executable has been integrated to Plasma. We saw that this plugin was really powerful by allowing to express many properties directly using C or GDB syntax. Although, in some cases the Plasma structure can limit the features possibilities.

To improve the simulation speed, we could only focus on interesting states and not force the debugger to stops at each instruction.

The plugin is currently limited to execute one function of the program. A improvement could allow specifying a set of functions and sub-functions the simulator needs to watch. Other aspects can also be monitored during the simulation such as signals or system calls.

This plugin can be extended in many ways in order to focus on other aspects of programs verification than functional testing. Simulating a multi-threaded application could be possible by watching the actions of each thread with the debugger. Testing several threads interleaving orders could help finding bugs like data-races and deadlocks.

Other debuggers can also be integrated to the simulator plugin to extend the supported program format. For example integrating the Java Debug Interface to support JAVA software.

The second plugin added features to the existing BLTL logic of Plasma. LTL

properties can now contains GDB expression based on C code for example. When an invalid trace is found, its now either printed or saved to a file allowing further inspections by the programmer.

The features of BLTL can be further developed by allowing to express properties about operations order to work with multi-thread programs and detect data-races conditions.

The last plugin links the fault injected variants with the software simulator and checker. An example demonstrated the feasibility of the interactions and was able to find interesting consequences of fault injection on a software.

In the future, the plugin can be extended to support more types of fault combinations and a more intelligent generation. For example, applying a fault on the DWARF data does not have any impact on the program execution and is not necessary to test.

To conclude, this work presented two complete workflows to check reliability issues about software systems and showed their feasibility. The two key points of this master's thesis are the simulation aspect which allows to be faithful with respect to the original system and the statistical model checking approach allowing obtaining mathematically asserted properties.

APPENDIX A

UML diagrams of the executable simulator plugin

References

- [1] P. Mehlitz and J. Penix, “Expecting the Unexpected - Radiation Hardened Software”, en, in *Infotech@Aerospace*, Arlington, Virginia: American Institute of Aeronautics and Astronautics, Sep. 2005, ISBN: 978-1-62410-069-7. DOI: 10.2514/6.2005-7088. [Online]. Available: <http://arc.aiaa.org/doi/10.2514/6.2005-7088> (visited on 06/14/2020).
- [2] *Emp high power electromagnetic weapons railguns microwaves*, en-us, Library Catalog: www.militaryaerospace.com, Nov. 2019. [Online]. Available: <https://www.militaryaerospace.com/power/article/14072339/emp-high-power-electromagnetic-weapons-railguns-microwaves> (visited on 06/14/2020).
- [3] H. Ziade, R. Ayoubi, and R. Velazco, “A Survey on Fault Injection Techniques”, en, vol. 1, no. 2, p. 16, 2004.
- [4] S. K. Bukasa, R. Lashermes, J.-L. Lanet, and A. Legay, “Let’s shock our IoT’s heart: ARMv7-M under (fault) attacks”, in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018, Hamburg, Germany: Association for Computing Machinery, Aug. 2018, pp. 1–6, ISBN: 978-1-4503-6448-5. DOI: 10.1145/3230833.3230842. [Online]. Available: <https://doi.org/10.1145/3230833.3230842> (visited on 06/01/2020).
- [5] T. Given-Wilson, N. Jafri, and A. Legay, “The State of Fault Injection Vulnerability Detection”, en, in *Verification and Evaluation of Computer and Communication Systems*, M. F. Atig, S. Bensalem, S. Bliudze, and B. Monsuez, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 3–21, ISBN: 978-3-030-00359-3. DOI: 10.1007/978-3-030-00359-3_1.

- [6] H. Madeira, D. Costa, and M. Vieira, “On the emulation of software faults by software fault injection”, en, in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, New York, NY, USA: IEEE Comput. Soc, 2000, pp. 417–426, ISBN: 978-0-7695-0707-1. DOI: 10.1109/ICDSN.2000.857571. [Online]. Available: <http://ieeexplore.ieee.org/document/857571/> (visited on 06/08/2020).
- [7] M. Kooli and G. Di Natale, “A survey on simulation-based fault injection tools for complex systems”, en, in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, Santorini, Greece: IEEE, May 2014, pp. 1–6, ISBN: 978-1-4799-4972-4. DOI: 10.1109/DTIS.2014.6850649. [Online]. Available: <http://ieeexplore.ieee.org/document/6850649/> (visited on 06/08/2020).
- [8] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, “An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT – Extended Version”, en, Apr. 2017. [Online]. Available: <https://hal.inria.fr/hal-01400283>.
- [9] C. Baier and J.-P. Katoen, *Principles of Model Checking*, en. MIT Press, Apr. 2008, Google-Books-ID: 5dvxCwAAQBAJ, ISBN: 978-0-262-30403-0.
- [10] A. Legay and B. Delahaye, “Statistical Model Checking : An Overview”, en, *arXiv:1005.1327 [cs]*, May 2010, arXiv: 1005.1327. [Online]. Available: <http://arxiv.org/abs/1005.1327> (visited on 06/08/2020).
- [11] A. Pnueli, “The temporal logic of programs”, en, in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Providence, RI, USA: IEEE, Sep. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. [Online]. Available: <http://ieeexplore.ieee.org/document/4567924/> (visited on 06/08/2020).
- [12] M. J. Eager and E. Consulting, “Introduction to the DWARF Debugging Format”, en, p. 10, 2007.
- [13] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, “A comparison of static, dynamic, and hybrid analysis for malware detection”, en, *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, Feb. 2017, ISSN: 2263-8733. DOI: 10.1007/s11416-015-0261-z. [Online]. Available: <http://link.springer.com/10.1007/s11416-015-0261-z> (visited on 06/14/2020).
- [14] N. Jafri, *Nisrine/Fault-Injection-Tool*, original-date: 2017-10-15T19:44:41Z, Jan. 2020. [Online]. Available: <https://github.com/nisrine/Fault-Injection-Tool> (visited on 05/22/2020).

- [15] *Stack Overflow Developer Survey 2020*, Library Catalog: insights.stackoverflow.com. [Online]. Available: https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020 (visited on 06/14/2020).
- [16] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, en, Library Catalog: [software.intel.com](https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-2a-2b-2c-and-2d-instruction-set-reference-a-z.html). [Online]. Available: <https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-2a-2b-2c-and-2d-instruction-set-reference-a-z.html> (visited on 05/22/2020).
- [17] A. Ltd, *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition / ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, en, Library Catalog: [developer.arm.com](https://developer.arm.com/docs/ddi0406/cd/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition). [Online]. Available: <https://developer.arm.com/docs/ddi0406/cd/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition> (visited on 05/22/2020).
- [18] *LLVM Bitcode File Format — LLVM 10 documentation*. [Online]. Available: <https://www.llvm.org/docs/BitCodeFormat.html> (visited on 06/14/2020).
- [19] D. Bøgstød Poulsen, *Lodin / Software Verification of LLVM Code*, en-GB, Library Catalog: lodin.boegstedpoulsen.dk. [Online]. Available: <https://lodin.boegstedpoulsen.dk/> (visited on 05/23/2020).
- [20] A. Legay, D. Nowotka, D. B. Poulsen, and L.-M. Tranouez, “Statistical Model Checking of LLVM Code”, en, in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds., vol. 10951, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 542–549, ISBN: 978-3-319-95581-0. DOI: 10.1007/978-3-319-95582-7_32. [Online]. Available: http://link.springer.com/10.1007/978-3-319-95582-7_32 (visited on 06/09/2020).
- [21] S. B. Yadavalli and A. Smith, “Raising binaries to LLVM IR with MC-TOLL (WIP paper)”, in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019, Phoenix, AZ, USA: Association for Computing Machinery, Jun. 2019, pp. 213–218, ISBN: 978-1-4503-6724-0. DOI: 10.1145/3316482.3326354. [Online]. Available: <https://doi.org/10.1145/3316482.3326354>.
- [22] *RetDec :: Home*. [Online]. Available: <https://retdec.com/> (visited on 05/23/2020).

- [23] *Clang C Language Family Frontend for LLVM*. [Online]. Available: <https://clang.llvm.org/> (visited on 06/14/2020).
- [24] *LLBMC: The Low-Level Bounded Model Checker: Usage*. [Online]. Available: <http://llbmc.org/usage.html> (visited on 05/22/2020).
- [25] *GDB: The GNU Project Debugger*. [Online]. Available: <https://www.gnu.org/software/gdb/> (visited on 06/14/2020).
- [26] *Debugging with GDB: Top*. [Online]. Available: <https://sourceware.org/gdb/onlinedocs/gdb/> (visited on 06/14/2020).
- [27] C. Jegourel, A. Legay, and S. Sedwards, “A Platform for High Performance Statistical Model Checking – PLASMA”, en, in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Flanagan, and B. König, Eds., vol. 7214, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 498–503, ISBN: 978-3-642-28755-8. DOI: 10.1007/978-3-642-28756-5_37. [Online]. Available: http://link.springer.com/10.1007/978-3-642-28756-5_37 (visited on 06/09/2020).
- [28] B. Boyer, K. Corre, A. Legay, and S. Sedwards, “PLASMA-lab: A Flexible, Distributable Statistical Model Checking Library”, en, in *Quantitative Evaluation of Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, K. Joshi, M. Siegle, M. Stoelinga, and P. R. D’Argenio, Eds., vol. 8054, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–164, ISBN: 978-3-642-40195-4. DOI: 10.1007/978-3-642-40196-1_12. [Online]. Available: http://link.springer.com/10.1007/978-3-642-40196-1_12 (visited on 06/09/2020).
- [29] A. Legay, S. Sedwards, and L.-M. Traonouez, “Plasma Lab: A Modular Statistical Model Checking Platform”, en, in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, T. Margaria and B. Steffen, Eds., vol. 9952, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 77–93, ISBN: 978-3-319-47165-5. DOI: 10.1007/978-3-319-47166-2_6. [Online]. Available: http://link.springer.com/10.1007/978-3-319-47166-2_6 (visited on 05/22/2020).
- [30] *Hoeffding’s inequality*, en, Page Version ID: 957989849, May 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hoeffding%27s_inequality&oldid=957989849 (visited on 06/14/2020).

- [31] *The 20 Best Linux Debuggers for Modern Software Engineers*, en-US, Library Catalog: www.ubuntupit.com, Jan. 2020. [Online]. Available: <https://ubuntupit.com/best-linux-debuggers-for-modern-software-engineers/> (visited on 06/14/2020).
- [32] *Debugging with GDB: GDB/MI*. [Online]. Available: https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI (visited on 06/14/2020).
- [33] *Buffer overflow*, en, Page Version ID: 956538856, May 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Buffer_overflow&oldid=956538856 (visited on 06/14/2020).
- [34] K. Havelund and T. Pressburger, “Model checking JAVA programs using JAVA PathFinder”, en, *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, Mar. 2000, ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s100090050043. [Online]. Available: <http://link.springer.com/10.1007/s100090050043> (visited on 06/09/2020).
- [35] Capstone, *The Ultimate Disassembly Framework*, Library Catalog: www.capstone-engine.org. [Online]. Available: <http://www.capstone-engine.org/> (visited on 06/14/2020).
- [36] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay, “An automated and scalable formal process for detecting fault injection vulnerabilities in binaries”, en, *Concurrency and Computation: Practice and Experience*, vol. 31, no. 23, e4794, 2019, ISSN: 1532-0634. DOI: 10.1002/cpe.4794. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4794> (visited on 06/01/2020).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl