Summary

The risk of fault injections has increased due to the Internet of Things [20]. These fault injections include attacks such as bits being flipped to zero or one. It has been shown that this type of fault injection attack can be used to bypass the PAM mechanism (Pluggable Authentication Modules). This report concerns the detection of vulnerabilities in RISC-V programs due to bitflip fault injection attacks. We utilize the Fault-Injection and Simulation Secure Collection (FISSC) to evaluate our solution. This collection contains PIN checkers, where each program implements a different number of countermeasures to ensure a certain level of security.

To detect these vulnerabilities we formalize the RISC-V instruction set architecture by defining its syntax and semantics, as well as the program configuration of a RISC-V program. Furthermore, we have chosen to focus on bitflips occurring in the registers described by our fault model.

The RISC-V formalization is utilized to create a static analysis that can detect bitflips. We have chosen to create a value-set analysis, which for each program point determines a set of possible values. To ensure soundness and monotonicity, we utilize the monotone framework. We have therefore created a domain which is a complete lattice. This domain contains the possible values for each register. In addition to this, it includes the possible bitflips that could happen in each register. The domain also includes the possible flipped and non-flipped values stored in the heap. We also created transfer functions for each RISC-V instruction, which describes the effect the instruction has on the environment based on the domain.

To show that the defined value-set analysis is able to detect bitflip vulnerabilities in registers, we implement a tool based on the theory above called BitflipperVild. BitflipperVild takes a RISC-V program as input and creates a CFG, which is then analyzed with our value-set analysis. To achieve a more efficient analysis, we have created a backward slicing module.

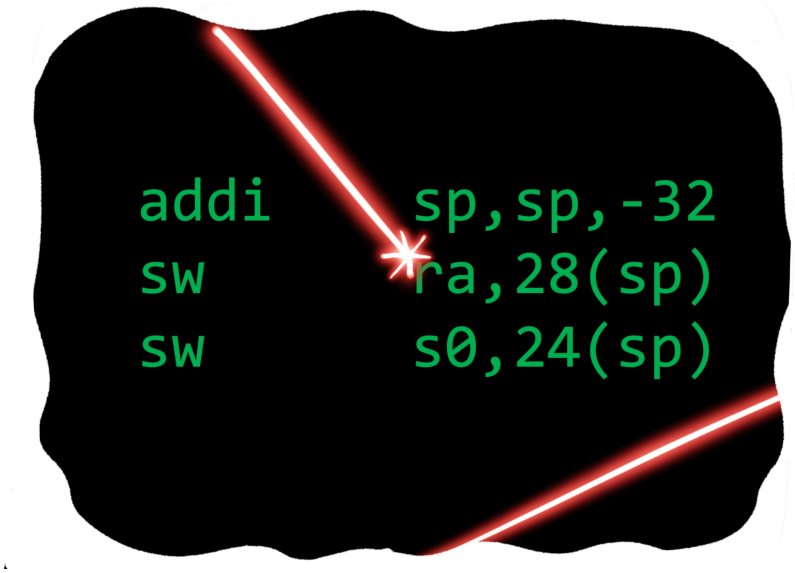
BitflipperVild is evaluated using FISSC and is able to show that the countermeasures implemented in the collection affects the amount of vulnerabilities in the programs. To do this, we compare different versions of the same program and show that some bitflips in a less secure version could result in reaching the privileged point. We were able to show that all countermeasures preventing bitflips occurring in registers limited ways of reaching a privileged point.

Value-set Analysis for RISC-V

Detecting Bitflip Vulnerabilities

Master thesis - P10

cs-23-ds-10-02



Aalborg University
The Technical Faculty of IT and Design
Department of Computer Science

Department of Computer Science
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Value-set Analysis in RISC-V

Theme:
Specialization in Software

Project Period:
Spring Semester 2023

Project Group:
cs-23-ds-10-02

Participant(s):
Ida Thoft Christiansen
Lena Said Ernstsen

Supervisor(s):
René R. Hansen
Danny B. Poulsen

Copies: 1

Page Numbers without appendix: 60
Page Numbers with appendix: 63

Date of Completion:
June 9, 2023

Abstract:

Bitflip attacks have been shown to be a real-life security issue, as demonstrated on the PAM mechanism [13]. Thus, this report concerns a proof of concept for detecting bitflip vulnerabilities in RISC-V programs using a value-set analysis. This is achieved by formalizing the RISC-V language and creating fault models describing different bitflip attacks. Based on this formalization, a value-set analysis is defined that utilizes the monotone framework. In the analysis, we have defined a domain, which has been shown to be a complete lattice, as well as monotone transfer functions for all instructions. The defined analysis is implemented as a tool called BitflipperVild. BitflipperVild is shown to be able to detect all register-relevant bitflip vulnerabilities in the programs found in the collection FISSC [9]. Thus, we are able to use our tool to show that some possible bitflips can result in an attacker reaching a privileged point without authentication.

Contents

Preface

This report is the 10th-semester master thesis written at Aalborg University within the Department of Computer Science. We would like to thank our supervisors René R. Hansen and Danny B. Poulsen for their continuous supervision during our studies. In this project, we have defined and developed the tool BitflipperVild, which can be found in [5].

In Chapter 1, we introduce and motivate a real-life problem concerning bitflip attacks. In Chapter 2, we revise the syntax and semantics describing RISC-V, as well as the conventions of this ISA. In Chapter 3, we define the scope of our project by defining a set of requirements and assumptions. In Chapter 4, we describe the monotone framework, as well as define the theory behind the value-set analysis which can detect bitflip vulnerabilities. In Chapter 5, we describe the implementation of the different modules in our analysis. This implementation is then evaluated in Chapter 6, using FISSC and performing benchmark tests. Lastly, we draw a conclusion in Chapter 7 and discuss future work in Chapter 8.

Aalborg University, June 9, 2023

Lena Said Ernstsen
lsaid18@student.aau.dk

Ida Thoft Christiansen
ichris18@student.aau.dk

Chapter 1

Introduction

The risk of fault injection has increased, due to the number of hardware devices introduced by Internet of Things [20]. Fault injection attacks are physical attacks that can be used to bypass authentication [6]. A type of fault injection attack is bitflip fault injection, where the attack targets a single bit using techniques such as electromagnetic pulses or lasers.

This has been shown to be an actual problem. An experiment using the tool ChipWhisperer [13] has been conducted on a PAM mechanism (Pluggable Authentication Modules) which is used in operating systems, such as Linux and Unix. [21] The ChipWhisperer tool injects bitflips into the system clock, which can result in instructions being skipped. In this experiment, the author managed to bypass the authentication itself, which granted them administrator privileges.

An example of a program which would be vulnerable to similar attacks is the PIN checker seen in Listing 1.1. In the PIN checker, the bits in the registers are targeted. This is the verifier used in a PIN checker called VerifyPIN found in the Fault-Injection and Simulation Secure Collection (FISSC) [9]. VerifyPIN compares each byte in the actual PIN with the PIN entered by a user and returns 1 if the PIN is correct.

Listing 1.1: An insecure PIN checker vulnerable to bitflips [9].

```
1 BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size) {
2     for(int i = 0; i < size; i++) {
3         if(a1[i] != a2[i]) return 0;
4     }
5     return 1;
6 }
```

There are several ways a single bitflip could result in the function returning 1 with an incorrect PIN. One way is to avoid entering the `for` control structure in line 2 by changing either variable `i` or `size`. Assuming that the PIN size is 4, it is possible with a single bitflip to flip `size` from 4 to 0 or flip `i` from 0 to a number equal to or greater than 4, which both make the condition in the for-loop false. Another possibility is to change the integer being returned in the for loop in line 3. Thus, if the integer is changed from 0 to 1, the function returns as if the entered PIN was correct. Lastly, if only one of the digits is incorrect, it could be possible to flip a bit in the incorrect PIN to result in the correct PIN. For example, if the correct PIN is 1234 and the entered PIN is 1235, the last digit could be flipped from 5 to 4.

As mentioned, the example is taken from FISSC [9], which is a public collection of C programs that includes different versions of VerifyPIN. FISSC can be used to analyze the robustness of code against fault injection attacks, due to the countermeasures applied in the programs, which try to counter fault injection attacks. In the collection, the number of countermeasures differs for each new version, which makes it possible to use the collection for the evaluation of analysis tools. Therefore, the problem addressed in this master thesis is being able to analyze and find the effect of bitflips on the PIN checkers in FISSC.

Chapter 2

Formalizing RISC-V

To reduce the scope of our project, we have chosen to focus on analyzing bitflips in the RISC-V Instruction Set Architecture (ISA). The FISSC C-programs will therefore be compiled using a RISC-V cross-compiler to get the equivalent RISC-V instructions. In our previous work [8] we defined the syntax and semantics of the RISC-V ISA. In this chapter, we will revise and expand these.

2.1 RISC-V

RISC-V [1] is an open-source ISA, which defines instructions for both 32-bit and 64-bit architectures. It has a minimal and flexible design, as it only defines a minimal base instruction set that can be extended further based on a given use case. Furthermore, there are established extensions already defined by the developers of RISC-V. In our previous work [8], we chose to focus on the 32-bit architecture to narrow our scope, and we will continue focusing on this architecture for this project. The domain for the 32-bit architecture includes 32-bit values, where the formalization of this domain is the following:

$$\text{Val} = \mathbb{B}^{32}$$

Where $\mathbb{B} = \{0, 1\}$ and 32 indicates that a total of 32 zeroes and ones is used to represent a given value. The instructions of this architecture have different formats, which describe what type of value is stored in the 32 bits for each instruction. An example of this format can be found in Figure 2.1.

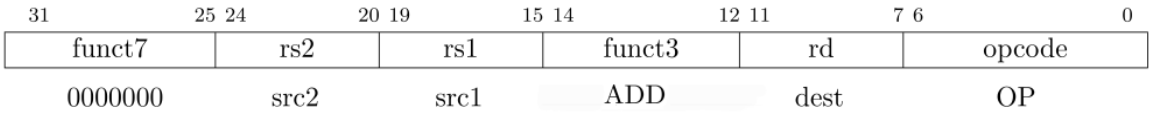


Figure 2.1: The format of a 32-bit RISC-V `ADD` instruction describing how the bits are allocated.

All instructions have an opcode which determine the instruction type. In the example, `OP` is the opcode, since it is an `ADD` instruction. Instructions can include up to two different source registers and one destination register. Moreover, instead of a source register, instructions can include immediate values. For example, instead of `rs2`, which represents a value from a register, a constant could be stored in an immediate, turning the instruction into an `ADDI` instruction.

RISC-V specifies 32 different registers that can be utilized by a program, which we can formalize as follows:

$$\text{Register} = \{\texttt{x0}, \ldots, \texttt{x31}\}$$

The registers have different predefined purposes, such as the stack pointer and function arguments. However, due to the flexible design of the RISC-V ISA, these registers can be used for other purposes if needed. The complete list of registers, their assembly names, and their predefined purposes can be found in Appendix A.

RISC-V also includes additional pseudo-instructions, which can be rewritten into base instructions. Some common pseudo-instructions include `mv` and `jr`, which are translated into `addi` and `jalr`, respectively. See section 2.2 for their semantics. The complete list of pseudo-instructions and their equivalent base-instructions can be found in [22].

2.2 Syntax and Semantics

In order to construct the bitflip analysis, we revise the syntax and semantics for the 32-bit RISC-V ISA, which we first described in [8]. In this section we define which components constitute a program configuration for a RISC-V program. Furthermore, we categorize the different types of instructions and create an abstract syntax rule for each instruction. Lastly, we define the semantics for each of these rules.

As described in [8], different fault models can be defined to specify where a bitflip could occur. The fault models describe a bitflip in the instructions, in the program counter, and in the registers. To ensure that our analysis is able to capture all of these types of faults, these three components must be included in our program configuration. We thus define a configuration as follows:

$$\begin{aligned} \text{Configuration} = \text{Program} \times \text{Heap} \times \text{Registers} \times \text{pc}, \\ \text{where} \quad \text{Program} = \text{Addr} \to \text{Instr} \\ \text{and} \quad \text{Heap} = \text{Addr} \to \text{Val} \\ \text{and} \quad \text{Registers} = \text{Register} \to \text{Val} \end{aligned}$$

Program maps from addresses to instructions, as an address in a RISC-V program refers to a specific instruction. Heap maps from addresses to values, since the heap represents the memory, where given an address a specific value can be retrieved. Registers maps one register to a value as the contents of a register is a set of binary values.

Using the defined configuration, we can generalize the different RISC-V instructions into abstract instructions and define their syntax. The following table is taken from [8].

Instr	::=	`OP` rd, rs_1, rs_2	Perform OP on rs_1 and rs_2 and store in rd
	|	`OPI` rd, rs_1, $\texttt{imm}_{12}$	Perform OPI on rs_1 and $\texttt{imm}_{12}$ and store in rd
	|	`LOAD` rd, `imm`(rs_1)	Load the value at offset + rs_1 and store in rd
	|	`STORE` rs_2, `offset`(rs_1)	Store the content of rs_2 at offset + rs_1
	|	`LUI` rd, $\texttt{imm}_{20}$	Copy $\texttt{imm}_{20}$ to rd
	|	`AUIPC` rd, $\texttt{imm}_{20}$	Sum pc with $\texttt{imm}_{20}$ and store result in rd
	|	`LI` rd, $\texttt{imm}_{32}$	Load an immediate value into rd
	|	`CMP` rs_1, rs_2, L	Compare rs_1 with rs_2 and if true transfer control to L
	|	`JAL` rd, `offset`	Call subroutine at `offset`. Save next pc in rd
	|	`JALR` rd, `offset`(rs_1)	Invoke subroutine at `offset` + rs_1. Save next pc in rd
	|	`NOP`	Move pc to the next instruction

The `OP` and `OPI` abstract instructions describe arithmetic and logical operations on contents of registers and immediates, respectively. To express this, a way to semantically evaluate the specific type of operation must be introduced. We specify this in Appendix B, where for example the semantic evaluation of `ADD v1, v2` is `v1 + v2`. Using these semantic evaluation functions, we define the following semantics for `OP` and `OPI`:

$$[\text{OP}] \quad \frac{P(pc) = \texttt{OP}\ rd,\ rs_1,\ rs_2 \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \qquad x = [\![OP]\!](R(rs_1),\ R(rs_2))$$

$$[\text{OPI}] \quad \frac{P(pc) = \texttt{OPI}\ rd,\ rs_1,\ \texttt{imm}_{12} \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \qquad x = [\![OPI]\!](R(rs_1),\ \texttt{imm}_{12})$$

In the conclusion, both the `OP` and `OPI` abstract instructions update the destination register, rd, to contain the result of performing the operation on the values. This value, x, is calculated in the side condition by calling the semantic evaluation function for the abstract instruction. Furthermore, both instructions also move the program counter, pc, to the next instruction.

The `LOAD` and `STORE` instructions describe interactions between the heap and the registers. The difference between them is that `LOAD` fetches the values from the heap and saves it in the register, whereas `STORE` fetches the contents of the register and saves it in the heap.

$$[\text{LOAD}] \quad \frac{P(pc) = \texttt{LOAD}\ rd,\ \texttt{imm}(rs_1) \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \qquad x = H(R(rs_1) + \texttt{imm})$$

$$[\text{STORE}] \quad \frac{P(pc) = \texttt{STORE}\ rs_2,\ \texttt{offset}(rs_1) \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H[x \mapsto R(rs_2)], R, pc'\rangle} \qquad x = R(rs_1) + \texttt{offset}$$

The `LOAD` instruction finds the value, x, in the heap by accessing the contents of the source register at the offset specified by the immediate. This is expressed by the side condition. The value, x, is then used to update the destination register. The `STORE` instruction calculates the address, x, in the side condition using the source register and an offset. However, instead of updating the register in the conclusion, it updates the heap at address, x, with the contents of the second source register, $rs2$.

The LUI and LI abstract instructions are similar in that they both update the destination register with an immediate in the conclusion. The difference is found in the immediate, where LUI concatenates 12 zeroes onto the immediate to turn it into a 32-bit value. The concatenation is described by $||_{\mathbb{B}}$ in the side condition.

$$[\text{LUI}] \quad \frac{P(pc) = \texttt{LUI}\ rd,\ \texttt{imm}_{20} \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \qquad x = \texttt{imm}_{20} \,||_{\mathbb{B}}\, 0_{12}$$

$$[\text{LI}] \quad \frac{P(pc) = \texttt{LI}\ rd,\ \texttt{imm}_{32} \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto \texttt{imm}_{32}], pc'\rangle}$$

The CMP instruction describes a comparison between the contents of two registers. To distinguish between the different types of comparisons, we have defined a semantic evaluation function, see Appendix B. To express that the result of performing a comparison is either true or false, we have defined two semantic rules for the CMP instruction:

$$[\text{CMP-T}] \quad \frac{P(pc) = \texttt{CMP}\ rs_1,\ rs_2,\ L \qquad pc' = addr(L)}{\langle P, H, R, pc\rangle \implies \langle P, H, R, pc'\rangle} \quad \text{if } [\![CMP]\!](R(rs_1),\ R(rs_2)) \to tt$$

$$[\text{CMP-F}] \quad \frac{P(pc) = \texttt{CMP}\ rs_1,\ rs_2,\ L \qquad pc' = pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R, pc'\rangle} \quad \text{if } [\![CMP]\!](R(rs_1),\ R(rs_2)) \to ff$$

The difference between the rules is what the program counter, pc, is updated to in the premise. If the semantic evaluation of the comparison is true, then CMP-T is applied, and the program counter is updated to the address of the specified label, L. Otherwise, CMP-F is applied and the program counter is updated to be the next instruction.

The two jump instructions, JAL and JALR, handle direct jumps and indirect jumps, respectively. This can be seen in the premise, where JAL updates the program counter with an offset, while JALR updates the program counter with both an offset and the contents of a given source register. However, both instructions save the next instruction in the destination register.

$$[\text{JAL}] \quad \frac{P(pc) = \texttt{JAL}\ rd,\ \texttt{offset} \qquad pc' = pc + \texttt{offset}}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \qquad x = pc + 2$$

$$[\text{JALR}] \quad \frac{P(pc) = \texttt{JALR}\ rd,\ \texttt{offset}(rs_1) \qquad pc' = pc + y}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle}$$

where $x = pc + 2$
and $y = \texttt{offset} + R(rs_1)$

The last instructions are AUIPC and NOP. AUIPC concatenates 12 zeroes to the immediate and then adds the resulting value to the program counter, saving it in the destination register, rd. The NOP instruction only updates the program pointer to point to the next instruction,

as seen in the conclusion of its semantic rule.

$$[\text{AUIPC}] \quad \frac{P(pc) = \texttt{AUIPC}\ rd,\ \texttt{imm\_20} \qquad pc' \mapsto pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R[rd \mapsto x], pc'\rangle} \quad x = pc + (\texttt{imm}_{20} \,||_{\mathbb{B}}\, 0_{12})$$

$$[\text{NOP}] \quad \frac{P(pc) = \texttt{NOP} \qquad pc' \mapsto pc + 2}{\langle P, H, R, pc\rangle \implies \langle P, H, R, pc'\rangle}$$

With the above definition of the abstract syntaxes and their corresponding semantics, the 32-bit RISC-V ISA has been formalized. To understand how these instructions are used in an actual RISC-V program, we will discuss the conventions of a typical RISC-V program.

2.3 RISC-V Conventions

As our goal is to analyze bitflip vulnerabilities in a RISC-V program, we need to understand the conventions used when compiling to RISC-V code. This includes conventions for the structure of a function, as well as conventions for the instructions. The conventions presented in this section are based on observations of RISC-V programs produced by cross-compiling the C-programs found in FISSC using the compiler found at [17].

All RISC-V programs include functions, which are each specified with a label representing the name of the function. Since it is necessary to perform certain instructions for all programs, default functions, such as `start` and `exit`, are always included to ensure correct startup and termination. Typically, there is also a `main` function but it depends on the contents of the program.

A function usually starts with updating the stack and frame pointer to allocate memory for local variables in the given function. The stack pointer is updated by adding an offset to its current value, which represents the amount of memory space that should be allocated for the stack. Additionally, the value of the frame pointer is first stored in the heap before the same offset is applied. Before the function returns, both the stack and the frame pointer are reset to the original values before the offsets were applied. The stack and frame pointer are normally not changed except at the start and the end of the function.

All instructions have an address written as a hexadecimal number. Control transfer functions usually jump only to addresses within the same function or to the first instruction of another function. Moreover, control transfer instructions that branch, such as `blt`, normally refer only to addresses within the same function. An example of a RISC-V function can be seen in Table 2.1.

In this example the function `main` first updates the stack and frame pointer with the offset -16. Afterward, there is a jump to another function at the address `10338`. After the function has been executed, it returns to the next instruction in `main` at line `6`. This instruction stores the content of register `ra` into the heap at the location indicated by `0(s0)`. Lastly, the stack and frame pointer are reset by adding 16 to `sp` and loading the original frame pointer from the heap.

```
1  00010318 <main>:
2    10318:          addi sp,sp,-16
3    1031c:          sw s0,12(sp)
4    10320:          addi s0,sp,16
5    10324:          jal ra,10338
6    10328:          sw ra,0(s0)
7    1032c:          lw s0,12(sp)
8    10330:          addi sp,sp,16
9    10334:          ret
```

Table 2.1: Example of the structure of a function in a RISC-V program.

When analyzing the RISC-V code, we assume that the mentioned conventions are followed. Since the analyzed code is C-code and compiled using a cross-compiler [17] which follows these conventions, this should be trivial. As the syntax, semantics, and convention for RISC-V have been defined, we can specify the focus of our project and determine the necessary requirements to reach our goal prior to defining our bitflip analysis.

Chapter 3

Scope

In our previous work [8], we defined a Value-Set Analysis (VSA) that could determine whether a RISC-V program was vulnerable to bitflips. We also provided a short example, where we analyzed a simple RISC-V program manually to show how our analysis could detect vulnerabilities when implemented. We will use our previous analysis, as well as the syntax and semantics we redefined in Chapter 2, as a foundation for developing an analysis which examines bitflips in a program. Based on this new analysis we will develop a tool that can analyze the programs in FISSC for bitflip vulnerabilities. Henceforth, we will refer to this tool as BitflipperVild. To define the scope of BitflipperVild, we need to specify the requirements, as well as the assumptions regarding the tool.

3.1 Requirements

As mentioned in Chapter 1, the analysis must be able to show that the countermeasures implemented in FISSC prevent bitflip attacks from accessing the privileged point. In this section, we will thus define the requirements we deem necessary to address the problem stated in Chapter 1, and define more concretely which vulnerabilities BitflipperVild should be able to detect.

To analyze a given program, we must visit all program points the necessary number of times. This can be achieved by revisiting the worklist algorithm specified in our previous work [8]. This algorithm requires a Control Flow Graph (CFG) as input in order for the analysis to be performed. Therefore, it is a requirement for BitflipperVild to perform a CFG recovery on a given program. Furthermore, our worklist algorithm should be able to utilize both a backward slicing method and VSA.

To guarantee that the analysis terminates and provides a usable result, the parameters of the analysis should satisfy the properties of the monotone framework. More specifically, the domain should be a complete lattice that follows the ascending chain condition and the transfer functions should be monotone. This is elaborated further in section 4.1.

As mentioned in section 2.1, we have defined three fault models in [8], which describe bitflips occurring in the program counter, the instruction, and the register. When a bitflip occurs in the program counter, a new faulty edge is created in the CFG, which points to the instruction at the flipped program counter. For an instruction, the bitflip occurs in the 32-bit binary encoding of the instruction, which changes the instruction into another instruction. Thus, when the bitflipped instruction is executed, the program is not changed,

as the CFG is not affected. However, the contents of the memory and registers are updated to correspond to the result of executing the bitflipped instruction. Lastly, bitflips in the register change the value being stored in the register. In addition to these three fault models, we can expand the possible scope of the analysis with a fourth fault model describing bitflips occurring in the heap. This fault model can be seen below.

$$[\text{HSEU}] \quad \frac{v = H(x) \qquad v' \equiv_1 v \qquad x \in Heap}{\langle P, H, R, pc\rangle \Longrightarrow_{HSEU} \langle P, H[x \mapsto v'], R, pc\rangle} \tag{3.1}$$

This fault model is similar to the fault model describing bitflips in the registers, see Equation 3.2. However, instead of a bitflip changing a value stored in registers, a bit in a value stored in the heap is flipped. In [8], we chose to focus on the fault model describing bitflips occurring in the values saved in the registers. We will continue focusing on this fault model.

$$[\text{RSEU}] \quad \frac{v = R(x) \qquad v' \equiv_1 v \qquad x \in Register}{\langle P, H, R, pc\rangle \Longrightarrow_{RSEU} \langle P, H, R[x \mapsto v'], pc\rangle} \tag{3.2}$$

The premise of the RSEU fault model states that given a value, v, from the register, a fault has occurred if there is a difference of exactly 1 bit. This is expressed by the operator $\equiv_1$, which describes the Hamming distance [14]. The Hamming distance can be written as Equation 3.3, where v' is the faulty bitstring of the bitstring v.

$$v' \equiv_1 v \quad \text{iff} \quad \exists i \, \forall j \, (i, j \in [0..31] \land v'(i) \neq v(j) \iff i = j) \tag{3.3}$$

The conclusion of the RSEU fault model states that the register, x, is updated to the faulty value, v'. When analyzing FISSC programs, we will thus only examine faults that concern bitflips in a register. Furthermore, as the programs in FISSC consist of multiple functions, the analysis should be interprocedural. This means that the CFG recovery should be able to handle control flow between functions. In addition, the analysis should be RISC-V specific as stated in section 2.1. Thus, it must be possible to determine at which labels an instruction could cause a bitflip vulnerability. This is necessary, as it allows the user to determine where countermeasures should be implemented. To summarize, the requirements in Table 3.1 should be satisfied.

The goal of BitflipperVild is to prove that it is possible to use VSA to detect bitflip vulnerabilities in RISC-V code, and not to create a polished tool that is ready to be released. Based on these requirements, we will first examine whether we can use existing tools as the foundation for BitflipperVild.

3.2 Existing Tools

There already exist tools that can perform static analyses on binaries. As mentioned in our previous work [8], it is possible to use tools like angr [19] and BAP [7] to perform specific tasks, such as CFG recovery and backward slicing. As these two tools have similar features, we will in this section only examine angr to determine whether they can be used

General	Analyze a FISSC program
	Prove countermeasures prevent bitflip vulnerabilities
	Determine value-sets for each program point
	Be RISC-V specific
	Terminate
CFG	Recover a CFG
	Interprocedural
	Resolve indirect jumps
Backward Slicing	Create a slice based on a privileged point
Bitflip Analysis	Detect bitflip vulnerabilities in registers
	Determine at which label a vulnerability can occur

Table 3.1: Summary of the requirements for the bitflip analysis.

to satisfy our requirements.

angr is an open-source binary analysis framework [19]. It integrates state-of-the-art analysis techniques. In this section, we will present the architecture of the angr framework, as well as different state-of-the-art techniques that are implemented in the framework.

Architecture

An overview of angr's architecture can be found in Figure 3.1.

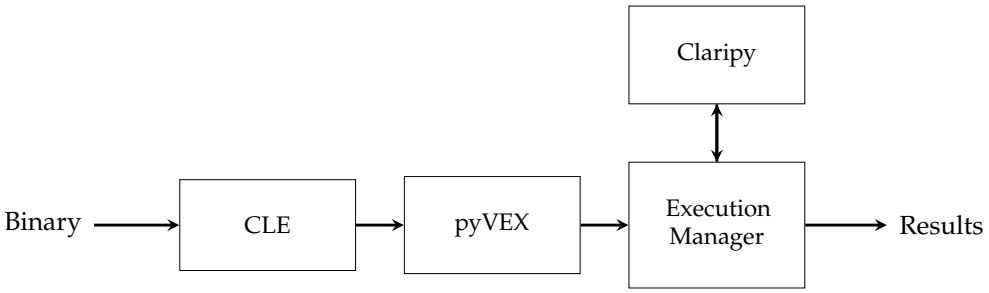


Figure 3.1: An overview of angr's architecture [3].

As input, angr loads a binary executable through the CLE module. This module supports binaries generated on different operating systems, such as Windows and Linux. The binary is represented in a way that makes it easy to work with by loading it into binary objects, which are mapped into a single memory space. However, support for RISC-V binaries requires an extension [4].

The representation is still dependent on the CPU architectures. To address this, the code is transformed into an intermediate representation (IR) called VEX before being analyzed [18]. VEX models instructions in a unified way using different classes of objects, such as expressions and operations. In Listing 3.1, an example can be found of a subtraction operation in RISC-V translated into the VEX IR.

Listing 3.1: Example of the VEX representation produced from the RISC-V instructions [16].

```
1 // RISC-V instructions
2 subi x2, x2, 8
3
4 // Corresponding VEX IR
5 t0 = GET:I32(16)
6 t1 = 0x8:I32
7 t3 = Sub32(t0,t1)
8 PUT(16) = t3
9 PUT(68) = 0x59FC8:I32
```

When the VEX IR is generated, it is possible to perform different analyses through the Execution manager module. This module uses symbolic execution assisted by the Claripy module, which performs SMT solving. angr has a wide range of analysis techniques available, which is still expanding. We will describe some of these analysis techniques more in-depth in the next section.

Static Analysis Techniques

As mentioned in section 3.1, it is necessary to implement a CFG recovery module, a backward slicing module, and a VSA algorithm that can identify bitflip vulnerabilities. angr implements these three modules to some degree.

In angr, the CFG recovery is implemented in the execution manager module. The algorithm runs in a recursive manner in which possible exits are found for each basic block. Exits are basic blocks, which are possible successors to the analyzed block. When these successors are identified, they are added to the CFG accordingly. The process is repeated recursively until no new exits are found.

As mentioned in our previous work [8, pp. 14–15], one of the challenges when recovering CFGs is indirect jumps, as the jump destination is represented by a value in a register or a memory location. Thus, the goal of CFG recovery is to resolve as many indirect jumps as possible to achieve a complete CFG. To achieve a sound CFG, all possible control flow transfers should be included in the graph, resulting in no edges being false negatives. On the other hand, a complete CFG means that all edges included in the graph are possible edges, and thus no edges are false positives.

To achieve higher completeness for the CFG recovery, angr has implemented an algorithm that performs backward slicing to remove unresolved jumps. Additionally, to achieve higher code coverage, angr has implemented another algorithm that performs backward slicing to remove dead code. This algorithm starts by identifying functions in the application, where the direct jumps are recovered by using recursive disassembly. Then different strategies implemented by angr are utilized to identify jump tables and resolve indirect call targets, making it possible to reach more of the code. This in turn increases the code coverage. The graph produced by this algorithm is not complete but is still useful for some analysis techniques.

angr also provides a VSA, which is performed by the Claripy module. This VSA attempts to identify a tight over-approximation of the program state at any given point in the program. These program states represent values in memory and registers. VSA uses an abstract domain to approximate possible values that registers may hold at each program point. The analysis is performed until a fixed-point is reached, and a tight over-approximation is found for all program points. This VSA is a sound analysis but lacks accuracy making it incomplete. Furthermore, the VSA provided by angr does not implement any bitflip analysis, which means we would have to extend the VSA component with a bitflip analysis.

Requirement Realization

As angr does not analyze the RISC-V code directly but instead makes an IR, each instruction being analyzed is changed into multiple instructions. This means that the transfer functions defined in [8, pp. 21–22] and revised in section 4.3, should instead be created based on the basic block in the VEX IR. However, after using pyVEX on a RISC-V program ourselves, we found that one RISC-V instruction did not correspond to a basic block, but that a single basic block contains several instructions. This was also the case for BAP. Using pyVEX and angr would thus require extensive manual adjustments to make it possible to specify in which instructions a bitflip could have occurred. Since this is one of the requirements specified in section 3.1, we deem angr and BAP to be unsuitable to fulfill our specific case.

3.3 Assumptions

To limit the complexity of the analysis and the scope of the implementation several assumptions have been made. This is due to the implementation being a proof of concept and not a polished tool ready to be released. The assumptions include both requirements regarding the RISC-V code, as well as abstractions of the analysis' implementation.

To limit the scope of the proof of concept, the number of different programs that BitflipperVild is able to analyze has been reduced. To analyze programs beyond the scope, we can make adjustments to the RISC-V code before it is analyzed. One of these adjustments is to minimize the number of function calls, where the start and exit functions are removed, which the compiler creates to allocate and deallocate memory to the program. Thus, only the actual functions found in the C program are analyzed.

Following this, because the stack pointer is always summed with an immediate at the start of a function, see section 2.3, we can initialize the stack pointer to be equal to its immediate. A consequence of this is that some values which should be saved in the heap are not saved at the correct address. To mitigate this, the contents of the register can be set to consist of a symbol representing all possible 32-bit values if a load is executed and nothing is saved in the specified location. This ensures that we do not create an under-approximation.

To reduce the complexity, we can use a symbolic value to access the heap instead of calculating the actual address. This can potentially lead to an analysis that is not sound. However, as our focus is on the programs found in FISSC, this is not an issue within our scope. In the collection, we have only found one instance where two different symbolic values refer to the same address. However, the analysis remains sound, as we over-approximate in this case. A way to ensure that local variables cannot be accessed from other methods, the function name should be part of the symbolic value. As an example, the symbolic value for Table 2.1 would be `main:12(sp)` in line 3.

One of the challenges we have observed is due to stack pointers between functions. For each function in a given program, the stack pointer is increased with an offset. When the function ends the stack pointer is decreased with the same offset. However, this becomes a challenge when one function is called multiple times. An illustration of this challenge and our solution can be seen in Figure 3.2.

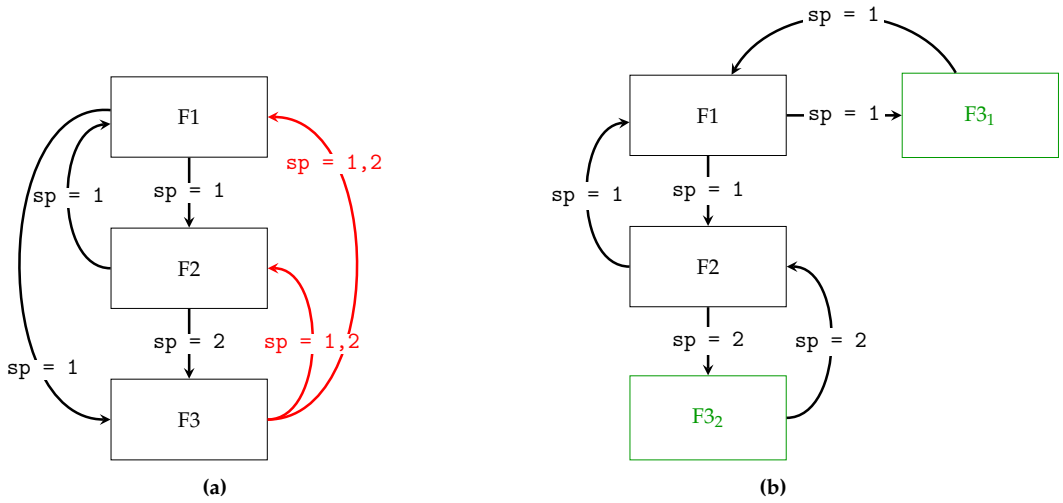


Figure 3.2: (a) is an example of calls between three functions leading to heavy over-approximation, where the problematic calls are marked with red. (b) is an example of how we have solved this challenge, where the changes are marked with green.

In Figure 3.2a, three functions can be seen, where each function increases and decreases the stack pointer by one. The initial value of the stack pointer is 0 in `F1`. The red arrows represent the problematic returns, as two different values for the stack pointer are possible and will be returned to both `F1` and `F2`. This is the case when the stack pointer, received from the caller functions, has different values, as we do not know which value belongs to which function. Therefore, as an over-approximation, both values are returned to both functions. This is a problem, as the worklist will add the nodes in `F1` to the worklist again since the environments have changed with the new stack pointers. When the nodes of `F1` is analyzed a second time, `F3` will return $\texttt{sp} = 1, 2, 3$. The value-set will increase until an overflow occurs leading to a very large over-approximation.

To address this challenge, we can simplify the RISC-V code by introducing duplicates of functions being called multiple times. This will make our analysis polyvariant [15] since the same function will be analyzed multiple times. This will correspond to the function being analyzed once at each call site, which achieves a more precise result. The duplication is illustrated by the green boxes in Figure 3.2b. In this case, `F3` is duplicated and the value-

set for the stack pointers in F1 and F2 have not been changed. This can lead to a fixed point being reached earlier.

Another change that can be applied to the RISC-V code is replacing calls to the <printf> and <scanf> functions with an I/O mapping. Examples of these changes can be seen below.

```
jal ra, 1065c <printf>  ↦  sw _print, 0
jal ra, 10270 <scanf>   ↦  li _scan, 0
```

In the example, the jump instructions are changed to store and load immediate instructions, respectively. For the print instruction, we store 0 in a register called _print, and for the load instruction, 0 is loaded into a register called _scan. The number being stored and loaded, in this case 0, is irrelevant, as it is a replacement for their respective instruction.

To be able to perform arithmetic operations on the values in a program, we can specify certain types, which should be checked during the analysis to ensure that we calculate the correct result. We have assumed that these types only include signed and unsigned decimal numbers, as well as hexadecimal numbers. As observed in section 2.3, addresses are always written as hexadecimal numbers. Therefore, if one of the arguments consists of a hexadecimal value, we can assume that the arithmetic operation is performed on an address. Otherwise, the type is determined based on whether the instruction performs a signed or an unsigned operation.

An assumption we make is that a CFG does not contain bitflips. More specifically, when generating the CFG a bitflip cannot occur. This is because we only consider one bitflip. Thus, if a bitflip has already occurred in the creation of the CFG, no other bitflips can take place. Thus, the following bitflip analysis would be redundant, since none of the calculated values would be possible.

Another challenge can occur when loops access the heap before checking the loop condition. This is illustrated in the below example based on a snippet from FISSC.

```
102b0:  sw zero,-20(s0)   Initializes the heap at address -20(s0) to zero
102b4:  j 102c4           Skips the first addition
102b8:  lw a4,-20(s0)     Loads the first condition argument from heap
102bc:  addi a4,a4,1      Adds 1 to the first condition argument
102c0:  sw a4,-20(s0)     Saves the first condition argument in the heap
102c4:  lw a4,-20(s0)     Loads the value of the first condition argument from the heap
102c8:  addi a3,a3,1      Adds 1 to a variable
102cc:  li a5,1           Loads 1 into the second condition argument
102d0:  bge a5,a4,102b8   Compares the condition arguments
102d4:  ret
```

The example includes a loop since the destination of the bge instruction refers to an address above it. The challenge occurs because register a4 is loaded from the heap and then saved after being updated. Since both paths should always be taken in the analysis for the CMP instruction, register a4 will be increased by one until an overflow is reached, as the

environment changes for each iteration due to the heap.

A naive solution would be to stop the loop after the condition is not satisfied for one of the values, which in this case is when register `a4` contains the value two. However, this results in an under-approximation due to the inclusion of bitflips. If a bitflip happens in register `a4`, for example changing the value from one to zero, it is possible to continue in the iterative structure for an additional time. In this case, register `a3` could contain the values `{1,2,3}` at `102d4`. This will not be reflected in the results of the analysis if we naively stop when the condition is not satisfied anymore. Thus, it would not be a sound solution.

Instead, we can manually perform loop unrolling on the RISC-V code and assume that all loops have been unrolled before performing the analysis. The unrolled version of the above RISC-V snippet can be seen below.

```
102b0:  sw zero,-20(s0)   −20(s0) = 0
102b4:  lw a4,-20(s0)     a4 = 0
102b8:  addi a3,a3,1      a3 = 1
102bc:  li a5,1           a5 = 1
102c0:  blt a5,a4,102e0   Assert
102c4:  lw a4,-20(s0)     a4 = 0
102c8:  addi a4,a4,1      a4 = 1
102cc:  sw a4,-20(s0)     −20(s0) = 1
102d0:  addi a3,a3,1      a3 = 2
102d4:  li a5,1           a5 = 1
102d8:  blt a5,a4,102e0   Assert
102dc:  ret
102e0:  nop
102ef:  j 102dc
```

Since the loop is iterated twice, the loop is duplicated once. The condition is replaced with a `blt` since we want to jump out of the loop when the original condition is false, and `blt` results in the opposite values as `bge`. The jump label of the `blt` instruction points at the address of a `nop` instruction which is a "dummy node". This node is used to detect which combination of bitflips and values could result in the loop terminating prematurely.

To reduce the runtime of the analysis we have also made assumptions regarding the implementation of BitflipperVild. This includes refraining from looking at bitflips in the stack and frame pointers. In this way, we can reduce the amount of memory space the analysis requires and thereby reduce the runtime. Furthermore, as register `zero` is the assembly name for register `x0`, see Appendix A, we can ignore all writes to this register. The analysis will still produce an over-approximation, as this register is always hard-wired to 0.

The above assumptions can be summarized to:

1. Setup functions are removed
2. Stack pointers are initialized to zero
3. When addresses are not found in the heap, loads result in all 32-bit values
4. I/O functions are replaced with I/O mapping

5. Functions called multiple times are duplicated
6. Loops are unrolled
7. The stack and frame pointers have no bitflips
8. Writes to register `zero` are ignored

Based on the listed assumptions, some changes to RISC-V programs are necessary to achieve a favorable analysis result. Some of these assumptions result in more over-approximation. However, other assumptions are implemented to improve the overall runtime or reduce complexity.

Chapter 4

Value-set Analysis

As mentioned in section 3.1, we have decided to perform the bitflip analysis using VSA. Furthermore, we will also use this analysis to recover the CFG. However, to use VSA we must define the domain and the transfer functions of the VSA specific to the RISC-V instructions. To ensure the transfer functions are monotone and that the analysis terminates, we can utilize the monotone framework, which we first introduced in [8].

4.1 Monotone Framework

The monotone framework is a conceptual framework, in which given different parameters, different types of analyses can be achieved. These parameters are the domain and the transfer functions. The input for an analysis in the monotone framework is a CFG, which represents the control flow of the program being analyzed. In broad strokes, a monotone framework consists of a complete lattice, which must satisfy the ascending chain condition, as well as a space of monotone transfer functions.

A lattice is a partial order, ensuring reflexivity, anti-symmetry, and transitivity of the ordering. [11, 12] It has the added requirement that there exists a join, least upper bound, and a meet, greatest lower bound, for every pair of elements in the underlying set. A complete lattice is a lattice where a greatest lower bound and a least upper bound always exist for an arbitrary non-empty subset. It thus follows that given any non-empty set of elements, we can determine their least upper bound.

For a complete lattice to satisfy the ascending chain condition, the height of the lattice must be finite. [12] This entails that all chains in the lattice have a finite height. A chain is a sequence of nodes in the lattice, where each ascending node is larger than or equal to the one before it.

Monotonicity of a transfer function guarantees that the order of the values is preserved after applying the function. [11, 12] Furthermore, it determines how the domain is affected when a statement is applied to values within the domain. Thus, the transfer functions of the VSA must be monotone to ensure that our analysis complies with the monotone framework.

The monotone framework ensures that the analysis eventually reaches a fixed point and thereby terminates, see Kleene's definition [10]. Thus, if our VSA analysis follows the monotone framework we can guarantee that the requirement concerning termination is

satisfied. To follow the monotone framework, one of the aspects we must consider is defining our domain to be a complete lattice following the ascending chain condition.

4.2 Domain and Environment

The domain from our previous work [8] has been revised to increase its precision. The revised domain can be seen in Equation 4.1.

$$\text{VSA} = \text{Label} \to ((\text{Registers} \to \text{ValSet}) \times (\text{Heap} \to \text{ValSet})) \tag{4.1}$$

Where Label $= \mathbb{N}$. The domain asserts that a specific label maps to a tuple containing functions, which each map to a lattice of registers and a lattice of addresses in the heap, both defined in section 2.2. Registers and Heap each map to ValSet which represents a set of value-sets.

A given value-set in ValSet can be a set of values where a bitflip has not occurred. Additionally, ValSet can include multiple value-sets which contain an interval of labels, $(\text{Label} \times \text{Label})_\perp$, representing at which label a possible bitflip could have occurred. Thereby, $\mathcal{P}(\text{Val})$ contains the bitflipped values for the specified interval. This can be seen below. The interval includes a bottom element, $\perp$, which represents that no bitflips have been observed. This bottom element is necessary to ensure that our domain is a complete lattice. Furthermore, the top element of our lattice is always a finite interval, since it depends on the number of instructions in the program.

$$\text{ValSet} = (\underbrace{\mathcal{P}(\text{Val})}_{\text{Non-flipped value-set}} \times \underbrace{\mathcal{P}((\text{Label} \times \text{Label})_\perp \times \mathcal{P}(\text{Val}))}_{\text{Flipped value-sets}})$$

A concrete instance of a domain for a given program point is called the environment. This means that given a specific program point, the environment keeps track of the possible values in the domain. An example of an environment can be seen below. In a real example, the value-set where bitflips have occurred would have at least 32 values, representing the 32 possible bitflips for a single value:

$$\begin{aligned} \text{L1} \to \quad & (\{\texttt{x1}: \quad \{\{1,2,3\}, \\ & \qquad\quad ([1,4],\{1,2,3\})\}\}, \\ & \{\texttt{0(s0)}: \quad \{4,5\}\}) \end{aligned}$$

In the example, L1 represents the current label, `x1` represents a register, and `0(s0)` represents a symbolic address in the heap. To ensure that the domain is a complete lattice we need to define the order of the value-sets within the environment.

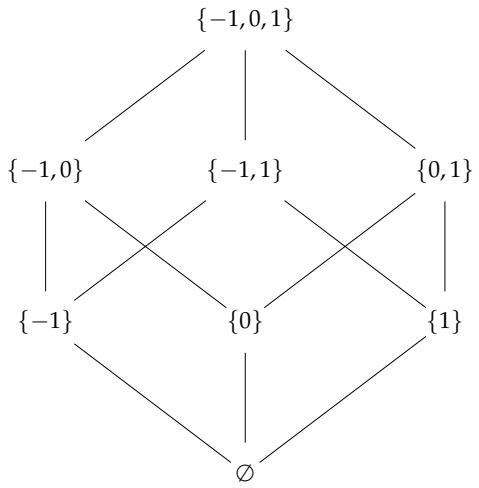
Ordering

To describe the ordering of the values within the domain, we define how to compare different environments. This is formalized in Equation 4.2.

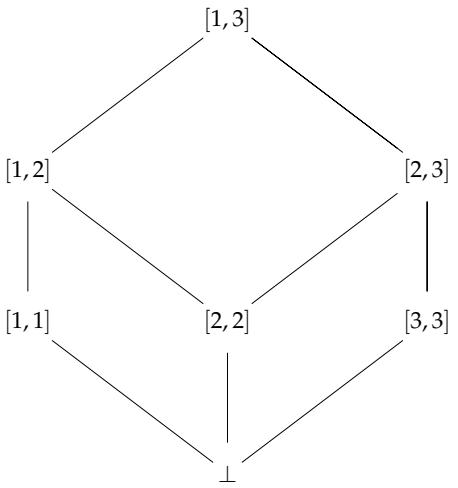
$$\begin{aligned} VS_1 \sqsubseteq VS_2 \implies &\forall\, v_1 \in VS_1,\ \exists! v_2 \in VS_2 : v_1 \sqsubseteq v_2, \\ &\text{where } \forall\, v_1, v_1' \in VS_1 : v_1 \parallel v_1' \\ &\text{and } \forall\, v_2, v_2' \in VS_2 : v_2 \parallel v_2' \end{aligned} \tag{4.2}$$

In the above formalization VS_1 and VS_2 are ValSets, while v_1, v_1', v_2, and v_2' are value-sets. Equation 4.2 states that VS_1 is partially ordered under VS_2 if and only if it holds that for every value-set v_1 in VS_1, there must be exactly one value-set, v_2 in VS_2, such that v_1 is partially ordered under v_2. This disallows environments with duplicate ValSets. In addition, a condition on the ValSets are applied, which ensures that they are discrete, meaning that no two value-sets can be compared. The $\parallel$ symbol reflects this and symbolizes that two elements are incomparable.

To ensure that the condition of a complete lattice is satisfied, we also need to describe the ordering between the value-sets in the domain. When comparing value-sets, we know that each set contains a finite number of values. This is because we have defined the possible values in the value-set to be within 32 bits, see section 2.1. Thus, the top value in our value-set lattice will be the set containing all possible 32-bit values, whereas the bottom value will be the empty set. The lattice also adheres to the ascending chain condition, since all sequences of nodes have a finite height. A simplified example can be seen in Figure 4.1a, which is a complete lattice for sets containing only 2-bit signed values.



(a) An example of a lattice for sets of signed 2-bit values.



(b) An example of a lattice for intervals between 1 and 3.

In the case where the interval for both value-sets is the same, we can simply find the least upper bound for the two sets of values. For example, if we wanted to join the value-set $\{-1, 0\}$ with $\{1\}$ in the 2-bit lattice example, the resulting value-set would be $\{-1, 0, 1\}$.

In the case where the intervals for the value-sets are different from each other, we must also ensure that these bounds can be partially ordered, which can be described by an interval lattice. Since the largest bound is determined by the largest label, the size of the interval is finite. Moreover, it is always possible to find the least upper bound and greatest lower bound between two intervals. To illustrate this, we have made a simple interval lattice for a program with three labels, see Figure 4.1b. To limit the size of the over-approximation,

we only join two intervals if one of the bounds is contained within the other or they have adjacent bounds and contain the same value-sets. For our example, we would join the intervals $[1,2]$ and $[2,3]$ only when they have the same value-sets. However, if one of the intervals is contained within the other, then we would join the value-sets.

Example

To explain how two environments can be joined within our domain, we have created a simple example with one register. Note that in a real example, the flipped values would in most cases contain at least 32 values.

Given two environments, $env1$ and $env2$, both containing the register, `x1`, with following value-sets:

$$\begin{aligned} env1 \rightarrow \{\texttt{x1} :\{&\{1,2,3\},\\ &([1,4],\{1,2,3\})\}\} \\ env2 \rightarrow \{\texttt{x1} :\{&\{2,7\},\\ &([3,4],\{3,4,5\}),\\ &([5,6],\{5,6,7\})\}\} \end{aligned}$$

We can join $env1$ and $env2$ by first considering the non-flipped values. For these values, the least upper bound can be found, which in this case is $\{1,2,3,7\}$. For the flipped values, we compare the intervals, where $[3,4] \sqsubseteq [1,4]$ resulting in the join of the values for these two value-sets. Note that this join is an over-approximation, as labels 1 and 2 now also have the values of the interval $[3,4]$. To minimize the size of the over-approximation, we do not join the intervals $[1,4]$ and $[5,6]$, as neither is contained within the other and they do not have the same value. The resulting environment from joining $env1$ and $env2$ is env seen below:

$$\begin{aligned} env \rightarrow \{\texttt{x1} :\{&\{1,2,3,7\},\\ &([1,4],\{1,2,3,4,5\}),\\ &([5,6],\{5,6,7\})\}\} \end{aligned}$$

Based on the above, we determine our domain for VSA to be a complete lattice that follows the ascending chain condition. Thereby, we have established one of the parameters that define a monotone framework. We also need to define the second parameter, which is the monotone transfer functions.

4.3 Transfer Functions

A transfer function specifies the effect a given statement has on the environment. This function describes the environment for a given label, l, before the statement is executed, $IN(l)$, and after the statement has been executed, $OUT(l)$. In this section, we will define the functions that determine the IN and OUT environments.

The IN environment is the input for the transfer functions. As a given node can have multiple incoming environments from different branches in the CFG, we must formalize a way to join these environments. The intuition for both the registers and the heap is to join all value-sets where no bitflip has occurred, since these are all possible values for the current node. Furthermore, the intervals of the value-sets containing bitflips should be extended if possible, as described in section 4.2. Based on this intuition, we can formalize the IN environment to be $IN(l) = JOIN(l)$, where $l \in Label$ and

$$JOIN(l) = \bigsqcup_{l' \in PRED(l)} OUT(l')$$

where $PRED(l)$ contains all predecessors to the label, l. To simplify the notation used to access specific labels in the environment, we say that for an element $(R, H) \in IN(l)$, the following holds:

$$\begin{aligned}
(R,H)(x) &= R(x), \text{ if } x \in \text{Registers} \\
(R,H)(x) &= H(x), \text{ if } x \in \text{Heap} \\
(R,H)[x \mapsto v] &= (R[x \mapsto v], H), \text{ if } x \in \text{Registers} \\
(R,H)[x \mapsto v] &= (R, H[x \mapsto v]), \text{ if } x \in \text{Heap}
\end{aligned}$$

When x is a register, as defined in section 2.1, the function accesses or updates the Registers, whereas if x is an address, the heap is accessed or updated. Based on this notation, we define $OUT(l)$, which is dependent on which instruction a given label, l, describes. Below, the OUT environment is defined for each of the abstract instructions found in section 2.2.

$$\begin{array}{rcll}
OUT(l) & ::= & IN(l)[rd \mapsto \{IN(l)(rs_1) \times_{[\![OP]\!]} IN(l)(rs_2)\}] & \text{if } l \mapsto \texttt{OP } rd, rs_1, rs_2 \\
& | & IN(l)[rd \mapsto \{IN(l)(rs_1) \times_{[\![OPI]\!]} \texttt{imm}_{12}\}] & \text{if } l \mapsto \texttt{OPI } rd, rs_1, \texttt{imm}_{12} \\
& | & IN(l)[rd \mapsto IN(l)(\texttt{imm} + rs_1)] & \text{if } l \mapsto \texttt{LOAD } rd, \texttt{imm}(rs_1) \\
& | & IN(l)[\texttt{offset} + rs_1 \mapsto IN(l)(rs_2)] & \text{if } l \mapsto \texttt{STORE } rs_2, \texttt{offset}(rs_1) \\
& | & IN(l)[rd \mapsto \{\texttt{imm}_{20} ||_{\mathbb{B}} 0_{12}\}] & \text{if } l \mapsto \texttt{LUI } rd, \texttt{imm}_{20} \\
& | & IN(l)[rd \mapsto \{\texttt{imm}_{32}\}] & \text{if } l \mapsto \texttt{LI } rd, \texttt{imm}_{32} \\
& | & IN(l)[rd \mapsto \{l+1\}] & \text{if } l \mapsto \texttt{JAL } rd, \texttt{offset} \\
& | & IN(l)[rd \mapsto \{l+1\}] & \text{if } l \mapsto \texttt{JALR } rd, \texttt{offset}(rs_1) \\
& | & IN(l)[rd \mapsto \{addr(l) + (imm_{20} ||_{\mathbb{B}} 0_{12})\}] & \text{if } l \mapsto \texttt{AUIPC } rd, \texttt{imm}_{20} \\
& | & IN(l) & \text{if } l \mapsto \texttt{NOP}
\end{array}$$

The transfer functions generally reflect the semantics of the different instructions, see section 2.2. The transfer function for the `OP` and `OPI` instructions utilizes the semantic evaluation functions to fetch the actual arithmetic or logical operation. To compute the new environment for labels describing these two instructions, a concatenation between the set of values in the two source registers is performed. The actual operation is then performed between the elements in each tuple in the resulting set.

The `LOAD` and `STORE` instructions both access the heap and the registers. `LOAD` fetches value-sets from the heap and stores them in rd. `Store` saves value-sets found in rs_2 at the address derived from $\texttt{offset} + rs_1$, which is a symbolic value representing a location in the heap.

The LUI and LI instructions are similar to their semantic rules, in that they save an immediate in the destination register. The JAL and JALR instructions both update the destination register with the next label by adding one to the current label, l. For the AUIPC instruction, we need to get the address of the current label. Therefore, we use the function *addr* which fetches the address based on a label. This address is summed with an immediate and then saved in the IN environment. Lastly, since no values are changed in the NOP instruction, the $OUT(l)$ is the same as $IN(l)$.

To minimize over-approximation and runtime for loops, we have limited which values are passed to the successors of the CMP instructions. The CMP instructions include beq, bne, blt, bltu, bge, and bgeu. Thus, only the values which satisfy the condition should be passed to the destination node, whereas only the values which do not satisfy the condition should be passed to the node following the branch instruction. The notation below depicts the true and false branches of the transfer function for the CMP instructions. For example, BEQ-T is the case when CMP-T is applied using the beq semantic evaluation function.

$$
\begin{array}{rcll}
OUT(l) & ::= & IN(l)[rs_1 \mapsto \{IN(l)(rs_1) \times_{<} max(IN(l)(rs_2))\}, & \text{if } l \mapsto \texttt{BLT-T}\ rs_1, rs_2, L \text{ or} \\
& & \quad rs_2 \mapsto \{IN(l)(rs_1) \times_{>} min(IN(l)(rs_2))\}] & \text{if } l \mapsto \texttt{BGE-F}\ rs_1, rs_2, L \\
& & & \\
& | & IN(l)[rs_1 \mapsto \{IN(l)(rs_1) \times_{\geq} min(IN(l)(rs_2))\}, & \text{if } l \mapsto \texttt{BGE-T}\ rs_1, rs_2, L \text{ or} \\
& & \quad rs_2 \mapsto \{IN(l)(rs_1) \times_{\leq} max(IN(l)(rs_2))\}] & \text{if } l \mapsto \texttt{BLT-F}\ rs_1, rs_2, L \\
& & & \\
& | & IN(l)[rs_1 \mapsto \{IN(l)(rs_1) \cap IN(l)(rs_2)\}, & \text{if } l \mapsto \texttt{BEQ-T}\ rs_1, rs_2, L \text{ or} \\
& & \quad rs_2 \mapsto \{IN(l)(rs_1) \cap IN(l)(rs_2)\}] & \text{if } l \mapsto \texttt{BNE-F}\ rs_1, rs_2, L \\
& & & \\
& | & IN(l)[rs_1 \mapsto IN(l)(rs_1), rs_2 \mapsto IN(l)(rs_2)] & \text{if } l \mapsto \texttt{BNE-T}\ rs_1, rs_2, L \text{ or} \\
& & & \text{if } l \mapsto \texttt{BEQ-F}\ rs_1, rs_2, L
\end{array}
$$

As seen above, we ignore unsigned comparisons, so the same transfer function is used to describe both blt and bltu instructions. These instructions check whether the value of the first argument is less than the value of the second argument. To compare value-sets for the node following the true branch, the value-set for the first argument contains all values that are less than the largest value in the value-set of the second argument, as seen in the transfer function for BLT-T. The value-set for the second argument contains all values that are greater than or equal to the smallest value in the value-set of the first argument. The opposite computation is performed for the node following the false branch.

The bge and bgeu instructions check whether the value of the first argument is greater than or equal to the value of the second argument. Thus, the environment of the node following the true branch does the same computations as blt and bltu's environment of the node following the false branch and vice versa.

The beq instruction checks whether the values in the arguments are equal. The value-sets passed to the destination node are calculated by taking the intersection of the values from the two value-sets, as these represent the values they have in common. Thereby, the intersection operator ignores the intervals in the value-sets and only performs an intersection between the values in the value-sets. For BEQ-F, the value-sets are not changed since it is

not possible to remove any values and still ensure a sound approximation without reasoning about the contents of the sets. The `bne` does the exact opposite of `beq` and thus keeps the original value-sets for the destination node and finds the intersection for the following node.

To make a more precise analysis, it is possible to make a tighter approximation for `BNE-T` and `BEQ-F` in a special case. This is the case when the second source register contains a value-set with a singleton. This value can safely be removed from the other value-set by finding the difference between the two sets, as seen below. The difference operator is only applied to the values in the value-set and ignores the intervals.

$$
\begin{array}{rcll}
OUT(l) & ::= & IN(l)[rs_1 \mapsto IN(l)(rs_1), & \text{if } l \mapsto \texttt{BNE-T}\ rs_1, rs_2, L \text{ or} \\
& & \quad rs_2 \mapsto \{IN(l)(rs_2) \setminus IN(l)(rs_1)\}] & \text{if } l \mapsto \texttt{BEQ-F}\ rs_1, rs_2, L \\
& & & \text{and } |IN(l)(rs_1)| = 1 \\
& & & \\
& | & IN(l)[rs_1 \mapsto \{IN(l)(rs_1) \setminus IN(l)(rs_2)\}, & \text{if } l \mapsto \texttt{BNE-T}\ rs_1, rs_2, L \text{ or} \\
& & \quad rs_2 \mapsto IN(l)(rs_2)] & \text{if } l \mapsto \texttt{BEQ-F}\ rs_1, rs_2, L \\
& & & \text{and } |IN(l)(rs_2)| = 1
\end{array}
$$

Based on the definition of the transfer functions, the nodes in the CFG must include the name of the instruction to determine which transfer function should be applied. Furthermore, the destination and source registers are also necessary information, as we need to know which registers to access and update. Because of the `JAL` and `JALR` instructions, the address of the node is also necessary to determine which node to jump to.

We have now defined how each of the transfer functions influences the environment when applied. However, to ensure that all of our transfer functions are monotone we further need to establish that the ordering of the values is preserved, as mentioned in section 4.1.

Ordering of Transfer Functions

The monotonicity of the transfer functions depends on whether the output of the functions preserves the ordering. This can be formalized as:

$$
\begin{aligned}
\forall\, l \in \mathit{Label}, \forall x \in \mathit{Register} : IN(l)(x) \sqsubseteq IN'(l)(x) \\
\implies OUT(l)(x)(IN(l)(x)) \sqsubseteq OUT(l)(x)(IN'(l)(x))
\end{aligned}
\tag{4.3}
$$

The above equation describes the order of the output of a transfer function. It states that given a label and a register, when an IN environment is partially ordered under another IN environment, IN', it implies that the OUT environment with $IN(l)(x)$ as input is also partially ordered under the OUT environment with $IN'(l)(x)$ as its input.

For example, for a given $v_1 = ([1,2],\{1,2,3\})$ in VS_1 and $v_2 = ([1,2],\{1,2,3,4\})$ in VS_2, the order is satisfied, since $v_1 \sqsubseteq v_2$, implying that a given transfer function also preserves the order. However, VS_2 cannot also contain the value-set $([1,2],\{3,4\})$, since it has the same interval as v_2. It should be noted that this example never occurs in our analysis, due to the JOIN function, which merges the value-sets with intervals that are within the other. However, the value-set $([4,5],\{3,4\})$ can be in VS_2, since it has a different interval.

Thus, the partial order for each value-set in ValSet should be preserved, meaning that Equation 4.3 must be true for all transfer functions. To show that the transfer functions preserve the ordering, the function for each abstract instruction is discussed.

For some of the transfer functions, the IN environment does not affect the OUT environment. This includes transfer functions for `LI`, `LUI`, `NOP`, `AUIPC`, `JAL`, and `JALR`. In this case, Equation 4.3 still holds since the value-sets stored in the *rd* registers for both OUT environments become equal.

In other cases, the IN environment does affect the OUT environment. This includes transfer functions for `OP`, `OPI`, `CMP`, `LOAD`, and `STORE`. To compare the order, we can look at each register separately and ensure that the ordering is preserved for all registers. If the value-sets stored in *rd* are unequal and all other value-sets in the other registers are equal, the order is preserved since the value-sets in *rd* will be equal after the transfer function has been applied. If either rs_1 or rs_2 differs, then the value-sets stored in *rd* will preserve the order from the differing register.

Thus, the transfer functions are determined to be monotone, as the order of the values remain the same when applying the transfer functions. We can therefore conclude that our analysis satisfies the conditions of a monotone framework.

4.4 Application of VSA

Taking advantage of the properties of the monotone framework, we can guarantee that our analysis will terminate, which is a requirement stated in section 3.1. However, VSA can be applied in different use cases, which we will describe in this section.

As mentioned in section 3.2, the challenge with CFG recovery is indirect jumps. However, this challenge could be resolved by using VSA to over-approximate which addresses could be the destination of an indirect jump. In this case, the analysis will continue until it is certain that all possible jump addresses have been found.

VSA can also be utilized to find possible values in the register. Using these values, we can derive which values could occur when a bitflip has occurred. To achieve this, we must apply an additional function after executing the transfer function to determine the OUT environment of a given node. For each non-flipped value in the value-sets of the registers, this function should calculate 32 new values, since values each consist of 32 bits. For each of these new values, only one bit has been flipped. We formalized this function in our previous work [8]:

$$flip(l) = \forall_{(R,H)(x,y)\in OUT(l)} : \{x' | x' \equiv_1 x\}$$

In this function, R represents the Registers and H represents the heaps. Furthermore, as our fault model only describes bitflips occurring in registers, we only flip the registers, $x \in R$, and ignore the heap, $y \in H$.

The flip function is performed immediately after the transfer function is performed. As

an example, if the current node has the label 1 and is an `addi` instruction, where the immediate 1 is added to the values in the register `x1:{2}`. When this node is analyzed, we calculate the initial OUT environment by applying the transfer function for `addi`, see section 4.3. The contents of register `x1` will be updated to the value-set `{3}`. Afterward, the flip function, seen above, is applied, which results in the following two value-sets:

$$\{\texttt{x1}: \{\{3\}, \\ ([1,1], \{-2147483645, 1, 2, 7, 11, 19, 35, 67, 131, 259, 515, 1027, 2051, 4099, 8195, \\ 16387, 32771, 65539, 131075, 262147, 524291, 1048579, 2097155, 4194307, \\ 8388611, 16777219, 33554435, 67108867, 134217731, 268435459, 536870915, \\ 1073741827\})\}\}$$

The resulting value-set contains the 32 values, corresponding to flipping each of the 32 bits representing the value in `x1`. When all the flipped values are calculated based on the non-flipped values in each register, the next node can be analyzed, wherein the transfer function for its instruction may also use the flipped values. Since we assume that only one bitflip can occur, the flip function is only applied to non-flipped values. Based on the static analysis defined in this chapter, we can now implement the two cases described in this section.

Chapter 5

Implementation

Using the RISC-V and VSA formalization as a foundation, we can implement a verification tool to determine bitflip vulnerabilities. We use VSA to implement the domain, as well as a CFG recovery and a bitflip analysis module. Furthermore, we also implement a backward slicing module to reduce the analysis scope, as well as a worklist algorithm, which is used by the different implementations of VSA. To provide an overview of the different modules making up BitflipperVild, we have created an illustration of the architecture.

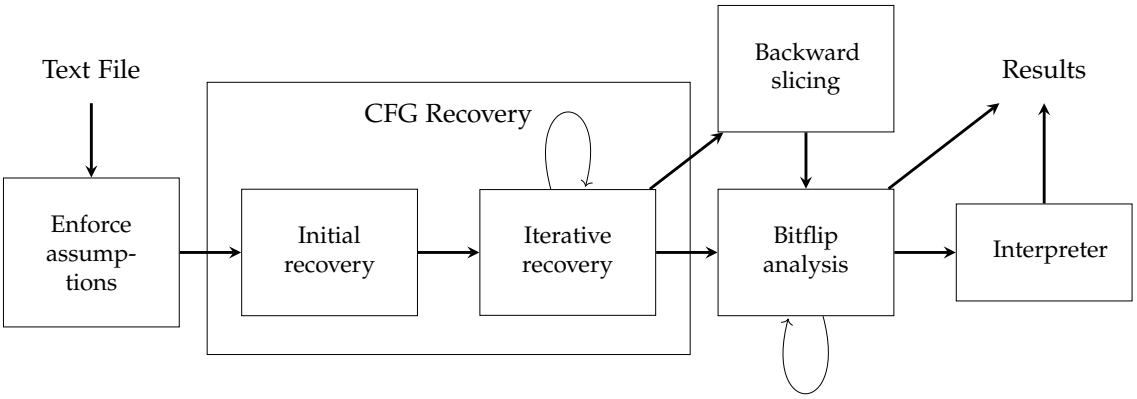


Figure 5.1: An overview of the architecture for BitflipperVild.

As seen in Figure 5.1, the input to BitflipperVild is a text file, which contains the RISC-V code to be analyzed. Before analyzing the text file, it is formatted to adhere to the assumptions described in section 3.3. The instructions in the file are then used to generate a CFG in the CFG Recovery module. This is accomplished in two steps as explained in section 5.4. The CFG is then either sliced in the backward slicing module or used as is and then analyzed iteratively in the bitflip analysis module. The results of the analysis can then be saved in a file. Moreover, the analysis results can also be the input of the interpreter module, which checks which of the bitflips result in an authentication bypass.

5.1 Automatic Adherence to Assumptions

To analyze a RISC-V program with BitflipperVild, it is in most cases necessary to manually modify the code to ensure that it adheres to the assumptions described in section 3.3. However, this process is cumbersome and prone to errors. This is due to new instructions being added because of loop unrolling and duplicated functions, making it also necessary update the jump addresses to point to the correct instructions.

This is addressed by creating the module "enforce assumptions", which automatically formats the program to adhere to the assumptions. As seen in Figure 5.1, this module receives a text file as input. An overview of the components that make up this module can be seen below.

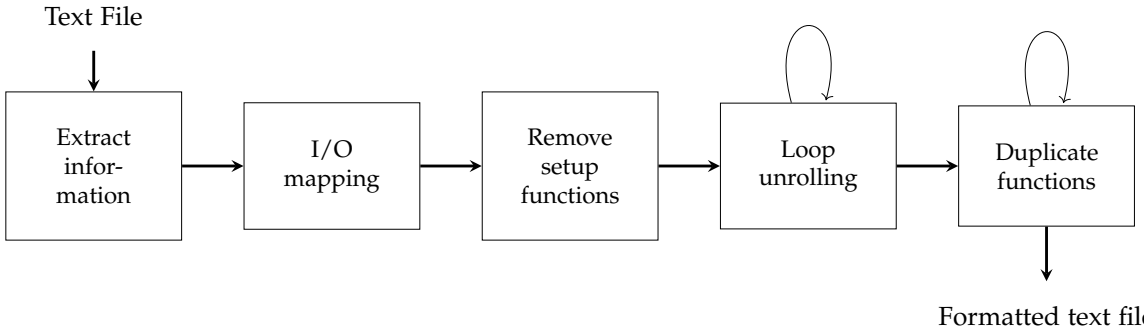


Figure 5.2: An overview of the components that format the text file, such that it adheres to our assumptions.

To keep track of which instruction belongs to which function, we have chosen to save this information in a dictionary, where the key is a function name, and the value is a list of instruction objects. Thus, information about each program point is extracted from the text file and saved in objects with properties corresponding to the different parts of the instruction. These objects are then saved in the list. While extracting the information, BitflipperVild checks for calls to the `printf` and `scanf` functions, as these need to be replaced with an I/O mapping due to item 4 found in section 3.3.

To adhere to item 1 in our assumption list, the module removes the setup functions. To do this, BitflipperVild utilizes a list of function names that it should analyze. Based on this list, we remove all keys in our dictionary, which do not match any of the names in the list.

The module also unrolls the loops in all functions to adhere to item 6 in section 3.3. As seen in Figure 5.2 this component is executed multiple times. This is due to the implementation which only unrolls one loop at a time, see Listing 5.1. In line Line 3, a loop is found and extracted from a function. This is done by checking whether a branch instruction jumps to an address that is smaller than its own address. If this is the case, the `loops` variable will contain a list of nodes in the loop, as well as the index of the next node after the loop. If no loop is found, this dictionary will be empty.

In the for-loop, the loop is unrolled. The logic for the branch instruction needs to be changed, as we want to jump to our "dummy" node when the condition is false. However, as RISC-V jumps to the destination label when the condition is true, we must flip the branch instruction to represent the opposite instruction, such that it returns true when the original condition returned false. This is done in line Line 10.

After the branch instruction has been flipped, nodes that are skipped in the first iteration must be removed, see line 14. To find these nodes, we examine the instruction right before the loop. If this node is a jump and its destination is an address inside the loop, the nodes from the start of the loop until the destination of the jump instruction are the nodes that should be skipped. This is necessary to do since these nodes will never be reached and thus would become a new entry point for the CFG.

In line 16, the module unrolls the loop a number of times depending on the value of `number_of_iterations`. This is achieved by inserting a copy of the loop nodes into the list of nodes. Lastly, the index of the node following the loop is updated, as this is used to return from the "dummy" node to the correct address. Furthermore, the addresses of the nodes in the function that had been unrolled must be updated.

Listing 5.1: Function used to unroll a loop.

```
1  def unroll(function_nodes: dict, number_of_iterations: int):
2      visited_lines, loops, function, index =
3          find_and_extract_loop(function_nodes)
4      no_removed = 0
5
6      for index, loop_nodes in loops.items():
7          destination_node = function_nodes[function][index]
8
9          branch_node = function_nodes[function][index - 1]
10         branch_node = flip_branch_instruction(branch_node)
11
12         nodes = function_nodes[function]
13         nodes, index, no_removed =
14             remove_skipped_nodes(nodes, loop_nodes, index)
15
16         for _ in range(number_of_iterations):
17             new_nodes = copy.deepcopy(list(loop_nodes.values()))
18             nodes = nodes[:index] + new_nodes + nodes[index:]
19
20         index = nodes.index(destination_node)
21
22     return function_nodes, function, index, no_removed
```

The last component in Figure 5.2 duplicates and renames the functions that are called multiple times, such that they have unique names. This is necessary since BitflipperVild is a polyvariant analysis. First, the functions which are called multiple times are found by examining the destination of all jump instructions. If a function is called multiple times, it is duplicated and the function name is renamed by adding a number at the end. Then the addresses are fixed and the callers of this function are updated, such that no two callers call the same function. As seen in Figure 5.2, this component is called iteratively, since a duplicated function can also contain calls to other functions. If the component is only called once, the issue described in section 3.3 is not resolved.

This module makes BitflipperVild more usable, as BitflipperVild then requires less manual work. The module also prevents human errors which could result in the tool providing incorrect results or not being able to terminate. Thus, this module outputs a formatted text file adhering to the assumptions, which can be used by the rest of the modules in BitflipperVild.

5.2 Worklist

The next module in Figure 5.1 is the CFG recovery. However, to be able to perform VSA on a CFG, we have created a worklist algorithm that keeps track of which nodes should be analyzed. The implementation of the worklist algorithm can be seen in Listing 5.2. As input, this function takes a domain, a CFG, and optionally a target address. `Domain` is an abstract class requiring an overload of the join and transfer methods, see section 5.3 for implementation details. The CFG is a representation of the program's control flow, and `target_address` is a string representing the address of the privileged point in the program. The `target_address` is optional as it is only necessary if the CFG is a slice, see section 5.5.

Listing 5.2: The worklist function used to perform VSA with different domains.

```
1  def worklist_algorithm(domain, cfg, target_address = ""):
2      worklist = cfg.get_entry_nodes()
3      domain.initialize_environment(cfg, cfg.get_entry_nodes())
4
5      while worklist:
6          node: CFGNode = worklist.pop(0)
7          for successor in cfg.get_successors(node):
8              if successor.address == target_address:
9                  domain.environment[successor.label] = join(domain,
                       cfg, successor)
10                 continue
11             IN = join(domain, cfg, successor)
12             OUT = domain.transfer(cfg, successor, IN)
13             worklist = update_worklist(domain, successor, OUT,
                   target_address, worklist)
```

The `worklist_algorithm` function first initializes the worklist and environment of the domain with the list of entry nodes, see lines 2 and 3. For each node in the worklist, we iterate through its successors, where we first check whether the address of the successor is equal to the target address, see line 8. If this is the case we should only perform the `join` method on the successor, without calling the `transfer` method. This is because we have reached our privileged point and our analysis should therefore terminate. If the successor is not the target node, we instead calculate its `IN` environment in line 11, which is then used to calculate the `OUT` environment in line 12. Lastly, we update the worklist by checking whether the environment has changed for the given node in line 13. If the environment has changed, the successor is added to the worklist. Otherwise, we have reached a fixed point. This also theoretically makes it possible to handle iterative control structures, since they will eventually reach a fixed point for the analysis.

This worklist algorithm is used for all VSA analyses, which includes the CFG recovery and the bitflip analysis. To be able to perform these analyses, we first need to define the domain.

5.3 Domain

To allow different static analyses to be performed, we have created an abstract class, which defines the necessary methods for our domain. This ensures modularity in BitflipperVild since the worklist algorithm can utilize any analysis implementing these methods. This is illustrated in Figure 5.3.

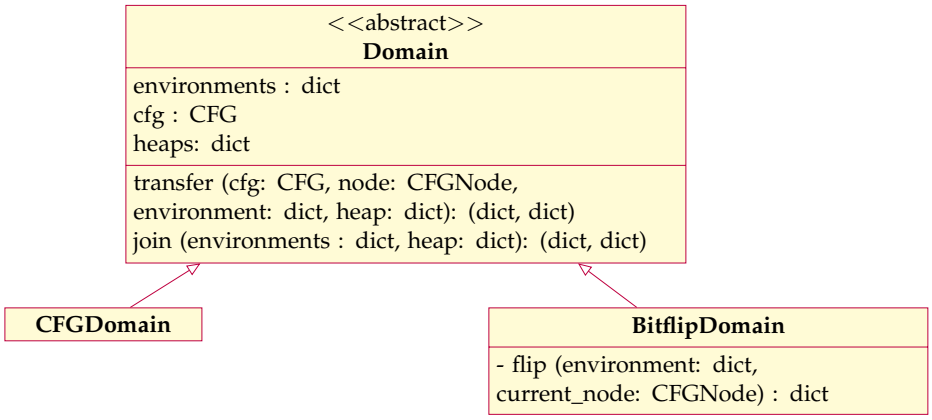


Figure 5.3: A UML diagram of the domain implementation for the analysis.

The abstract class, `Domain`, contains the `cfg`, `environments`, and `heaps` properties. The `cfg` keeps track of the CFG being analyzed, while the `environments` keeps track of the abstract environment containing the resulting values for each program point. Lastly, the `heaps` keeps track of the values stored in the heap for each program point.

The `environments` property is a set of key-value pairs wherein the key is a label and each of the values is a specific environment for that label, as seen below. Thus, we can keep track of the environment for each program point in the analysis. This is necessary when a node has multiple predecessors since we need to know the environment for each predecessor when performing a join to calculate the `IN` environment.

$$\text{environments} \rightarrow \{\text{label} : \{\text{environment}\}\}$$

As our environment must keep track of which values a given register can contain, we implement it as a dictionary with key-value pairs. The keys are the names of the registers, while the values are sets of value-set objects for the given register, as seen below.

$$\text{environment} \rightarrow \{\text{register} : \{\text{value-set}\}\}$$

We have chosen to implement the environment as a dictionary to ensure that there is only one set of value-sets for each register. Furthermore, we can efficiently access the set of value-sets given a specific register. The value in the dictionary is a set, because we want to keep track of both the values that have been bitflipped and those that have not while disallowing duplicates. In the case of joining two environments, it is possible that a value in the dictionary can contain several bitflipped value-sets for the same register.

Each value-set object reflects the value-set defined in section 4.2 and thus consists of two elements in the case of a bitflip: a tuple of lower and upper bounds and a set of values,

as seen below. The values in the value-set are contained in a set, as we want to avoid duplicates and disregard their order.

$$\text{value-set} \rightarrow (\text{lower, upper}), \{\text{values}\}$$

The type of values in the value-set can be hexadecimal, signed, or unsigned. The hexadecimal values are used to represent addresses. The type of value is determined by the transfer function. If the instruction is a control transfer instruction, the type will be addresses. Furthermore, if the instruction is an operation, the type can be determined based on whether the instruction is unsigned or signed, or the value is written as a hexadecimal. For value-sets representing the absence of bitflips, the lower and upper bounds are not initialized.

The `heaps` property of the abstract class is a dictionary, which represents a heap for each label. Each key is a label and the values are dictionaries which each represent a single heap illustrated below. Each key in each heap is a symbolic value, such as `-15(sp)`, representing where in memory the values should be stored. The value of the heap's key-value pairs is a set of value-sets. The heap reduces the over-approximation of the analysis and its inclusion will thus result in `LOAD` instructions updating their destination register with concrete values instead of all possible 32-bit values.

$$\text{heap} \rightarrow \{\text{address} : \{\text{value-set}\}\}$$

The `Domain` abstract class defines two abstract methods called by the worklist algorithm: `transfer` and `join`. The transfer method implements the effects of a given instruction for each node, which are defined in section 4.3. As output, it returns a tuple containing two dictionaries. The first dictionary is the updated environment after applying the transfer method and the second is the updated heap. The join method also takes two dictionaries as input: the registers and the heaps that should be joined. Thus, the join returns a new environment and a new heap that corresponds to the environment and heap before the current node has been analyzed.

As seen in Figure 5.3, two concrete domain classes inherit from the abstract `Domain` class. One of these classes is the `CFGDomain`, which we use to perform the CFG recovery. The second class is used to determine bitlfip vulnerabilities and is called `BitflipDomain`. The two classes each implement the transfer and join in a way that satisfies their use case. We will discuss this in the next sections.

5.3.1 Bitflip Domain

To perform our bitflip analysis, we have implemented a `join` and `transfer` method. The `join` method iterates through all the predecessors for a given program point in the analysis. If the value-sets of the predecessors have the same bounds for a given register, the joined value-set in the environment will be the union of the two sets. If there are value-sets in one predecessor, which are not contained in another, they will be added to the environment. The heaps are joined in a similar manner, where the union is found if two

predecessors contain the same symbolic value.

The environment and heap returned by the `join` method are then used as input for the `transfer` method. This method applies a transfer function, determined by the instruction type. The transfer functions for our analysis are defined in section 4.3. The environment and the heap are then updated according to the transfer function. For example, if the instruction is a type of `STORE` instruction, the content of the second source register is saved in the `heap` at the symbolic value specified by the first source register as a key-value pair. If the instruction is a `LOAD`, the specific value from the `heap` is fetched based on the source register and used to update the destination register. If the value cannot be fetched from the `heap`, the destination register of the instruction will have its sets of value-sets updated to be a single set containing all possible 32-bit values.

Since our VSA also keeps track of which values are possible when a bitflip has occurred, another method, `flip`, is called after the transfer function has been applied. The `flip` method takes the output from the `transfer` method and the current node as input and returns an updated environment. The `flip` method iterates through all value-sets in the environment. If the value-set already contains flipped values they will not be flipped, since we only handle the case where a single bitflip occurs. Moreover, if the set contains all possible 32-bit values, they will not be flipped, since the value-set already contains all possible values. To get the flipped values based on a value set, the 32-bit strings for each value are obtained. For each string, 32 different values are produced by flipping each of the bits. Then, a new value-set with all the bitflipped values is added to the environment, where the lower and upper bounds are set to be the current label.

5.3.2 CFG Domain

The CFG domain is used to create an accurate CFG representation using a VSA which can handle indirect jumps. As our focus is bitflips occurring in the register and not in the program counter, we do not include bitflips in the transfer function for this domain. Since this class also performs VSA, the two domain classes have identical `transfer` methods, but the CFG domain has no `flip` method. The join method is similar to that of the bitflip domain. Since there is only one value-set for each register due to flipped values being omitted, the join function for the CFG Domain can merge the value-sets for each register.

5.4 CFG Recovery

As mentioned in the requirements established in section 3.1, CFG recovery is necessary to perform the bitflip analysis. The recovery is made up of two steps: an initial recovery and an iterative recovery. In the initial step, we compute an initial CFG based on a text file containing the RISC-V instructions for a given program. Given the initial CFG, the iterative step is executed using `CFGDomain` to compute the destinations for the indirect jumps.

5.4.1 Initial Recovery

The initial CFG recovery is based on a formatted text file, which is created by first compiling C-code using a RISC-V cross-compiler, then fetching the instructions using `objdump`, and lastly reformatting the file such that the assumptions are enforced. For each line in the text file, the initial CFG recovery creates a CFG node. A line consists of three parts: the address of the instruction, the encoded instruction, and the actual instruction. As mentioned in section 4.3, the address and the actual instruction are required for each node in the analysis. Therefore, when creating a CFG node, the address and instruction are saved. Furthermore, the destination and source registers are extracted from the arguments of the instruction when present. For example, given the line:

Address	**Encoded instruction**	**Actual instruction**
`10280:`	`00e686b3`	`add a3, a3, a4`

The instruction is parsed as "add" with `a3` as the destination register, and `a3` and `a4` as the source registers. Moreover, `10280` is saved as the address. In the case of a pseudo-instruction, the instruction is parsed as its corresponding set of base instructions. For example, `mv rd, rs1` is parsed as `addi rd, rs1, 0`, according to [22]. Based on which instruction the node represents, we also set a `NodeType` property which represents its abstract instruction type defined in section 2.2.

After all the nodes have been created in the CFG, edges are added between the nodes. If the node is not a jump instruction then one edge is added between the current node and the node for the next line in the text file. If the node contains a jump instruction, the edge is added to the node at the address specified by the instruction. For comparison nodes, an additional node is added to the CFG, such that each comparison has a node representing the false and the true branches. An edge is added between the false node and the next instruction, while an edge is added between the true branch node and the node containing the address specified by the comparison instruction.

However, using this approach it is not possible to handle indirect jumps, since we cannot resonate about which values are saved in the registers. More specifically, it is not possible to add edges for the `jalr` instructions. To make this possible, we perform VSA with the CFG domain on the graph to compute which values a given register could contain at a program point with a `jalr` instruction.

5.4.2 Iterative Recovery

The iterative CFG recovery uses an incomplete CFG to generate a CFG, which includes indirect jumps. An example of an incomplete CFG could be the CFG created by the initial recovery for a program containing `jalr` instructions.

This step is iterative since it is possible to have several `jalr` instructions pointing at each other, which results in it not always being possible to determine the complete CFG when only performing the iterative CFG recovery once. Thus, VSA with the CFG domain is

performed on the resulting graph until a fixed point is reached, meaning no new edges are added.

Each time the VSA is performed on the incomplete CFG, we update the edges for the nodes containing a `jalr` instruction with the new information obtained from performing the VSA. Moreover, in the case of a node with a jump instruction to a different function, an edge is added from the function's return node to the node containing the next instruction after the jump in the original function.

When the fixed point is reached, the CFG has been recovered and is ready to be analyzed. However, before performing the bitflip analysis on the CFG, we can reduce the analysis scope by first performing backward slicing on the graph.

5.5 Backward Slicing

To limit the number of nodes being analyzed in the bitflip analysis using the `BitflipDomain`, we can perform backward slicing. To do this, the graph constructed by the CFG recovery and a target node is used to construct a slice of the CFG. The intuition is to discover which registers affect the target instruction, and then slice the CFG when all of these registers have new values loaded into them. This is because the content of the registers will be overridden and therefore the previous values before the loads have no effect on the target node. The implementation can be seen in Listing 5.3.

Listing 5.3: Method used to perform backward slicing on a CFG.

```
1  def __execute(self, cfg: CFG) -> CFG:
2      new_cfg = nx.DiGraph()
3      new_cfg.add_node(self._target)
4
5      entry_nodes = cfg.get_entry_nodes()
6
7      worklist, skipped_nodes = [self._target], []
8      self.__update_relevant_registers(self._target)
9
10     while worklist:
11         node: CFGNode = worklist.pop(0)
12
13         for predecessor in cfg.get_predecessors(node):
14             if self.__is_relevant_node(predecessor):
15                 if predecessor.type != NodeType.CTRL:
16                     self._relevant_registers[predecessor.destination]
                           = True
17                 self.__update_relevant_registers(predecessor)
18
19             if self.__is_graph_finished(entry_nodes, new_cfg):
20                 skipped_nodes.append(predecessor)
21             elif self.__should_update_worklist(new_cfg, worklist,
                   node, predecessor, entry_nodes):
22                 worklist.extend(skipped_nodes)
```

```
23                 worklist.append(predecessor)
24
25             self.__add_node(new_cfg, predecessor, node)
26
27     if any(["(" in relevant_register for relevant_register in self.
           _relevant_registers]):
28         return self.__finalize(CFG(graph=new_cfg), cfg)
29
30     return CFG(graph=new_cfg)
```

To create the slice, we initialize a new CFG with the target node in lines 2 - 7. Furthermore, we find which registers affect the target node in line 8. We then iterate through the predecessor nodes in the CFG, starting from the target node. For each of these predecessors, we determine whether they override the content of a relevant register in line 14. There are two cases where this is true. The first case is when the type of instruction for the predecessor node is either an `OP`, `OPI`, `STORE`, or `LOAD` instruction and their destination register is deemed relevant. The second case is when the instruction type is a control transfer and one of the registers in its arguments is deemed relevant. If one of these cases is true, we set the destination register to be True in line 16, since the destination register is overwritten by the current instruction. Furthermore, we update the relevant registers in line 17.

In line 19, we check whether all relevant registers have been overwritten. If this is the case, we have found all of the nodes that must be included in the CFG slice and we should therefore not add the predecessor node to the worklist. Instead, the predecessor should be added to a list keeping track of the nodes that have been skipped. This is necessary as we can come across a node, which was added to the worklist earlier, determining that the graph is not finished. Thus, the nodes in the list should be analyzed before terminating. In line 25 we add the predecessor node, which we have just analyzed, to the CFG slice along with its edges.

In line 27, we check whether one of the addresses in `relevant_registers` is from the heap. If this is the case and there is more than one entry node in the slice, we make a larger over-approximation. This over-approximation is created by making a subgraph that contains all nodes between the smallest and the largest label of the slice.

As seen in subsection 8.2.2, a CFG slice can be used to perform the bitflip analysis, which outputs all of the possible bitflips that could occur in the given slice. To find which of these bitflips actually make the program unsafe, we need to reduce the amount of non-critical bitflips.

5.6 Interpreter

To reduce the amount of manual work required to check which bitflips can bypass the authentication, we have implemented an interpreter. Our approach for the interpreter is to iterate through all the flipped value-sets produced by the analysis. For each value in the value-sets, the interpreter should then check whether it is possible to reach the privileged

point when the value is injected.

To achieve this, we have created a new domain, where only one flipped value is injected for each run. Furthermore, we have modified the worklist for this domain, such that only one of the branch paths would be taken, resulting in each non-flipped value-set only including one value. However, this value can also be a range in cases where the source from a load is undefined.

To limit the number of over-approximations, we modified the transfer functions concerning the heap. In the bitflip analysis, all heap addresses except for the global pointer are represented by symbolic values, which consist of the function name and the arguments rather than their actual address. This makes the heap address local, see section 3.3. However, we observed that this resulted in addresses being unreachable from other functions, which led to a range being loaded instead. This is not an issue for the bitflip analysis, as the only time a range is loaded, is when fetching the values of the user and card PINs. When these values are ranges, it represents that the PINs can be any 32-bit value. Thus, we try to bypass the authentication without knowing what the PIN is. However for this interpreter, we instead save the actual address, which results in less over-approximation.

The focus of the interpreter is to assist the user in analyzing the results found by the bitflip analysis. We have therefore not focused on proving the soundness of this component. However, we have manually checked the results achieved by using the interpreter on the results from running the bitflip analysis on the FISSC programs. Based on results of the interpreter, we have determined that the output of the interpreter appears to be correct.

To ensure that the implementation of the modules in subsection 8.2.2 addresses the requirements found in section 3.1, we need to evaluate this implementation.

Chapter 6

Evaluation

To analyze a RISC-V program for bitflip vulnerabilities, we must ensure that BitflipperVild includes the necessary properties. This can be ensured by evaluating whether we have addressed all requirements specified in section 3.1. These requirements are summarized below in Table 6.1.

General	Analyze a FISSC program
	Prove countermeasures prevent bitflip vulnerabilities
	Determine value-sets for each program point
	Be RISC-V specific
	Terminate
CFG	Recover a CFG
	Interprocedural
	Resolve indirect jumps
Backward Slicing	Create a slice based on a privileged point
Bitflip Analysis	Detect bitflip vulnerabilities in registers
	Determine at which label a vulnerability can occur

Table 6.1: Summary of the requirements for BitflipperVild.

Some of the requirements depend upon a program to be analyzed before evaluating them. Thus, the programs in FISSC will be used as a foundation for this evaluation. The requirements involving the analysis of a program include the CFG, backward slicing, and bitflip analysis categories. The requirements in each of these categories will be evaluated in this chapter. As the enforce assumptions and the interpreter modules are not a part of our requirements, they are therefore not critical for our project and we will not evaluate these modules. However, the interpreter module has been used to verify the manual analysis performed in section 6.4.

6.1 General Requirements

To evaluate the requirements in the general category, we can examine whether our analysis has the required properties. We do this by evaluating each of the five requirements. However, the second requirement is closely related to the requirements in the bitflip analysis

category. We will therefore evaluate these requirements together in section 6.4.

The first requirement states that BitflipperVild must be able to analyze a FISSC program. After compiling the program to RISC-V code and formatting the programs to follow the assumptions described in section 3.3, we were able to produce a bitflip analysis for all programs in the collection. The results of the bitflip analyses showed the possible value-sets for each program point in the program. This satisfies the third requirement, which states that BitflipperVild must be able to produce this. Furthermore, the fourth requirement stating that our analysis must be RISC-V specific is also satisfied. All analysis results can be found in [5].

Furthermore, all of the FISSC programs terminated. Since we have shown that our analysis follows the monotone framework in section 5.3, it will always be the case that the analysis terminates and the fifth requirement regarding termination is thereby also fulfilled.

6.2 CFG Recovery

One of the requirement categories establishes requirements about the CFG, as seen in Table 3.1. These include that we should be able to create a CFG based on the RISC-V code. More specifically, we must be able to handle interprocedural programs, as the programs in FISSC contain multiple functions. Furthermore, we should also be able to resolve any indirect jumps.

To ensure that we can create a CFG for RISC-V programs using the minimal instruction set described in section 2.1, we have specified how the CFG node and edges should be created for each of these instructions. This can be seen in section 5.4. Thus, the CFG recovery implementation should be able to handle all RISC-V programs that follow the assumptions described in section 3.3. We were able to create a CFG for all versions of VerifyPIN. Since these include interprocedural programs, we have fulfilled this requirement. The resulting CFG of VerifyPIN 0 can be seen in Figure 6.1.

In the CFG, the different methods have different colors. However, the entry node is colored green. The nodes in `<main>` are blue, whereas `<initialize>`, `<verifyPIN>`, and `byteArrayCompare` have pink, purple, and orange nodes, respectively.

Since none of the VerifyPIN versions include indirect jumps, we have created a small RISC-V program with an indirect jump at address `105e4`, as seen on the left-hand side in Figure 6.2. This is to show that our recovery implementation fulfills this requirement as well.

The indirect jump instruction should jump to two different nodes, depending on how many times the instruction in address `105d4` is executed. At address `105d0` the value `105d0` is stored in register `a5`. At the next address, this value is replaced with the value `105d4`. Thus, when we reach address `105e4` an edge is added between this node and the node at the address found in `a5`, which is `105d4`. At this address, the value in `a5` is replaced with `105d8`, as we perform the `addi` instruction again. Therefore, another edge between addresses `105e4` and `105d8` is also added. In short, edges from the indirect jump node to

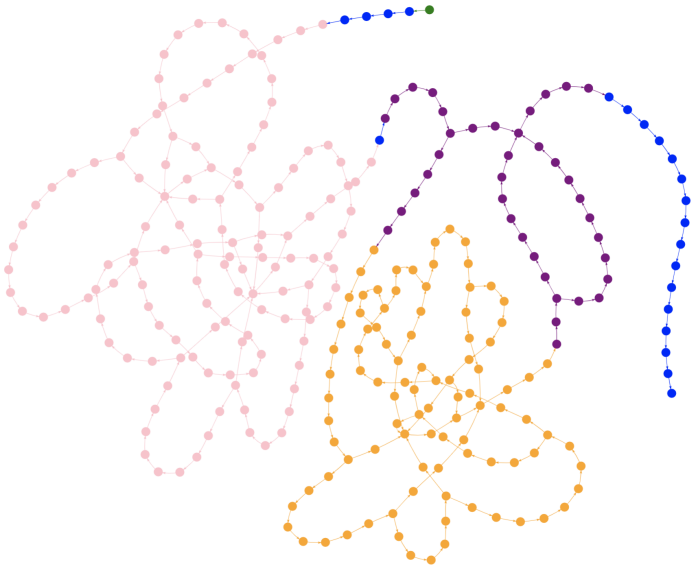


Figure 6.1: The CFG that corresponds to VerifyPIN 0. The green node represents the entry node, while the blue, pink, purple, and orange nodes represent the functions `<main>`, `<initialize>` `<verifyPIN>`, and `byteArrayCompare`, respectively.

the nodes at addresses `105d4` and `105d8` should be present in the CFG. To avoid looping until an overflow occurs, the `beq` instruction checks whether the address stored in register `a5` is equal to address `105dc`.

The CFG produced by BitflipperVild when analyzing the small program can be seen on the right-hand side in Figure 6.2. The green node represents the entry node for the CFG. Furthermore, the CFG represents the control flow of the program correctly, as it has two outgoing edges, one that jumps to the instruction at address `105d4` and another at address `105d8`. Moreover, two `beq` nodes are added which each represent either the true or false branch. As seen, our tool is able to resolve indirect jumps in RISC-V programs.

As mentioned in section 3.3, we do not consider bitflips when generating CFGs. Thus for the example in Figure 6.2, we do not consider whether the value stored in register `a5` has been flipped. Since our analysis can include false positives, the value-set in the destination register for `jalr` can contain several addresses. In this case, the CFG will be an over-approximation, resulting in some edges being false positives. Thus, there can be many edges from the jump node. However, as the functions in FISSC do not contain any indirect jumps, this has not been an issue in our case.

As described above, our CFG recovery implementation is able to create CFGs that use the syntax described in section 2.1 and follow the assumptions found in section 3.3. Moreover, it is also able to resolve indirect jumps. Thus, all requirements regarding the CFG recovery are fulfilled.

```
105d0:  auipc  a5,0x0
105d4:  addi   a5,a5,4
105d8:  nop
105dc:  auipc  a4,0x0
105e0:  beq    a5,a4,105e8
105e4:  jalr   0(a5)
105e8:  ret
```

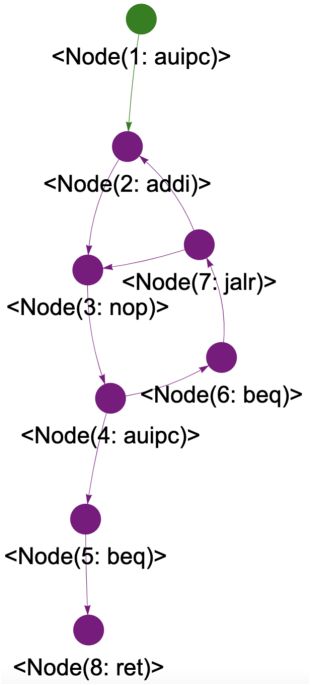


Figure 6.2: A RISC-V program which includes two indirect jump and its corresponding CFG. The green node in the CFG represents its entry node.

6.3 Backward Slicing

The next category of requirements states that BitflipperVild must be able to perform backward slicing based on a given privileged point. This requirement has been addressed by implementing the module backward slicing, see section 5.5. To evaluate the implementation, we have constructed the example seen below based on a modified snippet from VerifyPIN 0.

In Figure 6.3, the code snippet on the left-hand side describes a small program, which compares the values in registers `a4` and `a5`. The content of register `a5` is an immediate which is loaded at address `105ec`, while the content of `a4` is dependent on the heap. Given that the privileged point is the instruction at address `105f4`, the computed slice should include the instructions that write to the registers `a4` and `a5`.

In the above example, register `a5` is written to at address `105ec`, and register `a4` is written to at address `105f0`. However, `a4` is influenced by the heap address represented by `-20(s0)`. Thus, a `STORE` must be present for this symbolic value, which can be seen at address `105e8`, as all affected registers and heap addresses should be written to. Moreover, since the earliest instruction which influences the value of register `a4` is at address `105dc`, this address should be the first node in the slice. Thus, the CFG slice produced by BitflipperVild is shown to be accurate and our tool can indeed correctly perform backward slicing.

However, to ensure that the slice is always sound, we sometimes make a wider over-approximation. This is especially the case, when the slice depends on the heap and the slice contains several entry points. In this case, all nodes between the smallest and largest label are added to the slice, ensuring that there is only one entry point. This is necessary,

```
105d0:  addi  sp,sp,-48
105d4:  sw    s0,44(sp)
105d8:  addi  s0,sp,48
105dc:  sw    zero,-20(s0)
105e0:  lw    a5,-20(s0)
105e4:  addi  a5,a5,1
105e8:  sw    a5,-20(s0)
105ec:  li    a5,0
105f0:  lw    a4,-20(s0)
105f4:  bge   a4,a5,105f8
105f8:  ret
```

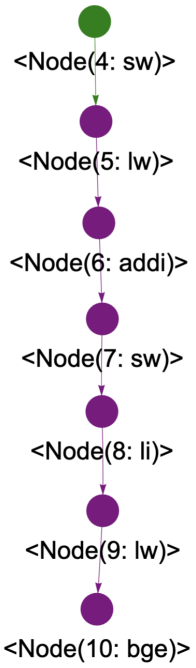


Figure 6.3: On the left is a RISC-V program of a comparison. On the right is a slice produced by BitflipperVild, where address `105f4` is the privileged point. The green node in the CFG represents its new entry node.

as the heap can be written to with new values in one of the nodes that were originally not added to the slice. These updates to the heap can influence the values in the registers. Therefore, without over-approximating the slice our representation would not be sound. A downside of this solution is the increase of nodes that will be analyzed by the bitflip analysis. However, as we prioritize accuracy higher than efficiency, we deem this solution appropriate.

Based on the above, our backward slicing module is able to create a CFG slice from a CFG created by the CFG recovery module and a privileged point. Thus, the requirement for backward slicing, described in Table 3.1, is satisfied.

6.4 Bitflip Analysis

To check whether the requirement "prove countermeasures prevent bitflip vulnerabilities" has been fulfilled, we can utilize FISSC to compare the different countermeasures implemented in the collection and examine whether BitflipperVild can detect vulnerabilities.

As mentioned in Chapter 1, the programs in FISSC [9] verify whether the given PIN is correct. If this PIN is correct a variable `g_authenticated` is set to true and the user is granted access to the privileged operations. Furthermore, the programs include a try counter (PTC), which keeps track of how many attempts the user has left. When the counter reaches zero, the program terminates without providing access to the user.

On the FISSC website [2], two conditions are stated to determine whether a successful attack has occurred:

1. A successful authentication with an erroneous PIN
2. A non-decrement of the PTC in case of a failed authentication

Thus, to check if a program is vulnerable to these attacks, we need to examine whether we can exit the verification as if the correct PIN was given when the actual PIN is incorrect. Furthermore, we need to examine whether the PTC is decremented each time the user enters an incorrect PIN.

There are seven different versions of VerifyPIN in FISSC. These all include different countermeasures and thus have different levels of security. An overview of these countermeasures can be found in Table 6.2.

	v0	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
Hardened Booleans		×	×	×	×	×	×	×
Fixed-time loop			×	×	×	×	×	×
Inlined functions				×	×		×	×
PTC decremented first					×	×	×	×
PTC backup					×			
Loop counter					×			
Double calls						×		
Double test							×	×
Step counter								×

Table 6.2: The countermeasures and programming features taken for each VerifyPIN program in FISSC [9].

Some of these countermeasures are beyond the scope of our analysis since they do not secure the registers but instead the control flow. This includes the double test countermeasure which performs two checks on the same condition instead of one. Another countermeasure beyond our scope is the step counter countermeasure which is used to ensure that no parts of the program are skipped. This is achieved by incrementing the variable `step_counter` for each step and checking whether it has been increased and triggering a countermeasure if it has not. As both of these countermeasures check the control flow and there are no values in the registers that can affect them with a single bitflip, they are not part of our scope. We have therefore decided not to examine these countermeasures further.

To evaluate BitflipperVild, we examine whether we can determine that it is possible to perform a bitflip attack which results in either of the two conditions being satisfied. To do this, we will for each of the countermeasures examine and compare VerifyPIN versions with and without the countermeasure. If we can show that an attack has been prevented with the countermeasure, we deem that our tool can detect the specific bitflip vulnerabilities protected by the countermeasure.

6.4.1 Hardened Booleans

The first countermeasure is hardened Booleans, which instead of representing true and false as one and zero, represents them as 0xAA and 0x55, respectively. This prevents a single injection attack to flip the value from false to true, as it requires additional flips to

achieve 0xAA from the value 0x55. All VerifyPIN versions, except for v0, use hardened Booleans. Therefore, to evaluate whether BitflipperVild can show that some attacks are prevented with hardened Booleans, we examine the result achieved by performing our analysis on v0 and v1. As this countermeasure focuses on the first condition for a successful attack, we will not consider the second condition for this evaluation. There are several places in the program where hardened Booleans are used. To limit the scope of the evaluation, we focus on the code snippets found in Listing 6.1 and Listing 6.2.

Listing 6.1: The code snippet from VerifyPin 0, which does not use hardened Booleans.

```
1  BOOL byteArrayCompare(UBYTE* a1,
       UBYTE* a2, UBYTE size) {
2      int i;
3      for (i = 0; i < size; i++) {
4          if (a1[i] != a2[i])
5              return 0;
6      }
7      return 1;
8  }
9
10 BOOL verifyPIN() {
11     ...
12     if (byteArrayCompare(
           g_userPin, g_cardPin,
           PIN_SIZE) == 1) {
13         // Authenticated
14     } else...
15 }
16
```

Listing 6.2: The code snippet from VerifyPin 1, which uses hardened Booleans.

```
1  BOOL byteArrayCompare(UBYTE* a1,
       UBYTE* a2, UBYTE size) {
2      int i;
3      for (i = 0; i < size; i++) {
4          if (a1[i] != a2[i])
5              return BOOL_FALSE;
6      }
7      return BOOL_TRUE;
8  }
9
10 BOOL verifyPIN() {
11     ...
12     comp = byteArrayCompare(
           g_userPin, g_cardPin,
           PIN_SIZE);
13     if (comp == BOOL_TRUE) { {
14         // Authenticated
15     } else...
16 }
```

The main difference between Listing 6.1 and Listing 6.2 can be seen in lines 5 and 7. In these lines, 0 is replaced with `BOOL_FALSE` and 1 is replaced with `BOOL_TRUE`, which represent the hardened Booleans. When analyzing the programs, we have determined that the privileged points are the `if`-statements in the `verifyPIN` function, as they check whether the PIN was correct. Therefore, a successful attack can occur when the incorrect PIN is given, but the value returned from `byteArrayComare` is equal to the value in the condition of the `if`-statement in the `verifyPIN` function.

The result of running BitflipperVild on these two versions of VerifyPIN can be seen in Listing 6.3 and Listing 6.4. The results show that there are multiple possibilities where an attack can lead to VerifyPIN 0 being vulnerable due to the lack of hardened Booleans. In the RISC-V code corresponding to the above code snippets, the registers `a4` and `a5` correspond to the two values being compared. Therefore, we only show the value-sets for these registers. Furthermore, we only show some of the flipped values, including the relevant ones that can cause a successful attack.

Listing 6.3: The result from analyzing VerifyPin 0.

```
a4:
<ValueSet(None: {0, 1})>
<ValueSet((184, 185): {-2147483648, 1 ...})>
<ValueSet((241, 242): {-2147483648, 1 ...})>
<ValueSet((250, 254): {-2147483647, 0, ...})>
<ValueSet((269, 271): {-2147483648, -2147483647, 0, 1, ...})>

a5:
<ValueSet(None: {1})>
<ValueSet((271, 271): {-2147483647, 0, ...})>
```

Listing 6.3 shows the results achieved for VerifyPIN 0 at the label corresponding to the privileged point. The non-flipped value can either be 0 or 1 since we include both the case where the correct PIN is given, as well as the case where it is incorrect. To check whether an injection attack is possible, we assume that the entered PIN is incorrect. Furthermore, to successfully attack the program, either register `a4` should be flipped to 1, or `a5` should be flipped to 0. Based on our results these two cases are both possible.

Register `a5` can be flipped to 0 at label 271, which is part of the RISC-V instructions making up the instruction in line 12 in Listing 6.1. It is only possible to flip it at a single label since the value is loaded into the register just before it is used.

The register `a4` can be flipped to 1 at labels 184-185, 241-242, and 269-271. The instructions at labels 184-185 and 241-242 correspond to line 5 in the source code, and consists of a `li` and `j` instruction in the RISC-V code. The labels 269-271 correspond to the value in the `if`-statement, which consists of the instructions fetching the contents of the return register, as well as the instruction assigning the value 1 to register `a5`. The labels 250-254 are not relevant, since they represent when the value is flipped from 1, indicating a correct PIN was entered. If this value is flipped from 1, it is not classified as an attack, as none of the attack conditions are met.

The shortened result of running BitflipperVild on VerifyPIN 1 can be found in Listing 6.4. As seen in Listing 6.4, the value 170 cannot be flipped to 85 in register `a5`, since it is not included in the flipped values for label 274 at the privileged point. Furthermore, it is not possible to flip 85 to 170 in register `a4`, as seen in the values of the value-set for label 185-186.

Listing 6.4: The result from analyzing VerifyPin 1.

```
a4:
<ValueSet(None: {170, 85})>
<ValueSet((185, 186): {21, 4181, 134217813, 65621, 1073741909, 213,
   262229, 536870997, -2147483563, 341, 2097237, 69, 81, 84,
   268435541, 33554517, 4194389, 597, 67108949, 1109, 8388693,
   16777301, 524373, 131157, 1048661, 87, 93, 8277, 2133, 16469, 117,
    32853})> // From 85
<ValueSet((242, 243): {...})> // From 170
<ValueSet((251, 255): {...})> // From 170
<ValueSet((271, 271): {...})> // From 170 or 85
```

```
<ValueSet((273, 274): {...})> // From 170 or 85

a5:
<ValueSet(None: {170})>
<ValueSet((274, 274): {138, 162, 168, 65706, 2218, 8362, 131242,
   262314, 16554, 1073741994, 536871082, -2147483478, 174, 171,
   16777386, 186, 42, 4266, 8388778, 4194474, 524458, 2097322,
   1048746, 33554602, 268435626, 134217898, 67109034, 32938, 426,
   682, 234, 1194})>
```

Based on the above, BitflipperVild shows that programs without hardened Booleans are susceptible to this kind of bitflip attack, as it is possible to enter the privileged point for VerifyPIN 0 and not in VerifyPIN 1. Thus, the effect of the countermeasure applied by VerifyPIN 1 is visible in our analysis.

6.4.2 Fixed-time loop

The second countermeasure we examine is fixed-time loop. This countermeasure checks whether the number of iterations of a loop is correct. As seen in Listing 6.5, it checks whether variable `i` in `byteArrayCompare` has been incremented to be equal to `size` in VerifyPIN 2.

Listing 6.5: The code snippet from VerifyPin 2, which uses fixed-time loop.

```
1  BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size) {
2      int i;
3      BOOL status, diff = BOOL_FALSE;
4      for (i = 0; i < size; i++) {
5          if (a1[i] != a2[i])
6              diff = BOOL_TRUE;
7      }
8      if (i != size)
9          countermeasure();
10     if (diff == BOOL_FALSE)
11         status = BOOL_TRUE;
12     else
13         status = BOOL_FALSE;
14     return status;
15 }
16
17 void countermeasure() {
18     g_countermeasure = 1;
19 }
```

As mentioned in section 3.3, we unroll all loops in the program being analyzed. Therefore, to check whether we can discover this type of bitflip attack, we first need to check whether we have exited the loop prematurely. Secondly, if this is the case we need to check whether the condition in line 8 leads to the countermeasure being be false. The results achieved from analyzing VerifyPIN 2 using BitflipperVild can be seen in Listing 6.6.

Listing 6.6: The result from analyzing VerifyPin 2.

```
a4:
<ValueSet(None: {0, 1, 2, 3, 4})>
... // Flips before first check
<ValueSet(((197, 198): {4, ...})>
... // Flips before second check
<ValueSet((216, 217): {5, ...})>
... // Flips before third check
<ValueSet((235, 236): {6, ...})>
... // Flips before fourth check
<ValueSet((254, 255): {7, ...})>
... // Flips before fifth check
<ValueSet((273, 274): {5, ...})>

a5:
<ValueSet(None: {4})>
<ValueSet((183, 189): {0, -2147483644})>
<ValueSet((196, 198): {-2147483644})>
<ValueSet((215, 217): {0, -2147483644})>
<ValueSet((234, 236): {0, -2147483644})>
<ValueSet((253, 255): {0, -2147483644})>
<ValueSet((272, 274): {0, -2147483644})>
<ValueSet((307, 310): {0, -2147483644})>
```

Listing 6.6 shows the value-sets of register `a4` and `a5` for the "dummy" node, which contains the possible values when the loop is terminated prematurely. Register `a5` contains the `size` variable, whereas register `a4` contains the variable `i`. Based on the value-sets in register `a5`, it is possible to terminate the loop when the value is flipped from 4 to either 0 or -2147483644 since these are the only values that are less than or equal to the values in $\{0, 1, 2, 3, 4\}$. Thus, for each iteration, a flip can occur at the labels seen in Listing 6.6 at register `a5`. A bitflip can also occur in register `a4`, which flips one of the values 0, 1, 2, 3, or 4 to be equal to or greater than 4. This will also result in the loop terminating prematurely.

However, due to the countermeasure, seen in Listing 6.5 at line 9, not all the described attacks will be successful. This is because the value stored in register `a4` needs to be equal to the value in `a5`. If a bitflip occurs in register `a5`, it will only be equal to the value in register `a4` if it is flipped during the first iteration and is flipped to zero. Thus compared to a version without the countermeasure, the labels where a bitflip can lead to a successful attack are reduced to 183 - 189.

For register `a4`, the number of possible program points where a bitflip leads to a successful attack is also reduced. Since we only allow one bitflip, register `a4` needs to be exactly 4 when terminating the loop to ensure that it is equal to the value in register `a5`. Thus, the loop needs to be terminated before the first iteration, since the only value in $\{0, 1, 2, 3, 4\}$ that can be flipped to 4 with a single bitflip is 0. This can be seen in the four value-sets in Listing 6.6. The first one contains 4, while the remaining value-sets for the other checks contain 5, 6, 7, and 5 as the smallest value, respectively.

When examining the possible values for the false-node of the fixed-time loop countermea-

sure at line 8 in Listing 6.5, BitflipperVild arrives at the same conclusion. The value-sets for this node can be seen in Listing 6.7. Thus, to avoid the countermeasure, the values in registers `a4` and `a5` both need to be either 0 or 4.

```
a4:
<ValueSet(None: {0, 4})>
<ValueSet((212, 213): {4})>
<ValueSet((231, 232): {4})>
<ValueSet((250, 250): {4})>
<ValueSet((277, 278): {4})>
```

```
a5:
<ValueSet(None: {4})>
<ValueSet((183, 189): {0})>
<ValueSet((276, 278): {0})>
<ValueSet((307, 310): {0})>
```

Listing 6.7: The result from analyzing VerifyPin 2. The snippet is for the instruction at line 8 in Listing 6.5.

Based on these observations, the fixed-time loop countermeasure only reduces the number of possible successful bitflip attacks in `byteArrayCompare`. It does not completely remove the vulnerability. Our tool is able to catch these vulnerabilities, as it can determine at which labels a bitflip can occur, resulting in a premature termination of a loop that also skips the countermeasure.

6.4.3 Inlined Functions

Another way that a FISSC countermeasure improves the robustness of the programs is by changing the features of the program by inlining functions. Since VerifyPIN 3 implements this programming feature, the function `byteArrayCompare` does not exist and its body is instead part of the function `verifyPIN`. This improves the robustness since the omission of functions avoids passing parameters, which could be targeted by an injection attack [9]. To examine whether our tool is capable of detecting this type of vulnerability, we compare the analysis of versions 2 and 3.

After running BitflipperVild on both VerifyPIN 2 and 3, we found that the number of labels at which a bitflip could occur in register `a4` for version 2 is higher than for version 3. More specifically, for version 2 there were nine different labels at which a bitflip could affect the value in register `a4`, whereas there were only four different labels in version 3. This is because the value is stored in the return register, `a0`, and afterward loaded into register `a4` from `a0`, which introduces five more instructions, increasing the risk of a bitflip occurring. Based on this, our tool is able to detect that there is a higher risk when functions are not inlined.

6.4.4 PTC Decremented First and Backup

VerifyPIN uses a global variable called `PTC`, which keeps track of how many tries the user has used to enter a correct PIN. A bitflip can occur in the try counter, such that the attacker obtains more than the specified number of tries. In theory, if the PIN was of size four and the attacker had 10,000 tries, they would at some point guess the correct PIN if they systematically try all PIN combinations. Because these countermeasures focus on

the second condition for a successful attack, we will concentrate on this condition when analyzing the vulnerabilities.

VerifyPIN 3 does not implement any countermeasures that prevent an attack directed at the `PTC`. When running BitflipperVild on VerifyPIN 3, we can see that the variable is saved as a byte. This means that the largest signed value it can contain is 127. However, as the `PTC` variable is 2 after being decremented once, the largest value it can be flipped to is 66. Moreover, as only one bitflip can occur, it is only possible to get 66 additional tries, which might not be enough to guess the correct PIN. If the try counter is flipped when it is 0 or 1 instead, the largest values would be 64 and 65, respectively. Even though the try counter can not be flipped to a value close to 10,000, it does still increase the vulnerability of the code.

VerifyPIN 4 implements two countermeasures to combat this vulnerability. The first countermeasure starts the verification by decrementing the try counter. The second countermeasure creates a backup copy of the try counter. This is used to continuously check whether the try counter and its backup are the same, preventing vulnerabilities caused by a bitflip in the try counter. The implementation of the backup try counter can be seen in Listing 6.8.

Listing 6.8: The two comparisons to the backup try counter at the start of verifyPIN

```
1  BOOL verifyPIN() {
2      ...
3      if (g_ptc > 0) {
4          if (ptcCpy != g_ptc) countermeasure();
5          g_ptc--;
6
7          if (g_ptc != ptcCpy-1) countermeasure();
8          ptcCpy--;
9          ...
10     }
11 }
```

When analyzing VerifyPIN 4 with BitflipperVild, we found that the PTC backup countermeasure makes the program more robust, as it is not possible to flip a bit in the `g_ptc` variable without triggering the countermeasure. This is due to the multiple checks made to ensure that the PTC backup and the actual PTC are always the same, as seen in lines 4 and 7. Our tool shows that there are fewer bitflip vulnerabilities in VerifyPIN 4 compared to VerifyPIN 3, as it is not possible to increase the try counter with a single bitflip in version 4.

6.4.5 Loop Counter

The countermeasure enforcing a loop counter checks the number of times a loop has been iterated through when verifying the PIN. In FISSC, this is achieved by including a variable, `loop_counter`, which is incremented at the end of each loop iteration. This countermeasure is only able to make the program more robust if it is used in combination with the

fixed-time loop countermeasure. Therefore, after the loop has terminated both the variable `i`, which is used in the for-loop and the `loop_counter` are compared to the `size` variable. Thus, all three variables need to be equal to the same integer. VerifyPIN 4 implements these countermeasures and after running the analysis we achieved the following results.

Listing 6.9: The value-sets for registers `a4` and `a5` at the label comparing the loop counter with `size`.

```
a4: // loop_counter
<ValueSet(None: {0, 4})>
<ValueSet((240, 241): {4})>
<ValueSet((262, 263): {4})>
<ValueSet((284, 284): {4})>
<ValueSet((316, 318): {4})>

a5: // size
<ValueSet(None: {4})>
<ValueSet((317, 318): {0})>
```

Listing 6.10: The value-sets for registers `a4` and `a5` at the label comparing the variable `i` with `size`.

```
a4: // i
<ValueSet(None: {0, 4})>
<ValueSet((243, 244): {4})>
<ValueSet((265, 266): {4})>
<ValueSet((287, 287): {4})>
<ValueSet((321, 323): {4})>

a5: // size
<ValueSet(None: {4})>
<ValueSet((322, 323): {0})>
```

Listing 6.9 shows the value-sets for registers `a4` and `a5` at the instruction which skips the call to the `countermeasure` function when comparing `loop_counter` and `size`. Listing 6.10 shows the value-sets at the instruction which skips the `countermeasure` function when comparing `i` and `size`. As seen, the value-sets for these two instructions are identical, aside from the labels. They both indicate that the `countermeasure` call can be skipped if the loop iterates the right amount of times or if a flip occurs in register `a4` from 0 to 4 or in register `a5` from 4 to 0.

However, the values in `a4` refer to different variables. Thus, if the value in `a4` referring to `i` is flipped to 4, it is not possible to also skip the countermeasure for `loop_counter`, since we assume that only one bitflip can occur. Therefore, it is only possible to perform a successful attack if register `a5` is flipped to 0, as both the `loop_counter` and `i` are initialized to 0.

In short, this countermeasure does not prevent a successful attack with a single bitflip completely but prevents more attacks compared to a version without the loop counter. Thus, BitflipperVild is able to detect this type of vulnerability.

6.4.6 Double Calls

Another way of making a program more robust towards bitflips is to call the critical function twice. This results in it being necessary to perform an attack twice to meet the conditions for a successful attack. In VerifyPIN 5, the critical function is `byteArrayCompare` and this is therefore called twice. Both of the calls need to return true for the PIN to be accepted and provide access to the privileged point.

As observed in subsection 6.4.2, to successfully attack the program by skipping the critical part in `byteArrayCompare`, a bitflip should occur either in the `size` variable or in the variable `i`. Therefore, as we call the `byteArrayCompare` function twice, it is only possible

to prematurely terminate the loop in one of the instances of this function. The variable `size` is a parameter that is passed to the `byteArrayCompare` function. In RISC-V, a constant is an immediate, which is loaded into a register using instructions, such as `li` and `lui`. Therefore, it is not possible to flip the size before it is passed to the functions.

A scenario where a successful attack would be possible using a single bitflip would be if the `size` variable was called by reference, meaning that the `size` variable at the location of the callee would be updated by reference with the flipped value. However, this would defeat the purpose of having the double call countermeasure.

To test whether this countermeasure makes VerifyPIN more robust, we have run our tool on VerifyPIN 5. We found that our assumptions were correct, as a bitflip must occur in both function instances to perform a successful attack. This is not allowed since we assume that only one bitflip can occur. As the results of BitflipperVild reflect this, we deem our tool able to detect this type of vulnerability.

6.4.7 Limitations

Based on the above analysis and evaluation, we can see that BitflipperVild can detect vulnerabilities in versions 0-5 of the FISSC programs. We have thus shown that our tool can find vulnerabilities caused by single bitflip attacks. However, more advanced attacks that change the control flow or perform multiple bitflips are not supported by BitflipperVild.

As mentioned, one limitation is that we cannot detect injection attacks where a bitflip affects the control flow. This is because it would require analyzing changes in the CFG, which is not part of our scope, see section 3.3. Thus, BitflipperVild cannot show that the countermeasures double test and step counter prevent attacks. To support the detection of attacks that should be prevented by these countermeasures, it would require constructing a new domain which applies our bitflip analysis on the destination for control flow transfers. This analysis would create all possible CFGs for a given bitflip.

Another limitation is that the result needs to be interpreted to find vulnerabilities. Thus, the user needs to have a certain knowledge of the context of the program, as well as an understanding of different possible bitflip attacks. Moreover, to ensure that the attack is not a false positive, it is necessary to construct a path that proves that this is not the case. This can be mitigated by using the interpreter described in section 5.6, but the user still needs to have some understanding of the program being analyzed.

In our evaluation of the bitflip analysis module, we have used and compared the different versions of programs found in FISSC to understand the impact of the different countermeasures. Without the comparison, it might be a challenge to detect and identify vulnerabilities. This is due to the countermeasures already defining specific vulnerabilities, which allows a user to identify which subset of the result they should consider, improving comprehensibility.

As seen, the requirements in the "bitflip analysis" category are satisfied, as we have shown that we can detect vulnerabilities in the registers. This is supported by the analyses per-

formed on the VerifyPIN versions. Moreover, as seen in the listings containing the results of the analysis, the labels at which a bitflip can occur are shown. Thus, both the requirements regarding proving countermeasures prevent vulnerabilities, as well as the bitflip analysis requirements have been fulfilled.

6.5 Benchmarks

To assess the performance of the analysis executed by BitflipperVild, we will in this section run benchmark tests on our tool. These include both individual tests on each module, as well as tests on how the tool as a whole performs. These tests were conducted on a MacBook Pro with 8 GB RAM and an Intel Core i5 with 2,3 GHz. The results can be seen in Table 6.3, which shows the average runtime of ten iterations for each component in seconds. Benchmarks tests were not conducted on the interpreter module, as it would be difficult to compare the different version due to the interpreter requiring a privileged point.

	Assumptions	**Complete CFG**	**Bitflip analysis**	**Entire analysis**
v0	0.03	0.59	15.98	22.13
v1	0.03	0.64	18.79	26.37
v2	0.03	0.80	38.40	43.25
v3	0.03	0.65	24.21	20.40
v4	0.04	1.11	44.21	52.99
v5	0.05	1.47	63.18	75.15
v6	0.04	0.85	24.80	25.94
v7	0.05	2.44	147.9	151.23
Avg	**0.04**	**1.07**	**47.18**	**65.93**

Table 6.3: The runtime of the different components on the seven versions of VerifyPIN. All results are shown in seconds.

As seen, the bitflip analysis module spends on average 47.1798 seconds to analyze each file. Compared with the other modules, which spend at most 1.07 seconds, the majority of the runtime is due to the bitflip analysis module. This is because the module analyzes the same nodes several times to ensure that we find a fixed point.

For example, the number of nodes in the CFG for v7 is 565. The number of nodes analyzed in the CFG recovery module is 1808, while it is 10667 for the bitflip analysis module. This means that every node in the CFG is analyzed on average 3 times in the CFG recovery module and 19 times in the bitflip analysis module.

The difference between the modules is also due to the bitflip analysis environment containing a larger amount of values, since the domain includes flipped values. Thus, a given environment of a program point is more likely to change when a node has multiple predecessors, as this can affect the environment of its successors. This can be seen in Appendix C.

In Table 6.3, the runtime for the entire analysis can be found. This includes performing automatic assumption adherence, a complete CFG recovery, and a bitflip analysis. The

connection between the number of calls to the transfer functions and the runtime for each file can be seen in Figure 6.4, where each point corresponds to a VerifyPIN version.

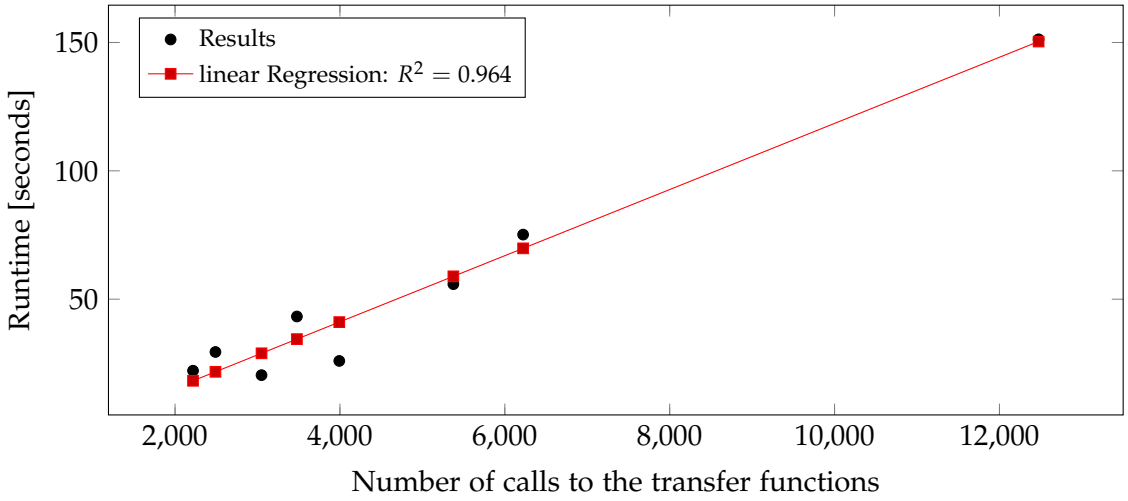


Figure 6.4: A linear regression plot showing the correlation between number of analyzed nodes and the runtime of BitflipperVild

It can be seen that there is linear growth when comparing runtime with calls to the transfer functions. VerifyPIN 7 spends approximately 2.5 minutes performing the entire analysis. However, this program contains multiple calls to duplicated `countermeasure` functions, which results in a large number of nodes with multiple predecessors being analyzed. Since different transfer functions have different implementations, the specific runtime for each instruction will vary. Thus, it is not an exact correlation.

Based on the above, we can see that the runtime is influenced by the amount of calls to transfer functions, which is influenced by the number of nodes in the given program, as well as the number of nodes with multiple predecessors. Furthermore, other factors such as which instructions are analyzed influence the runtime. Given that BitflipperVild is an analysis tool, we deem that its average runtime of approximately one minute for the programs in FISSC is acceptable.

Chapter 7

Conclusion

The problem presented in Chapter 1 addresses bitflip vulnerabilities and states that we must be able to analyze and find the effect of bitflips on the PIN checkers in FISSC. To further specify the different aspects of the problem, we have introduced requirements in section 3.1 necessary to solve the problem. All of these requirements were met, as concluded in Chapter 6.

The problem was solved by creating the tool BitflipperVild which can analyze RISC-V programs. This tool is a proof of concept. BitflipperVild can create a CFG based on a set of RISC-V instructions and then either perform backward slicing or a bitflip analysis on the CFG. To assist the user in gaining a higher comprehension of the results provided by the tool, we have also created an interpreter module. This module can find the bitflips which are able to bypass the authentication indicated by a given privileged point.

To construct BitflipperVild, we have created syntax and semantics describing the RISC-V instruction set architecture, as well as a fault model describing bitflip vulnerabilities occurring in registers. Based on these definitions, we have created a value-set analysis, by utilizing the monotone framework, we have defined the domain to be a complete lattice following the ascending chain condition and the transfer functions to be monotone.

Using these definitions we have implemented the analysis, which is able to successfully describe all possible bitflips in the program. However, the approach taken to implement BitflipperVild is based on assumptions described in section 3.3. These assumptions were made to reduce the amount of over-approximation produced by the analysis. The scope of the analysis is reduced to analyze programs found in FISSC, by introducing these assumptions. As a consequence, the authenticity of the analysis result is limited when running the tool on other programs.

As BitflipperVild is a proof of concept, we deem that the assumptions are appropriate for the stated problem. Furthermore, as seen in Chapter 6, we were able to find bitflips leading to an authentication bypass in the PIN checkers due to bitflips in the registers. Thus, we conclude that BitflipperVild successfully addresses the problem stated.

Chapter 8

Future Work

As mentioned in Chapter 7, BitflipperVild is able to address the problem stated in Chapter 1. However, since BitflipperVild is a proof of concept and is based several assumptions, future work can be performed to widen its scope and make it into a more usable tool. We therefore discuss ways to relax the assumptions and to scale the solution in this chapter.

8.1 Relaxing Assumptions

An alternative to automatically enforcing the assumptions on a given RISC-V program, as seen in section 5.1, is to either remove or relax these assumptions instead. This could involve handling `printf` and `scanf` functions, instead of replacing them.

One assumption we can relax is the removal of setup functions. This would make it possible to calculate the exact addresses where the memory is saved in the heap instead of using symbolic values. A downside to this relaxation is that it will likely increase the runtime since more nodes will be a part of the CFG, and we need to calculate the addresses where the memory is saved for when a `STORE` instruction is analyzed.

We could also handle loops instead of performing loop unrolling. However, this change would lead to a larger over-approximation, since we would need to execute the loop according to the flipped values to ensure soundness. Thus, we could risk executing the loop up to 2^{32} times. In our proof of concept, we do not remove this assumption, as this would cause a less usable result.

Another possible assumption removal would be to also flip the stack and frame pointer. This would increase the run time. However, bitflips in these registers only affect the location of the values saved in the memory. As we do not consider this type of fault model, we do not deem this an essential change to our proof-of-concept. However, if BitflipperVild was expanded to include this fault model, it would be necessary to remove this assumption.

8.2 Scaling

To increase the scope of BitflipperVild, it can be expanded with additional extensions. These include increasing the number of fault models which the tool is compatible with, as

well as the number of ISAs it can analyze.

8.2.1 Fault Models

A way to expand the scope of BitflipperVild is to extend the number of fault models. The two fault models discussed in this report, which are not included in our tool, are bitflips in the program counter, the instructions, and the heap.

To include the fault model regarding bitflips in the program counter, we would have to add a flip function to the transfer function of the `CFGDomain`. This function needs to take into account which address the program counter is flipped to. If the address is not a part of the program scope, then it should not be included in the CFG. However, if the address that the program counter jumps to, points to a smaller address, it means that it has created a loop. Assuming that our assumptions are not relaxed at this point, we would have to loop unroll each time this occurs. As a consequence, this would considerably increase the complexity of the CFG. Therefore, it would be advantageous if the assumption of loop unrolling was either relaxed or removed.

To make the analysis compatible with the fault model describing bitflips in the encoding of an instruction, we would need to implement a new flip function, as well as a new representation of the nodes in the CFG. The representation would take the encoding as input, and based on this encoding derive which instruction and registers are used. The flip function would take the encoding as input, instead of values in registers. Each bit in the encoding would be flipped and then checked for whether they represent actual instructions or registers. If this is the case, the transfer function would need to be applied multiple times for each possible encoding. In cases where the instruction is flipped to a control flow instruction, a more extensive modification would be required, as this would have to be reflected in the CFG.

To include compatibility with the fault model describing bitflips in the heap, a new domain would need to be added. This domain should handle flipping values stored in the heap. Similar to the domain described in section 5.3, this flip function would be part of the transfer function and would have a similar implementation as the domain for the current fault model. This implementation would thus have the same complexity as our current implementation.

There exist other types of fault models than those mentioned in this report. As we have implemented BitflipperVild as a modular program, it improves compatibility with other types of fault models by simplifying the implementation. However, the complexity of implementing another fault model depends on the properties of the model.

8.2.2 Architectures

Another way of scaling BitflipperVild is to include the 64-bit architecture for RISC-V as well. This would make it possible to analyze a larger number of programs. However, this addition will increase the runtime and memory usage of BitflipperVild, as the number of

possible flipped values are increased from 32 to 64. This might require a more efficient implementation of BitflipperVild.

Moreover, it is also possible to include other types of ISAs. This modification would increase the usability of our tool, as it would not be limited to a single type of architecture.

This expansion can be achieved in several ways. A naive way to scale the compatibility with different architectures would be to create new syntax, semantics, and transfer functions for each new ISA included. In the implementation, we would then need to implement transfer functions for each ISA and then choose which set of transfer functions should be used in the analysis based on the input. However, this would require a complex implementation, as well as duplicate code, since the different ISAs have similar instruction sets and functionalities. In short, this would reduce the reusability of BitflipperVild.

Another way would be to use a similar approach to angr [19], where we would convert the program to an intermediate representation (IR), which would then be analyzed. This would entail changing transfer functions to handle our IR instead of the RISC-V code. Thus, we would also need to define the syntax and the semantics for the IR. For each of the ISAs, we would need to implement a converter that converts the instructions to our IR. This would make it straightforward to include a new ISA without having to create a domain from scratch.

8.2.3 Conventions

Some of the decisions and assumptions made are based on the observed conventions of the RISC-V programs produced by the cross-compiler [17]. However, not all compilers create the same output. Furthermore, it is also possible to manually write RISC-V code without using a compiler, and therefore not follow these conventions. To expand the scope of BitflipperVild, we could thus generalize these conventions.

One of these conventions is the structure of a RISC-V program. We have observed that a function is always structured in columns following the order address, instruction encoding, instruction, and arguments. However, if this structure is not followed, an error would occur in our CFG module. To create a more general tool, we could automatically detect which columns contain the different parts. To identify the columns containing the instructions and arguments, we could specify a list of possible instructions and registers. Furthermore, addresses could be detected by comparing multiple lines and checking whether there is a difference of four between each hexadecimal value. Lastly, the instruction encoding would be the remaining column, assuming only these parts are included in the program.

Another convention specifies that all branching instructions jump to instructions within the same function. This convention allows us to simplify loop unrolling. However, it would be possible to not adhere to this convention by detecting whether the destination instruction is within the same function and then creating a special case for branching to other functions.

Taking these suggestions for future work into consideration, BitflipperVild could have the potential to be used in a wider range of cases and real-life cases, by expanding its scope and improving its usability.

Bibliography

[1] *About RISC-V*. URL: https://riscv.org/about/. (accessed: 10.03.2023).

[2] Université Grenoble Alpes. *FISSC: the Fault Injection and Simulation Secure Collection*. URL: https://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/. (accessed: 18.04.2022).

[3] angr. *throwing a tantrum, part 1: angr internals*. URL: https://angr.io/blog/throwing_a_tantrum_part_1/. (accessed: 17.05.2023).

[4] *angr-platforms*. URL: https://github.com/angr/angr-platforms. (accessed: 08.03.2023).

[5] *BitlfipperVild GitHub Repository*. URL: https://github.com/IdaThoft/BitflipperVild. (accessed: 27.04.2023).

[6] Jakub Breier and Xiaolu Hou. "How Practical Are Fault Injection Attacks, Really?" In: *IEEE Access* 10 (2022), pp. 113122–113130. DOI: 10.1109/ACCESS.2022.3217212.

[7] David Brumley et al. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 463–469. DOI: 10.1007/978-3-642-22110-1_37.

[8] Ida Thoft Christiansen and Lena Said Ernstsen. *Static Analysis in RISC-V*. 2022.

[9] Louis Dureuil et al. "FISSC: A Fault Injection and Simulation Secure Collection". In: *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*. Ed. by Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch. Vol. 9922. Lecture Notes in Computer Science. Springer, 2016, pp. 3–11. DOI: 10.1007/978-3-319-45477-1_1.

[10] Stephen Cole Kleene. *Introduction to Metamathematics*. Princeton, NJ, USA: North Holland, 1952. DOI: 10.2307/2268620.

[11] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University, https://cs.au.dk/~amoeller/spa/. Oct. 2018.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6.

[13] Colin O'Flynn and Zhizhang (David) Chen. "ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research". In: *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 8622. Lecture Notes in Computer Science. Springer, 2014, pp. 243–260. DOI: 10.1007/978-3-319-10175-0_17.

[14] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Trans. Reliab.* 51.1 (2002), pp. 63–75. DOI: 10.1109/24.994913.

[15] Jens Palsberg and Christina Pavlopoulou. "From Polyvariant flow information to intersection and union types". In: *Journal of Functional Programming* 11.3 (2001), pp. 263–317. DOI: 10.1017/S095679680100394X.

[16] *PyVEX*. URL: https://github.com/angr/pyvex. (accessed: 27.09.2022).

[17] *riscv-gnu-toolchain*. URL: https://github.com/riscv-collab/riscv-gnu-toolchain. (accessed: 4.11.2022).

[18] Yan Shoshitaishvili et al. "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware". In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL: https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware.

[19] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[20] *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally*. URL: https://iot-analytics.com/number-connected-iot-devices/. (accessed: 8.03.2023).

[21] *V4:Tutorial A2 Introduction to Glitch Attacks (including Glitch Explorer)*. URL: https://wiki.newae.com/index.php?title=V4:Tutorial_A2_Introduction_to_Glitch_Attacks_(including_Glitch_Explorer)&mobileaction=toggle_view_mobile. (accessed: 24.05.2023).

[22] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 2.2. RISC-V Foundation. May 2017.

Appendix A

RISC-V Registers

The registers mentioned in Chapter 2 are defined in Table A.1. The table is copied directly from our previous work [8].

Register name	Assembly name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate link register
x6-7	t1-2	Temporary register
x8	s0/fp	Saved register (frame pointer)
x9	s1	Saved register
x10-11	a0-1	Function argument / return value registers
x12-17	a2-7	Function argument registers
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporary registers

Table A.1: Mapping from register names to names produced by the compile. [22]

Appendix B

Semantic Evaluation Functions

To keep our semantic definitions generic in section 2.2, we use semantic evaluation functions to describe the semantics of the different specific instructions. Below the semantic evaluation for all arithmetic and logical operations, `OP`, can be found.

$$
\begin{aligned}
[\![ADD]\!](v_1, v_2) &\triangleq v_1 +_{32} v_2 \\
[\![SUB]\!](v_1, v_2) &\triangleq v_1 -_{32} v_2 \\
[\![SLL]\!](v_1, v_2) &\triangleq v_1 << v_2 \\
[\![SRL]\!](v_1, v_2) &\triangleq v_1 >> v_2 \\
[\![SRA]\!](v_1, v_2) &\triangleq v_1 >> v_2 \\
[\![AND]\!](v_1, v_2) &\triangleq v_1 \ \& \ v_2 \\
[\![OR]\!](v_1, v_2) &\triangleq v_1 \mid v_2 \\
[\![XOR]\!](v_1, v_2) &\triangleq v_1 \oplus v_2 \\
[\![SLT]\!](v_1, v_2) &\triangleq v_1 < v_2 \\
[\![SLTU]\!](v_1, v_2) &\triangleq v_1 < v_2
\end{aligned}
$$

Below the semantic evaluation for all immediate operations `OPI`, can be found.

$$
\begin{aligned}
[\![ADDI]\!](v, imm) &\triangleq v +_{32} imm \\
[\![SUBI]\!](v, imm) &\triangleq v -_{32} imm \\
[\![SLLI]\!](v, imm) &\triangleq v << imm \\
[\![SRLI]\!](v, imm) &\triangleq v >> imm \\
[\![SRAI]\!](v, imm) &\triangleq v >> imm \\
[\![ANDI]\!](v, imm) &\triangleq v \ \& \ imm \\
[\![ORI]\!](v, imm) &\triangleq v \mid imm \\
[\![XORI]\!](v, imm) &\triangleq v \oplus imm \\
[\![SLTI]\!](v, imm) &\triangleq v < imm \\
[\![SLTIU]\!](v, imm) &\triangleq v < imm
\end{aligned}
$$

Below the semantic evaluation functions for all comparison operations, `CMP`, can be found.

$$
\begin{aligned}
[\![BEQ]\!](v_1, v_2) &\triangleq v_1 = v_2 \\
[\![BNE]\!](v_1, v_2) &\triangleq v_1 \neq V_2 \\
[\![BLT]\!](v_1, v_2) &\triangleq v_1 < v_2 \\
[\![BLTU]\!](v_1, v_2) &\triangleq v_1 < v_2 \\
[\![BGE]\!](v_1, v_2) &\triangleq v_1 \geq v_2 \\
[\![BGEU]\!](v_1, v_2) &\triangleq v_1 \geq v_2
\end{aligned}
$$

Appendix C

Benchmark results

To show the relation between the number of calls to the transfer function compared with the number of nodes and predecessors, we have included the following table. This table is used in section 6.5.

	No. of calls to the transfer functions	**No. of nodes**	**No. of predecessors**
v0	2218	294	8
v1	2490	317	10
v2	3049	321	15
v3	3478	336	15
v4	3992	344	20
v5	5376	393	24
v6	6222	477	16
v7	12475	565	28

Table C.1: A table comparing the number of calls to the transfer functions in both the CFG recovery and bitflip analysis modules with the number of nodes in the CFG and the number of nodes with more than one predecessor.