SPECULOOS
ARCHITECTURE MANUAL

ARCHITECTURE VERSION 1.1
DECEMBER 6, 2016

ADRIEN MICHELET
CYRIL BRESCH
LAURENT AMATO
THOMAS MEYER

TABLE OF CONTENTS

TABLE OF FIGURES

ABOUT

INTRODUCTION

The goal of this document is to explain the principle of 2 different attacks on the memory of a OpenRISC system leveraging the stack limitations and describes how have been coded and implemented a hardware countermeasure known as the Speculoos defense module.

AUTHORS

Cyril Bresch	cyril.bresch.fr@gmail.com
Adrien Michelet	a.michelet-gignoux@hotmail.fr
Laurent Amato	amato.laurent@gmail.com
Thomas Meyer	thomasm0301@gmail.com
General e-mail	csaw.esisar@gmail.com

WORK IN PROGRESS

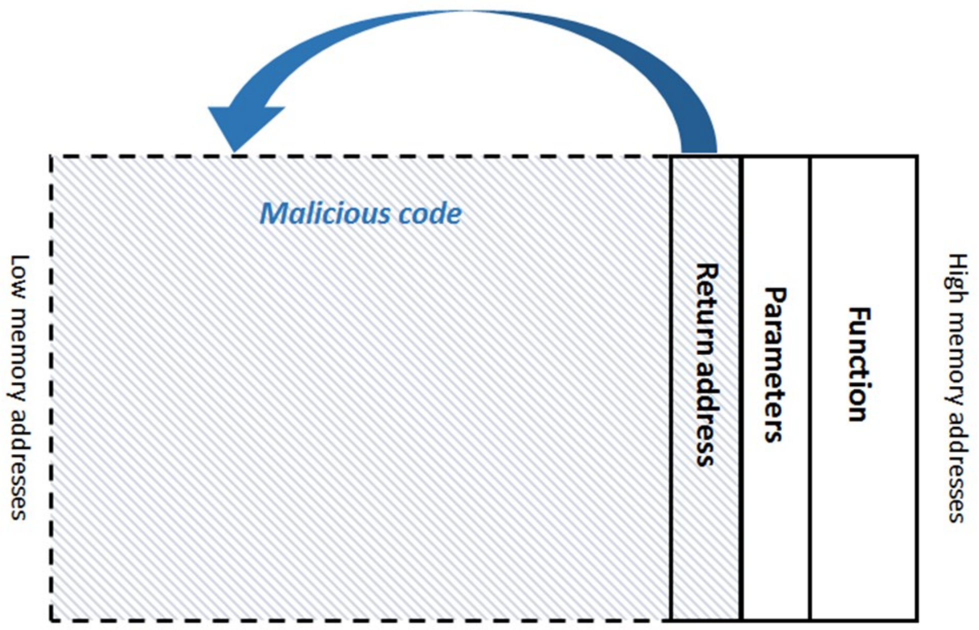
- Linux debugging
- Baremetal attacks and Speculoos on board test
- End of process detection and lock of the fsm
- Full stack management

ATTACKS

BUFFER OVERFLOW

A buffer overflow is a functional bug which happens when a program tries to write outside of an allocated buffer memory. When it happens, some assembly instructions of the program are erased and the functioning of this one became unstable. This kind of bug can be exploited in order to violate the security policy of a system, for instance it can allow one to get a shell and to gain privileges access. In C language, functions like “gets” or “strcpy” get characters from the standard input stream and put it into a buffer. These functions are vulnerable because they do not check if the character string in input matches with the length of the reserved buffer.

When a function is called, the instruction used by the OpenRISC is “jump and link”. After these parameters of the function are pushed on the highest addresses of the stack, just below, the return address of the function is pushed, the frame pointer, the local argument of the function and eventually the buffer. In a standard buffer overflow attack, the attacker uses assembly code injection, also called payload, into the vulnerable buffer. The length of the payload is bigger than the length of the buffer so the payload overwrites the local argument of the function, the frame pointer and the return address.



Once the return address of the function is modified, the function may never return into the calling function and an interruption like segmentation fault is raised. To avoid a segmentation fault, the attacker must replace the return address by the address of the top of the buffer as shown in figure 1, the buffer contains opcode instructions to get a shell as example.

BUFFER OVERFLOW EXPLANATION

In this section the buffer overflow code will be exposed and explain. It is important to remind that the OpenRISC architecture processor has its address in 32bits. In order to simulate a real attack with buffer overflow this attack will be launch just after a little bash script which will be explains in the section “Launching the Buffer Overflow Attack”.

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4
 5  void shell()
 6  {
 7      printf("=====Malicious Shell=====");
 8      setuid(0);
 9      system("/bin/sh");
10  }
11
12  void exploit()
13  {
14      char vulnbuffer[16];
15      char buffer[32];
16      int redirect = (int) shell; //adresse de exploit
17      int offset = 32-4;
18
19      printf("=====Exploit=====\n");
20      printf("Adresse de la fct : 0x%x\n", redirect);
21      memset(buffer, 0x41, 32);//On rempli le buffer de A
22      buffer[offset+3] = redirect & 0xff; //On stocke le dernier octet de l'adresse
23      buffer[offset+2] = (redirect >> 8) & 0xff;//On décale d'un octet pour stocker le 2nd octet de l'adresse
24      buffer[offset+1] = (redirect >> 16) & 0xff;
25      buffer[offset] = (redirect >> 24) & 0xff;
26
27      printf("buffer[31] : 0x%x\n", buffer[offset+3]);
28      printf("buffer[30] : 0x%x\n", buffer[offset+2]);
29      printf("buffer[29] : 0x%x\n", buffer[offset+1]);
30      printf("buffer[28] : 0x%x\n", buffer[offset]);
31
32      memcpy(vulnbuffer, buffer, 32);
33  }
34
35  int main(int argc, char** argv)
36  {
37      //int i;//start 28
38
39      printf("*****START*****\n");
40      //i = atoi(argv[1]);
41      exploit();
42      printf("*****END*****\n");
43
44      return 0;
45  }
```

The exploit is coded in C language. Libraries that are used for the exploit: stdio.h, stdlib.h, string.h, unistd.h (for system call).

Three function are declare in this code: main() function, shell() function, and exploit() function. In a normal execution the shell() function will never be called.

In fact the main() function call the exploit() function , then the exploit() function will return into the main.

In the exploit() function you can see that a buffer is reserved in the stack. The buffer may contain 32 characters. (Line 15)

```
char buffer[32];
```

Just before the 32 characters’ buffer there is a vulnerable buffer at Line 14 called vulnbuffer. This buffer may contain 16 characters.

```
        char vulnbuffer[16];
```

After that the function get the adresse of the malicious shell function.

```
        int redicrect = (int) shell;
```

At line 21 the memset function put 32 characters 'A' into the buffer.

```
        memset(buffer), 0x41,32);
```

So, once the buffer is full, the exploit function will put the redirection address to the function shell into the buffer. The address of the shell function has for size 4 octets because as it has been said before the OpenRISC architecture is in 32 bits. As a result of this operation the address will be put on the bottom of the buffer.

```
        buffer[offset+3] =  (redirect) & 0xff;
        buffer[offset+2] =  (redirect>>8) & 0xff;
        buffer[offset+1] =  (redirect>>16) & 0xff;
        buffer[offset] =  (redirect>>24) & 0xff;
```

When the function vuln has been called by the main function, the return address (4octets) and the frame pointer (4octets) has been pushed into the stack. After that, 16 octets has been reserved for the vulnbuffer, and 32 for the buffer. With the function memcpy the exploit copy the buffer into the vulnbuffer, as the length of these two buffers are different by 16 octets you can notice that the octet between 16 and 24 of the buffer will take the place of the frame pointer, and the octet between 24 and 32 will replace the return address. But the octet between 24 and 32 into the buffer is the address of the shell function. So at the end of its execution the function will return into the shell() function.

Shell() function details:

So if the shell function will be launch the system call setuid() will give the root privileges to the exploit and then the system call system() will return a shell to the attacker.

```
         setuid(0);
         system("/bin/sh");
```

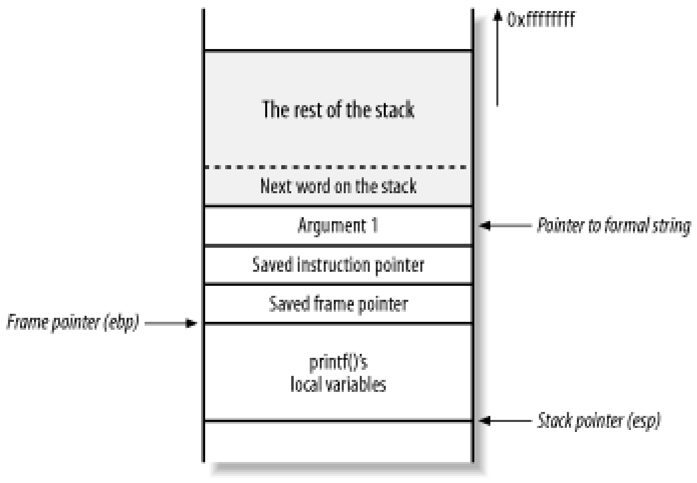
FORMAT STRING VULNERABILITY

In this subsection, we present another kind of attack. In fact, we use the printf/scanf vulnerability, which is also called the string format attack. This type of vulnerabilities is less present than a buffer overflow in programs, but their exploitation is as much as powerful as buffer overflow. We are going to demonstrate that is it possible to modify variables in a program with the function printf().

There are two different way to print a variable into the screen of a user.

This way: printf("%s",buffer);
Or this way: printf(buffer);

The second way is more economic in number of characters and takes less time to write for the programmer. But it opens a vulnerability. In fact the function printf() has no argument so if the buffer contains a string like "%x", the printf() function will try to return a value which is consider as argument. On the figure, you can see that the arguments of the function are just above the saved instruction pointer, but if the function has no argument it will return value into the stack.



Once it's done an attacker can fuzz he program and inject some formatter characters in order to explore the stack. But the printf() function has also a formatter which permit to write directly in memory: "%n". The formatter "%n" permit to write the number of character before him.

Example: aaaaa%n will write 5 in memory.

FORMAT STRING EXPLOIT EXPLANTATION

In its normal functioning, our vulnerable program gets a username and after that it calculates the ten first terms of the Fibonacci sequence. The program does not allow the user to modify any variables during the execution.

```c
#include<stdio.h>


int main()
{

    int a=0;
    int b=1;
    int c=0;
    char buff[20];

    int i=10;
    int j=0;


    printf("calcul of i= %d = %x first terms of Fibo, i is at address : %x \n",i,i,&i);

        printf("Enter name for database\n");
        scanf("%s",buff);

            printf(buff);

                printf("valeur de i = %d\n",i);

        for(j=0;j<i;j++)
        {
           c = a +b;
           a = b;
           b = c;
            printf("%d\n",c);

        }





return 0;

}
```

The exploit is coded in C language. Libraries that are used for the exploit: stdio.h. The variable i define the number of terms of the Fibonacci sequence which will be calculate.

First function asks for the name of the user:

```
printf("enter username for the database : \n\n");
```

Second function printf() just return the name of the user. As you can see, the printf function here is vulnerable.

```
printf(comments);
```

SPECULOOS MODULE

OVERVIEW

The speculoos module is composed by 3 different modules: a shadow stack, a finite state machine (FSM) and an observer.

- The shadow stack stores return addresses.
- The FSM manages the stack, checks if the return address is correct and orders interrupts of the system.
- The observer does the link between the OpenRISC architecture and the core of Speculoos by analyzing the instruction flow.

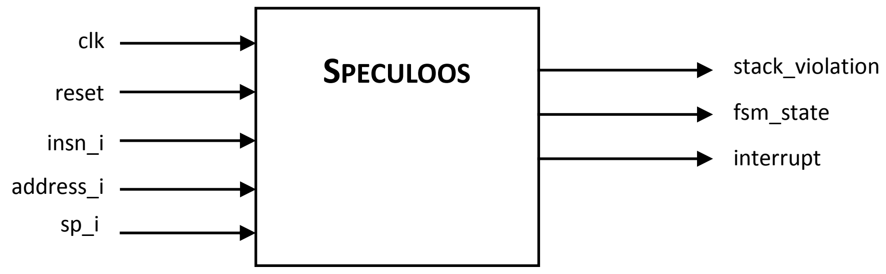


Figure 1: Speculoos block diagram

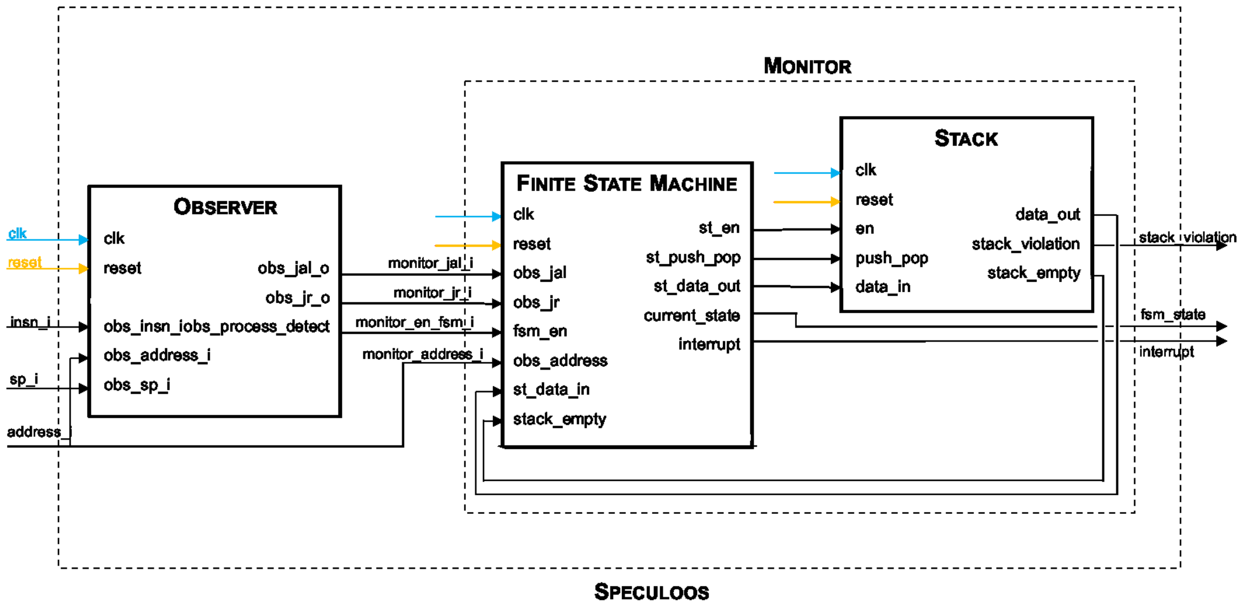


Figure 2: Speculoos detailed block diagram

STACK MODULE

The stack module is a shadow stack accessible only by a hardware connection. Its main role is to store return addresses. The stack can contain 128 vectors of 4 bytes. The value of the number of vector which can be store can be easily modify in the Verilog source code of the Stack module.

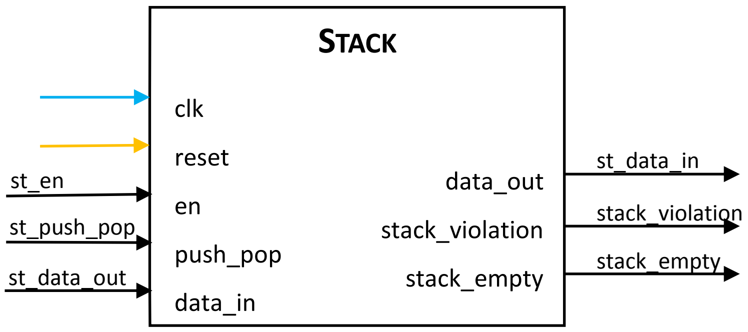


Figure 3: Stack block

Inputs

- clk: clock
- reset: reset
- en: enable the stack (necessary for pushing or popping the second stack)
- push_pop:
  - High for pushing an address in the stack
  - Low for popping the last address of the stack
- data_in: return address to store into the stack

Outputs

- data_out: return address popped
- stack_violation: stack full
- stack_empty: stack empty

FSM MODULE

The finite state machine module is part of the Speculoos core. It receives orders from the observer module and manages the shadow stack. Its role is also to check if the return address is correct and can order an interrupt of the system.

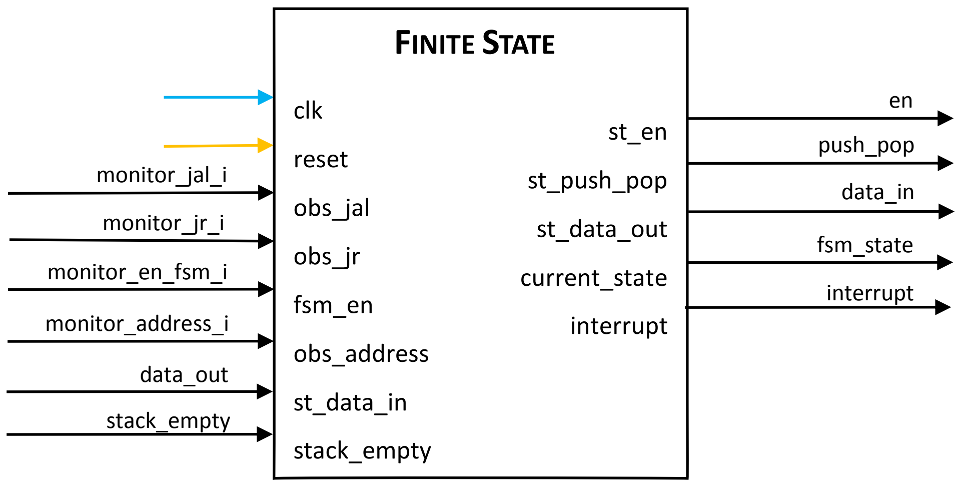


Figure 4: Finite state machine block

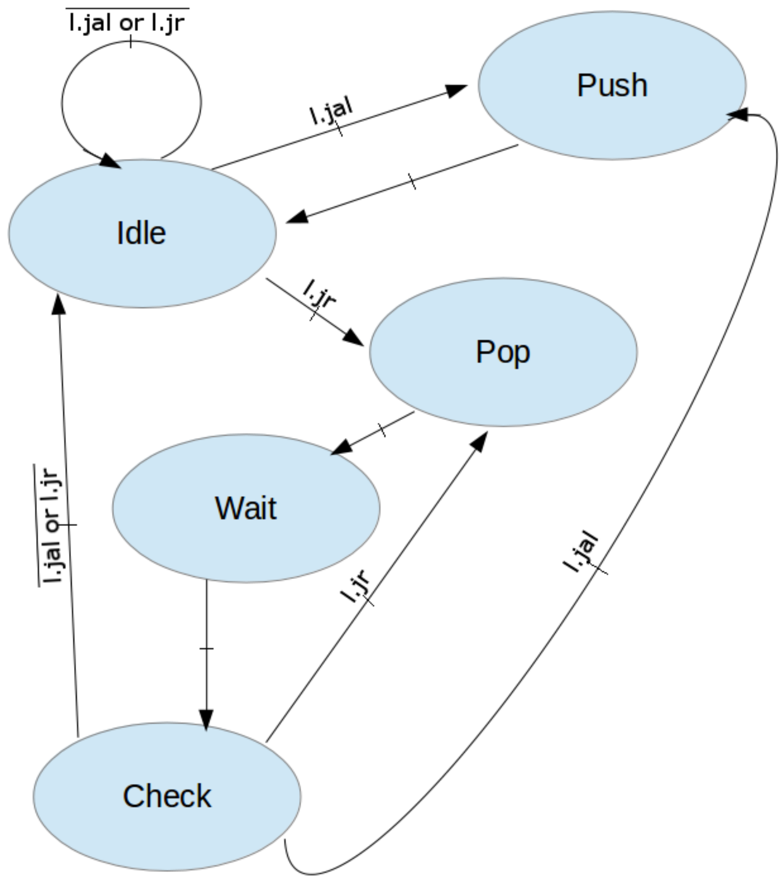
Inputs

- clk: clock
- reset: reset
- obs_jal: jump detection
- obs_jr: return detection
- fsm_en: enable the stack
- obs_address [31:0]: register R9 input
- st_data_in [31:0]: address popped from the stack
- stack_empty: stack empty

Outputs

- st_en: enable the stack
- st_push_pop:
  - High for pushing an address in the stack
  - Low for popping the last address of the stack
- st_data_out [31:0]: Return address from R9
- fsm_state: current state of the FSM (debug output)
- interrupt: high if the return address and the address popped are not the same

The FSM has 5 states: Idle, Push, Pop, Wait and Check, allowing it to order to push an address in the stack, to pop an address from the stack or to check if the address in R9 is correct. The way to move from a state to another is describe by the following diagram.



$\overline{\text{l.jal or l.jr}}$
l.jal
Push
Idle
l.jr
Pop
Wait
$\overline{\text{l.jal or l.jr}}$
l.jr
l.jal
Check

Figure 5: Finite state machine diagram

Idle

Firstly, the observer must detect a process to unlock the FSM which will be unlock until a reset. The register fsm_enable will be active high as soon as the process is detected.

If the observer detects a jump, it means that the execution program is calling a function and is going to save its return address on the top of the OpenRISC stack. So, the Speculoos module must also save this address into the shadow stack. The next state will be Push in order to save the return address of the caller function into the stack module.

If observer detects a return, it means that the execution function will end and return to the caller function. So, the Speculoos module must get the address of return and compare it with the last address saved in the stack module. In order to compare them, the next state will be Pop then Wait, and finally Check.

Push

In this state the stack will be enable with the output st_en. To push the address function, the output st_push_pop must be active high and the output st_data_out must be the address in question. After that the finite state machine will return to the state Idle.

Pop

In this state the stack will be enable with the output st_en. To pop the address function on the top of the stack the output st_push_pop must be active low. The input st_data_in will get the address pop by the stack. The next state will be Wait.

Wait

This state is used to synchronize the return address given by the stack on OpenRISC and the return address pop by the stack module. The next state will be Check in order to compare these two addresses.

Check

This state will compare with a XOR the address return popped by the stack and the address return in input of the finite state machine. If these addresses are not the same the interrupt output will be use to interrupt the program execution. Else if the observer detects a jump or a return the next state will be respectively Push or Pop. Else, the finite state machine will return to the state Idle.

OBSERVER MODULE

The observer block is the most critical block of Speculoos. In fact, its role is to link the OpenRISC architecture and the Speculoos core (inside the monitor). In order to ensure this, the Speculoos observer has 3 functions. The first one is to analyze the instruction flow as the instruction decoder. The second one is to detect and validate the begin of a process, the "jump and link" functions corresponding to the call of of intern or systems functions and a return into the caller. The last one is to inform the monitor especially the finite state machine at the right time.

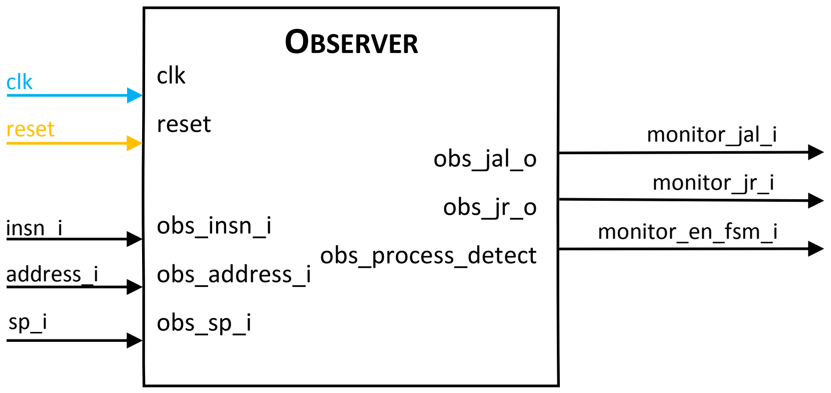


Figure 6: Observer block

Inputs

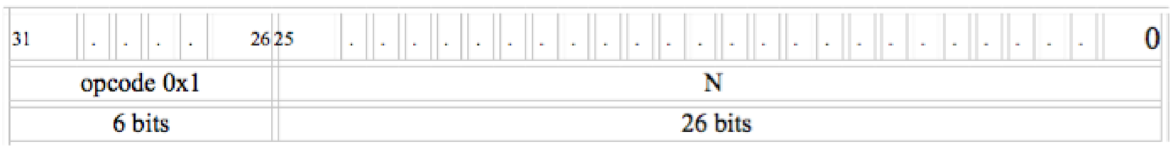
- clk: clock
- reset: reset
- obs_insn_i [31:0]: instruction code input
- obs_address_i [31:0]: register R9 input
- obs_sp_i [31:0]: stack pointer input (R1)

Outputs

- obs_jal_o: jump detection
- obs_jr_i: return detection
- obs_process_detect: process detection, unlock the fsm

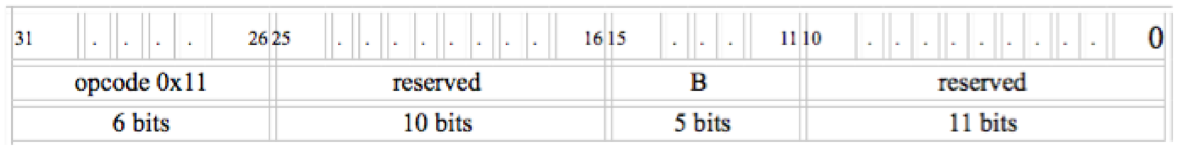
Most of instructions are coded in 32 bits with the first 6 bits corresponding to the opcode. The next 26 bits are reserved to parameters and are distributed by category of instructions. The Speculoos observer only focus on some determined instructions as l.sw, l.addi, l.lwz, l.jal and l.jr.

l.jal



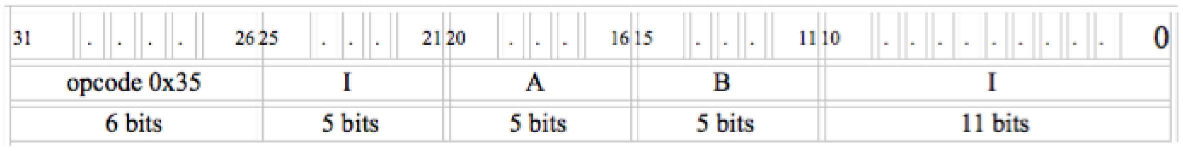
31 26	25 0
opcode 0x1	N
6 bits	26 bits

l.jr



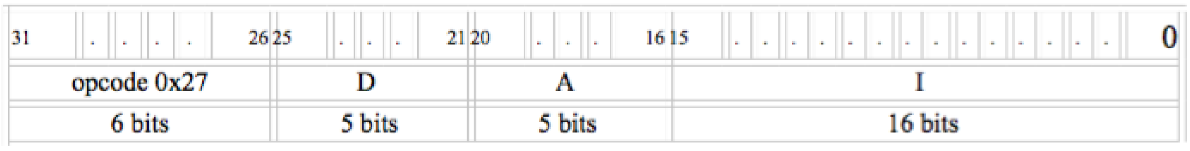
31 26	25 16	15 . . . 11	10 0
opcode 0x11	reserved	B	reserved
6 bits	10 bits	5 bits	11 bits

l.sw



31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
opcode 0x35	I	A	B	I
6 bits	5 bits	5 bits	5 bits	11 bits

l.addi & l.lwz



31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x27	D	A	I
6 bits	5 bits	5 bits	16 bits

Figure 7: Codes of instructions

At the beginning, the core of the Speculoos module is locked. The observers control the lock by detecting the start of the end of a process. (Currently the lock is only open at the detection of the first process). The main characteristic of the observer is its instruction flow redundancy analysis. Indeed, the observer does not detect only one instruction but a redundancy of specific instruction in order to specify and validate the start of a process, the jump ton another function or a return in the calling function. The observer checks 3 different instruction flow redundancy; the process, the jump and the return.

The process

- l.sw -8(r1),r2
    - opcode = 0x35
    - rA = 1
    - rB = 2
    - I = -8
- l.addi r2,r1,0
    - opcode = 0x27
    - rD = 2
    - rA = 1
    - I = 0
- l.sw -4(r1),r9
    - opcode = 0x35
    - rA = 1
    - rB = 9
    - I = -4

The jump

- l.jal (function)
    - opcode = 0x6
    - first 21 bits = 0
- empiric function
    - opcode = 0x5
    - rA =0
    - rB = 0

The return

- l.lwz r2, -8(r1) or l.addi r1, r1, 12
    - opcode = 0x21 or 27
    - rD = 2 or 1
    - rA = 1
    - I = -8 or 12
- l.lwz r9,-4(r1)
    - opcode = 0x21
    - rD = 9
    - rA = 1
    - I = -4
- l.jr r9
    - opcode = 0x11
    - rB = 9

Finally, when the observer has detected one the instruction flow redundancy, it has to inform the monitor. For the process, the observer will unlock the finite state machine and immediately give the order to stack into the shadow stack the address currently in r9. For the jump, the observer has to wait 2 clocks before giving the order to stack r9 because of r9 writing delay. And for the return no delay is necessary.

OPENRISC INTEGRATION

The Speculoos defense module has been especially design for the OpenRISC architecture. Speculoos is linked with OpenRISC in the block MOR1KX_CPU_CAPPUCCINO. The instruction code is directly recover from MOR1KX_FETCH_CAPPUCCINO and the interrupt is connected to the hard reset with a OR. The register R9 and R1 are located into a ram called RFA instantiated into MOR1KX_RF_CAPPUCCINO. This ram has been modified in order to recover these registers and is described into the module MOR1KX_SIMPLE_DPRAM_SCLK.

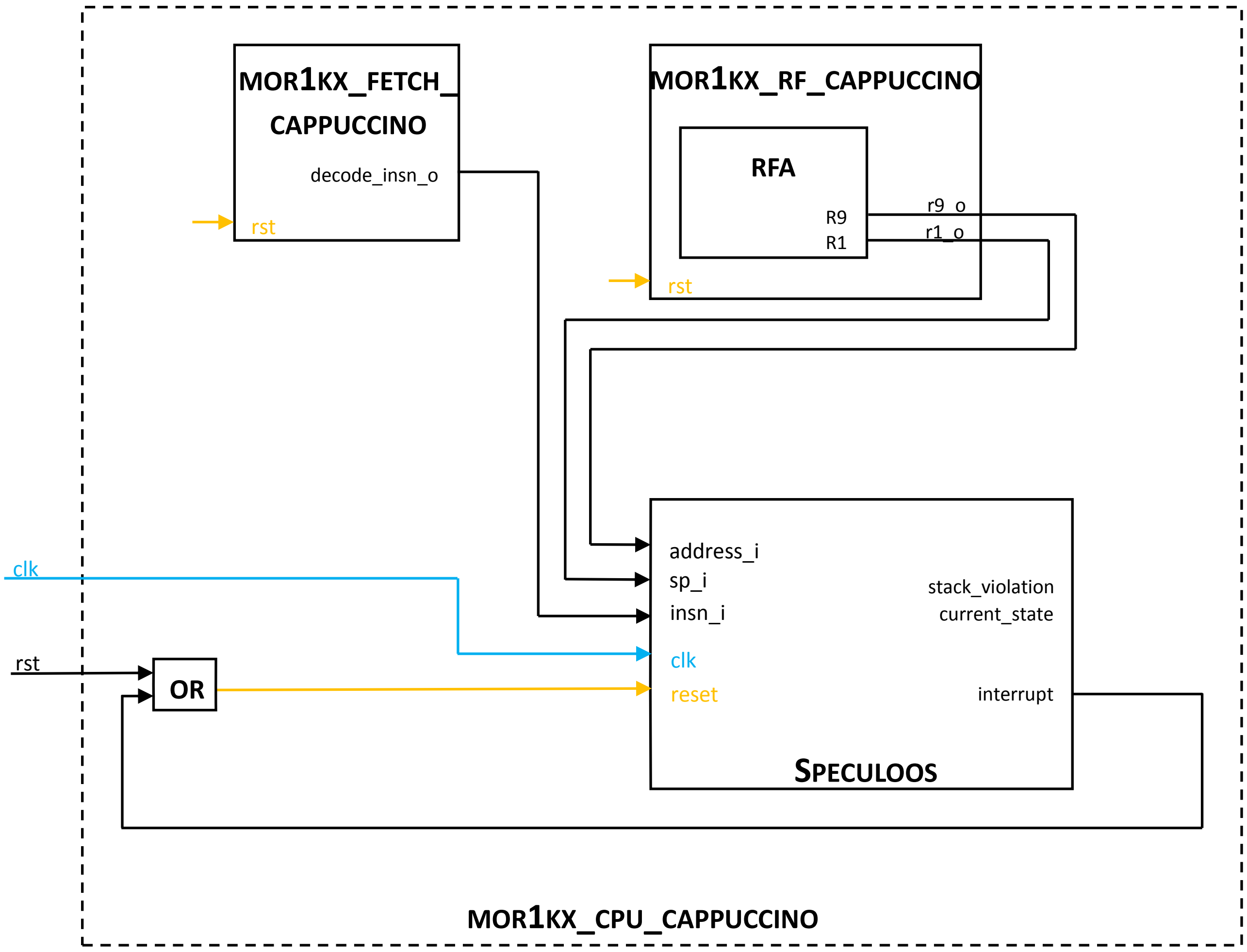
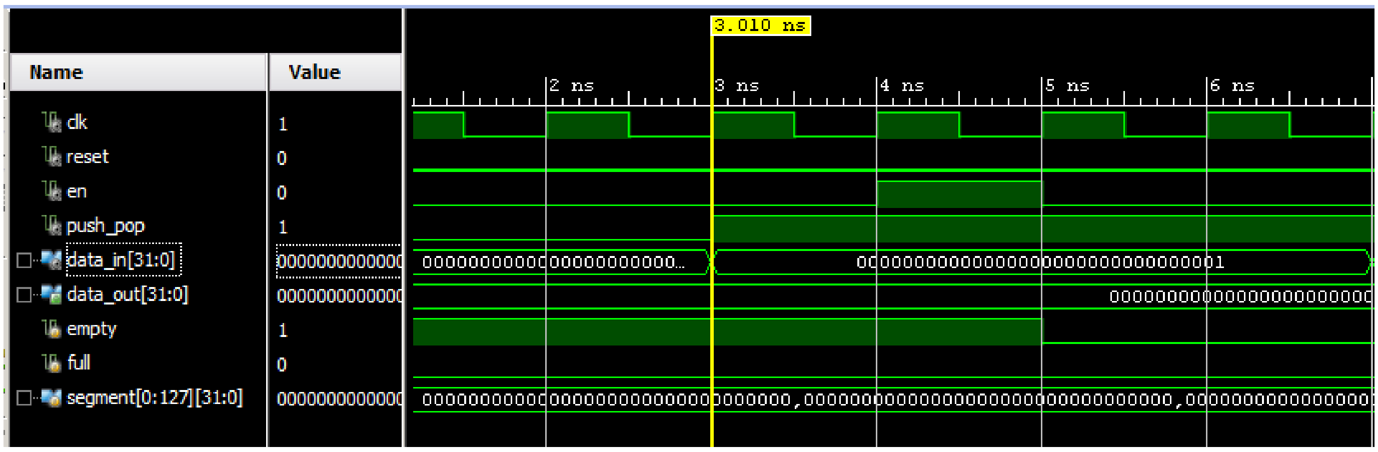


Figure 8: OpenRISC block diagram

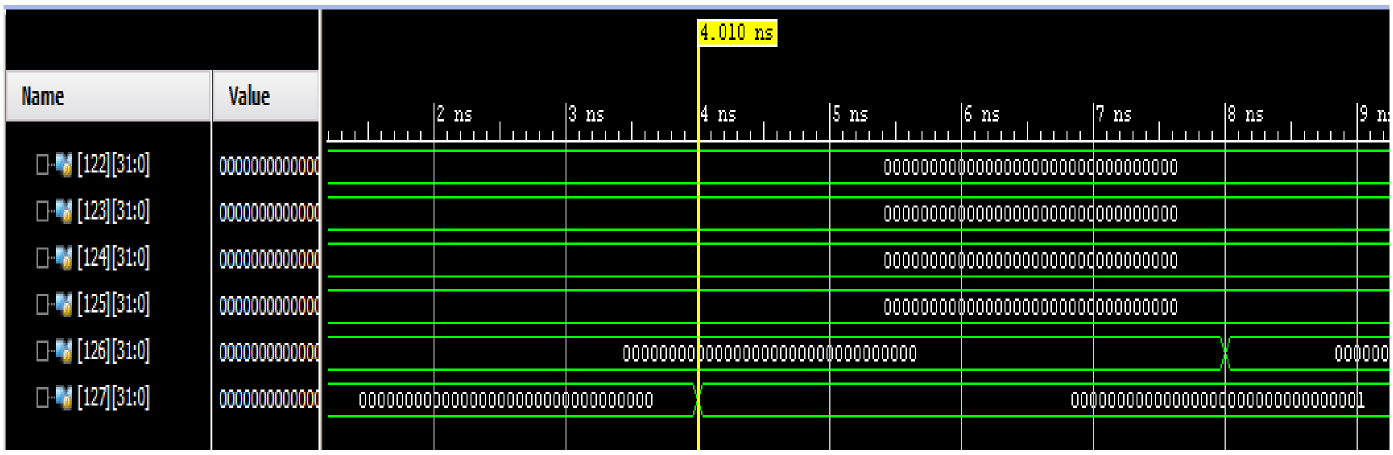
UNITARY TESTS & CORE VALIDATION

The Stack

- Stack Push function test

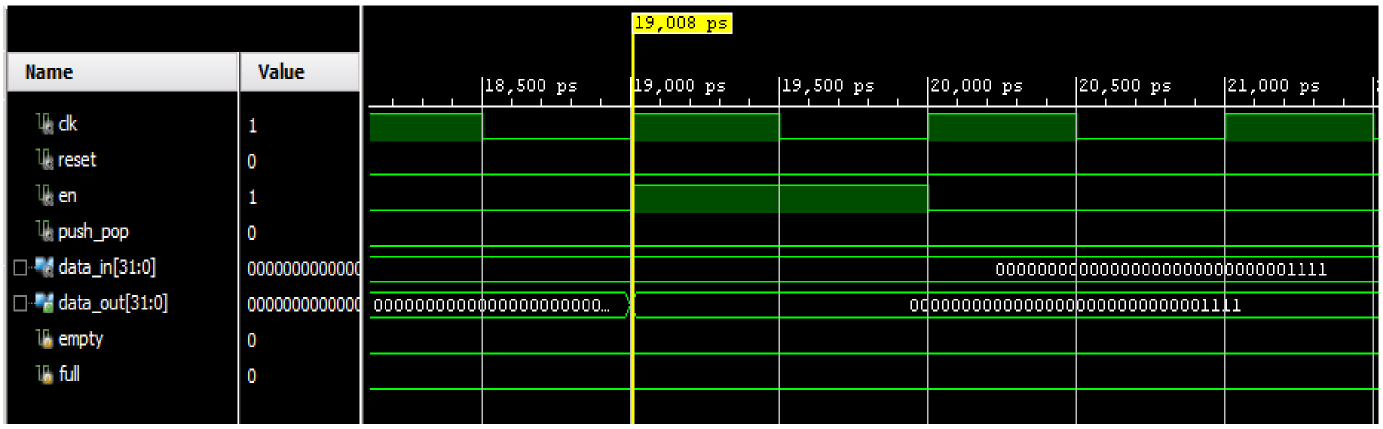


At 4 ns: Signal en is active high, Signal push_pop is active high, data_value is 1 in decimal.



At 4 ns, the value 1 is push on the top of the stack (position 127)

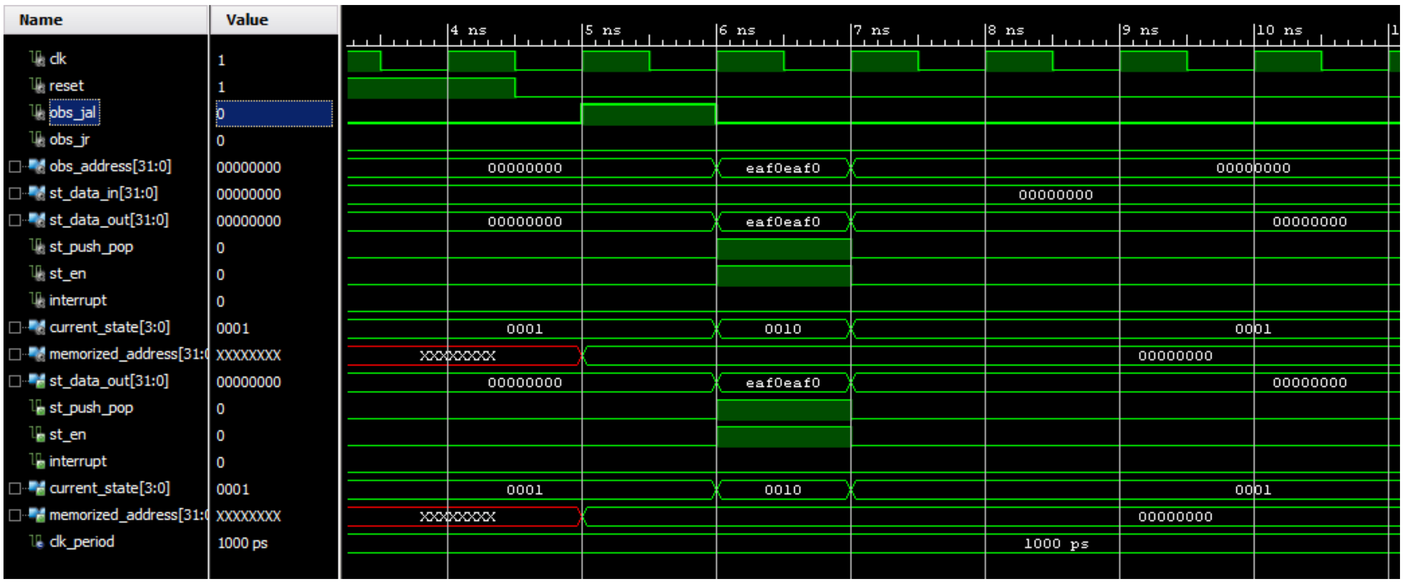
- Stack Pop function test



At 19 ps: signal en is active high, signal push_pop is active low, and we can see the last vector which has been push on the stack is 31 (check the data_in value). As result the data_out value take the value of 31 (top of the stack).

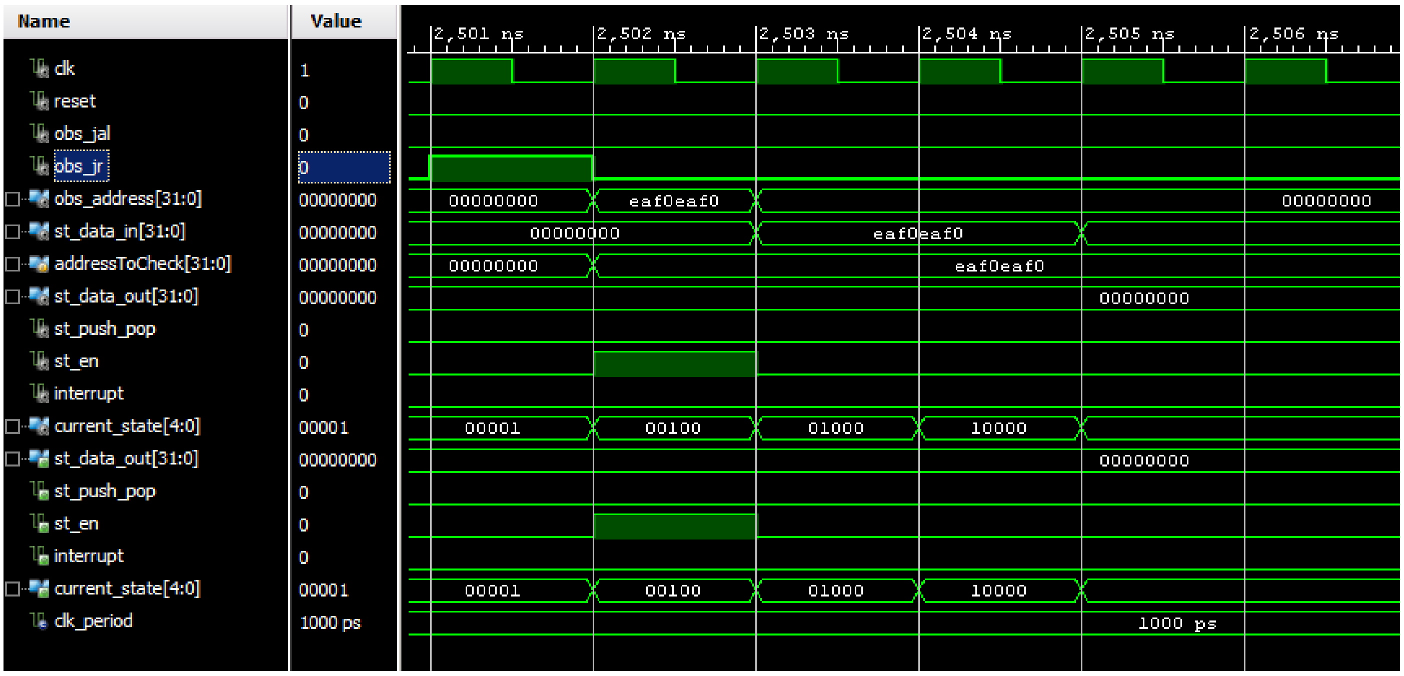
Finite State Machine

- Jump detection



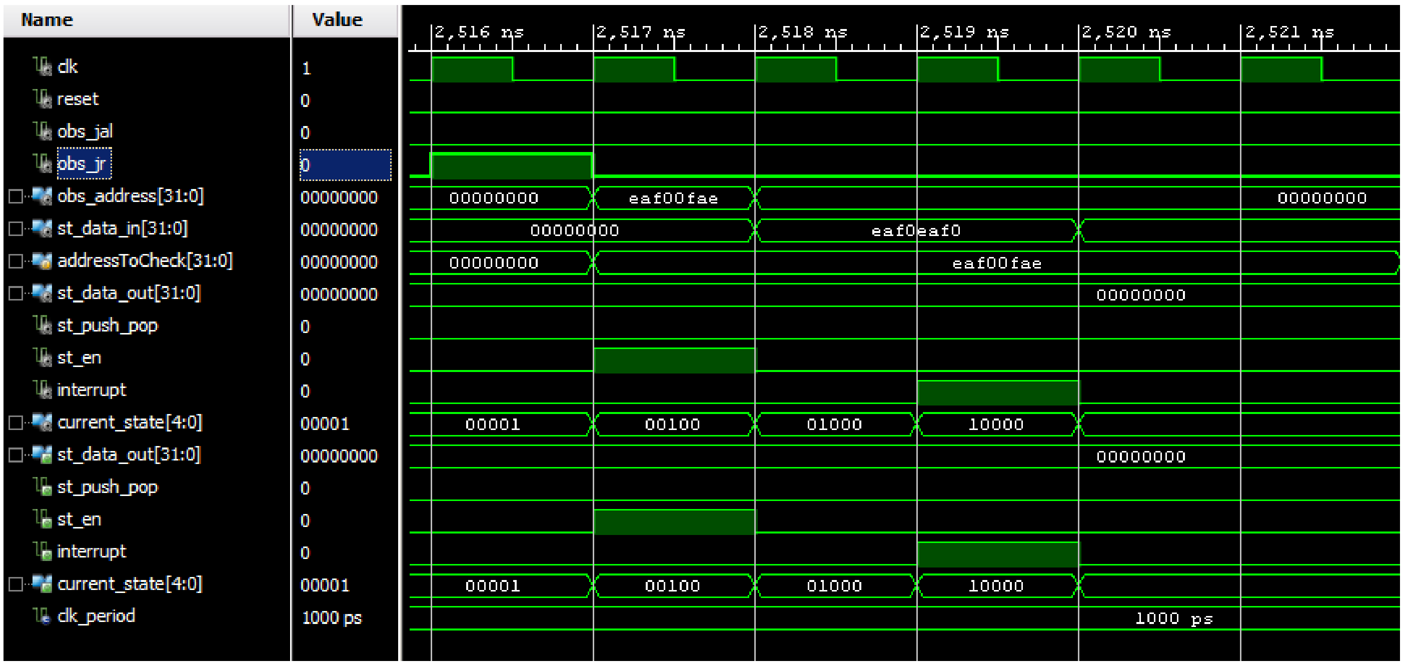
At 5ns we can observe that the input obs_jal is active high. The current_state output value is 1 it means that the current state is Idle. At the next clock (6 ns) the obs_address will get the return address of the calling function, so the finite state machine will put it in the stack. As a result, the current_state will change to the value 2 which is the Push state, the output value st_data_out is the address to put into the stack module. The st_push_pop is active high same as the st_in_en in order to push on the top of the stack module.

- Return detection



Name	Value
clk	1
reset	0
obs_jal	0
obs_jr	0
obs_address[31:0]	00000000
st_data_in[31:0]	00000000
addressToCheck[31:0]	00000000
st_data_out[31:0]	00000000
st_push_pop	0
st_en	0
interrupt	0
current_state[4:0]	00001
st_data_out[31:0]	00000000
st_push_pop	0
st_en	0
interrupt	0
current_state[4:0]	00001
clk_period	1000 ps

At 2501ns the jump return instruction is detected (obs_jr input is active high), at next lock (2505ns) the state of the FSM is pop in order to pop the value on the stack module (st_en is active high and st_push_pop is active low). At 2504ns the state of the FSM is check, the input st_data_in and the obs_address atre the same so no interrupt is raise.



Name	Value
clk	1
reset	0
obs_jal	0
obs_jr	0
obs_address[31:0]	00000000
st_data_in[31:0]	00000000
addressToCheck[31:0]	00000000
st_data_out[31:0]	00000000
st_push_pop	0
st_en	0
interrupt	0
current_state[4:0]	00001
st_data_out[31:0]	00000000
st_push_pop	0
st_en	0
interrupt	0
current_state[4:0]	00001
clk_period	1000 ps

At 2516ns the input obs_jr detect a jump return instruction, next clock (2517ns) the return address of the caller function is given to the FSM on input obs_address. Pop state active st_en and push_pop is active low, to pop the value on the top of the stack. The stack return a different value as the return address on input obs_address it means that the return address has been modified and interruption is raise at 2519ns.

Monitor

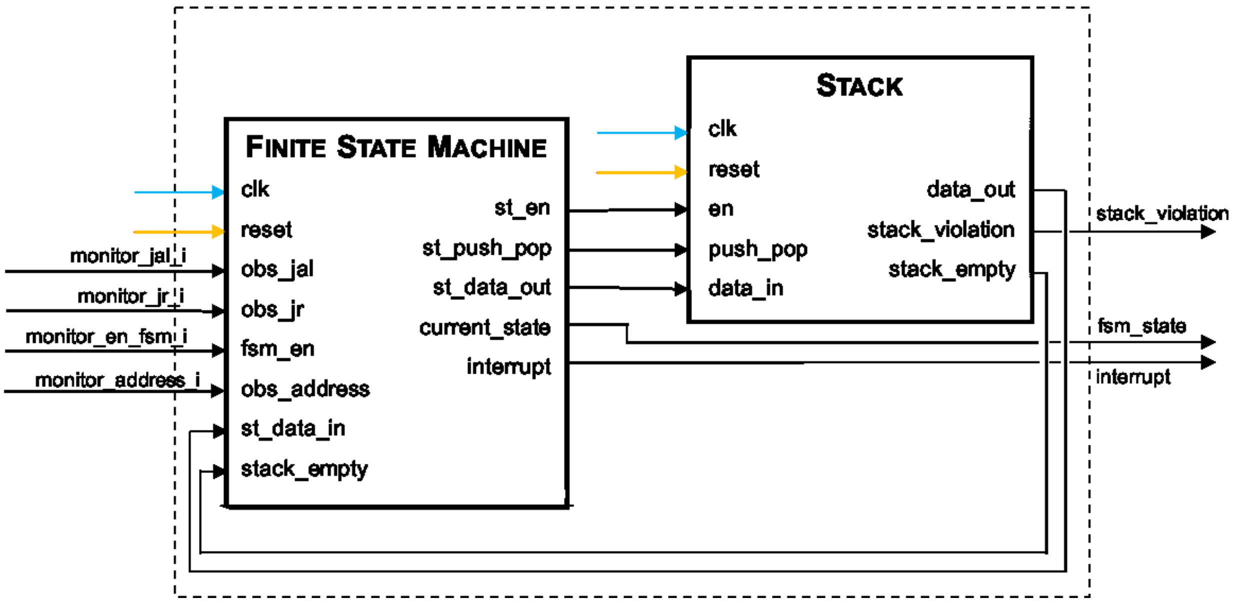
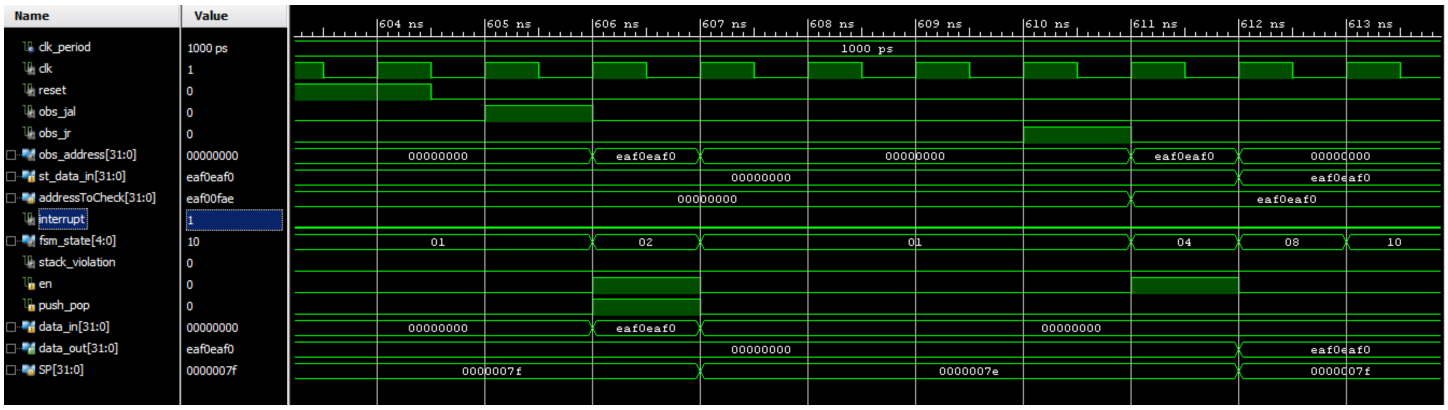


Figure 9: Monitor block diagram

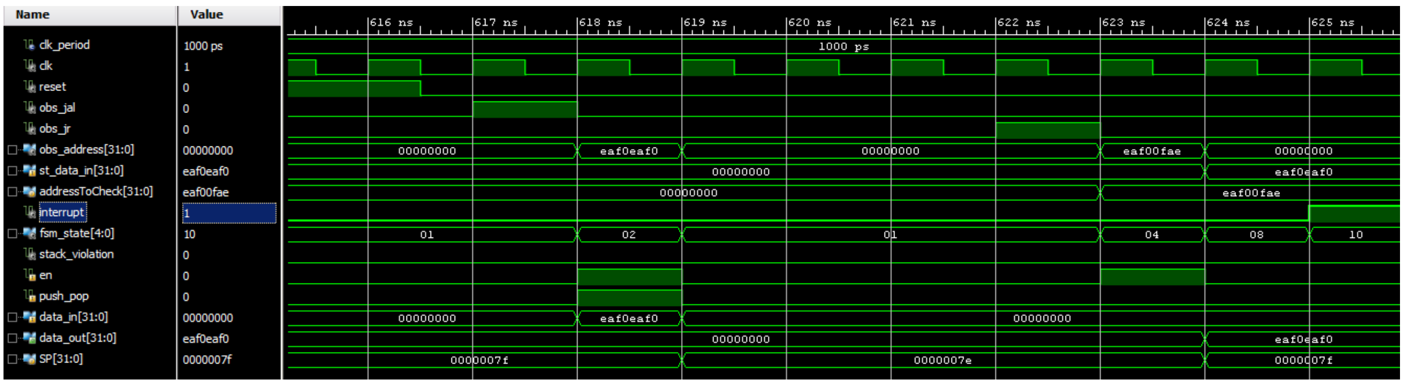
- Simulation with a normal returning of a function



Name	Value
clk_period	1000 ps
clk	1
reset	0
obs_jal	0
obs_jr	0
obs_address[31:0]	00000000
st_data_in[31:0]	eaf0eaf0
addressToCheck[31:0]	eaf00fae
interrupt	1
fsm_state[4:0]	10
stack_violation	0
en	0
push_pop	0
data_in[31:0]	00000000
data_out[31:0]	eaf0eaf0
SP[31:0]	0000007f

At 605ns a jal instruction is detect and the return address of the function (eaf0eaf0) is push on the stack. In fact, the stack_en signal and the push_pop signal is active high. At 610ns a jr instruction is target, so the stack_en is active high and the push_pop is active low. As a result, the address on the top of the stack is pop see st_data_in. As the address pop by our stack and the address in input are the same no interruption is raised.

- Simulation with a buffer overflow exploit



Name	Value
clk_period	1000 ps
clk	1
reset	0
obs_jal	0
obs_jr	0
obs_address[31:0]	00000000
st_data_in[31:0]	eaf0eaf0
addressToCheck[31:0]	eaf00fae
interrupt	1
fsm_state[4:0]	10
stack_violation	0
en	0
push_pop	0
data_in[31:0]	00000000
data_out[31:0]	eaf0eaf0
SP[31:0]	0000007f

At 618ns the address eaf0eaf0 is push on the stack, now we are looking at 622ns, at this time the instruction jr is detect by the top module block so the output en of the finite state machine is active high and the output push_pop is active low in order to pop the value stored on the stack.

As you can see the address in input (eaf00fae) of the top module block is not the same as the value pop by the stack (eaf0eaf0). It means that the return address of the calling function has been modified. The top module block detects it and raise an interruption at 625ns.

This confirms that the core of Speculoos can detect when an address has been modified by an attack as a buffer overflow.