Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

# Stack Smashing Attack Detection Methodology for Secure Program Execution Based on Hardware

Raphael Segabinazzi Ferreira
Advisor: Fabian Luis Vargas, Prof. PhD

Porto Alegre – RS, Brasil
2016

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Engenharia
Programa de Pós-Graduação em Engenharia Elétrica

# Stack Smashing Attack Detection Methodology for Secure Program Execution Based on Hardware

Raphael Segabinazzi Ferreira
Advisor: Fabian Luis Vargas, Prof. PhD

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Sinais, Sistemas e Tecnologia da Informação.

Linha de Pesquisa: Sistemas de Computação

Porto Alegre – RS, Brasil
2016

# STACK SMASHING ATTACK DETECTION METHODOLOGY FOR SECURE PROGRAM EXECUTION BASED ON HARDWARE.

**CANDIDATO: RAPHAEL SEGABINAZZI FERREIRA**

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

**DR. FABIAN LUIS VARGAS - ORIENTADOR**

**BANCA EXAMINADORA**

**DRA. FERNANDA GUSMÃO DE LIMA KASTENSMIDT - DO PROGRAMA DE PÓS GRADUAÇÃO EM MICROELETRÔNICA - PGMICRO - UFRGS**

**DR. SHISHPAL RAWAT - EDA**

**DRA. LETÍCIA MARIA BOLZANI POEHLS - DO PPGEE - FENG - PUCRS**

# AGRADECIMENTOS

Agradeço primeiramente ao professor Dr. Fabian Luis Vargas, por ter aceitado ser meu orientador durante este curso, e por todos os ensinamentos que me foram concedidos, sendo estes de vital importância para conclusão deste curso. Ainda, a todos os demais professores do programa que, de uma forma ou de outra, também contribuíram para a conclusão deste trabalho.

Aos colegas do laboratório EASE que também fizeram parte deste trabalho, e em muitos pontos a sua ajuda e conselhos foram essenciais para a conclusão das tarefas e análises realizadas durante esta dissertação.

Agradeço também ao Programa de Pós Graduação em Engenharia Elétrica e a Pontifícia Universidade Católica pela oportunidade dada neste curso. E a CAPES pela bolsa concedida a mim para a realização do curso de mestrado.

Agradeço, por fim, a minha família por estar sempre ao meu lado, dando apoio nas horas que mais precisei. A minha namorada, que também durante todo o curso de mestrado sempre me apoiou, me escutou e juntamente comigo se privou de momentos de lazer para que pudesse me dedicar às tarefas do mestrado.

# RESUMO

A necessidade de inclusão de mecanismos de segurança em dispositivos eletrônicos cresceu consideravelmente com o aumento do uso destes dispositivos no dia a dia das pessoas. À medida que estes dispositivos foram ficando cada vez mais conectados a rede e uns aos outros, estes mesmos se tornaram vulneráveis a tentativa de ataques e intrusões remotas. Ataques deste tipo chegam normalmente como dados recebidos por meio de um canal comum de comunicação, uma vez presente na memória do dispositivo estes dados podem ser capazes de disparar uma falha de software pré-existente, e, a partir desta falha, desviar o fluxo do programa para o código malicioso inserido. Vulnerabilidades de software foram, nos últimos anos, a principal causa de incidentes relacionados à quebra de segurança em sistemas e computadores. Adicionalmente, estouros de buffer (buffer overflow) são as vulnerabilidades mais exploradas em software, chegando a atingir, metade das recomendações de segurança do grupo norte americano Computer Emergency Readiness Team (CERT).

A partir deste cenário citado acima, o presente trabalho apresenta um novo método baseado em hardware para detecção de ataques ocorridos a partir de estouros de buffer chamados de Stack Smashing, propõe ainda de maneira preliminar, um mecanismo de recuperação do sistema a partir da detecção de um ataque ou falha. Comparando com métodos já existentes na bibliografia, a técnica apresentada por este trabalho não necessita de recompilação de código e, adicionalmente, dispensa o uso de software (como, por exemplo, um Sistema Operacional) para fazer o gerenciamento do uso de memória.

Monitorando sinais internos do pipeline de um processador o presente trabalho é capaz de detectar quando um endereço de retorno de uma função está corrompido, e a partir desta detecção, voltar o sistema para um estado seguro salvo previamente em uma região segura de memória.

Para validar este trabalho um programa simples, em linguagem C, foi implementado, este programa força uma condição de buffer overflow. Esta condição deve ser reconhecida pelo sistema implementado neste trabalho e, ainda, recuperada adequadamente. Já para avaliação do sistema, a fim de verificar como o mesmo se comporta em situações reais, programas testes foram implementados em linguagem C com pequenos trechos de códigos maliciosos. Estes trechos foram

obtidos de vulnerabilidades reportadas na base de dados Common Vulnerabilities and Exposures (CVE). Estes pequenos códigos maliciosos foram adaptados e inseridos nos fontes do programa de teste. Com isso, enquanto estes programas estão em execução o sistema implementado por este trabalho é avaliado. Durante esta avaliação são observados: (1) a capacidade de detecção de modificação no endereço de retorno de funções e (2) a recuperação do sistema. Finalmente, é calculado o overhead de área e de tempo de execução.

De acordo com resultados e implementações preliminares este trabalho conseguiu atingir 100% da detecção de ataques sobre uma baixa latência por detecção de modificações de endereço de retorno de funções salva no stack. Foi capaz, também, de se recuperar nos casos de testes implementados. E, finalmente, resultando em baixo overhead de área sem nenhuma degradação de performance na detecção de modificação do endereço de retorno.

# ABSTRACT

The need to include security mechanisms in electronic devices has dramatically grown with the widespread use of such devices in our daily life. With the increasing interconnectivity among devices, attackers can now launch attacks remotely. Such attacks arrive as data over a regular communication channel and, once resident in the program memory, they trigger a pre-existing software flaw and transfer control to the attacker's malicious code. Software vulnerabilities have been the main cause of computer security incidents. Among these, buffer overflows are perhaps the most widely exploited type of vulnerability, accounting for approximately half the Computer Emergency Readiness Team (CERT) advisories in recent years.

In this scenario, the methodology proposed in this work presents a new hardware-based approach to detect stack smashing buffer overflow attack and recover the system after the attack detection. Compared to existing approaches, the proposed technique does not need application code recompilation or use of any kind of software (e.g., an Operating System - OS) to manage memory usage.

By monitoring processor pipeline internal signals, this approach is able to detect when the return address of a function call has been corrupted. At this moment, a rollback-based recovery procedure is triggered, which drives the system into a safe state previously stored in a protected memory area.

This approach was validated by implementing a C program that forces a buffer overflow condition, which is promptly recognized by the proposed approach. From this point on, the system is then properly recovered. Having in mind to evaluate the system under more realistic conditions, test programs were implemented with pieces of known vulnerable C codes. These vulnerable pieces of codes were obtained from vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE). These code snippets were adapted and included in the test programs. Then, while running these programs the proposed system was evaluated. This evaluation was done by observing the capability of the proposed approach to: (1) detect an invalid return address and (2) to safely recovery the system from the faulty condition. Finally, the execution time and area overheads were determined. According to preliminary implementations and results this approach guarantees 100% attack detection with negligible detection latency by recognizing return address overwritten within a few processor clock cycles.

# FIGURES LIST

# TABLE LIST

# ABBREVIATIONS AND ACRONYMS LIST

| | |
|---|---|
| CERT | Computer Emergency Readiness Team |
| OS | Operational System |
| PC | Program Counter |
| SP | Stack Pointer |
| SSP | ShadowMem Stack Pointer |
| CVE | Commom Vulneabilities and Exposures |
| NIST | National Institute of Standards and Technology |
| NVD | National Vulnerability Database |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| SPARV-V8 | Scalable Processor ARChitecture Version 8 |
| SOC | System-on-a-chip |
| GPL | General Public License |
| IU | Integer Unit |
| FPU | Floating-Point Unit |
| CP | Coprocessor |
| MMU | Memory Management Unit |
| ALU | Aritmetical Logical Unit |
| CWP | Current Window Pointer |
| PSR | Processor State Register |
| FE | Instruction Fetch |
| DE | Decode |
| RA | Register Access |
| EX | Execute |
| ME | Memory |
| XC | Exception |
| WR | Write |
| WIM | Window Invalid Mask |
| CTIs | Control transfer |
| DCTIs | Delayed Control-Transfer Instructions |
| AMBA | Advanced Microcontroller Bus Architecture |
| FPGA | Field Programmable Gate Array |

| | |
|---|---|
| CPLD | Complex Programmable Logic Device |
| HDL | Hardware Description Language |
| JTAG | Joint Test Action Group |
| ISIM | ISE Simulator |
| ROB | Re-Order Buffer |
| SB | Store Buffer |
| SRAS | Secure Return Address Stack |
| ReDTPM | Reconfigurable Dynamic Trusted Platform Module |
| DTPM | Dynamic Trusted Platform Module |
| TPM | Trusted Platform Module |
| CFC | Control Flow Checker |
| TAB | Target Address Buffer |
| CFI | Control Flow Instruction |
| FFRR | Function Frame Runtime Randomization |
| LFSR | Linear feedback shift register |
| COTS | Commercial Off-The-Shelf |
| DDoS | Distributed Denial of Service |
| CPU | Central Processing Unit |

# SUMMARY

# 1. INTRODUCTION

The need to include security mechanisms in electronic devices has dramatically grown with the widespread use of such devices in our daily life. With the increasing interconnectivity among devices, attackers can now launch attacks remotely. Such attacks arrive as data over a regular communication channel and, once resident in the program memory, they trigger a pre-existing software flaw and transfer control to the attacker's malicious code. Software vulnerabilities have been the main cause of computer security incidents. Among these, buffer overflows are perhaps the most widely exploited type of vulnerability, accounting for approximately half the CERT advisories in recent years (CERT, Vulnerability Database).

## 1.1. OBJECTIVES AND MOTIVATION

According to GARTNER, most attacks are focused on the application layer (Gartner Newsroom, Announcements, Gartner Says 25 Percent of Distributed Denial of Services Attacks in 2013 Will Be Application-Based , 2013). Because of this, software security defects become the main concerns that security professionals deal with nowadays. This trend has motivated considerable research in the improvement of software development processes. As a consequence, security engineering becomes an important part of the business processes that protects corporate assets and information (NUNES, BELCHIOR e ALBUGUERQUE, 2010).

A common misunderstanding occurs, for example, when software engineers think that an identity and authentication control implemented in software to protect data confidentiality and integrity makes this software secure. Actually, this supposed secure software just implements a security function and cannot be considered secure. So, security function does not insure that software is safe (MCGRAW, 2004).

Safety-critical systems could be, potentially, cause of accidents. Software is hazardous if it can cause a hazard, for example, if it cause other components to become hazardous or if it is used to control a hazard. Software is deemed safe if it is impossible or at least highly unlikely that the software could ever produce an output that would cause a catastrophic event for the system that the software controls (SWARUP e RAMAIAH, 2008). Examples of catastrophic events include loss of

physical property, physical harm, and loss-of-life. Software engineering of a safety-critical system requires a clear understanding of the software's role in, and interactions with the system (LUTZ, 2000)(KNIGHT, 2002). Application areas for safety-critical systems include the following:

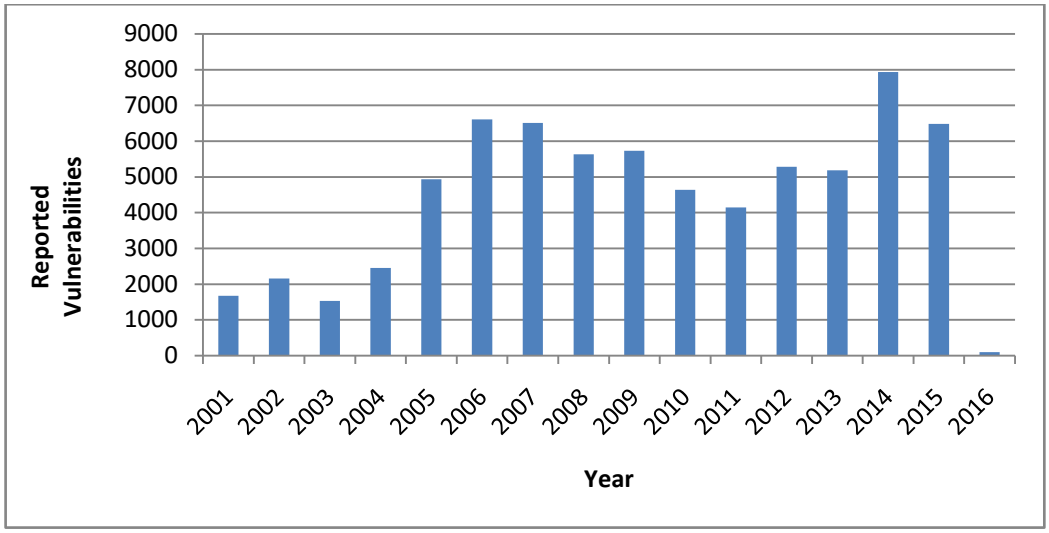
- Military: weapon delivery systems and space programs.
- Industry: manufacturing control where toxic substances are involved and robots.
- Transportation: fly-by-wire systems on board aircraft, air traffic control, interlocking systems for trains, automatic train control and computer systems in cars.
- Communication: ambulance dispatch systems and the emergency call part of a telephone system.
- Medicine: radiation therapy machines, medical monitoring and medical robots.
- Nuclear power plant control.

As is apparent from the above example areas, safety-critical systems are often real-time control systems. These systems require the utmost care in their specification, design, implementation, operation and maintenance, as they could lead to injuries or loss of lives and in-turn result in financial loss (HERRMANN, 2000) (SCHMID, 2002).

So, software security is an important issue, and as quoted above, a security breach could bring serious damages. In addition, software vulnerabilities can be the gateway to a security breach.

Nevertheless, second the National Institute of Standards and Technology (NIST) in the National Vulnerability Database (NVD), the software vulnerabilities reported in the last 4 years are more then 20.000. The evolution of the reported vulnerabilities in the last years is show in the Figure 1 (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY - NIST, 2016).

Figure 1 – Graf to illustrate the evolution of reported software vulnerabilities by the years in the National Vulnerabilities Database.




Reference: (National Vulnerability Database - Vulnerability search)

Also, according with NVD, the number of vulnerabilities caused by buffer errors represents more than 15% of the vulnerabilities reported in the year of 2015 (National Vulnerability Database - Vulnerability search). So, this issue represents a real and current problem that the work proposed by this dissertation intends to reduce.

In this scenario, the work proposed by this dissertation presents a new hardware-based approach to detect stack smashing buffer overflow attack. Compared to existing approaches, the proposed technique does not need application code recompilation or use of any kind of software (e.g., an Operational System - OS) to manage memory usage. According to preliminary implementations, this approach guarantees 100% attack detection, while resulting in negligible area overhead and zero performance degradation (since the Watchdog is fully independent from the processor and performs in parallel to the code execution).

# 2. PRELIMINARIES

In this section a background knowledge will be introduced, this knowledge will be necessary to understand this work: the problem, the methodology and the proposed solution.

## 2.1. A COMMON PROCESSOR MEMORY DIVISION

According to Patterson (PATTERSON e HENNESSY, 2012), a common processor typically divide memory into three parts (see Figure 2). The first part, near the bottom of the address space (starting at address 0x400000), is the text segment, which holds the program's instructions.

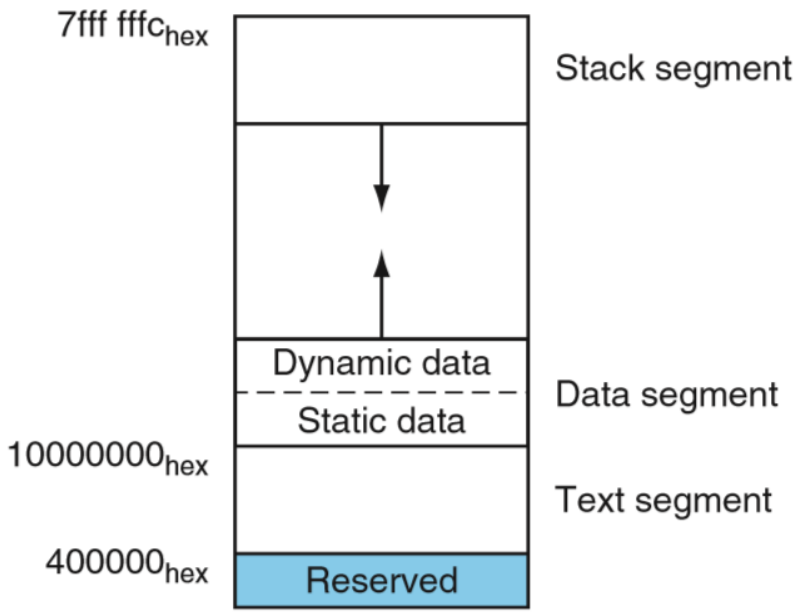
The second part, above the text segment, is the data segment, which is further divided into two parts. Static data (starting at address 0x10000000) contains objects whose size is known to the compiler and whose lifetime – the interval during which a program can access them – is the program's entire execution. For example, in C language programming, global variables are statically allocated, since they can be referenced anytime during a program's execution. The linker both assigns static objects to locations in the data segment and resolves references to these objects.

Immediately above static data is dynamic data area. This area, as its name implies, is allocated by the program as it executes. In C programs, the "malloc" library routine finds and returns a new block of memory. Since a compiler cannot predict how much memory a program will allocate, the operating system expands the dynamic data area to meet demand. As the upward arrow in the Figure 2 indicates, "malloc" expands the dynamic area with a system call, which causes the operating system to add more pages to the program's virtual address space immediately above the dynamic data segment.

The third part, the program stack segment, resides at the top of the virtual address space (starting at address 0x7fffffff). Like dynamic data, the maximum size of a program's stack is not known in advance. As the program pushes values on to the stack, the operating system expands the stack segment down toward the data segment.

This three-part division of memory is not the only possible one. However, it has two important characteristics: the two dynamically expandable segments are as far apart as possible, and they can grow to use a program's entire address space.

Figure 2 - Common processor layout of memory.




Reference: (PATTERSON e HENNESSY, 2012)

## 2.2. THE PROBLEM: STACK SMASHING ATTACK

Buffer overflow attacks exploit a lack of bounds checking on the size of an input being stored in a buffer array in memory. By writing data past the end of an allocated array, the attacker can make arbitrary changes to program data stored adjacent to the array. By far, the most common data structure to corrupt is the stack, so this is called "stack smashing" or "buffer overflow" attack (PARK, ZHANG e LEE, 2006).

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible (MILLER, KOSKI, *et al.*, 1995) (MILLER, FREDRIKSEN e SO, 1990).

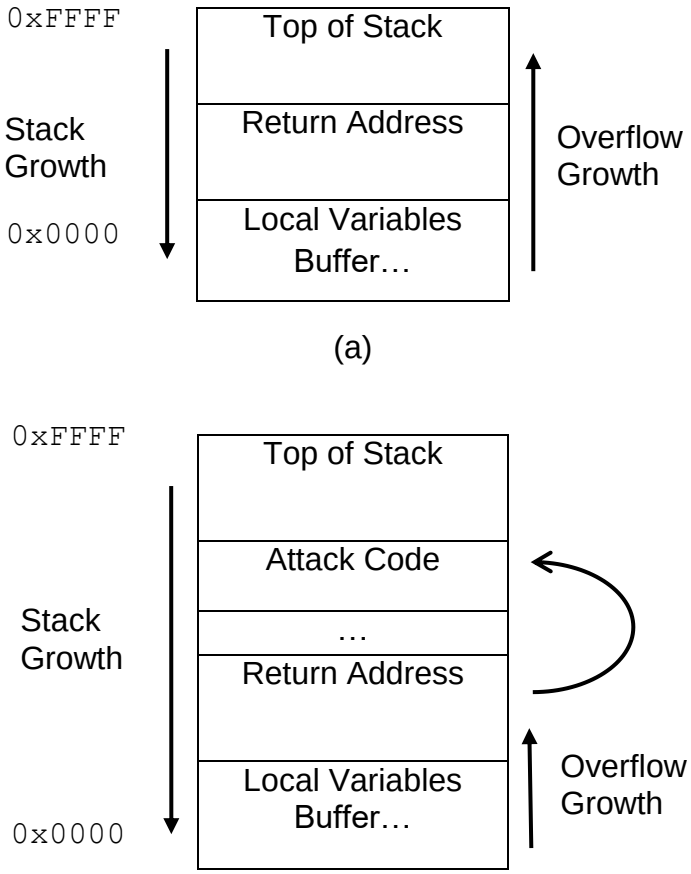
The common form of buffer overflow exploitation is to attack buffers allocated on the stack. Stack smashing attacks try to achieve two mutually dependent goals:

a) Inject Attack Code: The attacker provides an input string that is actually executable, binary code native to the machine being attacked. Typically this code is simple, and does something similar to exec("sh") to produce a root shell.

b) Change the Return Address: There is a stack frame for a currently active function above the buffer being attacked on the stack. The buffer overflow changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps to the attack code.

Figure 3 a) shows a stack with local variables and return address and Figure 3 b) shows the stack in a Stack Smashing situation, the local variables overwritten the return address data and introduced a malicious code. This code could be addressed by the new malicious return address, and finally, executed when the system returns from the current function.

Figure 3 - a) Stack during a normal execution. b) Stack with malicious code injected.



(b)

Reference: Segabinazzi (2016)

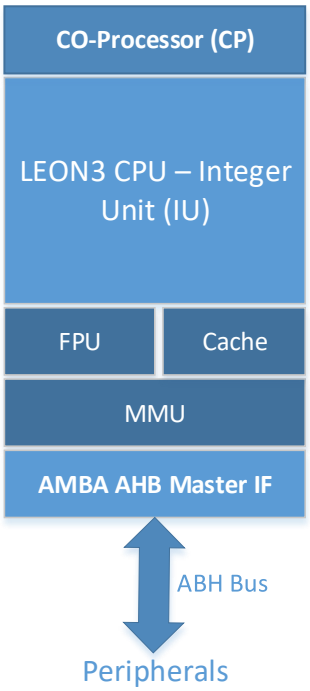
## 2.3. TARGET ARCHITECTURE: LEON3 PROCESSOR

This work was implemented on a LEON3 soft-core processor. LEON3 is a synthesizable Very High Speed Integrated Circuit Hardware Description Language (VHDL) model of a 32-bit 7-stage pipeline processor compliant with the IEEE-1754 (Scalable Processor Architecture Version 8 - SPARC-V8 - processor architecture) (SPARC INTERNATIONAL INC., 1992). The model is highly configurable, and particularly suitable for system-on-a-chip (SOC) designs. The full source code is available under the GNU General Public License (GPL) license, allowing free and unlimited use for research and education. LEON3 is also available under a low-cost commercial license, allowing it to be used in any commercial application to a fraction of the cost of comparable IP cores (COBHAM GAISLER AB, 2015).

A SPARC processor logically comprises an integer unit (IU), a Floating-Point Unit (FPU), and an optional coprocessor (CP), each with its own registers. This organization allows for implementations with maximum concurrency between integer, floating-point, and coprocessor instruction execution. All of the registers — with the possible exception of the coprocessor's — are 32 bits wide. Instruction operands are generally single registers, register pairs, or register quadruples (SPARC INTERNATIONAL INC., 1992).

Figure 4 describes the block diagram of the LEON3 processor core; the FPU, Cache, the Co-processor and the Memory Management Unit (MMU) are optional.

Figure 4 - LEON3 processor core block diagram.




Reference: modified from (COBHAM GAISLER AB, 2015)

### 2.3.1.Leon3 CPU

The Integer Unit contains the general-purpose registers and controls the overall operation of the processor. The IU executes the integer arithmetic instructions and computes memory addresses for loads and stores. It also maintains the program counters and controls instruction execution for the FPU and the CP (SPARC INTERNATIONAL INC., 1992).

The LEON3 integer unit is composed by 7 stages; it uses a single instruction issue pipeline. These seven stages are explained below (COBHAM GAISLER AB, 2015):

(1) Instruction Fetch (FE): The instruction is fetched from the instruction cache if the cache enable. Otherwise, the fetch is performed in the AHB bus.

(2) Decode (DE): The instructions are decoded and the target addresses of the CALL and Branch instructions are generated.

(3) Register Access (RA): Operands are read from the registers file or from internal data bypasses.

(4) Execute (EX): ALU, logical, and shift operations are performed. The addresses are generated for memory operations (e.g. LD) and for JMPL/RET instructions.
(5) Memory (ME): Data cache is read or written at this stage.
(6) Exception (XC): Traps and interrupts are resolved. The data is aligned as appropriate for cache reads.
(7) Write (WR): The result of Arithmetical Logical Unit (ALU), logical, shift or cache operations are written back to the register file.

#### 2.3.1.1. Integer Unit Register Windows

An implementation of the IU may contain from 40 to 520 general-purpose 32-bit *r* registers. This corresponds to a grouping of the registers into 8 global *r* registers, plus a circular stack of from 2 to 32 sets of 16 registers each, known as register windows. Since the number of register windows present (NWINDOWS) is implementation-dependent, the total number of registers is implementation-dependent.

At a given time, an instruction can access the 8 global and a register window into the *r* registers. A 24-register window comprises a 16-register set — divided into 8 in and 8 local registers — together with the 8 in registers of an adjacent register set, addressable from the current window as its out registers.

The current window is specified by the current window pointer (CWP) field in the processor state register (PSR). Window overflow and underflow are detected via the window invalid mask (WIM) register, which is controlled by supervisor software. The actual number of windows in a SPARC implementation is invisible to a user-application program (SPARC INTERNATIONAL INC., 1992).

### 2.3.2. Leon3 Instructions

The Leon3 instructions fall into six categories:

(1) **Load/store:** are the only instructions that access memory. They use two *r* registers or an *r* register and a signed 13-bit immediate value to calculate a 32-bit, byte-aligned memory address.

(2) **Arithmetic/logical/shift:** perform arithmetic, tagged arithmetic, logical, and shift operations. With one exception, these instructions compute a result that is a function of two source operands; the result is either written into a destination register, or discarded.

(3) **Control transfer (CTIs):** include PC-relative branches and calls, register-indirect jumps, and conditional traps. Most of the control-transfer instructions are delayed control-transfer instructions (DCTIs), where the instruction immediately following the DCTI is executed before the control transfer to the target address is completed.

(4) **Read/write control register:** read and write the contents of software visible state/status registers

(5) **Floating-point operate:** perform all floating point calculations. They are register-to-register instructions which operate upon the floating point registers

(6) **Coprocessor operate:** are defined by the implemented coprocessor, if any

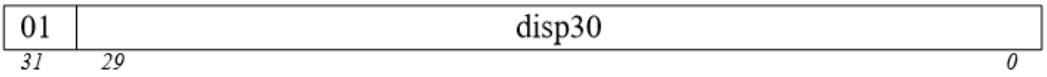
In this work the most important instruction category is the **CTIs**, this set of instructions implements the function calls and returns with the instructions CALL and JMPL. So, the branch and CALL instructions use PC-relative displacements. The jump and link (JMPL) instruction uses a register-indirect target address. It computes its target address as either the sum of two $r$ registers, or the sum of an $r$ register and a 13-bit signed immediate value. The branch instruction provides a displacement of ± 8 Mbytes, while the CALL instruction's 30-bit word displacement allows a control transfer to an arbitrary 32-bit instruction address (SPARC INTERNATIONAL INC., 1992).

The instruction CALL definitions done by (SPARC INTERNATIONAL INC., 1992) are showed in the Figure 5 and listed below:

Figure 5 – Instruction CALL format.

| *opcode* | *op* | *operation* |
|---|---|---|
| CALL | 01 | Call and Link |

Format (1):




| *Suggested Assembly Language Syntax* |
|---|
| `call` *label* |

Reference: (SPARC INTERNATIONAL INC., 1992).

Description:

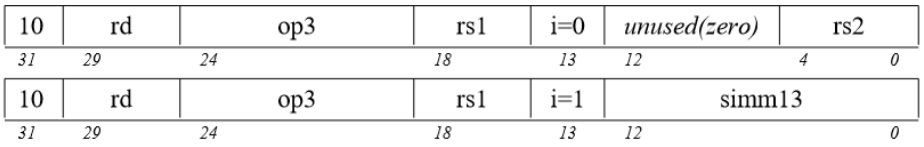
- The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address “PC + (4 × *disp30*)”. Since the word displacement (*disp30*) field is 30 bits wide, the target address can be arbitrarily distant. The PC-relative displacement is formed by appending two low-order zeros to the instruction’s 30-bit word displacement field.
- The CALL instruction also writes the value of PC, which contains the address of the CALL, into r[15] (*out* register 7).

Also, the description of instruction JMPL done by SPARC too is showed in the Figure 6 and explained below:

Figure 6 – Instruction JMPL format.

| *opcode* | *op3* | *operation* |
|---|---|---|
| JMPL | 111000 | Jump and Link |

Format (3):




| *Suggested Assembly Language Syntax* |
|---|
| `jmpl` *address* , $reg_{rd}$ |

Reference: (SPARC INTERNATIONAL INC., 1992).

Description:

- The JMPL instruction causes a register-indirect delayed control transfer to the address given by “r[*rs1*] + r[*rs2*]” if the *i* field is zero, or “r[*rs1*]+ sign_ext(*simm13*)” if the *i* field is one.
- The JMPL instruction copies the PC, which contains the address of the JMPL instruction, into register r[*rd*].
- If either of the low-order two bits of the jump address is nonzero, a mem_address_not_aligned trap occurs.

Programming Note:

- A JMPL instruction with *rd* = 15 functions as a register-indirect call using the standard link register. JMPL with *rd* = 0 can be used to return from a subroutine. The typical return address is “r[31]+8”, if a non-leaf (uses SAVE instruction) subroutine is entered by a CALL instruction, or “r[15]+8” if a leaf (doesn’t use SAVE instruction) subroutine is entered by a CALL instruction.

Implementation Note:

- When a RETT instruction appears in the delay slot of a JMPL, the target of the JMPL must be fetched from the address space implied by the new (i.e.

post-RETT) value of the PSR's S bit. In particular, this applies to a return from trap to a user address space.

Concluding, the instructions that will trigger a **function call** will be:

- The normal **CALL** instruction.
- The **JMPL** instruction with register $rd$ = 15.

And the instructions that will trigger **a function return** will be:

- The **JMPL** instruction with $rd$ non equal to 15.

### 2.3.3.Leon3 configuration

The registers windows, explained in the section 2.3.1.1, makes the processor more secure by creating new registers for every function call. As a consequence, overflows occurred in the stack program will not reach the current return address. However the overflow will reach other variables and return addresses saved in the stack when a window overflow situation occurs. So, the Watchdog proposed by this work will be useful in these situations too.

However, to helps the development of this work, by making easier to reproduce a stack smashing attack situation, these registers windows are deactivated in Leon3 configuration. This configuration makes the stack program more reachable and, as a consequence, more vulnerable.

These register windows are disabled by using the compiler flag *-mflat*, this flag makes the compiler to not call a new register window or restore to the old one when a new function is called or returned (not use SAVE and RESTORE instructions), So, the data will be saved and recovered normally from stack program.

The others Leon3 configurations are listed below:

- Cache disabled;
- FPU disabled;
- MMU disabled;
- CP disabled;
- One Leon3 core (one IU) enabled;

- The serial Debug Link is enabled;
- On chip ROM enable;
- On chip RAM – 64 kbyte - enabled;
- An Advanced Microcontroller Bus Architecture (AMBA) Master interface - enabled;

### 2.3.4. Environment

The Xilinx environment was used to implement this work. To evaluate area overhead the ISE Design Suite in the WebPACK version was used. This version is free and it is a fully featured front-to-back Field Programmable Gate Array (FPGA) design solution for Linux, Windows XP, and Windows 7. ISE WebPACK is the ideal downloadable solution for FPGA and Complex Programmable Logic Device (CPLD) design offering Hardware Description Language (HDL) synthesis and simulation, implementation, device fitting, and Joint Test Action Group (JTAG) programming (Xilinx - ISE WebPACK Design Software, 2016).

The ISE Simulator (ISIM), a complete, full-featured HDL simulator integrated within ISE (Xilinx - ISIM Simulator, 2016), was used to simulate and test the processor and our approach implementations.

## 2.4. RELATED WORKS

This section present the related works found in the literature. Approaches based on hardware and software which main objectives are to detect kinds of intrusions. The Table 1 shows the approaches presented as related works in this dissertation and classify then in hardware or software techniques.

Table 1 – Techniques based on hardware or software and his publication year.

| **Approach** | **Based on** | **Publication year** |
|---|---|---|
| StackGuard (COWAN, PU, *et al.*, 1998) (COWAN, BEATTIE, *et al.*, 1999) | SW | 1998 |

| | | |
|---|---|---|
| Dynamic Integrity Checking (KANUPARTHI, KARRI, *et al.*, 2012) (KANUPARTHI, ZAHRAN e KARRI, 2012) | HW | 2012 |
| SRAS (LEE, KARIG, *et al.*, 2004) | HW/SW | 2003 |
| Light-weight hardware return address and stack Frame Tracking (KAO e WU, 2009) | HW | 2009 |
| Separates the stack to two parts (DU e MAI, 2011) | SW | 2011 |
| ReDTPM (DAS, WEI e LIU, 2014) | HW | 2014 |
| FFRR (KUMAR e KISORE, 2014) | SW | 2014 |
| SafeStack – Memory Access Virtualization (CHEN, JIN, *et al.*, 2013) | SW | 2013 |

Reference: Segabinazzi (2016)

Several efficient software-based as well as hardware-based dynamic integrity checking techniques (CORLISS, LEWIS e ROTH, 2004)(OZDOGANOLU, VIJAYKUMAS, *et al.*, 2006)(PARK, ZHANG e LEE, 2006) have been proposed in the literature. However, software-based techniques suffer from performance overheads as high as 60%, while hardware-based approaches result in average overheads of about 18% (KANUPARTHI, KARRI, *et al.*, 2012). Additionally, some of these approaches (KANUPARTHI, KARRI, *et al.*, 2012) (SHUETTE e SHEN, 1987) need application code recompilation to compute specific information (hashes of application program's instruction addresses and opcodes) that are later used at runtime to detect attacks.

### 2.4.1.StackGuard

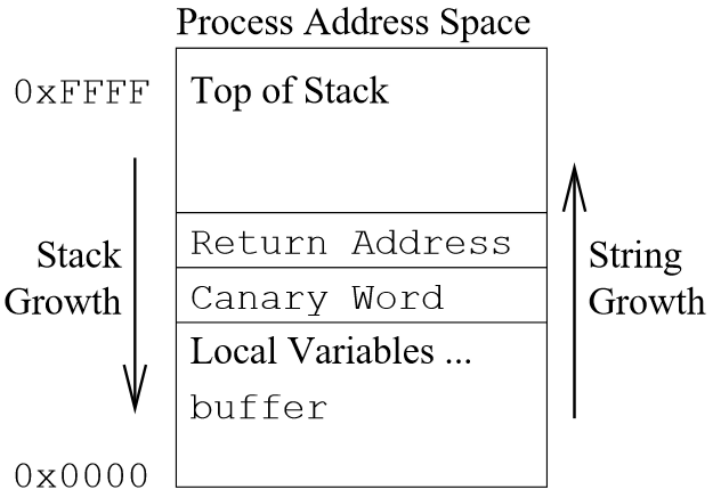
The StackGuard is proposed by (COWAN, PU, *et al.*, 1998)(COWAN, BEATTIE, *et al.*, 1999). This is a compiler extension that enhances the executable code produced by the compiler so that it detects and thwarts buffer-overflow attacks against the stack.

The detection is done by placing a "canary" word next to the return address on the stack, as show in the Figure 7. When the function returns, it first checks to see if the canary word is intact before jumping to the address pointed to by the return address word and assumes that the return address is unaltered if, and only if, the canary word is unaltered. The buffer overflow attack method exploits the fact that the

return address word is located very close to a byte array with weak bounds checking, so the only tool the attacker has is a linear, sequential write of bytes to memory, usually in ascending order. Under these restricted circumstances, it is very difficult to over-write the return address word without disturbing the canary word.

The StackGuard implementation is a patch to gcc compiler. The gcc *function_prologue* and *function_epilogue* functions have been altered to emit code to place and check canary words. The changes are architecture-specific (i386) and the additional instructions added to the function prologue are shown in pseudo-assembly form in Figure 8, and the additional instructions added to the instruction epilogue are shown in Figure 9.

Figure 7 - Canary word next to return address.




Reference: (COWAN, PU, *et al.*, 1998)

Figure 8 - Function Prologue Code: Laying Down a Canary.

```
move canary-index-constant into register[5]
 push canary-vector[register[5]]
```

Reference: (COWAN, PU, *et al.*, 1998)

Figure 9 - Function Epilogue Code: Checking a Canary.

```
move canary-index-constant into register[4]
 move canary-vector[register[4]] into register[4]
 exclusive-or register[4] with top-of-stack
 jump-if-not-zero to constant address .canary-death-handler
 add 4 to stack-pointer
 < normal return instructions here>
.canary-death-handler:
...
```

Reference: (COWAN, PU, *et al.*, 1998)

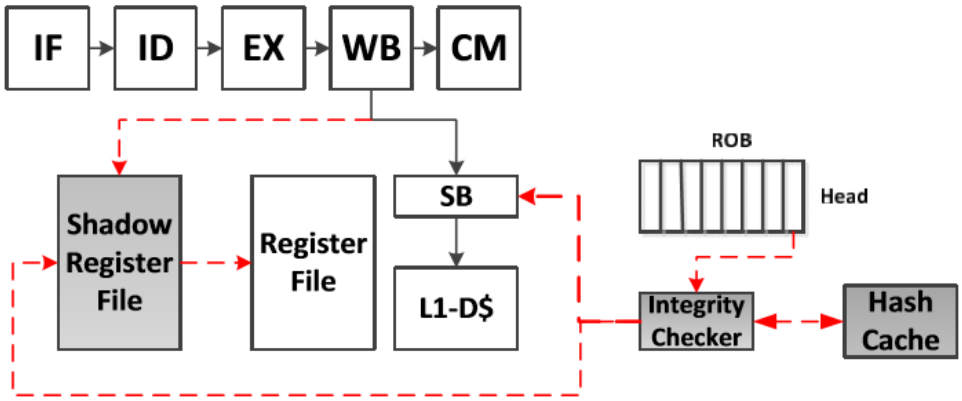
### 2.4.2. Dynamic integrity checking

Dynamic integrity checking involves calculation of hashes of the instructions in the code being executed and comparing these hashes against corresponding precomputed hashes at runtime. The approach proposed by (KANUPARTHI, KARRI, *et al.*, 2012) presents a hardware-based dynamic integrity checking that does not stall the processor pipeline. The approach permit the instructions to commit before the integrity check is complete, and allow them to make changes to the register file, but not the data cache. Then the system is rolled back to a known state if the checker deems the instructions as modified.

Modern out-of-order processors have recovery mechanisms to recover from incorrect branch speculations. Thus, it is possible to recover the architecture state to a previously check pointed state. We leverage this feature and allow all instructions to modify the architecture state. The results of the instructions that do not update the memory (and are currently being checked by the integrity checker) are written to a shadow register file, instead of the original register file. These values are held there until the integrity check is complete. Instructions that update the memory are held in the store buffer until the integrity check is complete.

Figure 10 shows the architecture of the proposed scheme. The shadow register file, integrity checker, and hash cache are the new components that are integrated with a processor pipeline. The instruction at the head of the re-order buffer (ROB) is non speculative and is ready to commit in program order. The instruction is allowed to commit and the result is written to the shadow register file or the store buffer (SB). Simultaneously, this instruction is also sent to the integrity checker, which collects instructions until all the instructions in the basic block 1 are available. The integrity checker then computes the hash of the instructions. While the hash computation is in progress, the checker accesses the hash cache to fetch the corresponding precomputed hash. The calculated hash is compared against the corresponding precomputed hash to detect any malicious modifications. Once the instructions are deemed safe, the original register file is updated with corresponding values from the shadow register file for instructions that do not update the memory. For instructions that update the memory, the pending data in the store buffer are allowed to update the data cache. The instruction decode and execute stages of the pipeline use the latest values from the shadow register file and the store buffer to resolve dependences and prevent hazards. In case of a mismatch in the calculated and precomputed hash, the system is rolled back to the last known correct state. This is accomplished by clearing the contents of the shadow register file and the store buffer.

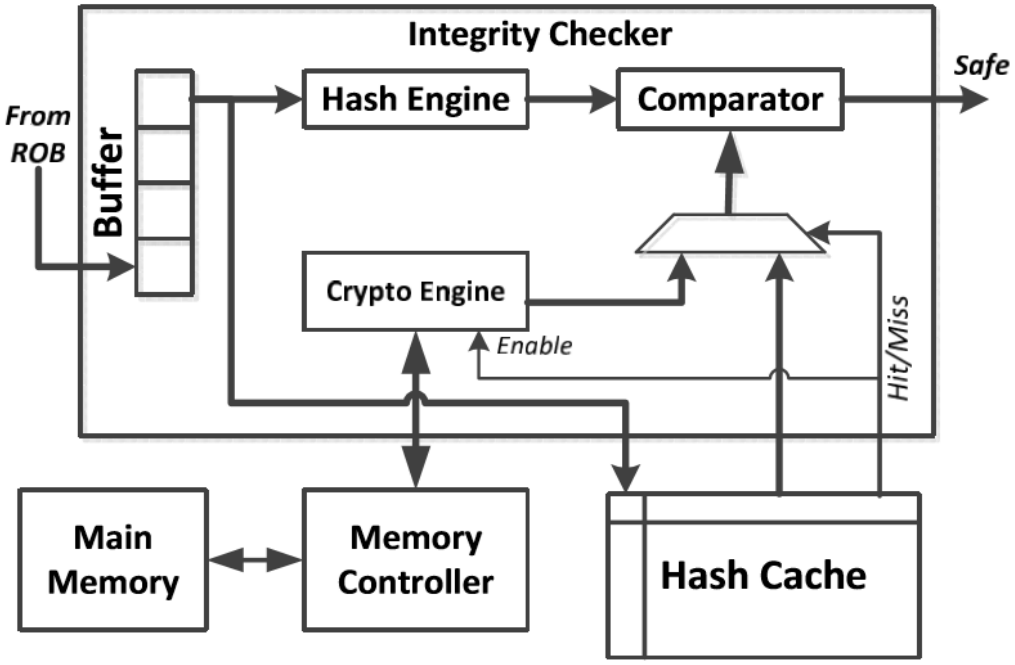
Figure 10 – Architecture of the scheme. Shaded blocks indicate the added components. The dashed lines indicate the new interconnections.




Reference: (KANUPARTHI, KARRI, *et al.*, 2012)

Figure 11 shows the internal structure of the integrity checker. The main components are a buffer, a hash engine, a crypto engine, and a comparator. The instructions are received by the integrity checker in program order from the reorder buffer and are held in the buffer until all the instructions that belong to a basic block or trace are available. The hash engine computes the hash of the instructions. The hash cache is accessed using the address of the first instruction in the basic block to fetch the corresponding precomputed hash. In case of a hash cache hit, the precomputed hash from the hash cache is used in the comparison. Otherwise, the encrypted precomputed hash is fetched from main memory, decrypted using the crypto engine, and is then used for comparison.

Figure 11 – Architectural details of the integrity checker.




Reference: (KANUPARTHI, KARRI, *et al.*, 2012)

### 2.4.3. Secure Return Address Stack (SRAS)

The work presented by (LEE, KARIG, *et al.*, 2004) describes a hardware-based secure return address stack (SRAS), which prevents malicious code injection involving procedure return address corruption. Only call and return instructions can

modify the contents of the SRAS. If the return address given by the SRAS hardware differs from that stored in the memory stack, then it is highly likely that the return address in the memory stack has been corrupted. A hardware SRAS structure contains a finite number of entries, which may be exceeded by the number of dynamically nested return addresses in the program. When this happens, the processor must securely spill SRAS contents to memory. The processor issues an OS interrupt to write or read SRAS contents to or from protected memory pages when SRAS overflow or underflow occurs. This SRAS overflow space in memory is protected from corruption by external sources by only allowing the OS kernel to access spilled SRAS contents.

### 2.4.4.Light-Weight Architecture Design

The paper enumerated by (KAO e WU, 2009) propose a light-weight architecture design change under the constraint to prevent from function return address attack by tracking the active return address and stack frame pointer. Memory write operations other than regular PUSH to the monitored location will be tagged and it will trigger the warning when the target address is actually used as return address. The advantage of tagging the location instead of saving the value of the return address is we do not need to track the consistency of the CALL/RET pair and the stack frame such as *setjmp* and *longjmp*. The approach is completely transparent to the software. The checking occurs within the micro-operation. The attackers cannot inject instructions to bypass the protection. The drawback is that since we do not keep every instance of the return address, and the use of the stack frame pointer register is optional, we cannot provide 100% coverage

### 2.4.5.Separates the stack in two parts

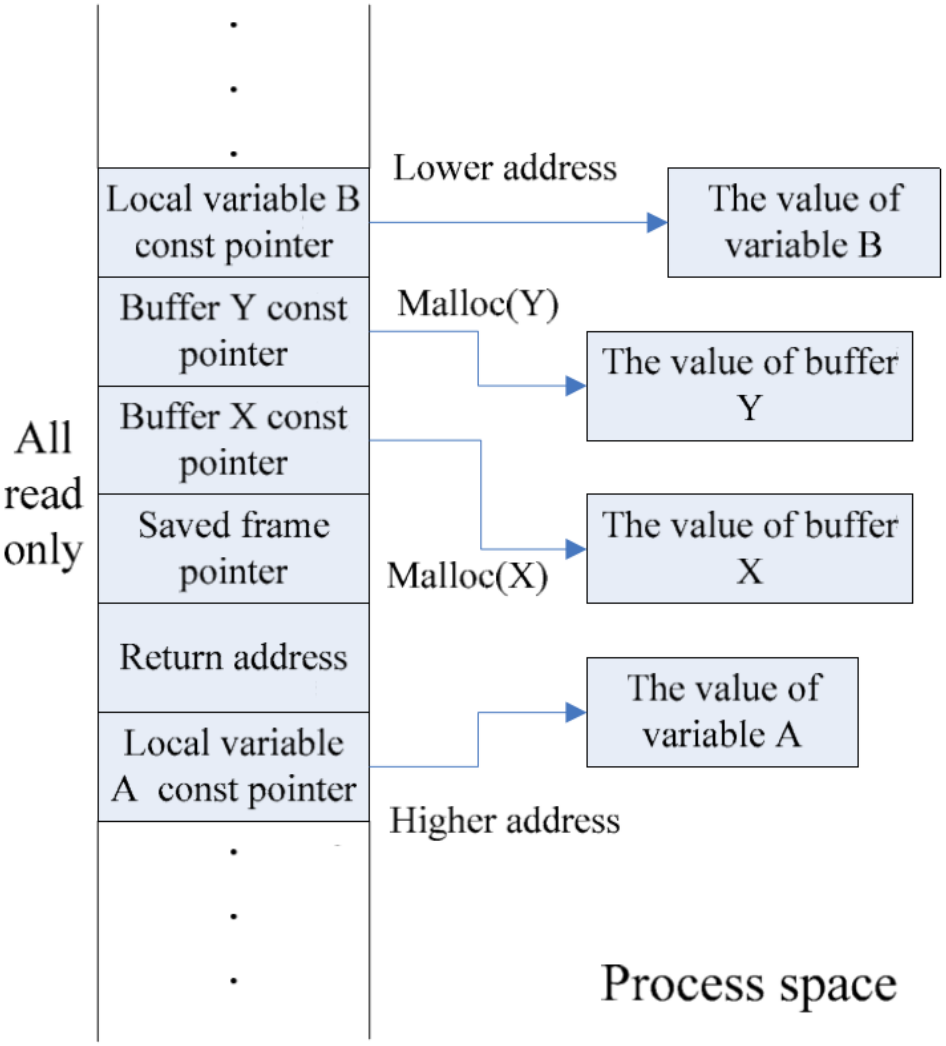
The method proposed by (DU e MAI, 2011) separates the standard stack to two parts, original stack saves the return address and the address of buffer with read permissions only, and the true values of buffer are saved in other space. A block of memory can be read, written and executed. In this approach, the space of stack will be set to read only, the features of it equal to a const pointer in C\C++ language.

That is means except initializing it we can’t change the space this pointer links to. Meanwhile, the stack frames store the addresses of local variables and buffer, and the value of them will be stored in heap or other memory space beyond the stack. In a sense, a stack attack would be not available with this approach, because the buffer overrun would not rewrite the return address and the saved frame pointer any more.

Taking an example to explain how this approach works. As the Figure 12 shows, the stored content of the stack of a program is not the real data of variables and buffers, but their memory addresses as a pointer. However, the structure is same as the original totally. Firstly, the return address is put into the stack, and then the saved frame pointer is put too. Meanwhile, some space would be improved for the local variables’ address and arrays’ address.

The distinctive is, there are two buffers, and a const pointer is put into the stack respectively. Once the const pointer has been initialized, it can be read only (STEVENS e RAGO, 2012). The value of buffers would be put into a space where the above pointer points to. Of course, the room must be in process space.

Figure 12 - Approach presented by (DU e MAI, 2011) to protect against stack overflow.




Reference: (DU e MAI, 2011)

### 2.4.6.Reconfigurable Dynamic Trusted Platform Module - ReDTPM

The method proposed by (DAS, WEI e LIU, 2014), implements a new DTPM. DTPM is an active security module as compared to the passive nature of TPM. Their DTPM design aims to perform the control flow checking for the software execution. This method contains two steps: offline profiling of the program and runtime control flow checking. By profiling the program, we generate the reference data which is used to verify the control flow at runtime. Control flow checking technique consists of Control Flow Checker (CFC) to perform the runtime checking and the Target Address Buffer (TAB) to buffer the addresses which need to be validated by CFC. The module

is implemented on FPGA to achieve the benefits of low cost, post-fabrication reconfigurability and sufficient performance.

Program execution flows from one instruction to the other in a sequential manner until a control flow instruction (CFI) is executed. After executing the CFI, the program counter (PC) takes the value of target address of the CFI, which may transfer the control to a different location. Thus, CFIs are responsible for changing the sequential flow of execution. An unexpected or deviated control flow is considered as an attack. Such attack tends to change the target address at runtime. Our method involves verification of these TAs in order to detect the deviation of the control flow from the normal execution. An alarm signal (e.g., a hardware interrupt to the OS) is generated to abort the program upon attack detection. Depending on ISA, there could be instructions apart from CALL, RET and JMP responsible for changing the control flow. Such instructions also need to taken into account for checking the control flow. As a proof of concept, this work limits to CALL, JMP and RET as CFI at present. However, new CFI instructions can be incorporated similarly.

### 2.4.7.Function Frame Runtime Randomization (FFRR)

The approach proposed by (KUMAR e KISORE, 2014) makes the memory location of the program objects on the stack (such as data objects and pointers of functions on the stack) a pretty more unpredictable. This is achieved by randomizing the relative distance (and in the process absolute address of the objects) between any two objects on the stack at run time (done at the beginning of the function execution). The basic idea is quite simple. In each function, we introduce as many random variables as the number of local variables declared in a function. These random values are used to add random number of words (padding) before the local variables are pushed on to the stack function frame at run time. The random numbers can be generated using a computationally inexpensive technique like linear feedback shift register (LFSR) during the function frame set up phase.

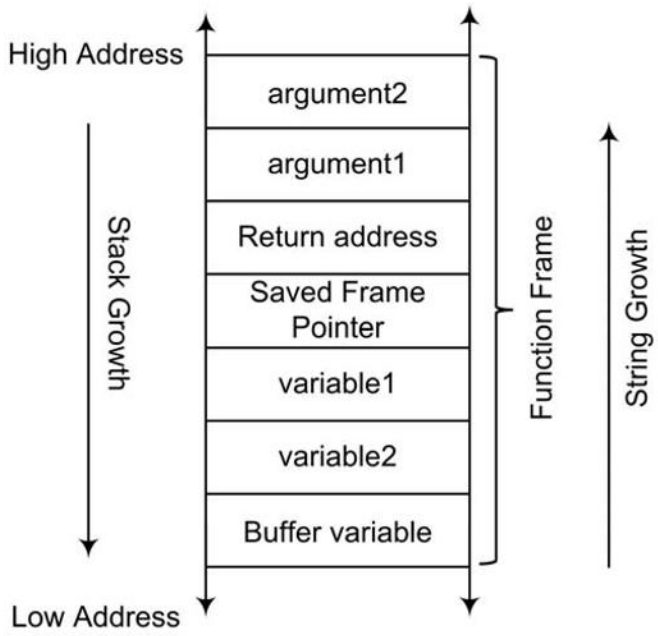
Consider a vulnerable function in a program whose stack memory layout is as shown in Figure 13. Assume that this function contains three local variables, in which the first two are non-buffer-type variables and third variable is a buffer type variable. The program vulnerability makes it possible to overflow the buffer-type variable and

overwrite the return address with address of malicious code to change the control of the program. Alternatively, in case the state of the adjacent non-buffer type variables is responsible for security validation, their corruption could result in bypassing certain security checks and thus resulting in gain of master/root privileges.

FFRR technique transforms the stack shown in Figure 13 to as shown in Figure 14(a). A random number of words are added before allocating the memory for local variables in stack function frame. The random numbers are represented as a function of time (t) because the random numbers vary from one function to another function (randomization in space domain) and also for each execution of the same function (randomization in time domain). These numbers are chosen first by the function prologue using LFSR. In Figure 14(a), δr1, δr2, δr3 are 2, 5 and 3 words respectively and the most recent random number is retained and serves as a seed for subsequent random sequence generation.

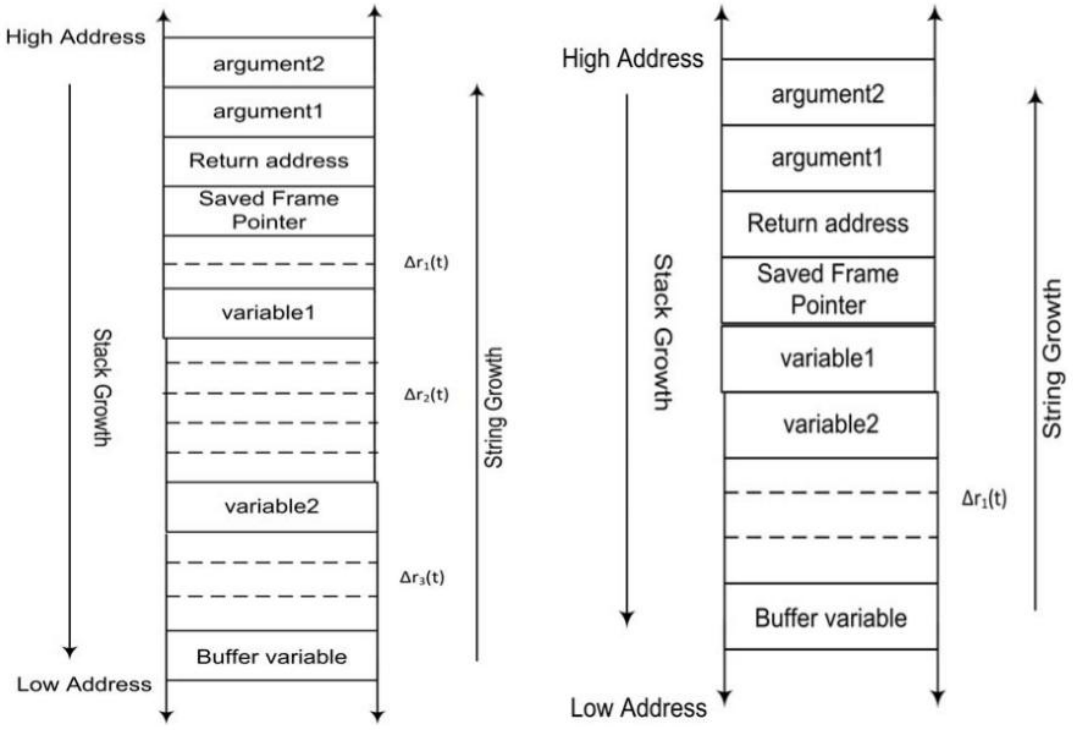
However, the above approach will introduce high overheads for each function call. Therefore we recommend applying this approach only for buffer-type local variables. Specifically, for each function we introduce the number of random variables (equal to number of local buffers in a function) and just as in the previous case, the random values are used to add the random number of words (padding) before the buffer-type local variables. The resulting function frame is as shown in Figure 14(b). FFRR does not impact the existing process for pushing software updates or patches as the proposed technique randomizes only the run time copy of the program binary.

Figure 13 - Layout of function frame on the stack.




Reference: (KUMAR e KISORE, 2014)

Figure 14 - Function Frame Runtime Randomization – (a) FFRR technique for all local variables; (b) FFRR technique for only buffer-type variable.




(a) (b)

Reference: (KUMAR e KISORE, 2014).

## 2.4.8. SafeStack - Memory access virtualization

The work proposed by (CHEN, JIN, *et al.*, 2013), presents a technique called Memory Access Virtualization. This technique allows the relocation of memory objects to other locations to maintain a program's functionality at runtime. With the ability of memory object relocation, for attack diagnosis the approach can move stack buffers to a monitored memory region to detect whether some of them have out-of-bound access, while for attack prevention the approach can move vulnerable buffers into protected memory areas, and then write values into (or take values out of) the corresponding protected memory areas instead of the original stack address space. Therefore, the approach can safely mask buffer overflow attacks such as an out-of-bound write or an out-of-bound read and make the program continue to execute normally, instead of throwing up an exception and terminating the program - an undesirable situation for an important business server program.

Based on this technique, the approach builds SafeStack to automatically diagnose and patch stack-based buffer overflow attacks. Specifically, it use memory access virtualization mechanism to relocate the bug-triggering buffer into protected memory areas, as shown in Figure 15b. In this way, the two vulnerable buffers are protected from being used to overrun the control data.

Memory access virtualization adopts an object-relocation table to map the original memory address of an object to a new address. The mapping is relatively simple if the access to a buffer is within that buffer. Otherwise, the mapping has to be done carefully as the system needs to consider whether it is an out-of-bound access or a legitimate access to a different variable.

This approach focuses the discussion on the memory access virtualization mechanism for simple arrays. The mechanism works in a similar way for other types of local buffers, such as structures, structure arrays, unions, and union arrays.

There are two main types of array element access: direct access and indirect access. For example, defining a character array of size 20 as `a[20]`, `a[5]` is a direct array element access using the constant index 5, and `a[i]` is an indirect array element access using the index variable `i`.

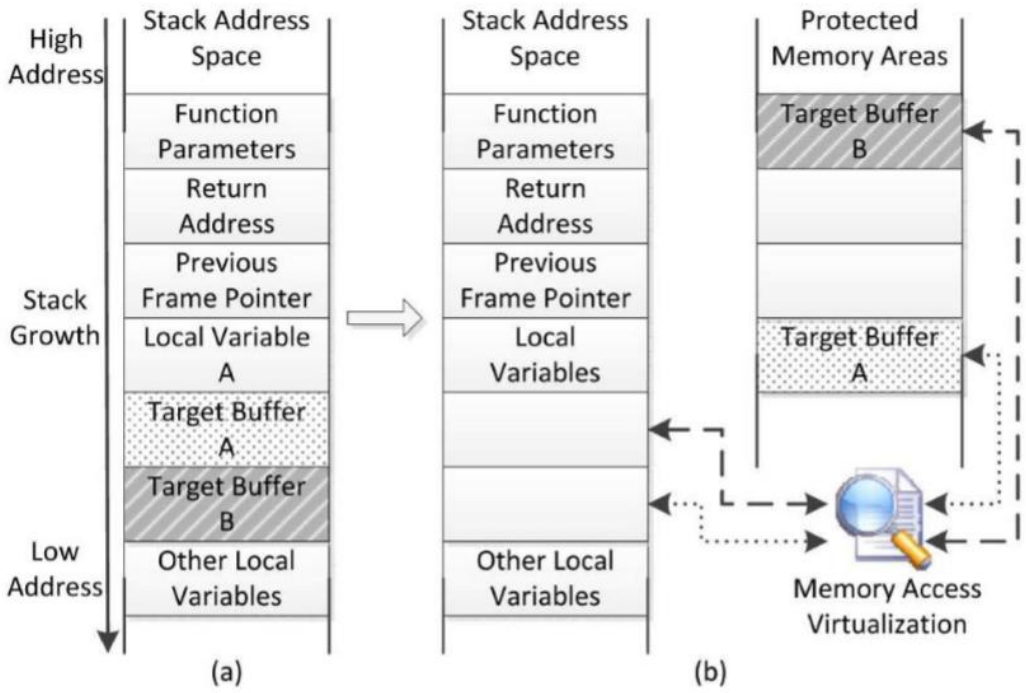
For the direct access, the array index is a constant. The offset from the frame pointer or the stack pointer is determined at the compilation time. In this case, the effective memory address is inside the object-relocation table, and the addressing mode is "*Base plus Offset."* The base is the base register (EBP or ESP) and the offset is the value between the variable (e.g., `a[20]`) and the base register. SafeStack simply replaces this original address with its corresponding new address before executing the instruction. There is a case that the stack buffer subscript is a constant and out of range, such as `a[24]`. Fortunately, this explicit out of bound access can be detected in the testing phase before the application is released, and it is reasonable that we assume the program does not have this explicit out of bound access.

For the indirect access, the array index is an expression. The effective memory address is the sum of the starting address of the array and the size of the array element multiplied by the array index, and the addressing mode is *"Base plus Index plus Offset.*" The base is the base register, the offset is the value between the starting address of the array and the base register, and the index is the index register

(storing value `i`, which is scaled by the size of the array element). SafeStack first calculates the starting address of the array, finds the corresponding new address according to the object-relocation table, calculates the final new address, and replaces the original address with it before executing the instruction.

In addition to these two types of array element access, arrays can also be accessed by pointers to arrays. For example, a program can define a pointer variable pointing to an array, and access array elements using the pointer variable. To handle this type of access, SafeStack first replaces the original address with the corresponding new address, and subsequently all the offset calculation based on this pointer can be directed onto the corresponding new address without the needs to map new address for each instruction subsequently. Similarly, most of the time the address of an array is passed as an argument to a function, for example, the address of an array is passed to the function `strcpy` to copy data into the array. In this case, the value of the argument has been replaced with the corresponding new address. This solution avoids searching the stack frame and checking every instruction for address mapping, and thus has better performance. For Single Instruction, Multiple Data (SIMD) instructions, they also specify the addresses of arrays to process. Therefore, SafeStack can replace them with corresponding new address before processing.

Figure 15 - (a) Stack without memory access virtualization and (b) stack with memory access Virtualization.




Reference: (CHEN, JIN, *et al.*, 2013).

## 2.5. COMPARISON AMONG RELATED WORKS

Table 2 compares the related works commented above. It shows advantages and disadvantages related to each methodology quoted in this work.

Table 2 – Comparison among the most common approaches.

| Index | Approach | Advantages | Drawbacks |
|---|---|---|---|
| 1 | StackGuard (COWAN, PU, *et al.*, 1998) (COWAN, BEATTIE, *et al.*, 1999) | • Independent of HW (processor architecture) or SW (application code description language and OS) platforms. | • Need application code recompilation.<br>• Based on SW, thus implying considerably performance degradation. |
| 2 | Dynamic Integrity Checking (KANUPARTHI, KARRI, *et al.*, 2012) | • High detection coverage<br>• Added stack in | • Need a pre-execution time for hash generation. |

| | | | |
|---|---|---|---|
| | (KANUPARTHI, ZAHRAN e KARRI, 2012) | HW with low area overhead | • Need hash's generation of every application to work well. |
| 3 | SRAS (LEE, KARIG, *et al.*, 2004) | • Added stack in HW with low area overhead | • Need an OS to manage memory. |
| 4 | Light-weight hardware return address and stack Frame Tracking (KAO e WU, 2009) | • Added stack in hardware | • Need an OS to manage the memory.<br>• Require more area than SRAS due to the additional need to save stack frame pointer. |
| 5 | Separates the stack to two parts (DU e MAI, 2011) | • Independent of the HW/SW platform | • Need application code recompilation. |
| 6 | ReDTPM (DAS, WEI e LIU, 2014) | • Independent of the HW/SW platform | • Need binary application profiling.<br>• Need memory space to keep extra data. |
| 7 | FFRR (KUMAR e KISORE, 2014) | • Independent of HW. | • High memory overhead.<br>• Not so safe, due to attackers could set values on the stack as they want, and discover the right return address location by brute force. |
| 8 | SafeStack – Memory Access Virtualization (CHEN, JIN, *et al.*, 2013) | • Independent of HW. | • Add execution overhead when reading and write data from memory. |

Reference: Segabinazzi (2016)

Additionally to Table 2 the Table 3 makes a comparison between the main related works and the proposed approach in this dissertation. The first two lines are related to the Watchdog and the Recovery Mechanism proposed by this dissertation, the lines that follow are related to approaches enumerated in Table 2. To better illustrate this table the advantages and drawbacks are summarized below:

- Advantages:

- 1: Independent of software (OS or extra software to support the approach);
- 2: Transparent to the hardware point of view (processor architecture);
- 3: High Detection coverage (more than 3 kings of vulnerabilities);
- 4: Low area overhead;

- Drawbacks:
  - 1: Dependent of software;
  - 2: Not transparent to the hardware;
  - 3: Low detection coverage (less than 2 kinds of vulnerabilities);
  - 4: High area overhead;
  - 5: Need application code recompilation;
  - 6: Imply considerable performance degradation;
  - 7: Need a static timing analysis (a pre-execution time analysis);

Table 3 – Comparison among main related works and the approach proposed by this dissertation.

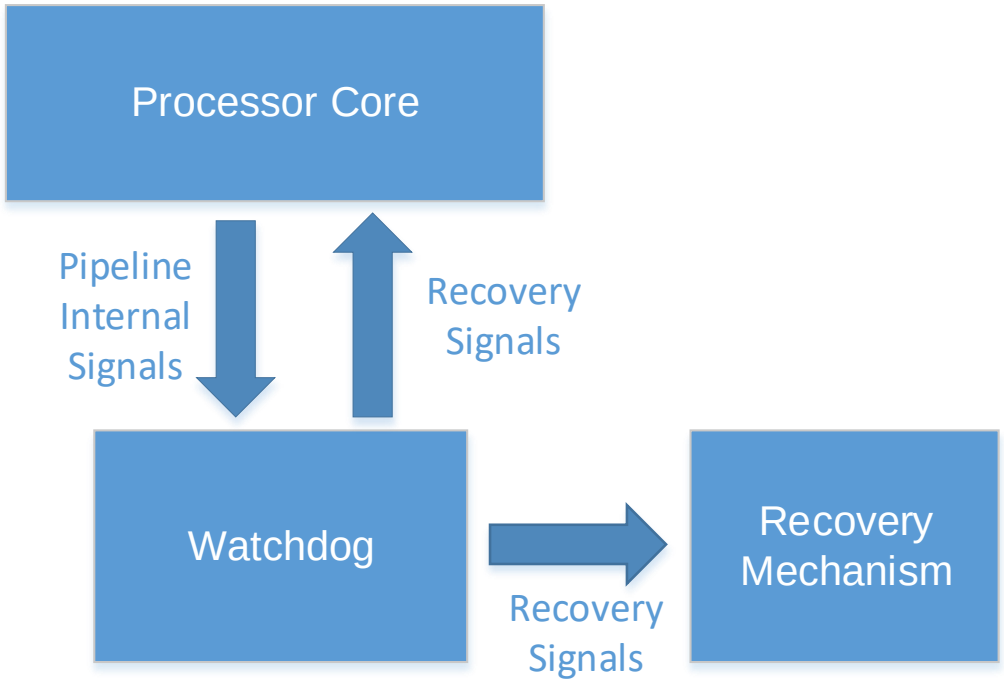
| **Approach** | **Advantages** | | | | **Drawbacks** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **Watchdog** | X | | | X | | X | X | | | | |
| **Recovery Mechanism** | X | | | | | X | X | X | | X | |
| **1 - Stack Guard** | | X | | X | X | | X | | X | X | |
| **2 - Dynamic Integrity Checking** | X | | X | X | | X | | | | | X |
| **3 – SRAS** | | | | X | X | X | X | | | | |
| **4 - Light-weight hardware return address and stack Frame Tracking** | | | | X | X | X | X | | | | |

Reference: Segabinazzi (2016)

# 3. THE PROPOSED APPROACH

The work proposed by this dissertation implements two main approaches (Figure 16): a Watchdog to detect malicious overwritten in the return addresses saved in the program stack, and a preliminary of a method to try to recover the system starting from the return address overwritten detection. This overwritten happens in buffer overflow situations, and could be a first step to a stack smashing attack. So, this work is capable to detect a tentative of attack by stacking smashing and could be able to recover the system to a safe point generated during normal execution.

Figure 16 – General architecture proposed by this work.




Reference: Segabinazzi (2016)

The proposed Watchdog is based on two specific structures (a) the logic implemented in hardware to detect the return address overwritten and (b) the memory block added in the Watchdog structure used to store functions return addresses. The paragraphs that follow explain in more details how this approach works:

- Every time a call instruction is executed by the processor, the return address is stored in the original stack (typically a memory address or a dedicated register inside the processor) and in added Watchdog memory block;
- Every time a return instruction is executed, the Watchdog performs a comparison between the return address stored in the original stack and in the Watchdog memory block. In this case, one of these two situations may occur:
    - In case of a positive comparison, the regular execution of code takes place;
    - In case of a negative comparison, the Watchdog raises a signal to the second part of this work: the recovery mechanism.

Therefore, in case of occurring an overflow on the original stack that corrupts the return address, such address remains unchanged in the Watchdog memory block. This condition guarantees the detection of the return address overwritten by comparison between copies of the stored return addresses. Finally, this approach do not allow the system to branch to some malicious code possible pointed by this corrupted return address and, in addition, raise a signal to start the recovery process.

Starting from the signal generated by the Watchdog, the recovery algorithm rollback the system to the last safe-point generated previously in run-time. This safe point is the last function call coming from *main* function. Then, when the system finishes the rollback, it starts to run again normally.

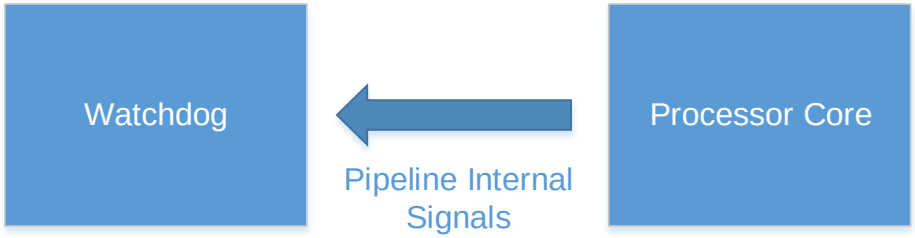
Given the above exposed, the proposed approach presents the following features and advantages compared to the existing techniques:

- Does not need application code recompilation;
- It is not based on any software component;
- The Watchdog detection mechanism requires a low area overhead;
- Negligible performance degradation to recognize the return address overwritten;
- Extremely low attack detection latency;
- Gives to the system the opportunity to recovery from a kind of attack by yourself.

## 3.1. WATCHDOG GENERAL ARCHITECTURE

Figure 17 depicts the general block diagram of the proposed Watchdog, where it is instantiated besides the processor core.

Figure 17 – Watchdog instantiated besides the processor core.




Reference: Segabinazzi (2016)

As observed in Figure 17 and Figure 18, the Watchdog monitors some internal signals from the execution stage of the processor pipeline. Such signals are described below:

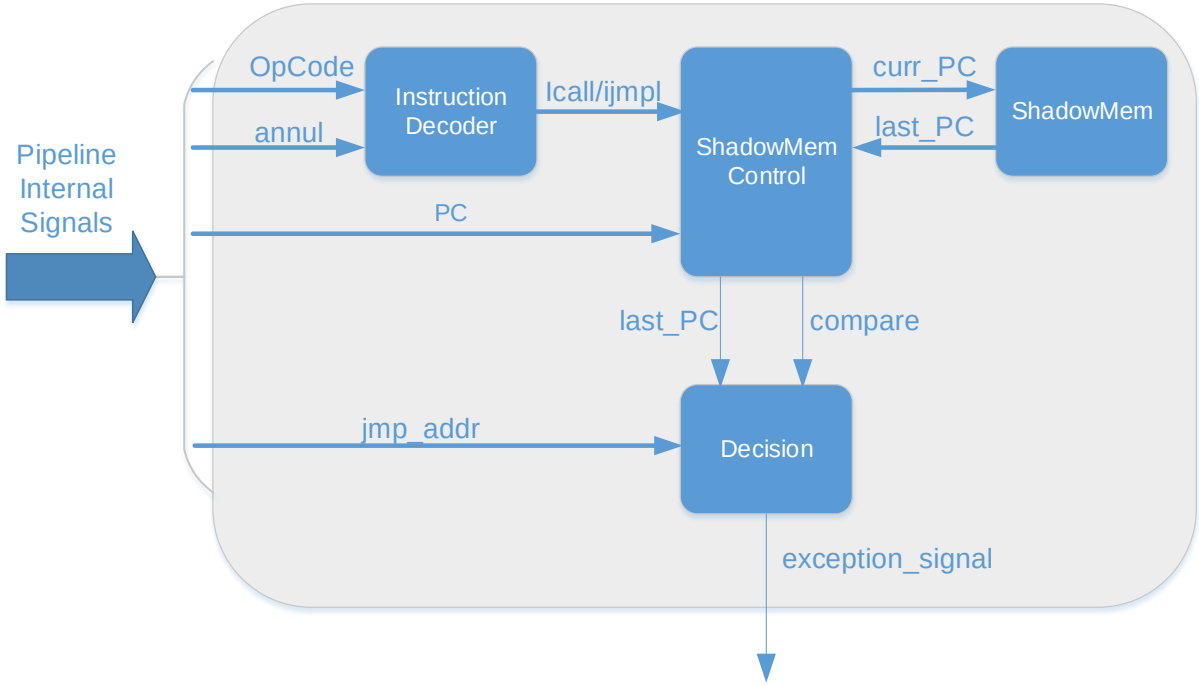
- The "*OpCode*" of the instruction that is leaving the Execution Stage of the pipeline;
- The bit "*annul*", whose function is to indicate if the instruction that is leaving the Execution Stage of the pipeline will be actually executed by the processor or it will be discarded due to speculative execution.
- The "*Program Counter*" (PC), which is saved into the Watchdog memory in case a function "CALL" is performed. After the function execution, the PC is defined as the return address that will be used to return processor control to the point where the application was interrupted.
- The "*jmp_addr*" signal, which points to the function return address that will be executed.

Figure 18 shows the internal blocks of the proposed Watchdog. As detailed above, it grabs a set of four specific pipeline internal signals. The Instruction Decoder Block uses the instruction "OpCode" and the "annul" signal to decode and check if

the current instruction will be executed. If the Instruction Decoder Block decodes a function “CALL”, it will send the *icall* signal to the ShadowMem Control Block. In this case, the ShadowMem Control Block will save the current PC retrieved from the pipeline (“Curr_PC” signal) into the ShadowMem Block. Instead, if the Instruction Decoder Block decodes a function return instruction (IJMPL), it will send the “ijmpl” signal to the ShadowMem Control Block that will recover from the ShadowMem Block the last PC saved therein and send it (together with *compare* signal) to Decision Block.

When the Decision Block receives the *compare* signal and the “last PC” it performs a comparison between this value and the *jmp_addr* retrieved from the pipeline. If this comparison returns true, no action is required. Nevertheless, if the comparison returns false the exception signal will be send to a Watchdog external block, the Recovery Mechanism that will be explained in the section 3.2.

Figure 18 - Internal blocks of the Watchdog.




Reference: Segabinazzi (2016)

## 3.2. THE RECOVERY MECHANISM

The recovery process proposed by this work includes in the processor architecture three main blocks: (1) the *Recovery_ctlr*; (2) the *Reg_recovery_ctlr; and* (3) the Reg_recovery_mem. Also, two extra memory blocks are proposed: (1) the Secondary_mem; and (2) the Recovery_mem. The general architecture is illustrated by Figure 19 and these five blocks are explained below:

(1) The ***Recovery_ctlr*** should gets the signals from Watchdog, generate safe points and saves data coming from address and data bus as necessary in the *Secondary_mem* or in the *Recovery_mem*;

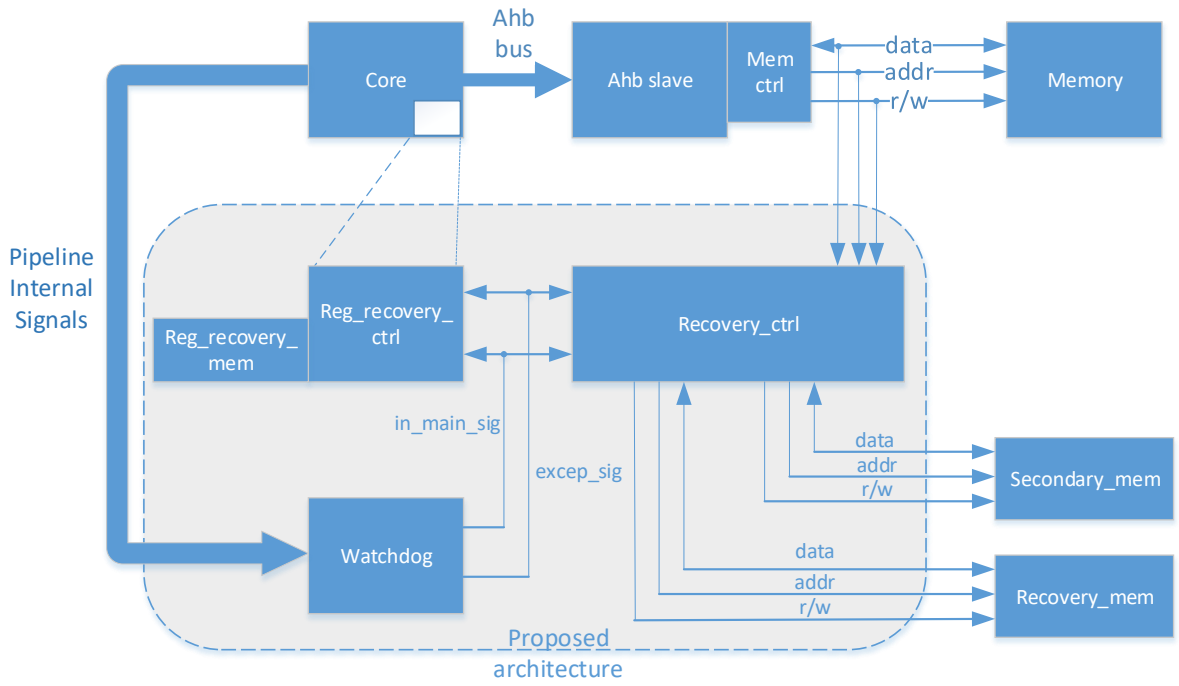
(2) The ***Reg_recovery_ctlr*** is a controller block added in the core of processor. This block receives the same signals from Watchdog, but it generate safe points by saving in the *Reg_recovery_mem* the data containing in the core registers;

(3) The ***Secondary_mem*** is a memory block that saves data received from *Recovery_ctrl* when the execution is in the *main* function;

(4) The ***Recovery_mem*** receives data from *Recovery_*ctrl too, but it saves all modifications occurred in the memory when the program goes out of main function. In the other words, when it is in function call situations every data and the respective data address are saved in the *Recovery_mem* in a queue order.

(5) The ***Reg_recovery_mem*** receives the data from *Reg_recovery_ctrl,* it saves the data coming from core registers.

Figure 19 – Recovery mechanism included in the processor.




Reference: Segabinazzi (2016)

As could be seen in the Figure 19 the *Recovery_ctlr* and the *Reg_recovery_ctlr* receive two signals from Watchdog:

- The ***exception_signal***: this signal is generated by Watchdog, and it is raised when the Watchdog detect a return address overwritten;
- The ***in_main_signal***: this signal is generated by Watchdog too. But it is raised or put down when the program execution is in or out of *main* function respectively.

So, to do the recovery, these blocks will work in the follow situations as described below.

### 3.2.1.Normal execution

This section describes the flow of the recovery process during normal execution. When the program is on *main* function, the execution data is saved in the

main memory (as a normal way) and in the Secondary_mem memory by the Recovery_ctrl. So when the execution goes out of *main* function by a function call:

- The program continues to save data in a normal way in the primary memory.
- The Watchdog put down the signal in_main_sig;
- The Recovery_crtl stops to save data in the Secondary_mem, but;

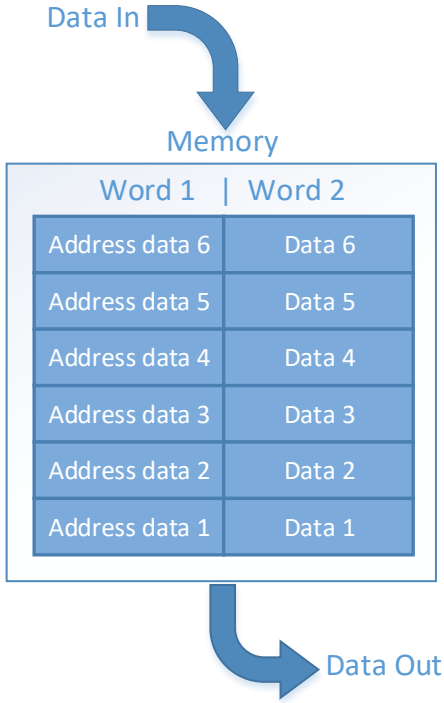
It starts to save on Recovery_mem the data itself and the respective address where any data was saved in the principal memory, these data are saved in this memory in a queue order as showed in the

- Figure 20;
- The Reg_recovery_ctrl generate the safe point by getting all internal register states and saving these data in the Reg_recovery_mem block.

When the execution returns to main function the signal *in_main_sig* is raised and:

- The data saved in the Recovery_mem will be committed to Secondary_mem in a FIFO (first in – first out) order;
- The data saved in this step is just the data owned by the piece of stack that was not released yet by the program execution and the data in other memory spaces with write permission (*e.g* heap and data).
- However if any data are required to save in memory space with non-write permissions this data will be discarded, in other words, if one “Address Data” corresponds to a memory address with non-write permissions, the “Data” owned by this “Address Data” will be discarded.
- Finally, the internal register states saved in the Reg_recovery_mem are discarded;

Figure 20 – The Recovery_mem memory block receiving data and the respective data address in a queue order.




Reference: Segabinazzi (2016)

### 3.2.2.In a return address overwritten detection

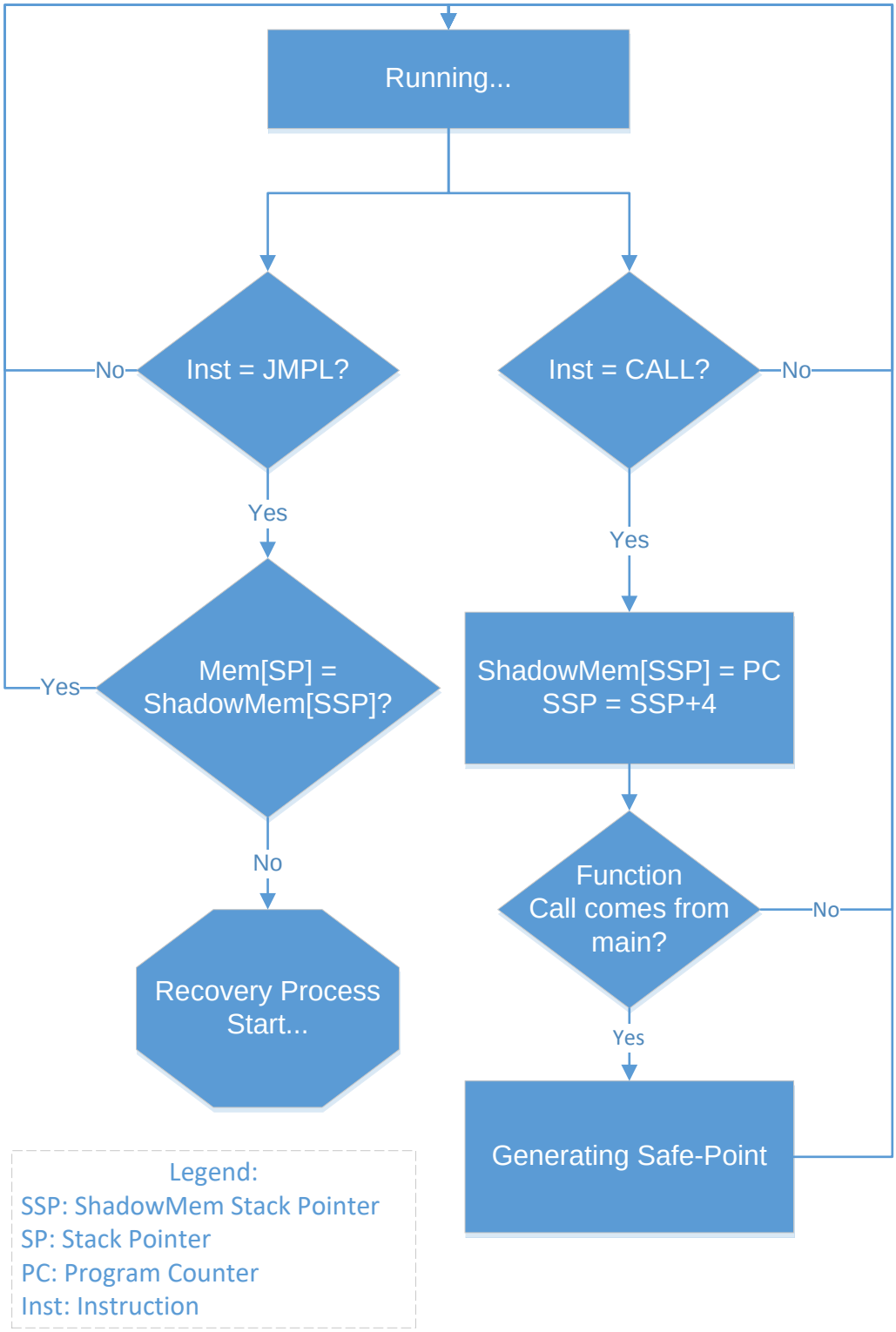
This section describes the Recovery Mechanism actions when the Watchdog detects a return address overwritten. So, in this situation, the Watchdog raises the exception_sig signal, and the Recovery Mechanism does the steps below:

- The program execution is stopped;
- The system is rolled back to the last generated safe point (the point where the first function was called in the main function), the Reg_recovery_ctrl will be responsible to save the register states saved in the Reg_recovery_mem in the official registers;
- All data saved in the Recovery_mem will be discarded;
- The data in the Secondary_mem memory will overwrite all data in the main memory data;
- Finally, the program starts to run again from the safe point.

## 3.3. DETECTION, CHECK-POINTING, AND RECOVERY

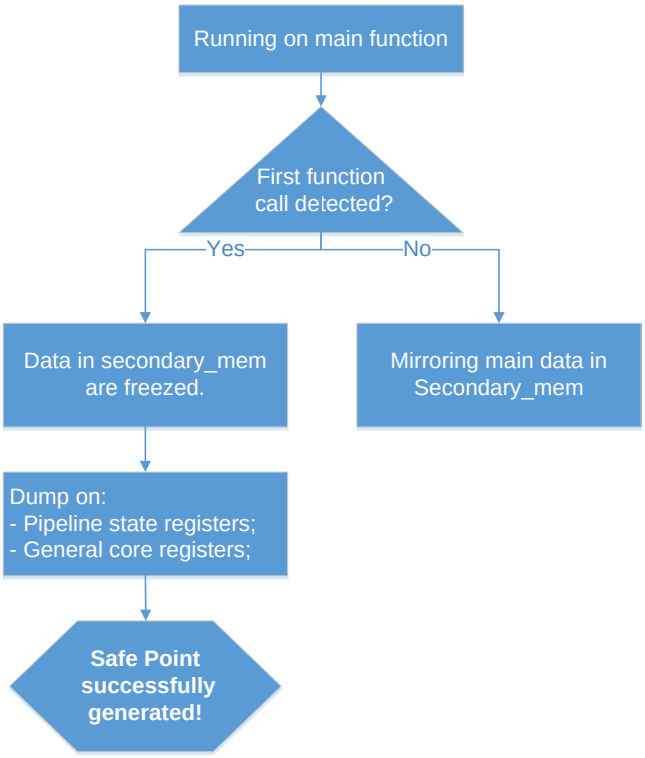
The flow chart in the Figure 21 describes the basic actions performed by detection process. Also, the recovery process is described as follows. The Figure 22 describes the operations while the system is on main function, the Figure 23 describes the operations while the system is out of main function and finally the Figure 24 describes the operations performed by the Recovery Mechanism when a return address overwritten was detected.

Figure 21 – Flow chart describing the basic operations performed by the Watchdog.



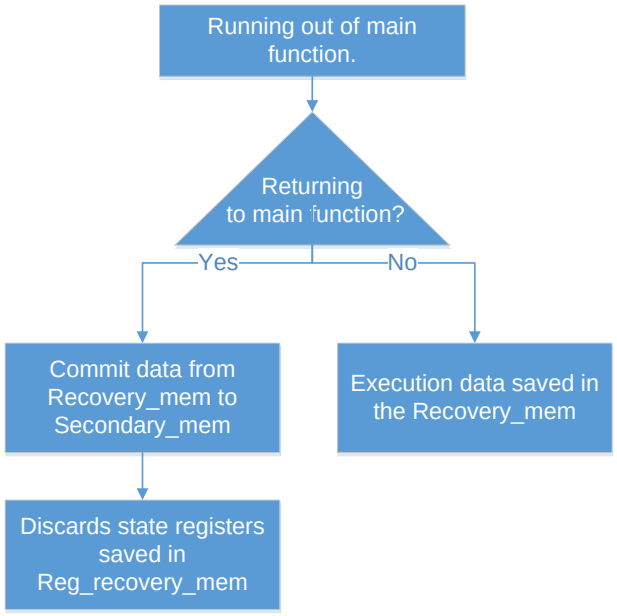

Reference: Segabinazzi (2016)

Figure 22 – Recovery mechanism operations when the execution is on main function.



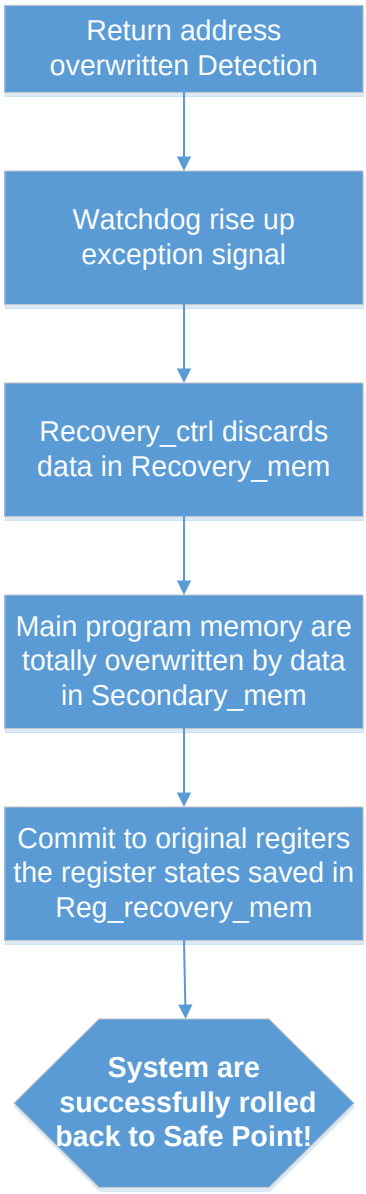

Reference: Segabinazzi (2016)

Figure 23 – Operation performed by Recovery Mechanism when the execution is out of main function.




Reference: Segabinazzi (2016)

Figure 24 – Recovery mechanism operations when an overwritten was detected in the recovery mechanism.




Reference: Segabinazzi (2016)

The Watchdog monitors the pipeline internal signals since the beginning of the program. And while program execution is on main function, the Recovery Mechanism uses the Secondary_mem memory block to save data as a mirror of main memory.

By monitoring processor pipeline internal signals, it is possible to detect the moment when a CALL instruction is executed. At this moment, the current PC is

simultaneously saved in the regular Stack and in the ShadowMem. In more detail, the following operations are performed:

```
Stack[SP+4]=PC                    (1)
ShadowMem[SSP+4]=PC               (2)
```

After saving PC context in the Stack and in the ShadowMem, the Stack Pointer (SP) and the ShadowMem Stack Pointer (SSP) are incremented as follows:

```
SP=SP+4                           (3)
SSP=SSP+4                         (4)
```

At this moment, the Recovery Mechanism checks if this function call is coming from *main* function (the first function call). If yes, the Recovery Mechanism starts to save data in the Recovery_mem memory and generate the first safe-point by saving all internal pipeline state register and general state register in the Reg_recovery_mem block.

By continuing the execution, when an instruction to return from function is executed by the processor, both stacks are read out by the Watchdog and their contents are compared:

```
Stack[SP]==ShadowMem[SSP]         (5)
```

When system returns to main function and no errors are detected, the Recovery Mechanism commits the data contained in the Recovery_Mem to Secondary_mem memory block. Also, at this point, the register states saved in the Reg_recovery_mem are discarded.

If the comparison made in the step (5) returns true, the PC register receives the PC stored in the original stack and the SP and the SSP are updated as follows:

```
PC=Stack[SP]                      (6)
SP=SP-4                           (7)
SSP=SSP-4                         (8)
```

If the comparison done in the step (5) returns false the Watchdog raises the *exception_sig* signal to start the recovery process. The Recovery Mechanism rollback the system to the safe point saved previously by attributing the registers states saved in the Reg_recovery_mem in the officials' core internal registers. All data saved after

this point will be discarded, and the main memory will be totally overwritten by data in the Secondary_mem. Finally, the execution is now reinitiated from this point.

# 4. VALIDATION

The validation of this work was made by implementing a simple C program (Figure 25), two points were verified with this code: (1) the Watchdog capacity to recognize the return address overwritten and (2) the Recovery Mechanism capacity to rollback the system to the last safe-point.

## 4.1. A SIMPLE C PROGRAM

The first step to validate this work was to implement a simple C program. This program performs a buffer overflow that overwrites a return address located in the stack. The C source code of this program is shown in the Figure 25:

Figure 25 – C source code used to check the Watchdog return address overwritten detection.

```
void sploit(void)
{
     char buff[8];
     memset(buff, 0xff, 1024);
     return;
}
void sploit1(void)
{
     sploit();
     return;
}
int main(void){
     sploit1();
     return 0;
}
```

Reference: Segabinazzi (2016)

This program uses the *memset* function to set a value in the char array called "*buff*". The *memset* function fills the first 1024 bytes of the memory area pointed by "*buff*" with the constant value "*0xff*". However the parameter size (1024 bytes) is much larger than the char array "*buff"* size (only 8 bytes). So, as explained in section 2.2, we have a buffer overflow situation. In this case, the *sploit1* function return address (which was saved in the stack) is overwritten by the value in *memset* function. Also, in this implemented program the safe point, accordingly to recovery mechanism, is the *sploit1* function call. Table 4 shows the safe point instruction address.

Finally, as expected result, the Watchdog should generate the exception signal and, additionally, the Recovery Mechanism should rollback the program to *sploit1()* function call.

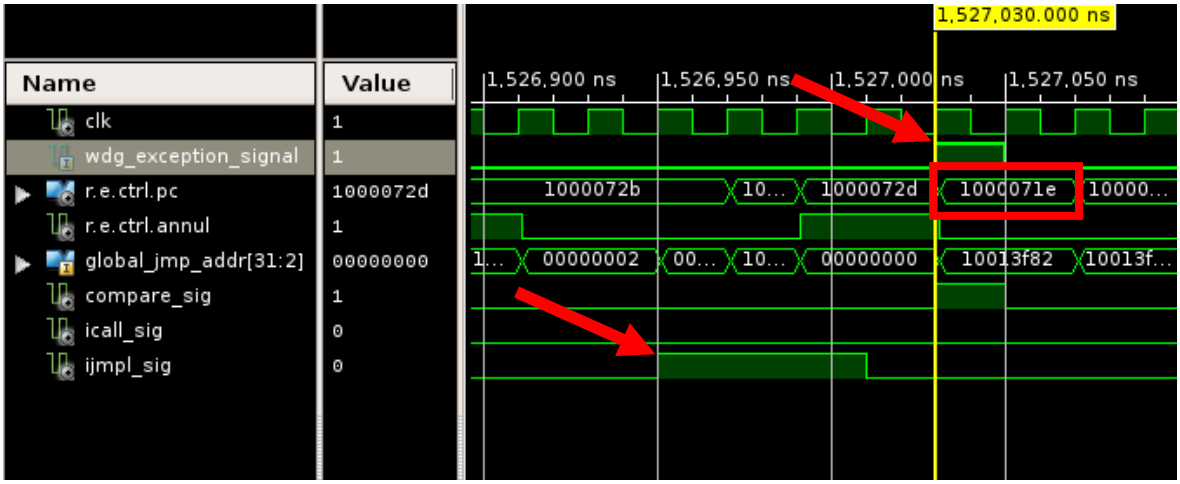
Table 4 - Safe point in the simple C code implemented.

| Main function call | Main function call instruction address [hex] | Instruction | Safe Point [hex] |
|---|---|---|---|
| *sploit1()* | *40001e1c* | *CALL 40001c78* | *40001c78* |

Reference: Segabinazzi (2016)

So, when processor executes the implemented program and tries to execute the return address, the Watchdog generates the exception signal due to return address overwritten detection, and also, the Recovery Mechanism rollback the system to the safe point. The Figure 26 shows the right moment when the Watchdog raises the wdg_exception_signal and attributes to the execution PC (*r.e.ctrl.pc*) the address *0x1000071e*, which after left shift *(2 << 0x1000071e)* it is equal to *0x40001c78*, the safe point listed in the Table 4.

Figure 26 - Moment when the Watchdog recognize the return address overwritten and the Recovery Mechanism rollback the system to the safe point.




Reference: Segabinazzi (2016)

# 5. EVALUATION

The evaluation of this approach was done in three steps: pieces of vulnerable source codes get from open source programs were used to evaluate this work in general situations, so the (1) Watchdog and (2) the Recovery Mechanism were tested using these codes, and finally, (3) the area and execution time overheads were calculated.

## 5.1. TEST CASES EVALUATION

To make an analysis under more realistic situations, test programs were implemented with pieces of known vulnerable C codes. These vulnerable pieces of C code were get from vulnerabilities published in the Common Vulnerabilities and Exposures (CVE). CVE is a dictionary of common names (i.e., CVE Identifiers) for publicly known cybersecurity vulnerabilities. CVE's common identifiers make it easier to share data across separate network security databases and tools, and provide a baseline for evaluating the coverage of an organization’s security tools (Common Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names, 2015) (National Vulnerability Database - Vulnerability search).

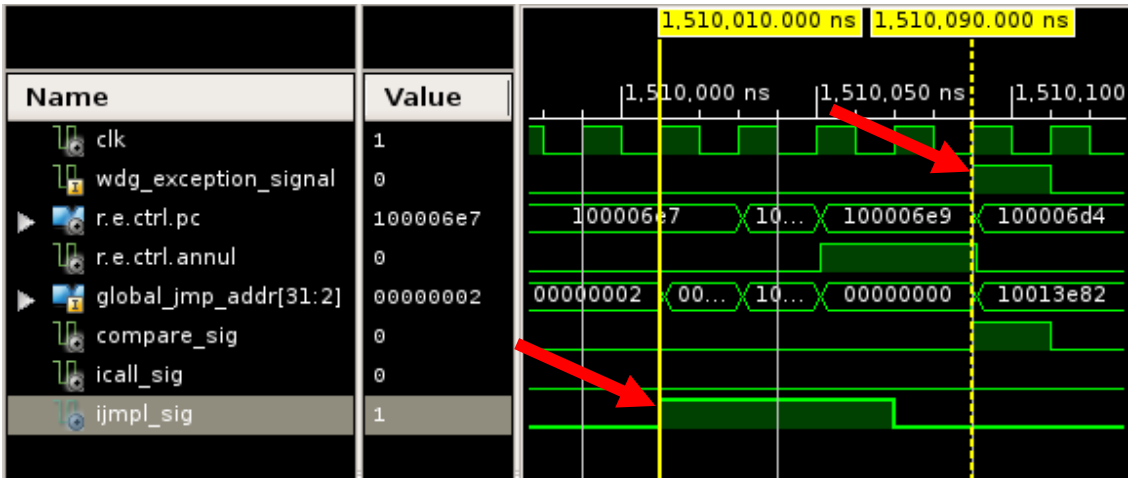
These test programs were implemented by including the snippet vulnerable code into the template C source code. This template code is quite simple, just initializes the program and calls the vulnerable code in such a way to force the buffer overflow situation. Finally, these programs were compiled and simulated on Leon3 environment constructed under this dissertation. So, while each one of these programs is running, the Watchdog and the Recovery Mechanism were evaluated.

### 5.1.1. Test cases evaluation – Watchdog detection

The overall results of the Watchdog detection were show in the Table 5. To better illustrate this result the Figure 27 were get when the test case Edbrowse is under the test, and it shows the right moment when the Watchdog generates the exception signal. In this figure the intermediary Watchdog signals could be observed, the raised signal *ijmpl_sig* shows that a function return instruction are been called. The exception signal named as *wdg_exception_signal* are raised when the

*compare_sig* signal are raised too. These signals configuration illustrates the situation where the Watchdog detects a return address overwritten successfully.

Figure 27 – Simulation moment when the exception signal is generated for the test case Edbrowse.




Reference: Segabinazzi (2016)

Table 5 - Watchdog detection using pieces of vulnerable codes obtained from vulnerabilities published in CVE.

| Vulnerable Programs | CVE Number | Severity (National Vulnerability Database - Vulnerability search) | Result |
|---|---|---|---|
| Edbrowse | CVE-2006-6909 | 10.0 high | Watchdog generates the exception signal |
| MADWiFi | CVE-2006-6332 | 7.5 high | Watchdog generates the exception signal |
| Samba | CVE-2007-0453 | 4.6 medium | Watchdog generates the exception signal |
| Sendmail | CVE-2003-0681 | 7.5 high | Watchdog generates the exception signal |

| | | | |
|---|---|---|---|
| Wu-ftpd | CVE-1999-0368 | 10.0 high | Watchdog generates the exception signal |
| Wu-ftpd | CVE-2003-0466 | 10.0 high | Watchdog generates the exception signal |

Reference: Segabinazzi (2016)

### 5.1.2. Test cases evaluation – Recovery mechanism

In this section the Recovery Mechanism is evaluated by using the same sniped codes from the section above. Starting from the exception signal generated by the Watchdog the capacity of the Recovery Mechanism to rollback the system to the last safe point and continue executing the program normally from this point were evaluated.

To make this evaluation the existing safe points in the compiled code needs to be checked. For the Edbrowse test case the safe points found in the compiled code are listed in the Table 6.

Table 6 – Safe points found in the test code done with Edbrowse sniped code.

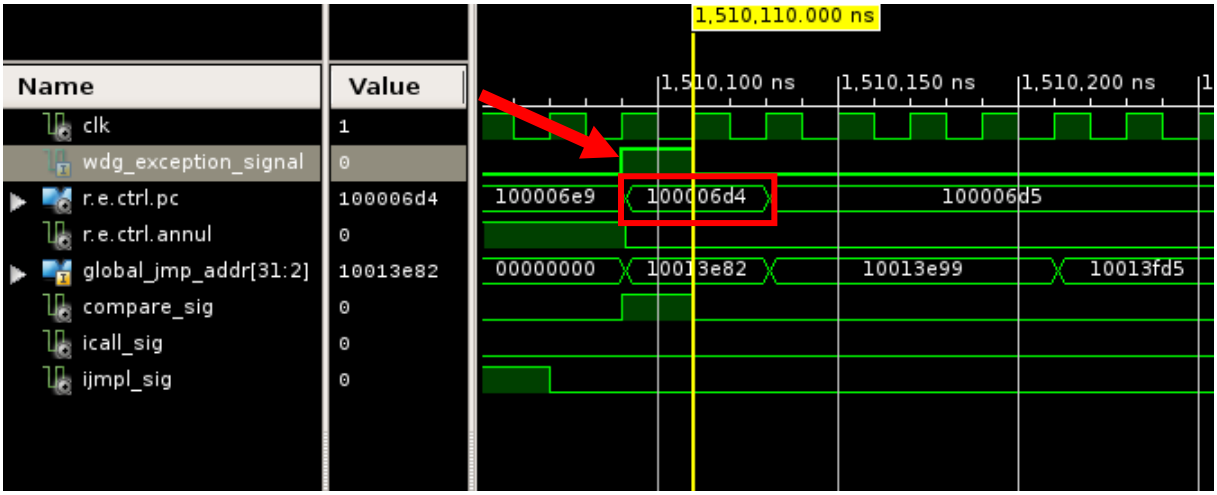
| Main call instruction address(hex) | Instruction | Safe Point |
|---|---|---|
| 40001eb4 | call 40002018 | 40002018 |
| 40001ebc | call 40001b50 | 40001b50 |

Reference: Segabinazzi (2016)

The Figure 28 shows when this benchmark is running and when the exception signal is raised. As could be see, when the *wdg_exception_signal* is raised the execution PC (r.e.ctrl.pc) gets the 0x100006d4 address, adjusting this address to have 32 bits by making a double left shift (2<<0x100006d4) the result get is 0x40001b50. This is one of the safe point addresses listed above. After that, the

system continues to run normally. So, as a conclusion, the Recovery Mechanism could rollback the system to the safe point successfully.

Figure 28 –Benchmark Edbrowse successfully recovered.




Reference: Segabinazzi (2016)

The Table 7 shows the results get from all benchmarks evaluation. As could be see, the Recovery Mechanism successful recovers the most part of benchmarks.

Table 7 – Recovery result when evaluating the Recovery Mechanism under the benchmarks implementations.

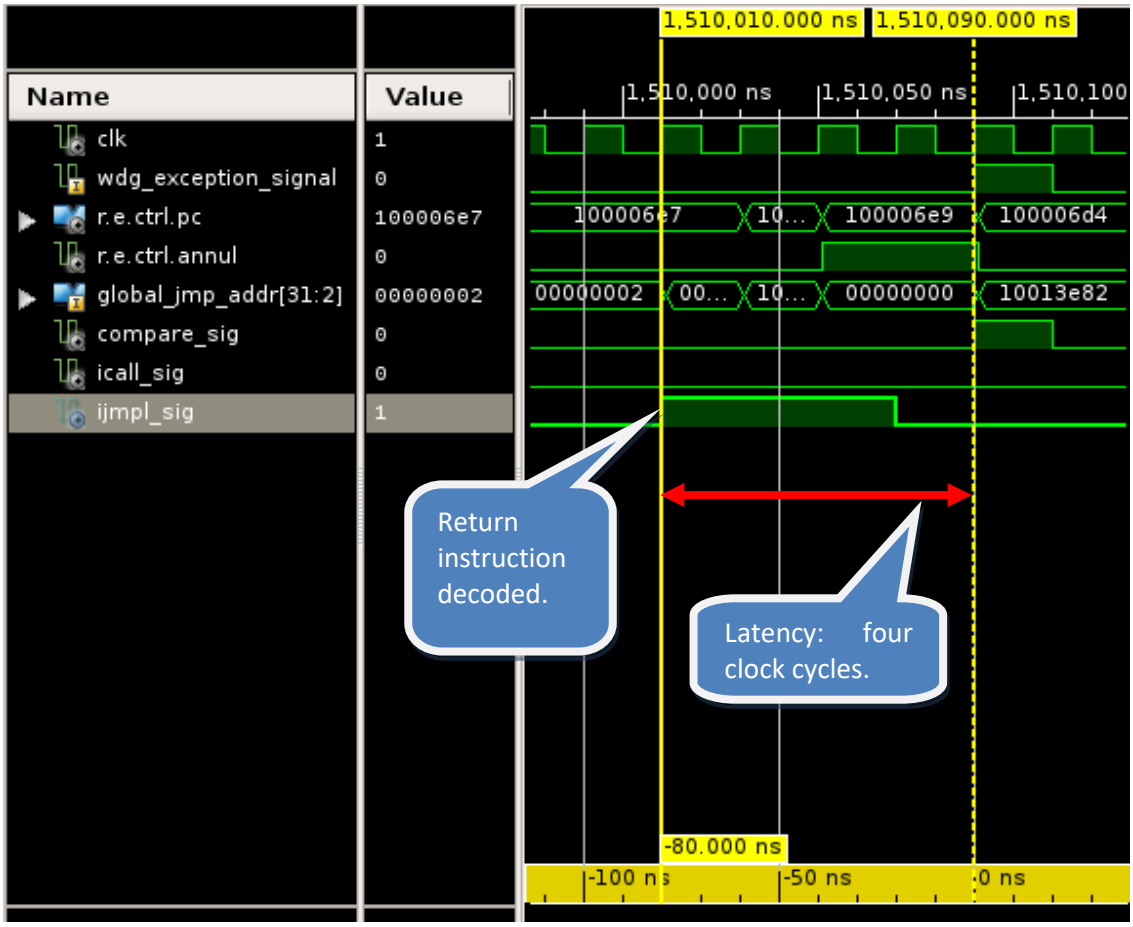
| Vulnerable Programs | CVE Number | Recovery Result |
|---|---|---|
| Edbrowse | CVE-2006-6909 | Successfully recovered |
| MADWiFi | CVE-2006-6332 | Successfully recovered |
| Samba | CVE-2007-0453 | Successfully recovered |
| Sendmail | CVE-2003-0681 | Successfully recovered |
| Wu-ftpd | CVE-1999-0368 | Successfully recovered |
| Wu-ftpd | CVE-2003-0466 | Successfully recovered |

Reference: Segabinazzi (2016)

## 5.2. DETECTION AND RECOVERY LATENCY

Additionally, the attack detection latency of the Watchdog was checked too. More precisely, the time between the Watchdog detects a return instruction (JMPL) by raising the signal *ijmpl* and the instant at which the Watchdog generate the exception signal by raising the *wdg_exception_signal* was measured. *A*s could be seen in the Figure 29 the measured latency was **four clock cycles**.

Figure 29 – Watchdog return address overwrite detection latency to generate the exception signal from a function return instruction decoded.




Reference: Segabinazzi (2016)

Finally, as can be observed in the figures above, when this approach are under simulation the Recovery Mechanism does not take any extra time to execute and recover the system. However, this behavior is not real, and when the system is under real situations, others than simulation, the Recovery Mechanism will take extra clock cycles to work. So, this implementation will introduce execution time overhead in the situations at follows:

- When the system returns to main function after a function call: the Recovery Mechanism will commit all data contained in the Recovery_mem memory block to Secondary_mem memory block. So, for every committed data, an execution overhead will be introduced by a number of clock cycles to write the data in the memory block. Therefore, the total amount of execution time overhead incurred by this action will depends on the total amount of data to commit.
- When a return address overwritten was detected: by recognizing this situation the Recovery Mechanism should overwrite all data in the main memory by the data contained in the Secondary_mem memory. Again, extra clock cycles will be necessary to commit these data. And, the execution time overhead will depends on the amount of data do commit too.

Finally, as can be noted in the Annex B, the pipeline internal registers and general registers data structures are duplicated, so, the state of these register are saved in these duplicated structures right at the local entity where the original registers are declared too. Therefore, to save the states of these original core registers and to commit data back to these registers the Recovery Mechanism does not take extra clock cycles, and this action will be performed in just one clock cycle.

## 5.3. AREA OVERHEAD

To calculate the area overhead, the vhdl code implemented in this work was synthesize and mapped to a Virtex-4 (XC4VFX12-10SF363) FPGA. In a general way, the number of Flip Flops and LUTs are illustrated, but additionally, the number of LUTRAMs used in the implementations is illustrated too. The LUTRAMs item shows the number of LUTS used as memory mapped by the ISE Design Framework as DRAMs. So, the total amount of LUTs used in implementation (DRAM + Others) is illustrated by LUTs, and, as additional information, the number of LUTs used as DRAMs are illustrated in the item LUTRAMs.

### 5.3.1. Watchdog Area Overhead

The Table 8 and Table 9 show the area overhead added by the Watchdog implementation. This table depicts results for two different implementations of the Watchdog according to its ability to monitor and capacity to store return addresses from nested function calls: in the first implementation, the Watchdog is able to handle 256 nested function calls, in other words, 256 return addresses, while in the second implementation it supports the monitoring and storage of 64 return addresses.

Table 8 - Area overhead yielded by the Watchdog implementation.

| **Return Address Capacity** | **Prymitive Type** | **Leon + Watchdog** | **Watchdog entity** | **Area Overhead** |
|---|---|---|---|---|
| **256** | Flip Flops | 2063 | 118 | 6% |
| | LUTs | 6817 | 895 | 13% |
| | **Total** | **8882** | **1013** | **11%** |
| **64** | Flip Flops | 2048 | 110 | 5% |
| | LUTs | 6349 | 394 | 6% |
| | **Total** | **8399** | **504** | **6%** |

Reference: Segabinazzi (2016)

Table 9 – Number of LUTS used as memory, mapped by ISE Design framework as DRAMS.

| **Return Address Capacity** | **Prymitive Type** | **Leon + Watchdog** | **Watchdog entity** | **Area Overhead** |
|---|---|---|---|---|
| **256** | LUTRAMs | 502 | 480 | 96% |
| **64** | | 142 | 120 | 85% |

Reference: Segabinazzi (2016)

The methodology proposed by (KANUPARTHI, KARRI, *et al.*, 2012) shows an area overhead of 4,25%. However, the memory area used to save some hashes was not been considered when this overhead was calculated.

So, been in mind these overheads above, the Watchdog proposed by this approach has a similar area overhead in comparison with other techniques.

### 5.3.2.Recovery mechanism area overhead

The area overhead incurred by the Recovery Mechanism was related to memory. As it is explained in the sections above, the standard Recovery Mechanism generates safe points from function calls originated in main function. In this situation the memory overhead is:

<total amount of memory> = <system memory capacity> x 3

Also, the quantity of logic blocks used and the area overhead incurred by the other recovery blocks such as Recovery_ctrl, Reg_recovery_ctrl and the Reg_recovery_mem blocks, are showed in the Table 10.

Table 10 - Logic blocks utilization and the area overhead incurred by the Recovery Mechanism blocks.

| **Prymitive Type** | | **Leon3** | **Leon3 + recovery blocks** | **Area Overhead** |
|---|---|---|---|---|
| Flip Flops | | 1938 | 2890 | 49% |
| LUTs | Total | 5955 | 7183 | 20% |
| | LUTRAMs | 22 | 10 | -54% |
| **Total** | | **7895** | **10075** | **27%** |

Reference: Segabinazzi (2016)

## 5.4. OVERALL OVERHEAD RESULT

Finally, the accumulated overhead incurred by the Watchdog and the Recovery Mechanism together are shown in the Table 11.

Table 11 – Accumulated overhead incurred by the two main approaches proposed by this dissertation.

| **Approaches** | **Area Overhead** |
|---|---|
| **Watchdog (capacity: 64)** | 6% |
| **Recovery mechanism** | 27% |
| **Total** | **33%** |

Reference: Segabinazzi (2016)

The number of equivalent gates was not mentioned in this evaluation because the equivalent gates estimation was removed from ISE Xilinx mapper report, since the estimation done in earlier versions was only an oversimplification (XILINX INC., 2016).

# 6. CONCLUSIONS

This paper presents a new Dynamic Integrity Checking technique based on a Watchdog and a Recovery Mechanism implemented in hardware. The Watchdog observes specific instructions in the code being executed through the processor pipeline, compares them against reference values generated at runtime and in the event of detecting a tentative of intrusion, the pipeline is stalled and the instructions are not allowed to commit by flushing them from the pipe, finally the Watchdog rises an exception signal that trigger the recovery process that should initiated by the Recovery Mechanism. The attack type treated in this work is Stack Smashing Buffer Overflow. Compared to the existing approaches found in the literature the advantages of this work are listed as follows: (a) Does not need application code recompilation; (b) It is not based on any software component; to recognize the return address corruption the Watchdog has (c) no performance degradation, (d) an acceptable area overhead; and (d) a low attack detection latency.

Experimental results obtained throughout simulations of test programs that were implemented with pieces of known vulnerable C codes obtained from vulnerable test benchmarks published in the CVE demonstrate that the technique is very efficient: so far, the totality of the simulated intrusions were detected and recovered by the Watchdog en the Recovery Mechanism. Furthermore, area overhead was measured against the implementation of a system based on the LEON3 softcore processor plus this approach, both described in VHDL and mapped into a Xilinx Virtex-4 FPGA. In this scenario, the observed area overhead was around of 6% of the LEON3 processor. Finally, the attack detection latency of the Watchdog was measured to be very low: 4 clock cycles.

## 6.1. FUTURE WORK

In this section some possibilities to improve the work proposed by this dissertation are exposed.

### 6.1.1. Recovery mechanism – improvement possibility

As explained in the section 3.2 the Recovery Mechanism uses two levels of extra memory and a register recovery memory to rollback the program execution to a function call in the main function.

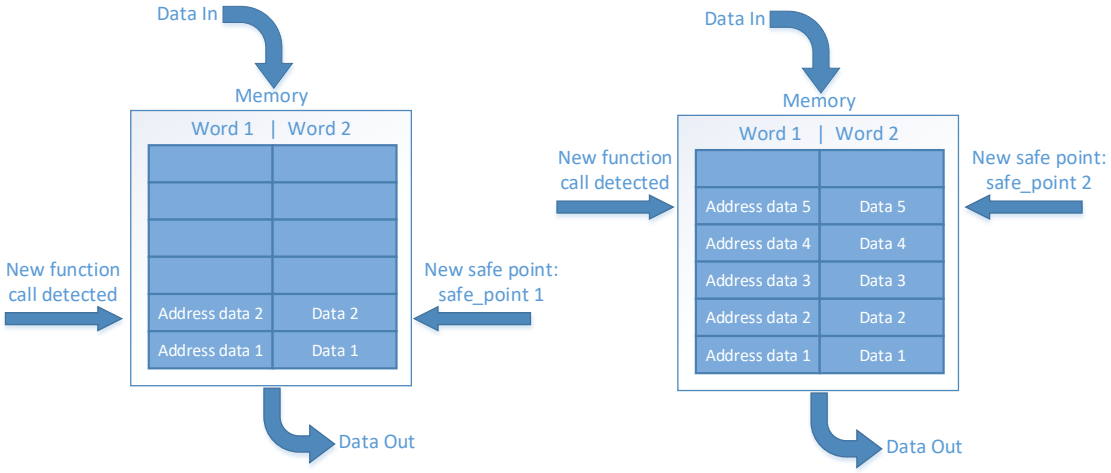
Also, an improvement to this mechanism could be done to support the rollback to other nested functions. This improvement could be done my marking points on the Recovery_mem memory block as new safe points when a new function call instruction is decoded (Figure 30(a) illustrate this behavior). So, when system returns from a function call the recovery memory continues to be used, and the last safe point created should be cleaned. The recovery memory will be committed in the secondary memory just when the execution returns to main function.

However, to support this improvement, new levels of register recovery memories should be included as more levels of nested function the mechanism support, additionally, a new memory block should be included to save the safe-points (Figure 30(b)).

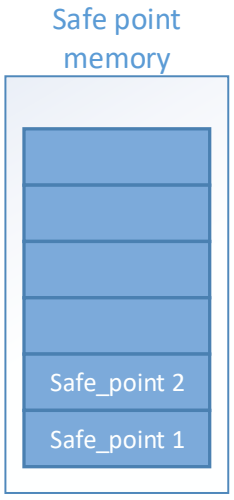
Register_recovery_mem_levels = 1 + <nº of nested functions to support>

When an attack situation is detected the secondary memory should be committed to main memory, and after, the recovery memory should be committed direct to main memory too, but only the data saved before the last safe point should be committed and the remaining modifications after safe point should be discarded. Finally, data in the Reg_recovery_mem should be committed to original registers and after that, the system will run again normally, starting from the last safe point.

Figure 30 - The safe pointing system (a) the recovery memory when new safe points were detected and (b) the safe point memory block with these new safe points.




(a)

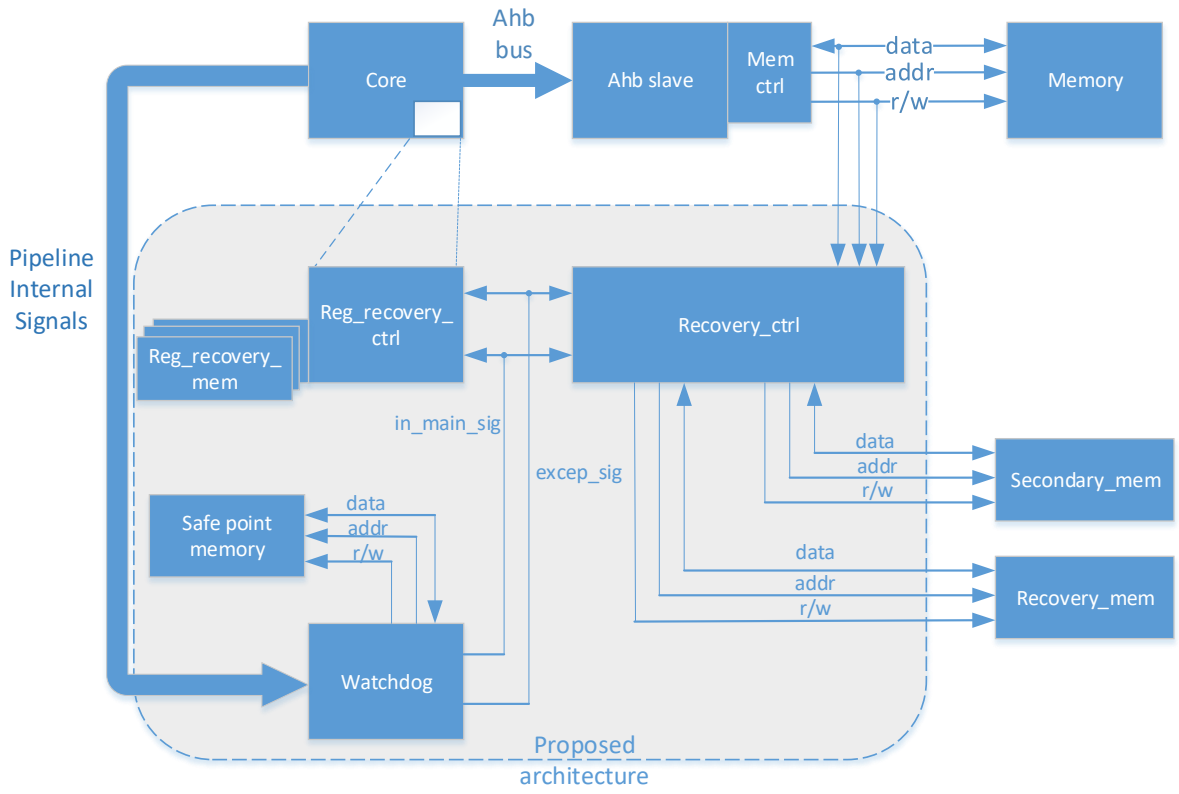



(b)

Reference: Segabinazzi (2016)

Finally, the Figure 31 shows the new architecture required for this improvement implementation.

Figure 31 – The new Recovery Mechanism architecture proposed by this improvement.




Reference: Segabinazzi (2016)

## 6.2. DISCUSSIONS

It is also worth discussing three important points: (1) the applicability of the proposed approach to different processor architectures, (2) the detection coverage of the Watchdog and, finally, (3) the Recovery Mechanism applicability.

### 6.2.1. Proposed approach applicability to different processor architectures

Concerning this first issue, the approach is easily adapted to any kind of open-source soft-core processor, considered that the four signals described in Section III can be retrieved (“OpCode”, “annul”, “PC” and “jmp_addr”). Concerning Commercial off-the-shelf (COTS) processors (for instance x86, PowerPC and ARM), it is more difficult to monitor pipeline internal signals. So, our solution can be extended to processors that use any instruction to return from functions, for example *ret*, *jmpl* or any others instructions in a situation that the Watchdog could use to trigger as a function call. So, the Watchdog just needs a way to differ these instructions from any others in the processor instruction set.

Moreover, assume, for instance, that a processor use a link register to save the current PC as a return address in function call situations. In this situation the PC is not stacked but the current data contained in the linker register is. However, the Watchdog get the return address direct from the PC (as explained in section 3.2) in the right moment when the instruction that calls the function is being executed. So, from the Watchdog point of view, it does not matter if this data will be saved first in the link register and then moved to the stack or saved directly from the PC to the stack.

### 6.2.2. Discussing about the Watchdog detection coverage

Considering the detection coverage, the Watchdog is not capable to analyze the code contained in a function that is being called. Also, there are programs that use indirect function calls, this is an specific situation where a pointer is used to call a function. However this pointer could be saved in the stack and be modified in run time by a buffer overflow in the stack. Therefore, the Watchdog will not detect if the

pointer used to call some function was modified in a malicious way, and the execution could be branched to an injected code without any action from Watchdog. Nevertheless, note that the proposed Watchdog is assumed to be used in critical applications, which by nature do not use function pointers in order to satisfy safe software design practices (O'CONNOR, 2004).

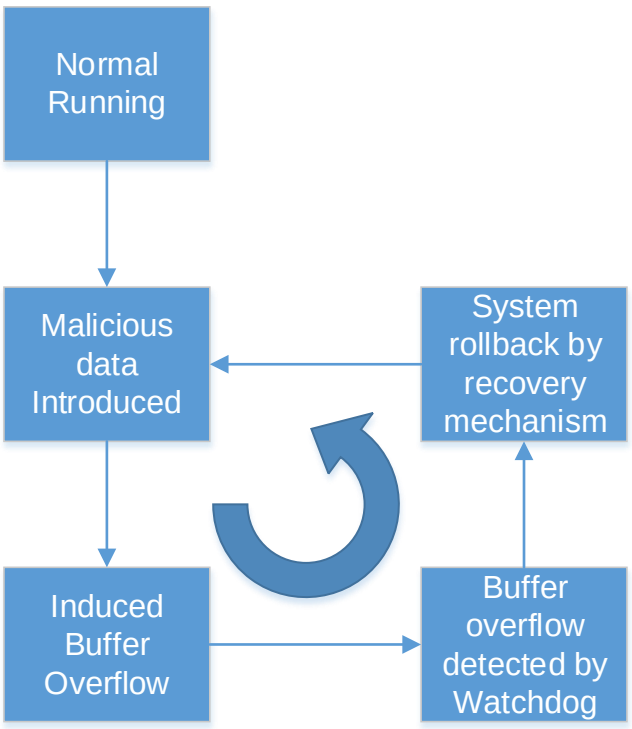
### 6.2.3. The Recovery Mechanism applicability

Making some analysis over the methodology proposed by this work, the Recovery Mechanism together with Watchdog could open a possibility to attackers drive a Distributed Denial of Service (DDoS) attack by forcing some buffer overflow situation in the system.

During DDoS, attackers send out targeted commands to applications to tax the central processing unit (CPU) and memory and make the application unavailable (Gartner Newsroom, Announcements, Gartner Says 25 Percent of Distributed Denial of Services Attacks in 2013 Will Be Application-Based , 2013).

The Figure 32 shows a flowchart of a tentative of DDoS attack in the system where the Watchdog and the Recovery Mechanism are running. In this situation the Watchdog will detect the buffer overflow and the recovery will rollback the system to the point where some malicious code could be inserted again and so on.

Figure 32 – Flowchart to describe a DDoS attack in the system where the Watchdog and the Recovery Mechanism are running.




Reference: Segabinazzi (2016)

Having this issue in mind, and an infinity of vulnerability situation found in programs source codes, there will be situations where the best action to take, after a buffer overflow detection, will be the overall system reset. So, in these situations, an extra verification should be included in the approach proposed by this dissertation. An especial logic to verify how many times the Recovery Mechanism rollback to the same safe point, and if it happens more than 2 or 3 times, the system should be overall restarted.

# REFERENCES

AEROFLEX GAISLER AB. **GRLIB IP Core User's Manual**. 1.3.7. ed. [S.l.]: [s.n.], 2014.

CERT, Vulnerability Database. **CERT, Vulnerability Database**. Disponivel em: <http://>. Acesso em: March 2015.

CHEN, G. et al. **SafeStack:** Automatically Patching StackBased Buffer Overflow Vulnerabilitie. IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING. [S.l.]: IEEE Computer Society. 2013. p. 368-379.

COBHAM GAISLER AB. Gaisler: Leon3 Processor. **Gaisler**, 2015. Disponivel em: <http://gaisler.com/index.php/products/processors/leon3>. Acesso em: 23 Jan 2016.

COMMON Vulnerabilities and Exposures - The Standard for Information Security Vulnerability Names. **Common Vulnerabilities and Exposures**, 2015. Disponivel em: <https://cve.mitre.org/>. Acesso em: May 2015.

CORLISS, M. L.; LEWIS, E. C.; ROTH, A. **Using DISE to Protect Return Addresses from Attack**. Workshop on Architectural Support for Security and Anti-Virus (WASSA). Philadelphia: [s.n.]. 2004.

COWAN, C. et al. **StackGuard:** Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proceedings of the 7th USENIX Security Symposium. San Antonio, Texas: [s.n.]. 1998.

COWAN, C. et al. **Protecting Systems from Stack Smashing Attacks with StackGuard**. 5th Linux Expo. Raleigh: [s.n.]. 1999.

DAS, S.; WEI, Z.; LIU, Y. **Reconfigurable Dynamic Trusted Platform Module for Control Flow Checking**. IEEE Computer Society Annual Symposium on VLSI. Tampa, FL: IEEE Computer Society. 2014. p. 166-171.

DU, J.; MAI, J. **A New Approach against Stack Overrun:** Separates the stack to two parts. Int. Conf. on Instrumentation, Measurement, Computer, Communication and Control. [S.l.]: [s.n.]. 2011. p. 441-444.

GARTNER Newsroom, Announcements, Gartner Says 25 Percent of Distributed Denial of Services Attacks in 2013 Will Be Application-Based. **Gartner**, 2013. Disponivel em: <http://www.gartner.com/newsroom/id/2344217>. Acesso em: 20 jan. 2016.

HERRMANN, D. S. Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors. In: HERRMANN, D. S. **Software Safety and Reliability**. [S.l.]: Wiley-IEEE Computer Society, 2000. Cap. 02, p. 13-31.

KANUPARTHI, A. K. et al. **A High-Performance, Low-Overhead Microarchitecture for Secure Program Executio**. IEEE International Conference on Computer Design (ICCD). [S.l.]: [s.n.]. 2012. p. 102-107.

KANUPARTHI, A. K.; ZAHRAN, M.; KARRI, R. Architecture Support for Dynamic Integrity Checking. **IEEE Transactions on Information Forensics and Security (TIFS)**, v. 7, n. 1, p. 321-332, Feb. 2012.

KAO, W.-F.; WU, S. F. **Light-weight Hardware Return Address and Stack Frame Tracking to Prevent Function Return Address Attack**. International Conference on Computer Science and Engineering. [S.l.]: [s.n.]. 2009. p. 859-866.

KC, G. S.; KEROMYTIS, A. D.; PREVELAKIS, V. **Countering Code-Injection Attacks With Instruction-Set Randomization**. 10th ACM Conference on Computer and Comunications. Washington: [s.n.]. 2003.

KNIGHT, J. C. **Safety Critical Systems:** Challenges and Directions. Proceedings of the 24th International Conference on software Engineering (ICSE). Orlando: [s.n.]. 2002. p. 547-550.

KU, K. et al. A buffer overflow benchmark for software model checkers. **Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering** , New York, 2007. 389-392.

KUMAR, K. S.; KISORE, N. R. **Protection against buffer overflow attacks through runtime memory layout randomization.** 2014 13th International Conference on Information Technology. Bhubaneswar, India: IEEE Computer Society. 2014. p. 184-189.

LEE, R. B. et al. Enlisting Hardware Architecture to Thwart Malicious Code Injection, 2004.

LUTZ, R. R. **Software Engineering for Safety:** A Roadmap. Proceedings of the Conference on The Future of Software Engineering. Limerick: [s.n.]. 2000. p. 213-226.

MCGRAW, G. Software Security. **IEEE Security & Privacy**, v. 2, n. 2, p. 80-83, 2004.

MILLER, B. P. et al. **Fuzz Revisited: A Reexamination of the Reliability of UNIX Utilities and Services**. University of Wisconsin. Madison, p. 23. 1995.

MILLER, B. P.; FREDRIKSEN, L.; SO, B. An Empirical Study of the Reliability of UNIX Utilities. **Comunications of the ACM**, v. 33, n. 12, p. 33-44, Dec 1990.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY - NIST. National Vulnerability Database - Home. **National Vulnerability Database - NVD**, 2016. Disponivel em: <https://nvd.nist.gov/home.cfm>. Acesso em: 30 July 2016.

NATIONAL Vulnerability Database - Vulnerability search. **National Vulnerability Database**. Disponivel em: <https://web.nvd.nist.gov/view/vuln/search>.

NUNES, F. J. B.; BELCHIOR, A. D.; ALBUGUERQUE, A. B. **Security Engineering Approach to Support Software Security**. IEEE 6th World Congress on Services. Miami: [s.n.]. 2010. p. 48-55.

O'CONNOR, B. **NASA Software Safety Guidebook**. National Aeronautics and Space Administration. [S.l.], p. 389. 2004.

OZDOGANOLU, H. et al. SmashGuard - A hardware solution to prevent security Attacks on the Functions Return Address. **IEEE Trans. on Computers**, v. 55, n. 10, p. 1271-1285, Oct. 2006.

PARK, Y.-J.; ZHANG, Z.; LEE, G. Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks. **IEEE Micro**, v. 26, n. 4, July-Aug 2006.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design:** Tthe Hardware/Software Interface. 4ª edition. ed. [S.l.]: Elsevier, 2012.

RAGO, W. R. S. A. S. A. **Advanced Programing**. Second. ed. [S.l.]: [s.n.], 2010.

SCHMID, D. C. Adaptive Middleware: Middleware for Real-time and Embedded Systems. **Comunications of the ACM**, v. 45, n. 6, p. 43-48, June 2002.

SHUETTE, M. A.; SHEN, J. P. Processor Control Flow Monitoring Using Signatured Instruction Streams. **IEEE Transactions on Computers**, v. C-36, n. 3, p. 264-276, March 1987.

SPARC INTERNATIONAL INC. **The SPARC Architecture Manual:** Version 8. Upper Saddle River, NJ, USA: Prentice-Hal, 1992.

STEVENS, W. R.; RAGO, S. A. **Advanced Programming in the UNIX Environment**. 2nd Edition. ed. [S.l.]: Pearson, 2012.

SWARUP, M. B.; RAMAIAH, P. S. **An Approach To Modeling Software Safety**. Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD '08. [S.l.]: [s.n.]. 2008. p. 800-806.

XILINX - ISE WebPACK Design Software. **Xilinx**, 2016. Disponivel em: <http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html>. Acesso em: 23 Jan. 2016.

XILINX - ISIM Simulator. **Xilinx**, 2016. Disponivel em: <http://www.xilinx.com/products/design-tools/isim.html>. Acesso em: 23 Jan 2016.

XILINX INC. XILINX Archived ISE issues. **XILINX Community Forums**, 2016. Disponivel em: <https://forums.xilinx.com/t5/Archived-ISE-issues-Archived/How-to-display-quot-Total-Equivalent-Gate-Count-for-Design-quot/td-p/21395>. Acesso em: 15 out. 2016.

# ANNEX A

The VHDL code below shown illustrates the Watchdog internal blocks implementation.

```
------------------------------------------------------------------
-- Watchdog Entity
------------------------------------------------------------------
entity watchdog is
  generic (
    pclow    : integer range 0 to 2 := 2
    );
  port (
    clk   : in  std_ulogic;
    rstn  : in  std_ulogic;
    g_inst  : in  std_logic_vector(31 downto 0);
    g_pc    : in  std_logic_vector(31 downto PCLOW);
    g_annul : in  std_logic;
    global_jmp_addr  : in  std_logic_vector(31 downto PCLOW);
    exception_signal : out std_logic;
    in_the_main_signal : out std_logic := '1'
    );
end;

...

------------------------------------------------------------------
-- ShadowMem declaration
------------------------------------------------------------------
  type   data_vector   is   array   (0   to   (64   -   1))   of
     std_logic_vector (31 downto PCLOW);
  signal shadow_mem : data_vector;
  signal ssp : integer range 0 to (64 - 1) := 128;
```

```
...

-------------------------------------------------------------
-- Decision Block Recovery Mechanism
-- In this block we generate the signals to recovery mechnism
-------------------------------------------------------------
          if(pc & "00" > main_low and pc & "00" < main_high )
     then
               in_the_main_signal <= '1';
               main_first := '1';
          -- just put down in_the_main_signal if the
          -- execution just reach the main function
          -- for the first time.
          elsif (main_first = '1') then
               in_the_main_signal <= '0';
          else
          end if;

...

-------------------------------------------------------------
-- Decision Block
-- This block receive the compare signal from ShadowMem
-- Control and check if jmp_addr and the shadow_mem data
-- are iquals.
-------------------------------------------------------------
          if (compare = '1') then
               -- comparing jmp_addr and shadow_mem data
               if(shadow_mem(ssp)+2 /= local_jmp_addr)then
                    -- rising up exception signal
                    exception_signal <= '1';
               end if;
               -- cleanup compare signal
               compare := '0';
```

```
          else
          end if;

...

-------------------------------------------------------------
-- ShadowMem Control
-- This block receive signal icall or ijmpl from Instruction
-- decoder and generate the compare signal as necessary
-------------------------------------------------------------
          -- FSM to actualize the SSP and generate the compare
     signal
          case (commit) is
          when 1 => --Call
               ssp <= ssp+1;
               commit := 0;
          when 2 => -- return
               ssp <= ssp -1;
               commit := 0;
               compare := '1';
          when others => dummy := '0';
          end case;

          -- checking icall signal
          if (icall = '1' and pc /= old_pc) then
               icall := '0';
               shadow_mem(ssp) <= old_pc;
               commit := 1;
               old_pc := pc;
          else
          end if;

          -- checking the ijmpl signal
          if (ijmpl = '1' and pc /= old_pc) then
```

```
                ijmpl := '0';
                commit := 2;
           else
           end if;

...

-------------------------------------------------------------
-- Instruction decoder
-- This block decode the instruction get from pc, decode
-- this instruction and generate icall or ijmpl to
-- ShadowMem controller.
-------------------------------------------------------------
           case op is
           -- decoding call instruction
           when CALL =>
                if(g_annul = '0') then
                     icall := '1';
                     old_pc := pc;
                end if;

           when FMT3 =>
                case op3 is
                -- decoding jmpl instruction (return)
                when JMPL => --pg 124 sparcV8 manual
                     if(g_inst(29 downto 25) = "00000" and
                         g_inst(13) = '1' and
                         g_annul = '0') then -- .rd = 0
                          ijmpl := '1';
                          local_inst := g_inst;
                          local_jmp_addr := global_jmp_addr;
                          old_pc := pc;
                     end if;
```

```
                -- another situation to decode call
instruction
                if(g_inst(29 downto 25) = "01111" and --rd
= 15 indirect call
                    g_annul = '0') then
                     icall := '1';
                     old_pc := pc;
                end if;
           when others => dummy := '1';
           end case;
     when others => dummy := '0';
     end case;
```

# ANNEX B

Below are shown pieces of commented VHDL code describing the reg_recovery_ctrl logic to generate and recovery from the safe-point.

```
-------------------------------------------------------------
-- Internal pipeline registers declaration
-------------------------------------------------------------
  type registers is record
    f   : fetch_reg_type;
    d   : decode_reg_type;
    a   : regacc_reg_type;
    e   : execute_reg_type;
    m   : memory_reg_type;
    x   : exception_reg_type;
    w   : write_reg_type;
  end record;

  signal r, rin : registers;
  signal my_rin : registers;  -- duplicate pipeline registers

...

-------------------------------------------------------------
-- Register recovery controller (Reg_recovery_ctrl)
-- Process to generate safe point when function goes
-- out of main function.
-------------------------------------------------------------
my_proc : process (clk, wdg_in_the_main_signal, r)
 variable local_ctrl : std_logic := '0';
 begin
   if rising_edge(clk) then

   if(wdg_in_the_main_signal = '0' and local_ctrl = '0') then
```

```
     my_rin <= r;
     local_ctrl := '1';
   elsif (wdg_in_the_main_signal = '1') then
       local_ctrl := '0';
   else
   end if;
 end if;
end process;

...

----------------------------------------------------------
-- Original process from Leon3 pipeline, this process
-- actualize the pipeline registers
----------------------------------------------------------
  reg : process (clk, wdg_exception_signal, my_rin)
  begin
    if rising_edge(clk) then
     -- logic introduced to recovery to the safe-point saved
     -- in the my_rin register structure.
    if(wdg_exception_signal = '1') then
         r <= my_rin;
    else

      if (holdn = '1') then
        r <= rin;
      else
        r.x.ipend <= rin.x.ipend;
        r.m.werr <= rin.m.werr;
        if (holdn or ico.mds) = '0' then
          r.d.inst <= rin.d.inst; r.d.mexc <= rin.d.mexc;
          r.d.set <= rin.d.set;
        end if;
        if (holdn or dco.mds) = '0' then
```

```
          r.x.data <= rin.x.data; r.x.mexc <= rin.x.mexc;
          r.x.set <= rin.x.set;
        end if;
      end if;
      if rstn = '0' then
        if RESET_ALL then
          r <= RRES;
          if DYNRST then
            r.f.pc(31 downto 12) <= irqi.rstvec;
            r.w.s.tba <= irqi.rstvec;
          end if;
          if DBGUNIT then
            if (dbgi.dsuen and dbgi.dbreak) = '1' then
              r.x.rstate <= dsu1; r.x.debug <= '1';
            end if;
          end if;
          if (index /= 0) and irqi.run = '0' then
            r.x.rstate <= dsu1;
          end if;
        else
          r.w.s.s <= '1'; r.w.s.ps <= '1';
          if need_extra_sync_reset(fabtech) /= 0 then
            r.d.inst <= (others => (others => '0'));
            r.x.mexc <= '0';
          end if;
        end if;
      end if;
--    end if; --exception signal
    end if;
  end process;


----------------------------------------------------------
-- The general registers declaration and the duplicated
```

```
-- structure.
-------------------------------------------------------------

  type mem is array(0 to numregs-1)
     of std_logic_vector((dbits -1) downto 0);
  signal memarr : mem;
  signal my_memarr: mem; -- duplicated general registers
...

-------------------------------------------------------------
-- Process to save general registers when system goes
-- out of main
-------------------------------------------------------------

 my_proc : process(wclk, wdg_in_the_main_signal, memarr)
   variable local_ctrl : std_logic := '0';
 begin
   if rising_edge(wclk) then
     if(wdg_in_the_main_signal  =  '0'  and  local_ctrl  =  '0')
     then
     my_memarr <= memarr;
     local_ctrl := '1';
     elsif (wdg_in_the_main_signal = '1') then
       local_ctrl := '0';
     else
     end if;
   end if;
 end process;

-------------------------------------------------------------
-- Original process from Leon3, this process
-- actualize the general registers
-------------------------------------------------------------
```

```
  main : process(wclk, wdg_exception_signal, my_memarr)
  begin

    if rising_edge(wclk) then
    -- commit saved state registers to original core registers
    -- when exception_signal is raised
    if (wdg_exception_signal = '1')  then
      memarr <= my_memarr;
    else
      din <= wdata; wr <= we;
      if (we = '1')
-- pragma translate_off
     and (conv_integer(waddr) < numregs)
-- pragma translate_on
      then wa <= waddr; end if;
      if (re1 = '1')
-- pragma translate_off
     and (conv_integer(raddr1) < numregs)
-- pragma translate_on
      then ra1 <= raddr1; end if;
      if (re2 = '1')
-- pragma translate_off
     and (conv_integer(raddr2) < numregs)
-- pragma translate_on
      then ra2 <= raddr2; end if;
      if wr = '1' then
        memarr(conv_integer(wa)) <= din;
      end if;
    end if; --exception_signal
    end if; --wclk
  end process;
```