

SECURE RISCV DESIGN FOR SIDE-CHANNEL EVALUATION PLATFORM

by

Bhavin Thakar

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2021

Approved by:

Dr. Fareena Saqib

Dr. Arindam Mukherjee

Dr. Tao Han

ABSTRACT

BHAVIN THAKAR. SECURE RISCV DESIGN FOR SIDE-CHANNEL EVALUATION PLATFORM. (Under the direction of DR. FAREENA SAQIB)

In the era of smart technology, IoT technology has been an integral part of the system. IoT systems are susceptible to many type of attacks such as buffer overflow attack and execution flow attacks. Information flow tracking is a technique to monitor the control flow of the program and mitigate the buffer overflow attack. The open source architecture has several applications in design and simulations of security applications such as Common Evaluation Platform(CEP). CEP is a RISC-V based simulation framework for side channel analysis to analyze power traces. These application rely on secure execution, and can have adverse affects if the underlying framework is compromised. Our research focuses on security of RISC-V architecture and its simulation framework of security enabled RISC-V design and simulation to enable hardening the design to be resilient to hardware attacks and capability of run time detection of any attacks. CEP version 1.2 is based on RISC-V ISA, so focus of this thesis is to develop the software simulation model of ISA Level Information Flow Tracking on RISC-V ISA which can be used as a parallel tool to evaluate these new security extension without the need of the hardware to test it. The steps discussed about Assembler modification in this thesis can be used for adding and deploying new instructions within the ISA. An attack setup is developed to manipulate the return address which results in the change of program control flow and also demonstrate the security extensions integrated in the simulation framework to illustrate the security extensions which can detect the attack at run-time.

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my gratitude and respect to my advisor and chair of the committee, Dr. Fareena Saqib, for her support and advice throughout my course of study. Her constant encouragement and support helped me cope with all the challenges that accompanied me. I sincerely appreciate all the efforts she has made to ensure that this thesis has the best of its ability. I am particularly grateful to the members of my thesis committee, Dr. Arindam Mukherjee and Dr. Tao Han for their valued suggestions, time and support during my thesis. I am always thankful for the constant support of my family and friends. Their sacrifice and love played a significant part in the success of this research work.

TABLE OF CONTENTS

LIST OF FIGURES	vii
LIST OF ABBREVIATIONS	viii
CHAPTER 1: INTRODUCTION	1
1.1. CONTRIBUTIONS	3
1.2. ORGANIZATION	3
CHAPTER 2: BACKGROUND	4
2.1. RISC-V ARCHITECTURE	4
2.1.1. RISC-V DEVELOPMENTS	6
2.1.2. RISC-V INSTRUCTION FORMATS	7
2.1.3. CALLING CONVENTION	8
2.2. SECURITY VULNERABILITIES OF RISC-V ARCHITECTURE	9
2.2.1. BUFFER OVERFLOW ATTACK	9
2.2.2. Fault Injection on Return Address	11
2.3. SECURITY COUNTERMEASURE OF RISC-V ARCHITECTURE	11
2.3.1. INFORMATION FLOW TRACKING	12
2.4. SOFTWARE ECOSYSTEM OF RISC-V ARCHITECTURE	14
2.4.1. RISC-V GNU TOOLCHAIN	14
2.4.2. SPIKE ISA SIMULATOR	17
2.5. COMMON EVALUATION PLATFORM	17

CHAPTER 3: PROPOSED SCHEME AND EXPERIMENTAL SETUP	19
3.1. PROPOSED SCHEME	19
3.2. EXPERIMENTAL SETUP	22
3.3. RISC-V GNU TOOLCHAIN INSTALLATION PROCESS	23
3.4. SPIKE ISA Simulator Installation Process	24
3.5. TOOLCHAIN MODIFICATION	25
3.6. SPIKE SIMULATOR MODIFICATION	26
3.7. DEMONSTRATION OF SECURITY PROPERTIES	27
3.7.1. ATTACK SETUP	27
CHAPTER 4: RESULTS AND SECURITY ANALYSIS	30
4.1. RESULTS	30
4.1.1. RETURN ADDRESS LOCATION DECRYPTION	30
4.1.2. BUFFER OVERFLOW ATTACK ON CONVENTIONAL RISC-V ARCHITECTURE	32
4.1.3. BUFFER OVERFLOW ATTACK ON MODIFIED RISC-V ARCHITECTURE	32
4.2. SECURITY ANALYSIS	33
CHAPTER 5: CONCLUSIONS AND FUTURE GOALS	35
REFERENCES	36

LIST OF FIGURES

FIGURE 2.1: RISC-V Instructions format	7
FIGURE 2.2: General stages for executing a function	8
FIGURE 2.3: Assembler mnemonics for RISC-V integer registers	9
FIGURE 2.4: Memory Structure of a Computer System	10
FIGURE 2.5: Different stages of program execution	15
FIGURE 2.6: Directory Structure for Assembler Modification	16
FIGURE 2.7: Field Format to add instruction	17
FIGURE 3.1: Load/Store Instruction format of RISC-V Architecture	20
FIGURE 3.2: E31 Core RISC-V Architecture with Tag Module	21
FIGURE 3.3: Pseudocode for Check tag and Store tag functions	27
FIGURE 3.4: Sample Attack code	28
FIGURE 3.5: Disassemble code to find the target address	29
FIGURE 4.1: Sample Code to perform attack along with its result on commandline	31
FIGURE 4.2: Result of Normal Control flow execution	32
FIGURE 4.3: Result of Buffer overflow attack execution	32
FIGURE 4.4: Result of normal control flow execution compiled with new instructions	33
FIGURE 4.5: Result of Hardware based Information Flow Tracking sys- tem to protect the system from buffer overflow attack	33

LIST OF ABBREVIATIONS

ABI Application Binary Interface

DARPA Defence Advanced Research Projects Agency

DIFT Dynamic Information Flow Tracking

ECE An acronym for Electrical and Computer Engineering.

EMTDS Ensembles of Moving Target Defenses

GDB GNU Debugger

GLIBC The GNU C Library

GLIFT Gate Level Information Flow Tracking

GSMA Global System for Mobile Communication Association

IFT Information Flow Tracking

IOT Internet of Things

RTLIFT Register Transfer Level Information Flow Tracking

SSITH Security Integrate Through Hardware and Firmware

CHAPTER 1: INTRODUCTION

Evolution of smart technology has paved a way to embed smart connectivity system in each and every device in recent years. Internet of Things (IOT), being the backbone of this technology proved to be a framework which allows communication from one device to another where each device acts as a data node which processes certain information received from another node or data retrieval from surrounding environment. IOT Technology is referred as the network of devices communicating with each other over the internet. According to Global System for Mobile Communication Association (GSMA), the number of IoT devices are predicted to increase upto 25.2 billion by 2025 which was marked about 10.3 billion in 2018 [1]. Due to its principle operation of sending and receiving data between nodes and its abundance has managed to attract many attackers to plant an attack to hijack the system. IOT systems are vulnerable to many attacks out of which control flow integrity is of major concern.

Control Flow integrity is a large family of attacks where the attacker tries to manipulate the memory address and register values during run-time to gain access to the system or change the control flow of the program. These attacks are used for bypassing software security policies or to run malicious code. In IOT embedded system, when data is transferred from one node to another allows a back door entry for the attacker to plant the attack, one of which is buffer-overflow attack. Buffer-overflow attacks occurs when a malicious data node tries to send information which exceeds the storage capacity of receiving node and then try to manipulate the memory location which the program is not allowed to access to change the control flow of the system. There are many schemes which are developed to stop this type of attack out of which one scheme is Information Flow Tracking (IFT). IFT is a technique that monitors the

flow of data received by the node and prevent the malicious data to manipulate the protected memory location.

RISC-V architecture is gaining recognition in last couple of years due to ease to use tools and big developer community [2]. The main advantage of RISC-V architecture is its free and open-source nature which attracts the developers from all over the world to contribute in this free and open-source project. It provides the flexibility to develop custom ISA by modifying or adding elements to the base ISA. RISC-V has rich software ecosystem which consists of emulators, simulators, compiler, debuggers etc. Due to all these advantages, it has encourage security engineers and researchers to incorporate security policies within the ISA.

The goal of this research is to propose a ISA level Information Flow Tracking scheme by adding instructions and tag cache module to stop a buffer-overflow attacks. This research focuses on two parts, one of which is to implement Information Flow Tracking simulation model that correlates to the hardware behaviour to provide security against buffer-overflow attack and other part is to learn the RISC-V GNU Toolchain directory structure and modify the toolchain to generate object files with newly added instructions.

The correlated hardware tool used for evaluation of this design is RISC-V that is the underlying architecture of Common Evaluation Platform [3]. Common Evaluation platform provides a set of tools to evaluate custom security extensions. This tool can be used to verify that the underlining functionality is still maintained even after modification. The goal of this research is to design the RISC-v based simulation model that can be used in Common Evaluation Platform to generate the custom ISA image along with the software simulation model to test the model without the need of hardware

For the purpose of these research, a simulation model is developed in Spike ISA Simulator with RISCV ISA(RV64I) following the latest instruction set manual docu-

ment version-2.2. Changes are made to RISC-V GNU Toolchain to generate software applications compatible with the modified ISA.

1.1 CONTRIBUTIONS

Specifically, this work makes following contributions:

- Survey of security vulnerabilities of RISC-V core and countermeasure of .
- Describes RISC-V security extensions and development a simulation model of security enriched RISC-V ISA.
- Information flow tracking integration with the RISC-V architecture
- Presents attack model to testbed to demonstrate buffer overflow and return address attack.
- Performs successful correlation between security design developed on hardware with software simulation model

1.2 ORGANIZATION

This document is organized as follows:

Chapter 2 introduces background knowledge and existing works related to topics involved in this work.

Chapter 3 discusses the proposed framework and review the steps needed to be carried out to perform experiments on proposed scheme. It also describes the buffer overflow attack setup with an example code.

Chapter 4 provides the results of each and every stage and demonstrates the difference in execution result between traditional RISC-V architecture and IFT based RISC-V architecture.

Finally, conclusion and future scope of this research is presented in Chapter 5.

CHAPTER 2: BACKGROUND

2.1 RISC-V ARCHITECTURE

RISC-V is an open source community project. It was started in the year 2010 by researchers at University of California at Berkeley. Its basic goal was to create a free and open-source ISA. The base Instruction set specification defines 32-bit and 64-bit address space variants while 128-bit variant is under development[2] [4] [5]. RISC-V base ISA consists of only base integer instruction set. There are other extensions available that can be combined with the base integer instruction set as per the purpose of operation. The extensions are Multiply and Division(M), Atomic Instructions(A), Floating Instruction(F), Double Precision Floating Point Instruction(D), Control and Status Register(Zicsr), Instruction-Fetch Fence(Zifencei), General purpose ISA(G) etc. There are various ISAs available in the market such as ARM, x86, MIPS etc. The biggest difference between this ISAs and RISC-V ISA is the cost and flexible of modification. RISC-V architecture is free and open-source ISA[2]. The authors of RISC-V Reader: An Open Architecture Atlas [6] has defined 7 metrics on which as ISA can be described. They are as follows:

1) Cost: Cost can be defined by the yield of processors on a single wafer. It can be concluded that smaller the die, higher the yield and thus reducing the cost of manufacturing a processor. It becomes very important for ISA designers to make a ISA as simple as possible to shrink the size of processors. RISC-V is a much simpler ISA than ARM ISA. The size of die for RISC-V Rocket Processor is $0.22mm^2$ whereas it is $0.53mm^2$ for ARM-32 Cortex A5 processor. Therefore, cost of manufacturing is less for RISC-V ISA. In RISC-V, Register x0 is dedicated to 0, only one data addressing mode, no conditional execution, no complex stack instructions whereas

for ARM ISA contains conditional instruction execution. Complex Data Addressing modes.

2) Simplicity: Simplicity can be defined as the measure of complexity of instruction to execute a single operation. Simpler instruction design reduces the cost and time required for designing and verification. It also reduces the cost of documentation and also reduces the difficulty for customers to understand and develop their software applications.

3) Performance: Performance is an important factor when it is compared with the cost of manufacturing of a chip, performance can be defined as amount of time required by a processor to perform an operation. Even if simple ISA might execute more instructions per program than a complex ISA, it can make up for it by having a faster clock cycle. For example for CoreMark benchmark[6] the performance of BOOM implementation of RISC-V is 14.26 secs/program whereas ARM-32 Cortex A9 processor takes approximately 18.15 secs/program.

4) Isolation of Architecture from Implementation: Due to RISC-V's free license and open source nature, it allows machine language programmer to modify or add instructions in the ISA. This helps programmers to write correct code along with optimizing the performance of execution. For example ARM and MIPS ISA have Multiple Load/ Store instructions which cannot be modified. These instructions can improve performance for single instruction pipeline design but lack performance for multiple instruction issue pipelines.

5) Room for Growth: With the end of Moore's Law, the only route to major performance-cost improvements is to add instructions for specific areas such as deep learning, augmented reality and graphics. For this, it is important for the ISA to reserve opcode space for future improvements.

6) Program Size: The only known downside to RISC-V is its larger program size than its competing ISAs. This increases the required area on the chip required for the

program's memory. This also results in more instruction cache misses. Based on the author's analysis, [6] RISC-V32 is 6-9% larger in code size than the x86 architecture.

7) Ease of Programming, compiling and linking: RISC-V features a very simple design and an extensive software ecosystem. Thanks to its open availability and strong community support, many researchers have been able to carry out their research on RISC-V. At most, all the instruction in RISC-V takes one clock for each instruction. This reduces the burden on the system and facilitates its use in developing and optimizing the code as required.

2.1.1 RISC-V DEVELOPMENTS

RISC-V's Open Source flexibility has made it an increasingly powerful and popular chip architecture for big companies like Seagate and Western Digital Corp, Alibaba, along with government initiatives backed by the U.S. military's Defence Advanced Research Projects Agency (DARPA) [7]. NVIDIA used RISC-V architecture in its graphics processing unit (GPU) chips, even while working to acquire Arm's proprietary Arm Holdings architecture. Intel can also help accelerate the adoption of RISC-V through its expanding foundry activity that aims to manufacture chips for other enterprises based on x86, ARM or RISC-V architectures. In 2019, Alibaba reported that it had developed the XuanTie 910 processor based on the RISC-V architecture through its semiconductor subsidiary Pingtou Ge. It has also helped many startups to design custom chips without the expensive licensing fees required to use proprietary chip architectures.

This also reflects in the security sector. Security has been a major concern of RISC-V and developing security applications with the architecture's flexibility has led to various advancements and programs [8]. In 2021, U.S. Defense Advanced Research Program Agency (DARPA) program called Security Integrate Through Hardware and Firmware (SSITH), University of Michigan demonstrated their new RISC-V processor called Morpheus. Around 580 cybersecurity researchers spent 13,000 hours trying to

break into this processor, but they all failed. Morpheus consists of ensembles of moving target defenses(EMTDs) mechanism which is hardware support that randomizes the undefined variables along with the churn mechanism that rerandomizes this values to protect it from probing. [9]

2.1.2 RISC-V INSTRUCTION FORMATS

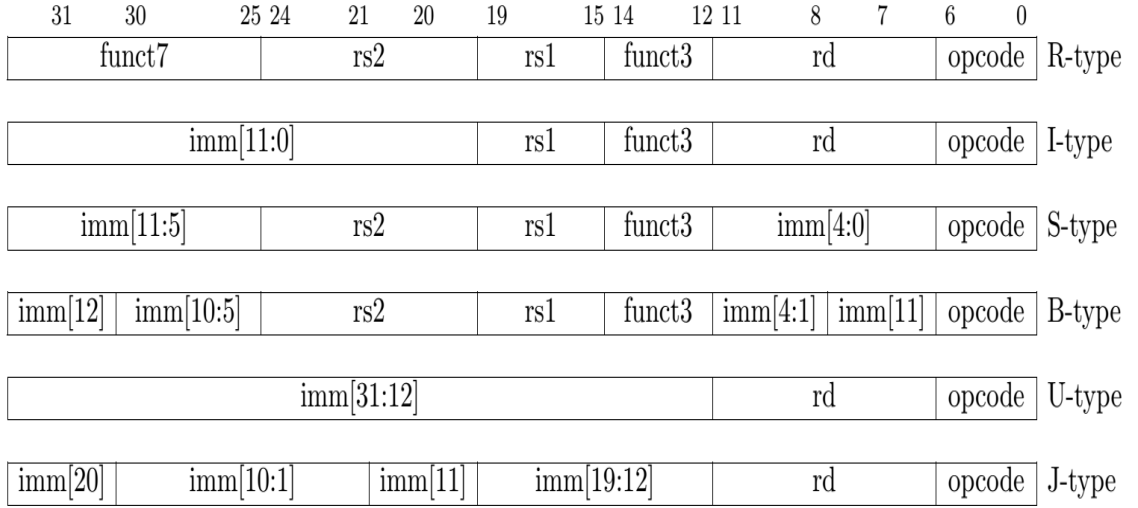


Figure 2.1: RISC-V Instructions format

Figure 2.2 shows the six base instructions format of RISC-V architecture. R-type for register-register operation, I-types for short immediate and loads, S-type for stores, B-type for conditional branches, U-type for long immediates and J-type for unconditional jumps [5] [6]. Each instruction in RISC-V architecture is 32 bit long except for Compressed Extension(RV32C, RV64C) which compresses 32 bit instructions into 16 bit long. As there are only six instruction formats and each 32 bit long simplifies the instruction decoding and hence increasing simplicity and reduces cost-performance. In comparison to x86 architecture, RISC-V ISA has three operands, rather than sharing the same field for source and destination register. In figure 2.2 rd stands for destination register, rs stands for source register, funct stands for function and imm

stands for immediate value. `funct7 + funct3` combined with opcode describe what operation to perform.

2.1.3 CALLING CONVENTION

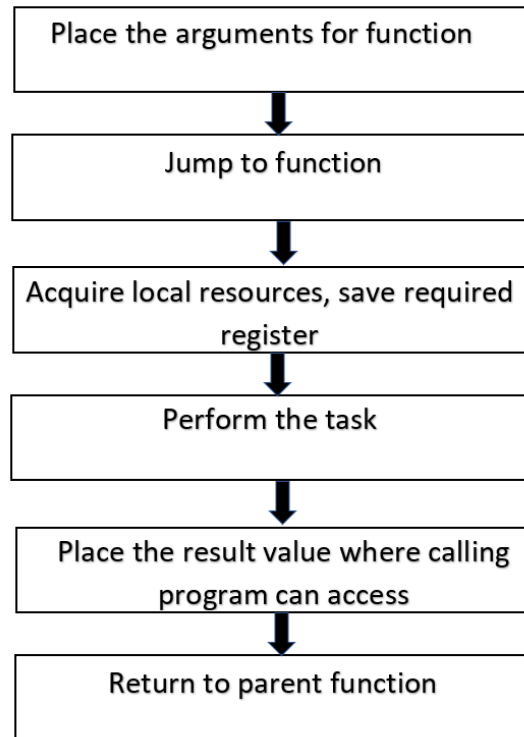


Figure 2.2: General stages for executing a function

Figure 2.3 describes the general stages when a function call is made. To obtain fast execution performance, it is important to keep all the variable in the registers to avoid going to memory frequently to save or restore these registers. Fortunately, RISC-V has enough registers to store variable in register hence reducing the time to save or restore them. It contains temporary registers to store the temporary values used for computation of a instruction and saved register to store the result between function and store important result during execution of the function. Each register in RISC-V architecture is classified in two parts: preserved and not preserved. Figure 2.3 lists the RISC-V application binary interface(ABI) names of registers with their

classification [6].

Register	ABI name	Description	Preserved across call
X0	Zero	Hard-wired zero	---
X1	ra	Return Address	No
X2	sp	Stack Pointer	Yes
X3	gp	Global Pointer	---
X4	tp	Thread Pointer	---
X5	t0	Temporary / alternate/link register	No
X6-7	t1-2	Temporary register	No
X8	s0/fp	Saved Register/frame pointer	Yes
X9	s1	Saved Register	Yes
X10-11	a0-1	Function Arguments/ return values	No
X12-17	a2-7	Function Arguments	No
X18-27	s2-s11	Saved Registers	Yes
X28-31	t3-6	Temporary register	No

Figure 2.3: Assembler mnemonics for RISC-V integer registers

2.2 SECURITY VULNERABILITIES OF RISC-V ARCHITECTURE

In the field of computer security, a vulnerability is the weakness that can be tapped by the attacker to trespass into privileged areas of computer system. RISC-V is also susceptible to many vulnerabilities such as buffer overflow, fault injection on return address etc. The RISC-V based Common Evaluation Platform being used to build a secure architecture are also prone to this type of attack.

2.2.1 BUFFER OVERFLOW ATTACK

In computer system, stack memory plays an important role in providing a mechanism that allows system's memory to be used as a temporary storage for the incoming information. It consists of certain important protected information which is responsible for keeping a track of function which a program is executing along with its parameters, return address after the function has completed execution and base pointer. The one form of buffer overflow attack is to target the return address on the stack to hijack the control flow of the system [10]. To understand more in detail, it is

important to look at memory structure of a computer system. The figure 2.1 shows an example of memory structure concentrating on stack memory structure and other elements of the memory

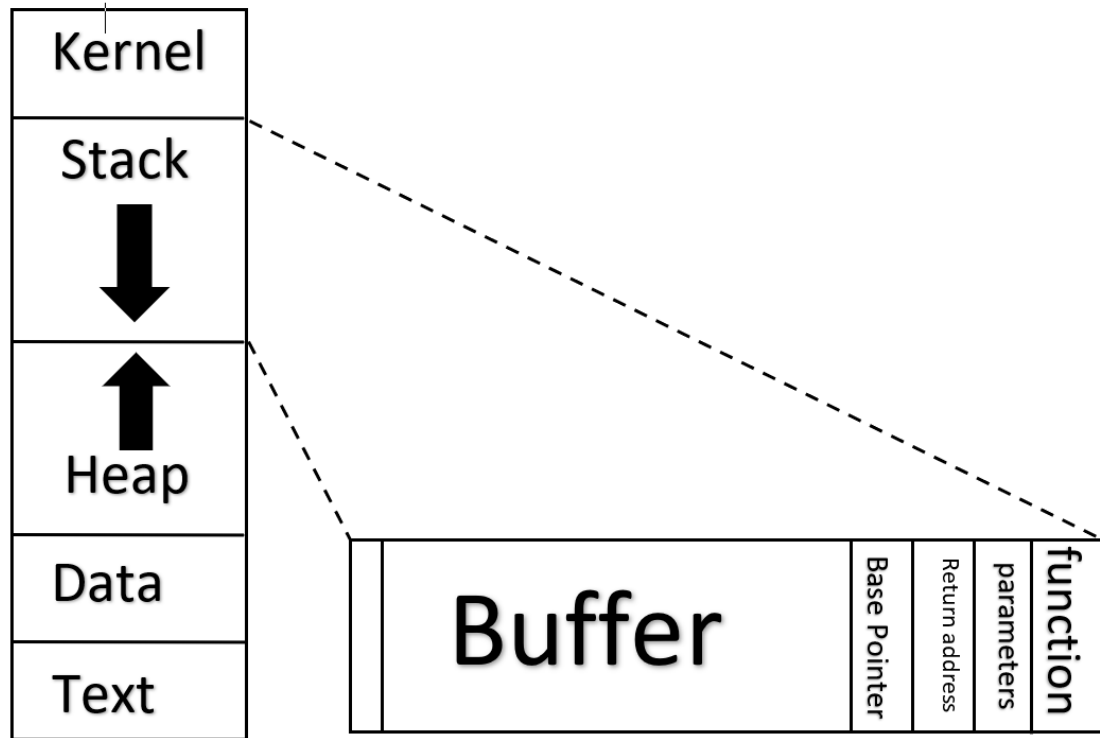


Figure 2.4: Memory Structure of a Computer System

Memory structure includes components such as kernel space, stack, heap, data and text. In RISC-V base architecture, stack grows downwards and heap grows upwards. Whenever a new program is loaded by the CPU, it starts creating a stack frame. A buffer of specific size is added to the stack frame to temporarily store the incoming data for processing. A buffer is usually created by the initialization of an array in the function. The return address is also stored on the stack frame which has the address to return after the function has completed its execution. The buffer grows in index from low address to high address. The attacker uses this principle to cause a segmentation fault by exceeding the set storage limit of the buffer and hence trespassing the limit to modify the return address to execute malicious code or to modify the execution

flow of the program.

Hence, it is important to monitor the information during run time and stop the execution if the return address is manipulated to avoid buffer-overflow attack.

2.2.2 Fault Injection on Return Address

Return address manipulation is the most common type of run-time attack [11]. It is done to change the control flow of the program. There are various hardware and software ways to perform this attack. The most common way to perform these attack is to cause buffer overflow attack to invade the protected data stored on the stack. This protected also contains the return address which tell the program to return after the successful execution of the function. The attacker modifies the return address stored on the guarded memory location stored on the stack. This result in change in the program flow which an intruder can use to bypass a security extension or run the malicious code on the system.

2.3 SECURITY COUNTERMEASURE OF RISC-V ARCHITECTURE

There are various countermeasure developed on the RISC-V architecture against buffer overflow attack at various level of operations. Morpheus chip developed at University of Michigan [9] uses Address space layout randomization(ASLR) policy to avoid buffer overflow attack. This policy randomizes the protected address stored on the stack after every program execution. This policy provides a higher accuracy against these type of attacks but the effectiveness of this policy it compromised by multiple constraints such as size of virtual address space, fragmentation problems, power consumption and compatibility [12] [13]. Information flow tracking is the another technique used against buffer overflow attack[14]. In this thesis, we explore the Information flow tracking technique and it's implementation on different levels of granularity. We also propose a ISA level information flow tracking on RISC-V architecture as security countermeasure against buffer overflow attack.

2.3.1 INFORMATION FLOW TRACKING

Information Flow Tracking has been a promising and effective analysis technique in security applications of determining the information leakage and detecting the malicious untrusted security applications by determining the information leakage and detecting the malicious untrusted data at run-time. Based on the static verification during the design phase or dynamic checking at the runtime, different IFT approaches have been implemented. The precision of the IFT logic determines the different levels of abstraction such as gate level information flow tracking, register transfer level information flow tracking, language based information flow tracking and dynamic information flow tracking.

Gate level Information flow tracking(GLIFT) has precise tracking rules for the set of universal gates with shadow logic [15]. Though it provides a formal basis for a system's root of trust with a compositional approach, it results in a computationally complex design for shadow logic functions with a high number of inputs and does not scale with design size. The gate level design focuses on a specific security critical module and the Datapath of the module to be executed is tracked. The inputs and outputs of the entire module are tainted and a shadow logic library for the tainted gates is implemented. The security policies are matched with tainted information for all the gates under test. The output of the tainted gates will be affected if there is an arbitrary change in the inputs that affect the output. The shadow logic function is implemented for all the gates and the output of the original module will yield a true value if the output is not affected by any tainted input and a false value if the tainted input and a false value if the tainted inputs affect the original logic.

Register transfer level (RTL) tracking is based on propagation rules with RTL expressions [16]. It enables verification of information at an early stage, at a higher level of abstraction and directly in RTL code. Although the verification is done on the early stage, it tradeoffs with IFT precision. The authors suggested that RTLIFT

is 5x faster than gate level IFT and it also minimizes the efforts required for designing Register Transfer Level IFT technique the gate level IFT [16].

Language-based IFT achieves a dynamic access-control mechanism with communication channels. It uses a software approach which focuses on security goals rather than protecting confidentiality by controlling the information flow[17].

The next approach is Dynamic Information Flow Tracking(DIFT). Dynamic Information Flow Tracking includes both hardware and software-based implementations depending on the architecture of the system and focused on the control as well as non-control-data attacks[18] [19]. Most of the implementations can be used to detect buffer overflow attacks, format string Attacks along with various memory corruption attacks. The DIFT mechanism flow in order to monitor and track the Untrusted data start by allocating a tag to the malicious channels and marks it as spurious. During program execution, the tag propagation unit keeps track of the information flow generated by the spurious data[20]. Finally, a tag checking unit detects the unsafe data by matching it with the security policies implemented for each untrusted channel and raises a security exception. The idea of DIFT is to tag certain inputs or data with some metadata known as tags and then propagating these tags as the program executes to know the information flow of the program. The metadata can be represented by one bit: either trusted or untrusted or it can be represented by a data structure. To achieve this idea of tags, a different memory is associated in DIFT which can be called as tag memory or shadow memory. It contains the collection of tags and also the relationship between this tags and memory addresses that own them.

The proposed IFT technique is based on Dynamic Information Flow Tracking implemented on ISA level. ISA is usually defined as the interface between hardware and software. ISA acts as an interface between hardware and software. ISA consists of a set of instructions which defines the operation hardware should perform and contains the parameters on which instruction is to be executed. The ISA level approach allows

us to add instructions to interact directly with the hardware to perform an operation. As it is overlaid between hardware and software, it makes it possible to leverage the advantage to execute a security policy with high accuracy and less overhead delay.

2.4 SOFTWARE ECOSYSTEM OF RISC-V ARCHITECTURE

RISC-V architecture has a rich software ecosystem which consists of simulators, emulators, compilers, debuggers etc [2]. The software toolchain for RISC-V supports GCC and Clang/LLVM compilers along with various benchmark environments [21]. Many simulators for RISC-V have been developed and the most common simulator which is used as a reference model for RISC-V ISA is the Spike simulator [22].

2.4.1 RISC-V GNU TOOLCHAIN

Toolchain is collection of tools combined for the purpose of software development of a processor. RISC-V toolchain consists of tools such as Binutils, DejaGnu, GCC, GDB, GLIBC, Newlib. Binutils consists two sets of important tools such as assembler and linker. The assembler processes assembly language programs to produce relocatable machine code. Library files and other object files must be linked together with the relocatable machine code produced by the assembler. The role of the linker is to link these multiple files together. Commands included in binutils are as (assembler), ld (linker), gprof (profiler), objcopy, objdump, addr2line, etc. DejaGnu is a framework for testing other programs. Its purpose is to provide a single front end for all tests. The GNU Compiler Collection (GCC) is a set of compilers used in Linux systems. Normally GCC is responsible for pre-processing, and compilation the source code that can be used by assembler to carry-out its operation. GNU Debugger (GDB) is the debugging tool that comes with the GNU toolchain. It is a programmer friendly tool, that is used to understand the flow of program by running it line by line to understand its operation. The GNU C Library (GLIBC) provides the cores libraries for GNU/ Linux systems. Some of might include open, read, write, malloc, printf,

getaddrinfo, dlopen, crypt, login, exit. Newlib is the standard library that provided minimalistic core libraries, intended for the used for embedded systems. Figure 2.5 shows different stages of program execution from source code to executable machine code along with its tools responsible for each block declared in the toolchain

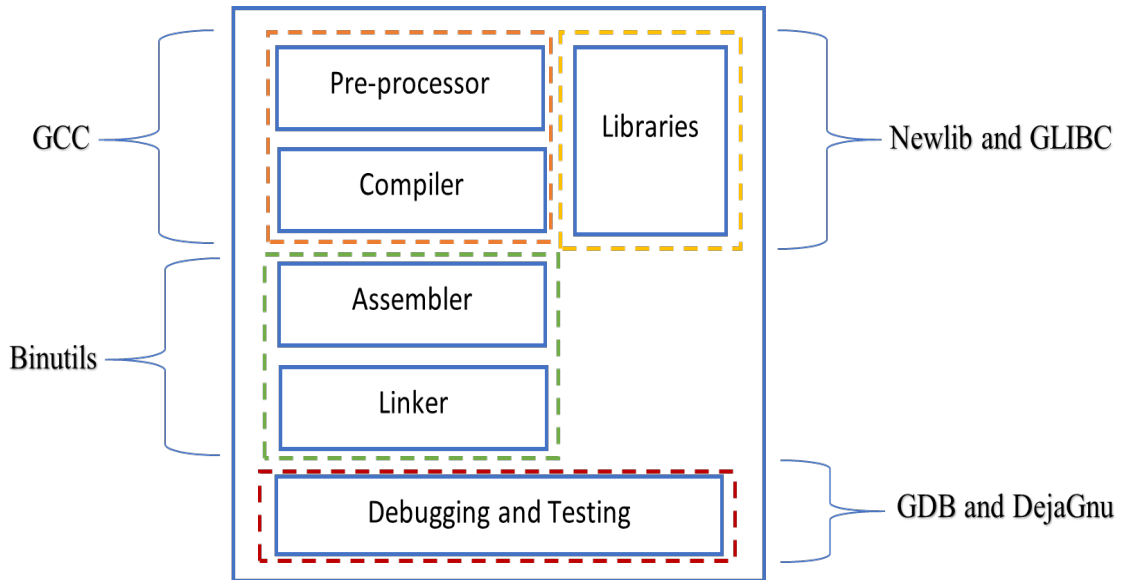


Figure 2.5: Different stages of program execution

2.4.1.1 ASSEMBLER MODIFICATION

RISCV is an open-source architecture. There are some steps that need to be carried out to add a new instruction in GNU assembler of RISCV toolchain. Firstly, we must declare the instruction with respect to its instruction format to generate matching and masking address value for the instruction. Secondly, we declare the instruction corresponding to the matching address and master address. Lastly, we must describe the instruction length, operands and functionality and processor info of that instruction.

The major goal of Assembler Modification is to extend the ISA for a number of instruction set extension. This section shows the directory tree of required file to add instructions in RISC-V ISA. All the following steps will be concentrated on RISC-V ISA but the changes can also be transferable to other architecture as well.

The first important step before modify the toolchain is to classify the instruction format and associate it with an opcode and function value to help the assembler to generate correct machine code. After the instruction format is generated, create the appropriate masking and matching code.

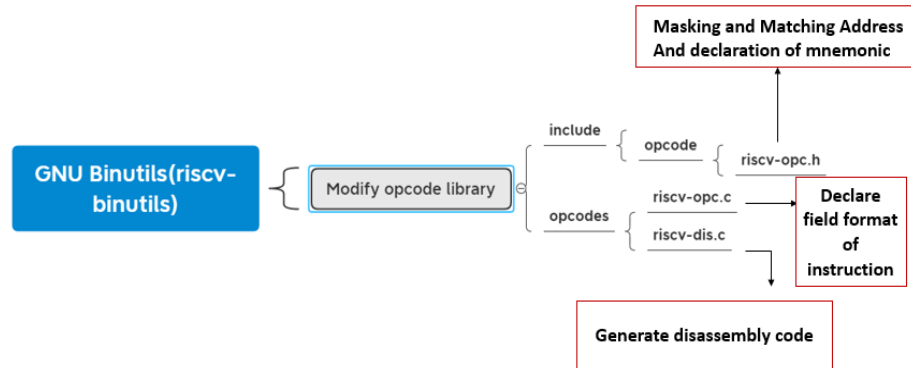


Figure 2.6: Directory Structure for Assembler Modification

Figure 2.6 shows the directory structure required for Assembler modification. To add a new instruction in the assembler, there are three major steps such as initializing the masking address and matching address along with its declaration with the mnemonic, declaration of field format of instruction and generate the disassembly code. Each step is briefly described below:

1) The first step is to add the computed Match address and Mask address and initialize the new instruction with this computed match and mask addresses. This can be done by declaring match and mask address globally in `riscv-opc.h` and declare the instruction using function `DECLARE_INSN`. This instruction is responsible for mapping match and masking address with the instruction.

2) Modify the `riscv-opc.c` to add the instruction to the `riscv_opcodes` structure which contains all of the instruction available in the RISC-V architecture. Figure 2.7 shows the field format to add the instruction in `riscv_opcode` structure.

Name	Xlen	Instruction Class	Instruction Operands	Match	Mask	Match opcode	Pinfo
------	------	-------------------	----------------------	-------	------	--------------	-------

Figure 2.7: Field Format to add instruction

Name: It represents mnemonic name of the instruction.

Xlen: It will be 32 if the instruction only belongs to RV32 , 64 for RV64 and 0 for both.

Instruction class: This determine the instruction which describes the instruction format.

Instruction operands: Defines the operands of the instruction.

Match and Mask address: Connect the Masking and Matching address with Mnemonic name of instruction.

Match Opcode: RISC-V generic match_opcode function uses the logic : encoding & MASK == MATCH.It helps to mask the opcode.

Pinfo: It determines the process information which is used to describing the instruction.

After Declaring the instruction in the toolchain rebuild the toolchain . Next step is to add modules in a ISA.

2.4.2 SPIKE ISA SIMULATOR

Spike ISA simulator is used to test the modification of ISA on software level. Spike ISA directory tree is divided in different modules. Each module is responsible to simulate a block of architecture. It contains blocks such as Memory Management Unit(MMU), Trap Unit, Encoding and Decoding Unit, Processor Unit etc. It is easy to add blocks in the Spike Simulator by adding functions within the simulation block.

2.5 COMMON EVALUATION PLATFORM

The RISC-V based Common Evaluation Platform (CEP) is vulnerable to the above mentioned security vulnerabilities that can affect the accuracy of the simulation model

and thus can corrupt the design flow of side channel resilient IP design. In this work we design security extensions that enable secure execution of instructions run time checking of any such attacks as discussed earlier[23]. The security extensions verification and integration is a tedious time consuming and require several iterations of testing on the hardware. We design a simulation model of verifying the security properties with the extended security enriched instructions and datapaths in the simulation model for fast verification process. We create a software simulation model that highly correlates with the working of hardware image.generated.

CHAPTER 3: PROPOSED SCHEME AND EXPERIMENTAL SETUP

3.1 PROPOSED SCHEME

The propose technique focuses on preventing memory corruption and protects the return address from software attacks. The control data flow integrity with tagged bits ensures that the return address matches the corresponding address after the context switching which prevents an adversary from hijacking the return addresses. The tag-based analysis is flexible in tracking the record of the data with minimal overhead if a single bit is used as a tag. Compiler-based modification is done to add the security policies and to assist with the additional new instructions added to the architecture.

Stack and data protection by tracking and detecting the tagged data eliminates software attacks which focuses on the return address [24]. In the tagged mechanism, labels are associated with the address of the data that are received from the untrusted source. With minimal hardware overhead, the tag mechanism is implemented by using a 1-bit tag to the data address [25]. When a program is executed during run time the tag mechanism assigns the tag bits to the spurious data through the tag propagation module. The tag propagation module assigns tags by using the new custom instructions implemented in the ISA architecture to store and check the tag bits. The tag bits are stored in the tag cache which reduced the dedicated memory assigned for storing the tag bits.

The RISC-V Rocket core is modified to incorporate the security features both in the ISA level and on the toolchain to detect and eliminate the buffer overflow and string format attacks. At the ISA level, the tag module consists of all the tag management units and the translated compiler modifications are developed in RISC-V GNU. Design verification and validation is performed on the RISC-V Spike Simulator.

This model focusses on the stack protection and the return address stored on the stack. The security policy functions are used check for the tag bits from the return address and the untrusted data source. An exception is raised when the framework detects a mismatch in the tag bit of the return and data address indicating a software attack has occurred.

The Load/Store architecture dedicates to copying The data in the memory. Loads are encoded in the I-type Format and stores are S-type format. Thus, the RISC-V core is Modified by adding a new Load/Store instructions which are Used to provide security checks for the 1-bit tag with matching The return addresses. Based on the load and store encoding Specified in the RISC-V core the new instructions are added:

- In the load instruction encoding: LDTCHECK
- In the store instruction encoding: SDTCHECK

Figure 2.8 shows the Load/Store Instruction format where the red highlighted fields (funct3 and opcode) are modified for the new instructions to specify the operations to be performed. When the data from an untrusted source is received the SDTCHECK will set the tag bit to 1 and LDTCHECK is used for loading the data word and checking if the tag bit is 0 or 1.

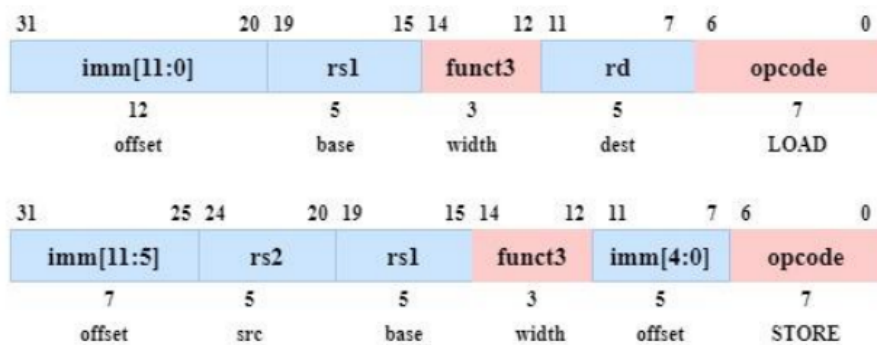


Figure 3.1: Load/Store Instruction format of RISC-V Architecture

The CSR address convention which provides accessibility for error checking by

using the large CSR space is used to check the read/write compatibility of the custom instructions with the tag bit where the loaded tag bit and the expected tag bit in the instruction match, if not it raises an exception. A separate Tag cache module consists of all the tag management mechanisms with tag initialization, tag propagation, and tag checking modules. This module reduces the overhead of physically adding a one-bit tag to the memory. When a one-bit tag is added to the instruction this module fetches the tag and store it in the tag cache with its own tag cache mapping which reduces a significant amount of overhead in the architecture. The tag cache acts as a cache simulator for the tags and this is used to validate the return addresses and the memory access of the untrusted data. Figure 2.9 shows the E31 core RISC-V architecture with the added IFT Tag module.

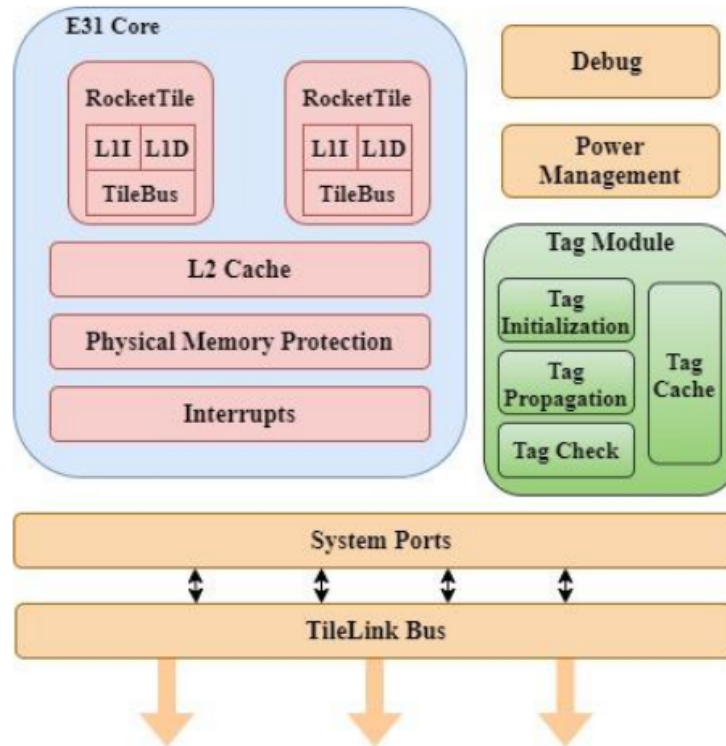


Figure 3.2: E31 Core RISC-V Architecture with Tag Module

The tag for each memory data is directly mapped and stored in the tag cache by virtual address spacing and lookup tables are used for mapping. During program

execution, the tag initialization module assigns the tag bits to the data for the new instructions added and the tag propagation module tracks the flow of the data and the return addresses of each data from the new instructions. Finally, the tag check module is used to compare the tag bits of the new instructions and if there is a return address miss-match it raises a security exception. This detects the return address modification attacks when an adversary tries to modify the return address by function call and context switching.

When sensitive data is sent from an untrusted source the tag module initials the tag mechanism using the new instructions with load and store request similar to lowRISC [26]. The L1 data cache holds the data with the tag value for both read and write request and the tag module checks for a mismatch condition based on the tag bits.

The IFT framework is tested on the Spike simulator by implementing a buffer overflow attack program. The new instructions are used with the stack operations where the return address is saved. Tag bits are assigned to the return address and the data to be stored on the stack. When the buffer overflow attack occurs the return address is checked for the tag value and if it's the same it executes the program. If there is a mismatch an exception is raised, and the program execution is stopped by eliminating buffer overflow attack. The newly added SDTCHECK is used to assign a tag bit for the return address and the LDTCHECK is used to check the tag corresponding the return address in the tag cache. For IFT the program raises an exception and stops without returning to the main function using the new instructions when there is a tag mismatch thus eliminating buffer overflow attacks. This framework protects the stack by using the customized instructions where the new return address compromised by an adversary is not loaded on the stack.

3.2 EXPERIMENTAL SETUP

Experimental setup includes shell commands to download and install prerequisite packages, RISC-V GNU Toolchain and Spike ISA Simulator, steps to add instruction

in the toolchain, add tag memory module is Spike ISA simulator, write test codes for simulating buffer overflow attack. We used Ubuntu 20.04 virtual machine installed on Oracle VirtualBox with 30 GB of memory space allocated. All the software are installed in this virtual machine.

3.3 RISC-V GNU TOOLCHAIN INSTALLATION PROCESS

Step 1: Open a new terminal and download the prerequisite using the following command:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev
libmpfr-dev libgmp-dev gawk git build-essential bison flex texinfo gperf libtool patchutils
bc zlib1g-dev libexpat-dev
```

Step 2: Download the RISC-V GNU Toolchain from RISC-V official git repository using the following command. Use this command to download the latest version of RISC-V ISA published on the master branch

```
$ git clone -recursive https://github.com/riscv/riscv-gnu-toolchain
```

Step 3: To build the toolchain get inside the downloaded RISC-V toolchain directory and run the following command. By default the installation binaries are set to RV64C use `-arch` flag to use different RISC-V ISA architecture.

```
$ cd riscv-gnu-toolchain
```

```
$ ./configure --prefix=/opt/riscv
```

`--prefix` flag points out the installed directory. You can change installation directory but make sure that the directory entered has read and write permissions.

Step 4: After configuring the build binaries use the make command to create a RISC-V toolchain image in the prefixed directory. The command is as follows :

```
$ make
```

It will take around 30 to 45 minutes depending on computer's configuration.

Step 5: Set an installation path and add the path to the environment variable permanently. To do so , type:

```
$ sudo gedit /etc/environment
```

Add your path in the file .For example. `:/opt/riscv/bin` and save it. Step 6: Check if the path is properly added in environment variable by using this command:

```
$ echo $PATH
```

If you see your RISC-V installation path, it is permanently added to environment variable. If not repeat step 5 and Step 6: Test the installation using the following command. This command shows the RISC-V compile version if installed correctly.

```
$ riscv64-unknown-elf-gcc -v
```

3.4 SPIKE ISA Simulator Installation Process

To continue with installation, make sure Toolchain is properly installed and to know the installation path of RISC-V GNU Toolchain.

Step 1: Open a new terminal and create a temporary environment variable pointing toolchain prefixed path.

```
$ export RISCVC=/opt/riscv
```

Step 2: Download the prerequisites using the following command:

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev
libgmp-dev libusb-1.0-0-dev gawk build-essential bison flex texinfo gperf libtool patchutils
bc zlib1g-dev device-tree-compiler pkg-config libexpat-dev
```

Step 3: Download the RISC-V Tools from RISC-V official git repository using the following command:

```
$ git clone -recursive https://github.com/riscv/riscv-tools
```

Step 4: Get inside the folder and run the `./build-spike-pk.sh` using the following commands

```
$ cd riscv-tools
```

```
$ ./build-spike-pk.sh
```

Step 5: To test your installation, create a new C file that prints hello world. Compile it with RISC-V compiler using following command


```
$ riscv64-unknown-elf-gcc hello.c -o hello
```

Try to run the object file using Spike simulator using the following command. It should print Hello World

```
$ spike pk hello
```

3.5 TOOLCHAIN MODIFICATION

In chapter 2 , we discussed the files which are needed to be modified to add instructions. So to modify the opcode library we will add computed Match address and Mask address in riscv-opc.h and use the declare function to map masking and matching address with the instruction's name. The changes are as follows:

```
#define MATCH_LDCHK0 0x57
#define MASK_LDCHK0 0x707f
#define MATCH_SDSET0 0x2057
#define MASK_SDSET0 0x707f
DECLARE_INSN(ldchk0, MATCH_LDCHK0, MASK_LDCHK0)
```

The first step is to edit the riscv-opc.c to declare the instruction in riscv_opcode structure. Changes will be as follows:

```
"ldchk0",64,INSN_CLASS_I,"d,o(s)",MATCH_LDCHK0,MASK_LDCHK0,match_opcode,
INSN_DREF|INSN_8_BYTE
"sdset0",64,INSN_CLASS_I,"t,q(s)",MATCH_SDSET0,MASK_SDSET0,match_opcode,
INSN_DREF|INSN_8_BYTE
```

Next step is to configure the assembler to use the new table entries. For this, ldchk0 and sdset0 are declared as macros. The reason for declaring it as a macro is that in RISC-V architecture all the load and store instruction are declared as macros as they directly load symbols from register. The instructions are as follows:

```
case M_LDCHK0:
```

```

    pcrel_load (rd, rd, imm_expr, "ldchk0",
    BFD_RELOC_RISCV_PCREL_HI20, BFD_RELOC_RISCV_PCREL_L012_I);

    break;

case M_SDSET0:

    pcrel_store (rs2, rs1, imm_expr, "sdset0",
    BFD_RELOC_RISCV_PCREL_HI20, BFD_RELOC_RISCV_PCREL_L012_S);

    break;

```

The final step is to recompile the toolchain. The command is as follows:

```

$ cd riscv-gnu-toolchain

$ make clean

$ make

```

3.6 SPIKE SIMULATOR MODIFICATION

To simulate the working of newly added instructions, LDCHECK and SDSET, a tag module is added in the the RISC-V base ISA with the functions to handle the setting and checking the tag, In this work. we demonstrate the security capabilities on return address attacks where an adversary can control the targets execution flow by corrupting the return address in the stack and executing an arbitrary code on the target. The security policies are described to check the flow of the tag bit in the tag cache and traversing through the newly added instructions. Figure 3.1 illustrates the security policy pseudocode for the `check_tag` and `store_tag` functions. The `check_tag` function executes the tag condition for the return address. If there is a mismatch it raises an exception without returning to the main address. The `store_tag` function checks for the untrusted channels and the return address before procedure calls and assigns a tag bit to it and stores the tag bit on the tag cache.

```

Function :check_tag
1 Offset decoding
2 Masking Address Decoding
3 Fetch tag value from tag cache for the masked address
4 condition( tag_bit = 0)
5   Function executes
6 else
7   Raise an Exception
8 end

Function :store_tag
1 Offset decoding
2 Masking Address Decoding
3 Fetch tag value from tag cache for the masked address
4 condition check on the untrusted source
5   Assign tag bit =1
6 else
7   Assign tag bit =0

8 Store tag bit on tag cache
9 Store the address masked

```

Figure 3.3: Pseudocode for Check tag and Store tag functions

The functions are added in mmu.h header file mentioned in the Spike Simulator. To check Tag , Masking address is decoded and check its entry in tag cache. If the value corresponding tag bit is 1 the function executes or else it raise an exception. In store tag function masking address is decoded, if the condition on the untrusted source, assign tag bit to 1 or else 0 and then store this address in the tag cache. For demo purposes, we assume that each instruction defined is untrusted.

3.7 DEMONSTRATION OF SECURITY PROPERTIES

3.7.1 ATTACK SETUP

This section discusses about the code to simulate buffer-overflow attack. To simulate buffer overflow attack we have created a C source code. Figure 3.4 shows the sample code to simulate buffer overflow attack for testing the modified ISA.

```

1 void target() {
2     printf("Buffer overflow successfully occurred");
3     printf("\n");
4     exit(0);
5 }
6
7
8 void vulnerable(char* str1) {
9     char buf[5];
10    strcpy(buf, str1);
11 }
12
13 int main() {
14     vulnerable("abcdefghijklmno\x50\x01\x01");
15     printf("This only prints in normal control flow execution");
16 }

```

Figure 3.4: Sample Attack code

In this code, main function calls a vulnerable function which do a string copy operation and add the data on the stack. In the vulnerable function, a character buffer is created which has a size 5 but during strcpy function due to undefined behaviour known as buffer overflow attack, we can bypass the segmentation fault trap and allow the program to access restricted locations. This can be used to modify the return address. The argument received to vulnerable function from the main function has the target function address which can be retrieved by using objdump tool to disassemble the object file. The instruction for RISC-V toolchain is:

```
$ riscv64-unknown-elf-objdump -d attack>attack.asm
```

Through this, we can retrieve the memory address of the function as shown in Figure 3.2. Location of stored return address on the stack is retrieved by intentionally causing the buffer overflow attack which will output the return address in the terminal and based on the input string provided predict the location of return address stored on the stack.

```

00000000000010150 <target>:
| 10150: 1141 addi sp,sp,-16

```

Figure 3.5: Disassemble code to find the target address

This modifies the return address stored on the stack and calls a target function which will print something in the shell, For the given example, the target function address is 0x010150 and if the attack is failed it will either run into segmentation fault if the value exceeds the buffer size and if it string value is under the buffer value. It will print out the statement from the main function. The modification of return address will result in the change of instruction flow.

For the proposed IFT system, the disassembled code is edited with the new instructions replacing all the load/store instructions to display the protection from this attack, The design with security enriched instructions are compiled and are made available to the simulation setup. It is an important thing to note that toolchain modification is important for this purpose because if the assembler cant recognize the instruction, it cannot change instructions into its binary format and will throw an illegal instruction exception.

CHAPTER 4: RESULTS AND SECURITY ANALYSIS

4.1 RESULTS

This section demonstrates the results of attacks on the traditional RISC-V core and its countermeasure using IFT. Furthermore, software simulation model is correlated to the hardware model to expedite the security verification model based on RISC-V GNU toolchain and Spike ISA simulator.

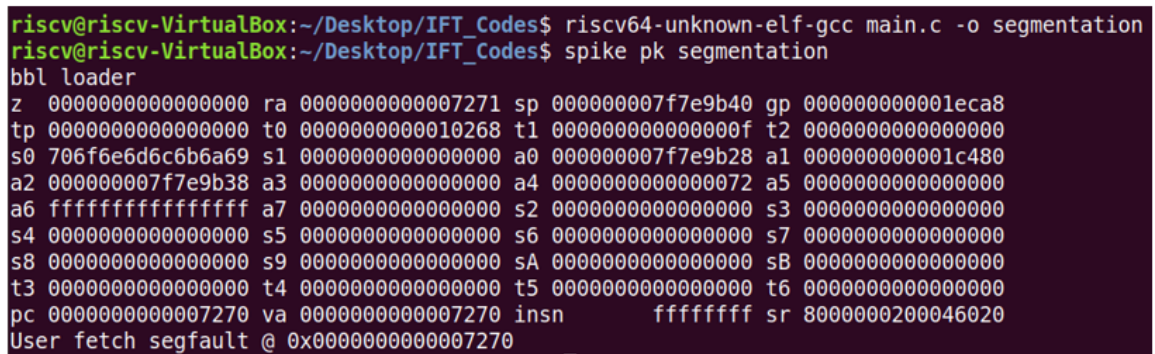
4.1.1 RETURN ADDRESS LOCATION DECRYPTION

The location of return address on the stack can be retrieved by intentionally performing a segmentation fault. It can be done by overflowing the char buffer in the attack source code. Figure 4.1 shows the attack source which will cause the segmentation fault and information stored on stack after executing the segmentation fault.

```

1 void target() {
2 printf("Buffer overflow successfully occurred");
3 printf("\n");
4 exit(0);
5 }
6
7
8 void vulnerable(char* str1) {
9 char buf[5];
10 strcpy(buf, str1);
11 }
12
13 int main() {
14 vulnerable("abcdefghijklmnopqr");
15 printf("This only prints in normal control flow execution");
16 }

```



```

riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ riscv64-unknown-elf-gcc main.c -o segmentation
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ spike pk segmentation
bbl loader
z 0000000000000000 ra 0000000000007271 sp 000000007f7e9b40 gp 000000000001eca8
tp 0000000000000000 t0 0000000000010268 t1 000000000000000f t2 0000000000000000
s0 706f6e6d6c6b6a69 s1 0000000000000000 a0 000000007f7e9b28 a1 000000000001c480
a2 000000007f7e9b38 a3 0000000000000000 a4 0000000000000072 a5 0000000000000000
a6 ffffffff s2 0000000000000000 s3 0000000000000000
s4 0000000000000000 s5 0000000000000000 s6 0000000000000000 s7 0000000000000000
s8 0000000000000000 s9 0000000000000000 sA 0000000000000000 sB 0000000000000000
t3 0000000000000000 t4 0000000000000000 t5 0000000000000000 t6 0000000000000000
pc 0000000000007270 va 0000000000007270 insn ffffffff sr 8000000200046020
User fetch segfault @ 0x0000000000007270

```

Figure 4.1: Sample Code to perform attack along with its result on commandline

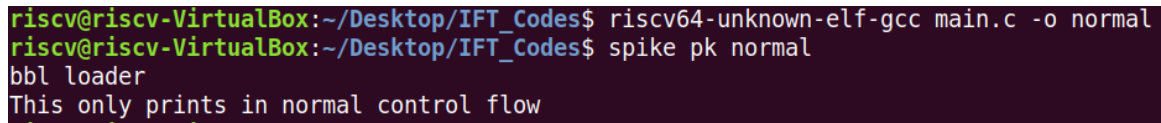
To cause segmentation fault, a string is passed with size longer than the buffer value. In the above example code string from A to R is passed which has a string length greater than the buffer size and hence, causes segmentation fault. In the above figure, return address (ra) shows 7271. This hex number is converted into ASCII number which outputs qr. Through this method, return address location stored on the stack is retrieved. In the above example, qr string is replaced with the target memory address.

This proves that even though segmentation fault exception exits the running process and avoids the program to access protected memory location stored on the stack, it

can still leak many important information.

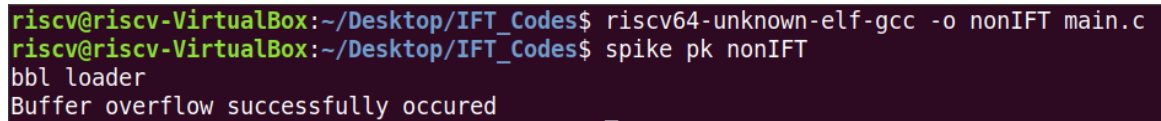
4.1.2 BUFFER OVERFLOW ATTACK ON CONVENTIONAL RISC-V ARCHITECTURE

The segmentation fault can be bypassed by executing a buffer overflow attack on the system as discussed in previous chapter. Figure 4.2 shows the result when the program is working in normal condition on conventional RISC-V architecture and Figure 4.3 shows the result of the buffer overflow attack on conventional RISC-V architecture.



```
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ riscv64-unknown-elf-gcc main.c -o normal
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ spike pk normal
bbl loader
This only prints in normal control flow
```

Figure 4.2: Result of Normal Control flow execution



```
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ riscv64-unknown-elf-gcc -o nonIFT main.c
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ spike pk nonIFT
bbl loader
Buffer overflow successfully occurred
```

Figure 4.3: Result of Buffer overflow attack execution

The results in the figure above depicts that the program will work correctly during normal execution where the input data is less than the buffer size limit but due to undefined behaviour called buffer overflow, which causes a fault injection on return address which causes the change in control flow of program.

4.1.3 BUFFER OVERFLOW ATTACK ON MODIFIED RISC-V ARCHITECTURE

In the proposed scheme, RISC-V core is hardened from the buffer overflow and return address to protect the system from this type of attack by stopping the load of return address from the stack and raising an exception to terminate the process. Same steps should be carried as discussed in chapter 3. Figure 4.4 shows the execution of normal control compiled with the new instructions. Figure 4.5 shows the result of

Hardware based Information Flow Tracking system to protect the system from buffer overflow attack.

```
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ riscv64-unknown-elf-gcc -o normalIFT main.s
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ spike pk normalIFT
bbl loader
This only prints in normal control flowriscv@riscv-VirtualBox:~/Desktop/IFT_Codes$
```

Figure 4.4: Result of normal control flow execution compiled with new instructions

In figure 4.4 verification of the new instructions which are recognized by the Spike ISA simulator and assembler generates proper binary formats for those instructions.

```
riscv@riscv-VirtualBox:~/Desktop/IFT_Codes$ spike pk IFT
bbl loader
z 0000000000000000 ra 0000000000010150 sp 000000007f7e9b10 gp 000000000001eca8
tp 0000000000000000 t0 000000000001026a t1 000000000000000f t2 0000000000000000
s0 000000007f7e9b40 s1 0000000000000000 a0 000000007f7e9b28 a1 000000000001c480
a2 000000007f7e9b38 a3 0000000000000001 a4 0000000000000000 a5 0000000000000000
a6 ffffffffffffffff a7 0000000000000000 s2 0000000000000000 s3 0000000000000000
s4 0000000000000000 s5 0000000000000000 s6 0000000000000000 s7 0000000000000000
s8 0000000000000000 s9 0000000000000000 sA 0000000000000000 sB 0000000000000000
t3 0000000000000000 t4 0000000000000000 t5 0000000000000000 t6 0000000000000000
pc 0000000000010186 va 0000000002011457 insn 02011457 sr 8000000200046020
Bufferflow Exception by IFT
```

Figure 4.5: Result of Hardware based Information Flow Tracking system to protect the system from buffer overflow attack

Figure 4.5 shows the successful execution of Hardware based Information Flow Tracking. The important thing to note here is that there is a change in return address due to buffer overflow, but due to tag checking mechanism the new address cannot be mapped to the address stored in tag cache, hence the proposed IFT model identifies the buffer overflow attack and stops the attack by terminating the process.

4.2 SECURITY ANALYSIS

The RISC-V is vulnerable to the security leaks such as buffer overflow, program counter attacks, fault attacks. The attacker leverages the bit flips or modification to the return address (RA) of the stack and can compromise the device. The hardware integration of the security features takes much longer for verification and security

analysis. The toolchain extension support helps in creating new software features by adding, modifying instructions and registers developed for our custom ISA inside the toolchain. Toolchain support provides the flexibility to add new security policies and develop test codes and software programs to carry out security analysis for the custom ISA. The proposed simulation model supports the design and verification of security extensions to the RISC-V processor. The simulation model has extended toolchain that supports new functions and instructions such as IFT enabled execution and encryption support.

CHAPTER 5: CONCLUSIONS AND FUTURE GOALS

In this research, we designed a software simulation model of hardware-based Information Flow Tracking framework with Tagged mechanism by assigning a 1-bit tag to the spurious data address and return address and translated the hardware architecture-specific extensions to compiler-specific simulation model. This is a novel contribution to integrate the hardware security to support the architectural integration for simulation model. The results of the implemented simulation model show that the framework tracks the tagged address and eliminates the buffer overflow attacks and the results are correlated to the hardware design. This implementation has minimal performance overhead with better precision logic and higher performance in terms of verifying the security extensions.

Our future work will concentrate more on checking the resilience of this system to avoid bypassing this system by any undefined behaviour of the program. We will also work on calculating the performance overhead of this design.

REFERENCES

- [1] S. Kechiche, “Iot connections forecast: the rise of enterprise,” December 2019.
- [2] RISC-V Foundation. [Online]. Available: <https://riscv.org>.
- [3] Common Evaluation Platform Repository Available: <https://github.com/mit-ll/CEP>.
- [4] RISC-V ISA Privileged Specification Version 2.2 Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf>.
- [5] RISC-V ISA Unprivileged Specification Version 2.2 Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>.
- [6] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.
- [7] “Risc-v star rises among chip developers worldwide,” April 2021. <https://spectrum.ieee.org/tech-talk/semiconductors/design/riscv-rises-among-chip-developers-worldwide>.
- [8] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, “Shakti: A risc-v processor with light weight security extensions,” *In Proceedings of the Hardware and Architectural Support for Security and Privacy* 7, 2017. <https://doi/10.1145/3092627.3092629>.
- [9] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. Austin, “Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn,” *Architectural Support for Programming Languages and Operating Systems*, April 2019. <https://doi/10.1145/3297858.3304037>.
- [10] D. Fu and F. Shi, “Buffer overflow exploit and defensive techniques,” January 2013.
- [11] S. Nashimoto, N. Homma, Y. ichi Hayashi, J. Takahashi, H. Fuji, and T. Aoki, “Buffer overflow attack with multiple fault injection and a proven countermeasure,” *Journal of Cryptographic Engineering*, July 2016.
- [12] H. Marco-Gisbert and I. Ripoll, “Address space layout randomization next generation,” *Security Issues and Solutions of Smart Contracts in Blockchain Technology*, July 2019. <https://doi.org/10.3390/app914292>.

- [13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [14] N. Timmers, A. Spruyt, and M. Witteman, “Controlling pc on arm using fault injection,” *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016. doi: 10.1109/FDTC.2016.18.
- [15] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Shephard, “Complete information flow tracking from the gates up,” *SIGARCH Computer Architecture. News* 37, March 2009. <https://doi.org/10.1145/2528521.1508258>.
- [16] A. Ardeschiricham, W. Hu, J. Marxen, and R. Kastner, “Register transfer level information flow tracking for provably secure hardware design,” *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017. 10.23919/DATE.2017.7927266.
- [17] A. Sabelfeld and A. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, 2013. 21. 10.1109/JSAC.2002.806121.
- [18] J. Criswell, N. Dautenhahn, and V. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” *IEEE Symposium on Security and Privacy*, 2014.
- [19] U. Erlingsson, M. Abadi, M. Vrabie, M. Budiu, and G. C. Necula, “Xfi: Software guards for system address spaces,” *Symposium on Operating Systems Design and Implementation*, 2006.
- [20] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, “Defeating memory corruption attacks via pointer taintedness detection,” *International Conference on Dependable Systems and Networks*, 2005. doi:10.1109/DSN.2005.36.
- [21] M. Poorhosseini, W. Nebel, and K. Gr  ettner, “A compiler comparison in the risc-v ecosystem,” *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020. doi: 10.1109/COINS49042.2020.9191411.
- [22] RISC-V Official Repository Available: <https://github.com/riscv>.
- [23] Q. Ge, Y. Yarom, and G. Heiser., “No security without time protection: we need a new hardware-software contract,” *Asia-Pacific Workshop on Systems (APSys)*, 2018.
- [24] B. Thakar, S. Geraldine, and F. Saqib, “Hardware secure execution and simulation model correlation using ift on risc-v,” *iiScience*, 2021.
- [25] B. Thakar, S. Geraldine, and F. Saqib, “Multi granular level based ift model for risc-v,” *GLSVLSI*, 2021.

- [26] A. Bradbury, G. Ferris, and R. Mullins, “Tagged memory and minion cores in the lowrisc soc,” *Technical report, Computer Laboratory, University of Cambridge*,, December 2014.