12-2019

Secure Embedded Computer Design using Emerging Technologies

Md Badruddoja Majumder
University of Tennessee

To the Graduate Council:

I am submitting herewith a dissertation written by Md Badruddoja Majumder entitled "Secure Embedded Computer Design using Emerging Technologies." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Garrett Rose, Major Professor

We have read this dissertation and recommend its acceptance:

Nicole McFarlane, Jinyuan Sun, Hoon Hwangbo

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Secure Embedded Computer Design using Emerging Technologies

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Md Badruddoja Majumder

December 2019

I would like to dedicate this work to my parents who always encouraged me for higher studies and gave every support in the way.

Acknowledgments

I would like to thank my Advisor, Dr. Garrett Rose for his guidance and support throughout my whole graduate studies. I also would like to thank Dr. Nicole McFarlane, Dr. Jinyuan Stella Sun and Dr. Hoon Hwangbo to be in my Ph.D. committee.

I am very grateful to Dr. Md Sakib Hasan for his valuable suggestions regarding my research. I would like to acknowledge all of my labmates: Mesbah, Aysha, Gangotree, Sherif, Sagar, Adnan, Ryan, Nick and Sam for their all along support.

Abstract

Security is one of the major design criteria for modern computing system. Computing systems are becoming more compact and connected day by day. Due to the large scale connectivity, various security threats are causing failure to confidentiality, integrity, availability and many other basic security criteria. On the other hand, the compact devices used in the embedded computing systems can hardly afford resource hungry security techniques. As improvements in conventional technologies has almost come to a saturation, many emerging technologies are drawing researcher's attention in this regard. This work proposes design techniques using emerging technologies in order to find more comprehensive security solutions for embedded computing system. Two major part of conventional Von-Neuman or Harvard architectures are memory and processing unit which requires individual attention for security against various threats. Chaotic oscillator based logic and RRAM based memory design are explored in this work to mitigate different existing security vulnerabilities with significantly lower overhead. Proposed design techniques are applied in a RISC-V microarchitecture to ensure memory integrity and enhance the security against unauthorized code execution and instruction reverse-engineering based on side channel power attack. Security of the proposed design techniques are found to be at the desired level while consuming a very low overhead as compared to existing mitigation techniques against the same set of vulnerabilities.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Security is a required design criteria in modern computing system. Security threats are expanding in a similar pace with the advancements in computing technologies. A computing system is vulnerable to various threats in both of its production and consumption phases. Reverse engineering, hardware trojan insertion, counterfeiting etc. are some of the security concerns for computing hardwares while being manufactured in outsourced and untrusted fabrication facilities [49]. There are also numerous ways a computing system can be attacked in its usage phase leveraging both hardware and software related vulnerabilities. Security aware designs are therefore necessary to minimize the vulnerabilities.

1.1 Motivation

Secure computing starts with the security of underlying computer architecture. Traditional security solutions often require high implementation cost. Specially, an embedded computing system cannot afford such solutions due to limited resources. Besides, most of the solutions are very specific to a particular vulnerability. More comprehensive solutions within the cost budget are required in order to mitigate the vulnerabilities.

Emerging technologies are being explored by the researchers to design more efficient computing systems [2]. Emerging technologies bring more opportunities to solve existing design problems within limited resource. However, security applications of these technologies have not been explored much yet. In this work, two emerging technologies, chaos computing and memristive memory have been explored in order to find more comprehensive security solutions for computer architecture [71, 15, 16, 57, 93].

Chaos computing is an unconventional method for computing where a chaotic oscillator can be used for computation. A chaotic oscillator governed by nonlinear dynamics generates large number of patterns which are sensitive to initial condition. Reconfigurable Boolean functions can be implemented using chaotic oscillators. Reconfigurability from chaos logic is not only limited to having different functionalities from a single device. Rather, each function itself is reconfigurable in such a way that each configuration causes unique physical characteristics such as power consumption, delay etc. Another aspect of the chaos computing is that minor process variations cause significant differences in device to device behaviors due to chaotic evolution. All these features can be leveraged in designing a secure computer processor capable of mitigating various security threats and can be applied in embedded systems with low resource budgets.

Designing secure memory is another important part of secure computer architecture. In this work, memristive memory has been explored to design a memory with an integrated integrity checking scheme. In a typical integrity checking method, a tag is generated from memory data during each write operation and saved. During the next read operation, the tag is regenerated and compared with the last saved tag. Any mismatch between the two tags indicates an alteration in the memory data. Conventionally, an external tag generating module is used to generate tag from the memory data which often requires high overhead in terms of area, power consumption, delay and prohibitive for embedded system applications. In this work, an integrity checking method has been developed based on the Resistive Random Access Memory (RRAM) where the memory itself can be used as the tag generating function. This tag generation method leverages the sneak path currents in a crossbar RRAM. Sneak path currents refer to the current flow in the unselected cells in a crossbar architecture of passive circuit elements. Due to the sneak paths, current flow in a read operation is dependent on each cell in the memory which can be leveraged to generate a tag for the memory data. However, sneak path currents affect the reliability of regular read and write operations. In order to mitigate this problem, 1T1R structure is considered for the RRAM where sneak path currents can be controlled using the transistor of each cell.

1.2 Research Goals

This work aims at providing more comprehensive security solution to traditional computing architecture for embedded system using emerging technologies. Two major parts in Von-Neuman or Harvard computer architectures are the memory and processing unit. The work is organized into two major parts: i) designing a secure memory using crossbar RRAM ii) designing a secure processing unit using chaos logic. These designs are applied in a RISC-V microarchitecture in order to develop a security enhanced embedded computer.

The goal of the secure memory design work is to design a memory having an integrated integrity checking protocol. 1T1R memristive crossbar array is considered as the memory technology for this work. The main part of the integrity checking protocol of this work is the tag generation method. Tag is computed from the memory data by using sneak path currents in the memristive crossbar array. Generated tag needs to meet certain statistical properties for ensuring the security of the integrity checking protocol. An analytical model of the tag generation is developed in order to guide a designer to find appropriate design parameters meeting the security goals.

Second part of this dissertation is designing a secure processing unit using chaos based reconfigurable logic gates. Chaos logic gates are developed using the state-of-the-art chaos computing research. Chaos logic can be reconfigured into different Boolean functions depending on certain control parameters named as chaos configurations. The same function can be realized using different chaos configurations. Each of these configurations in the chaos gate results in different physical characteristics (power, delay) from each other. An obfuscated arithmetic logic unit (ALU) is designed using chaos logic gates where each copy of the device uses a unique chaos configuration. The resiliency of this design against side channel power template attack is analyzed in this work. Power templates built from the instructions of an ALU can not be used successfully to classify the instructions of an ALU on another chip because each chip exhibits unique power profile for each instruction due to the process variation and differences in configurations. Different classification algorithms are used in order to demonstrate the attack on the chaos based ALU design.

The design of the secure memory unit and the arithmetic logic unit is incorporated in a RISC-V microarchitecture in order to design a logic obfuscated processor which can verify memory integrity in an efficient way and provides security against unauthorized code execution and side channel power template attacks.

1.3 Dissertation Overview

The dissertation is organized as follow. Chapter 2 provides relevant backgrounds on the RRAM memory and chaos based logic technology. Design of an RRAM based secure memory unit with an integrated integrity verification protocol is described in Chapter 3. Chapter 4 describes the chaos based reconfigurable logic design methodology and its application in design obfuscation. Design obfuscation in this section involves both functionality and power traces of ALU instructions. Implementation of a secure RISC-V microarchitecture using the RRAM memory and chaos logic based processing unit is presented in Chapter 5. Finally, the dissertation is summarized along with some future prospects of the work in Chapter 6.

Chapter 2

Background

Designing embedded computers are challenging due to the constraints in resources. Besides, embedded computing systems are connected to large scale networks with sensors, other computing devices and various output devices. Due to large scale connectivity, an embedded computer is vulnerable to a large number of security vulnerabilities. Adversaries target both memory and processing unit which are the two basic parts of traditional Von-Neuman or Harvard computer architectures (Fig. 2.1) used in embedded computing. Existing mitigation techniques require significant amount of resources. They are also very specific to mitigating a particular type of vulnerability. In this dissertation, unconventional design techniques using emerging technologies are applied for designing an embedded computer in order to enhance security against several attacks with significantly lower overhead.

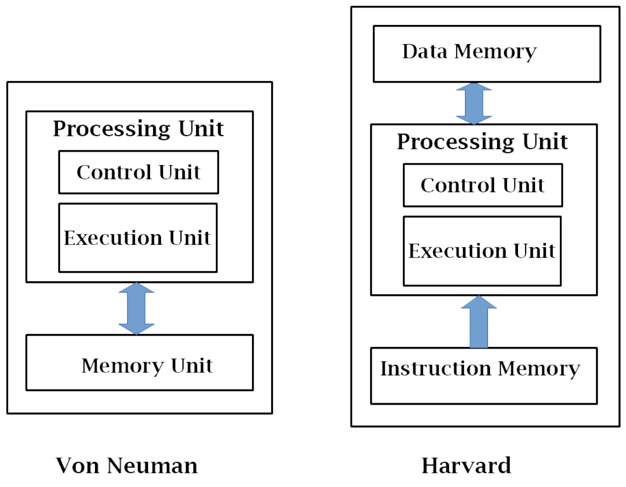


Figure 2.1: Traditional computer architectures.

A microarchitecture with enhanced security features for embedded computing platform is developed in this work. The microarchitecture is designed for a RISC-V instruction set [80, 28, 47]. Security enhancing measures for the design of memory and processing unit are applied to the developed microarchitecture. The RRAM technology is used for the design of memory. The processing unit is secured by applying chaos computing technology. In this chapter, necessary backgrounds are provided on RISC-V instruction set, vulnerabilities in memory and processing units of an embedded computer, RRAM and chaos computing.

2.1 RISC-V

RISC-V is an open source instruction set architecture (ISA) developed at The University of California, Berkeley for computer architecture research. RISC-V ISA is built using the basic features of reduced instruction set computer (RISC). It has a significant impact on the computer architecture research as there have been very few open source instruction sets earlier. Researchers from a variety of computing platforms use RISC-V and explore new opportunities to the existing computer architectures. RISC-V is being widely used in research toward lightweight and resource constrained computing in the embedded systems and IoT edge devices [80, 28, 47, 30, 46].

RISC-V is very flexible for a variety of microarchitectural implementations in different hardware platforms such as fully custom, ASIC, FPGA [30, 46]. The RISC-V ISA has multiple extensions to the base instruction sets. In addition, base instruction sets have different variants for 32, 64 and 128 bit which represents the width of the data registers. There are 2 base sets for 32 bits known as RV32I and RV32E. RV32I is the base set for integer instructions. RV32E is the same as the RV32I except for the number of registers supported. RV32E is used mainly for embedded applications. RV32I supports 32 registers where RV32E causes an illegal instruction exception if the register number is chosen between 16 and 31. 32 and 64 bit base instruction sets of RISC-V are known as RV32I and RV64I, respectively. Along with the base integer instructions, it also supports single, double and quad precision floating points with 'F', 'D' and 'Q', respectively. Decimal floating points are supported with the 'L' extension. Multiplication/division has been excluded from the base set which is supported with the ’M‘ extension. There are several other extensions named as ’A‘, ’C‘, ’B‘, ’J‘, ’T‘, ’P‘ ’V‘ and ’N‘ which supports atomic instructions, compressed instructions, bit manipulation, dynamically translated languages, transactional memory, packed-SIMD instructions, vector operations and user level interrupts, respectively. Parallel computing structures with multi cores are supported by the RISC-V. This work focuses on embedded computing for which the RV32E ISA is recommended [84]. However since the RV32I is same as the RV32E except for the register file size, here only the RV32I base instruction set is described.

2.1.1 RV32I

There are 4 core instruction formats in the RV32I base ISA [84]. The instructions are categorized as R-type, I-type, S-type and U-type. In addition to these 4 core types, there are two more variants for the instruction formats, B and J type based on the structure of the immediate data field. The instruction format is given in the Table 2.1. Detailed user level instruction set with the opcode and extended opcode (funct3, funct7) values are listed in Table 2.2. R-type instructions perform operation on the operands corresponding to the data of two source registers, rs1 and rs2. The result of the operation is loaded into the destination register, rd. The I-type instructions are very similar to the R-types except that one of the operands is a constant and comes from the 12-bit long immediate field of the instruction.

Control transfer instructions are of several types. Unconditional jump instructions JAL and JALR uses the J-type and I-type instruction encoding. The JAL instruction executes unconditional jump operation by loading the target address to the program counter used as the address of the instruction memory. The JALR instruction causes a program to jump

Table 2.1: Instruction format of RV32I instruction set [84].

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Type
funct7	rs2	rs1	funct3	rd	opcode	R
imm[11:0]		rs1	func3	rd	opcode	I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S
imm[12][10:5]	rs2	rs1	funct3	imm[4:1][11]	opcode	B
imm[31:12]				rd	opcode	U
imm[20‖10 : 1‖11‖19 : 12]				rd	opcode	J

Table 2.2: User level instruction subset of RV 32I [84].

Instruction	31:25	24:20	19:15	14:12	11:7	6:0
LUI	imm[31:12]				rd	0110111
AUIPC	imm[31:12]				rd	0010111
JAL	imm[20‖10 : 1‖11‖19 : 12]				rd	1101111
JALR	imm[11:0]		rs1	000	rd	1100111
BEQ	imm[12‖10 : 5]	rs2	rs1	000	imm[4:1‖11]	1100011
BNE	imm[12‖10 : 5]	rs2	rs1	001	imm[4:1‖11]	1100011
BLT	imm[12‖10 : 5]	rs2	rs1	100	imm[4:1‖11]	1100011
BGE	imm[12‖10 : 5]	rs2	rs1	101	imm[4:1‖11]	1100011
BLTU	imm[12‖10 : 5]	rs2	rs1	110	imm[4:1‖11]	1100011
BGEU	imm[12‖10 : 5]	rs2	rs1	111	imm[4:1‖11]	1100011
LB	imm[11:0]		rs1	000	rd	0000011
LH	imm[11:0]		rs1	001	rd	0000011
LW	imm[11:0]		rs1	010	rd	0000011
LBU	imm[11:0]		rs1	100	rd	0000011
LHU	imm[11:0]		rs1	101	rd	0000011
SB	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
SH	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
SW	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
ADDI	imm[11:0]		rs1	000	rd	0010011
SLTI	imm[11:0]		rs1	010	rd	0010011
SLTIU	imm[11:0]		rs1	011	rd	0010011
XORI	imm[11:0]		rs1	100	rd	0010011
ORI	imm[11:0]		rs1	110	rd	0010011
ANDI	imm[11:0]		rs1	111	rd	0010011
SLLI	0000000	shamt	rs1	001	rd	0010011
SRLI	0000000	shamt	rs1	101	rd	0010011
SRAI	0100000	shamt	rs1	101	rd	0010011
ADD	0000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
SLT	0000000	rs2	rs1	010	rd	0110011
SLTU	0000000	rs2	rs1	011	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
SRL	0000000	rs2	rs1	101	rd	0110011
SRA	0100000	rs2	rs1	101	rd	0110011
OR	0000000	rs2	rs1	110	rd	1100011
AND	0000000	rs2	rs1	111	rd	1100011

to the called procedure (subroutine) and saves the address of the next instruction to the destination register, rd.

Conditional branch instructions follow the B-type encoding. B-type instructions compare between two source registers, rs1 and rs2. The target address is calculated by adding the offset from the immediate field to the current program counter value. The program counter loads the target address value if the branch condition is found to be true.

Memory instructions are categorized into load and store instructions. Load and store instructions are used for the memory read and write operations, respectively. Load instructions follow the I-type encoding. The memory address for loading the data from is calculated by adding an offset with the source register, rs1. On the other hand, store instructions have a S-type format where the address for the memory is the addition of the source register, rs1 and the offset while the data to be written is in the source register, rs2.

There are several other instructions in the RV32I which require privileged access as listed in Table 2.3. Control and status register (CSR) instructions are a privileged instruction type which follows the I-type encoding. The immediate field of the I-type format are used as the CSR number in this type of instructions. RV32I also provides some timer instructions such as rdcycle, rdtime and rdinstret which are associated with execution time programs in terms of number of cycle, absolute time etc. ECALL and EBREAK are two other privileged instructions used for communicating with the operating system. For multi core operations,

Table 2.3: Privileged instruction subset of RV32I [84].

[31:28]	[27:24]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Instruction
FENCE	0000	pred	succ	00000	000	00000	0001111
FENCE,I	0000	0000	0000	00000	001	00000	0001111
ECALL	000000000000			00000	000	00000	1110011
EBREAK	000000000001			00000	000	00000	1110011
CSRRW	csr			rs1	001	rd	1110011
CSRRS	csr			rs1	010	rd	1110011
CSRRC	csr			rs1	011	rd	1110011
CSRRWI	csr			zimm	101	rd	1110011
CSRRSI	csr			zimm	110	rd	1110011
CSRRCI	csr			zimm	111	rd	1110011

RV32I supports FENCE instructions in order to manage scheduling the memory and I/O access by different processing cores.

The goal of this work is to augment the security of a RISC-V embedded processor by applying lightweight and comprehensive security solutions to the design of memory and processing units. An RRAM technology is chosen for the secure memory design for the work and chaos based logic design is used in the design of secure processing unit. Necessary backgrounds for RRAM and chaos computing are provided here.

2.2 Relevant Attacks and Existing Mitigations

2.2.1 Attack on Memory

Memory is one of the major parts of a conventional computer architecture. A computer program uses memory access instructions such as load/store quite frequently. Security of a program execution largely depends on the integrity of memory data. There are various ways a memory can be subject to unauthorized modification both in runtime and offline. Runtime attacks are usually software based attacks where memory is manipulated by a malicious program. Low level programming languages such as C/C++ facilitate direct memory manipulation which allows an evil software to cause unauthorized modification to the memory. Stack overflow is such an attack where data is stored into the memory for a number of times that adjacent memory segment that contains sensitive information is overwritten [13]. Direct memory access (DMA) is another runtime memory attack [66]. DMA is a feature found in modern computers that enables some special peripherals to access the memory, bypassing the operating system's supervision. DMA helps faster processing by enabling an I/O to communicate directly with the memory while the processor can perform other tasks. However, this feature can be leveraged to attack a memory with unauthorized modification.

Memory can also be attacked offline which refers to modifying memory content while the memory is not interacting with the legitimate processor. Evil Maid attack is a relevant example where an adversary attacks a powered off computer [74]. In this attack, the boot loader program stored in the non-volatile main memory is modified maliciously using an external device. The objective of this attack is to gain sensitive information from the login attempt of the authorized user using the compromised boot loader. This attack is analogous to an event where an adversary is in the disguise of a maid enters a hotel room and manipulates the boot loader program of the unattended computer left by the guest.

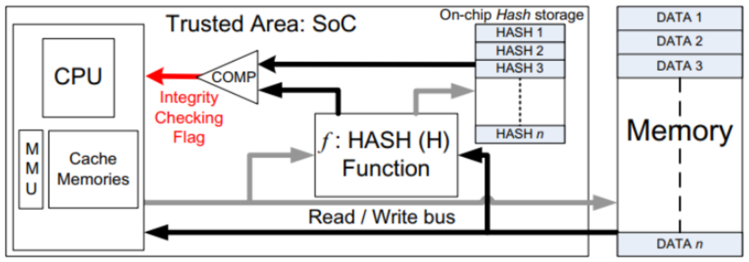
2.2.2 Mitigation Techniques for Memory Attack

General strategy for integrity checking is to generate a tag for the data to be stored in the memory as shown in the Fig. 2.2. Integrity of the data is verified during the read operation by comparing the regenerated tag with the saved one. Tag is a compressed representation of an arbitrary sized data which is usually based on cryptographic hash function, message authentication code (MAC) or block ciphers [61, 27, 32, 52, 21, 94, 73]. In order to avoid overwhelming implementation cost associated with the tag generation using standard crypto primitives, more lightweight approaches have been proposed for resource constrained applications [33, 51].

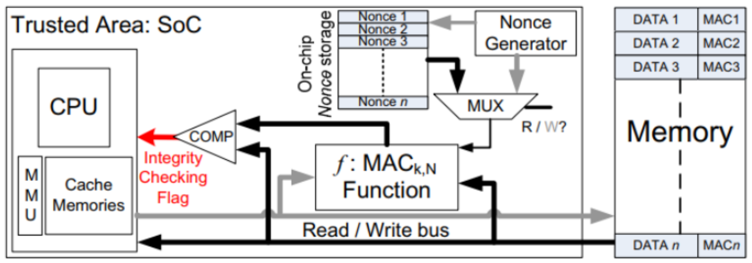
2.2.3 Attack on Embedded Processor

Computer processors are subject to a variety of attacks. Leveraging seemingly inevitable loop holes of the underlying architectures, an adversary can run malicious programs on a target machine which can affect all of the system components connected to the processor. Specially, when it is an embedded system, there are a very large number of sensitive components. Unauthorized code execution can affect confidentiality, integrity and availability of the information associated with these components in an embedded system. Return oriented programming (ROP) is leveraged by an attacker in order to take control of the instruction pointer and execute arbitrary codes on a processor [35, 68]. There are also Use-after-Free attacks where a freed memory is accessed in order to crash a program or execute an arbitrary code [76].

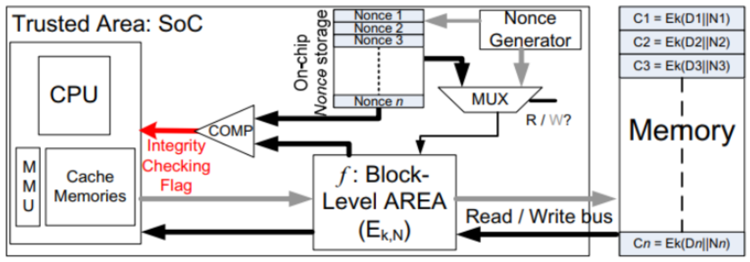
Side channel attacks are another type of attacks exploiting hardware specific implementation loop holes of software. Machine codes of an embedded processor can be reverse



Hash based



MAC based



Block cipher based

Figure 2.2: Existing mechanisms used for memory integrity verification [21].

engineered by using common side channel characteristics such as power, electro-magnetic emanation, timing information etc [63, 82, 69]. For power analysis based instruction reverse engineering, various machine learning classification methods are used on pre-collected power profiles for known instructions. Based on the classification algorithms and training power profiles, nearly 100% instructions of a micro-controller can be recognized [63].

2.2.4 Existing Mitigation Techniques

There are various defense mechanisms on different abstraction levels against the security vulnerabilities of an embedded processor. Architectural level mitigation techniques can prevent such attacks to some extent [85, 67]. However, these mitigation techniques are very specific to a particular implementation. Reconfigurable micro-architectures are proposed to secure a processor from a lower abstraction level [50, 49]. In this method, computing hardwares are logic encrypted using reconfigurable LUT. Executing a code on this platform requires the valid configuration information.

There are also separate defense mechanisms for countering the side channel power attacks. Hiding, masking intermediate values are very common approaches to mitigating different kinds of power analysis attacks. Dynamic logic circuits such as dual rail pre-charge logic, sense amplifier based logic (SABL) etc. can be used in hardware design to mitigate power analysis based attacks [53].

2.3 Emerging Technologies for Secure Computing

Techniques for mitigating attacks on an embedded computer described above exhibits significant amount of resource overheads in terms of power, area and delay. In addition, all these mitigation techniques are very specific to mitigating a particular type of vulnerability. Opportunities in the emerging technologies such as RRAM, chaos computing are explored in this work in order to find more comprehensive security solutions for an embedded processor. Relevant backgrounds for these two technologies are provided in this section.

2.3.1 Resistive Random Access Memory (RRAM)

RRAM is one of the prominent emerging memory technologies. In an RRAM, information is stored as a form of device resistance. Stored information in an RRAM is therefore non-volatile which makes it an attractive candidate for computer memory. The term RRAM is interchangeably used to refer a device called "memristor" [9]. Memristor is an abbreviated form of memory resistor. In order to form an RRAM array, memristors are placed simply between each cross-point of vertical and horizontal metal lines. RRAM thus has the potential to form a high density memory array. Besides, RRAM technology is compatible with the CMOS which keeps it a realistic choice from the perspective of fabrication cost. This section provides some background on memristor, memristor models and RRAM array structures.

Memristor Device Operation

Memristor is a resistive switching device. Metal-insulator-metal (MIM) device stacks usually exhibit property expected from a memristive device. Metal oxide memristors such as TiO_2 or HfO_2 based devices require a filament formation between the two metal layers in order to operate as a switching device. Before the filament formation, the resistance of the device is at a high state governed by the oxide materials. A high bias voltage is required in order to form the filaments. After the filament formation, a memristor can be switched between a high resistance state (HRS) and low resistance state (LRS) by applying appropriate voltage pulse. Current voltage relation of a memristor can be expressed as:

$$v(t) = i(t)M(t) = \frac{d\phi}{dq}i(t), \tag{2.1}$$

where $M(t)$ is the memristance i.e. the instantaneous resistance of the memristor, $q(t)$ and $\phi(t)$ are the charge and flux linkage, respectively. Flux linkage, $\phi(t)$ and charge, $q(t)$ can also be represented as the time integral of voltage and current, respectively:

$$\phi(t) = \int_{-\infty}^{t} v(\tau)d\tau, \tag{2.2}$$

$$q(t) = \int_{-\infty}^{t} i(\tau) d\tau. \tag{2.3}$$

A typical memristor switches by using voltage pulse of a specific magnitude, polarity and pulse width. For a uni-polar memristor, pulse magnitudes required for switching are different while having the same polarity [58]. On the other hand, for switching a bipolar memristor, voltage pulse of the opposite polarity needs to be applied. The HRS and LRS are encoded as the binary data '0' and '1', respectively. The resistive state of a memristor can be read out as binary data by applying a small non destructive voltage and using a sense amplifier.

Memristor Model

Memristors are basically metal oxide switching devices which has the metal-insulator-metal (MIM) structure [29], [77], [9]. Various transition metal oxides are used as the insulating layer in these devices. Several models have been reported for memristor's operation. The HP lab proposed a model based on the ionic drift of oxygen vacancies in TiO_2 memristive devices [79]. The conceptual device structure considered for this model is shown in Fig. 2.3. The oxide layer can be considered as a combination of doped and undoped region based on the amount oxygen vacancies. The doped and undoped regions have different resistances. The width of the doped region changes with the amount of applied bias. I-V characteristics of such devices can be expressed as follows:

$$v(t) = (R_{ON}\frac{w(t)}{D} + R_{OFF}(1 - \frac{w(t)}{D}))i(t), \tag{2.4}$$

where D is the thickness of the oxide layer, $w(t)$ is the width of the doped region, R_{ON} and R_{OFF} represents the resistance of the doped and undoped regions, respectively.

Mc. Donald *et.al.* proposed a model for HfO_2 memristors [60]. This model predicts the instantaneous rate of memristance change with respect to the applied voltage as governed by Eq. 2.5 and 2.6. For a positive bias, instantaneous memristance change is as follows:

$$M(t_{i+1}) = M(t_i) - \frac{\Delta r \Delta t V(t_{i+1})}{t_{swp} V_p}, \tag{2.5}$$

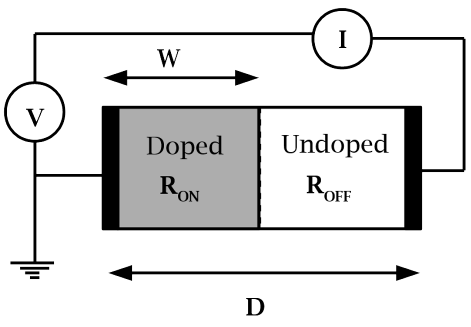


Figure 2.3: Memristor device structure considered for Ionic Drift Model proposed by HP lab [79].

whereas for a negative bias, the memristance is updated by:

$$M(t_{i+1}) = M(t_i) + \frac{\Delta r \Delta t V(t_{i+1})}{t_{swn} V_n}. \tag{2.6}$$

In the above equations, M is the memristance, Δr represents the difference between LRS and HRS, Δt is the simulator step size, V_p and V_n are positive and negative thresholds, respectively, t_{swp} and t_{swn} are the minimum switching time for low to high and high to low switching, respectively. An improved model for the HfO_2 memristors given in Eq. 2.7 have been proposed in [3] which considers resistance saturation and nonlinear resistance change with the applied bias.

$$\frac{dM}{dt} = \begin{cases} -C_{LRS}(\frac{V(t)-V_{tp}}{V_{tp}})^{P_{LRS}} f_{LRS}(M(t)), & V(t) > V_{tp} \\ -C_{HRS}(\frac{V(t)-V_{tn}}{V_{tn}})^{P_{HRS}} f_{HRS}(M(t)), & V(t) < V_{tn} \end{cases} \tag{2.7}$$

The term $\frac{\Delta r}{t_{swp}}$ $(\frac{\Delta r}{t_{swn}})$ of Eq. 2.5 and 2.6, respectively are considered in the term C_{LRS} and (C_{HRS}) of Eq. 2.7. Window function, f for fitting the model to experimental data is as follows:

$$f(M(t)) = \begin{cases} \frac{1}{1+e^{\frac{M(t)-\theta_{HRS}HRS}{\beta_{HRS}\Delta r}}}, & V(t) < V_{tn} \\ \frac{1}{1+e^{\frac{\theta_{LRS}LRS-M(t)}{\beta_{LRS}\Delta r}}}, & V(t) > V_{tp} \end{cases} \tag{2.8}$$

where, θ and β are fitting parameters.

RRAM Crossbar Array

RRAM crossbar array is a high density nanoelectronic structure where RRAM cells are placed at the cross points of horizontal and vertical metal lines. A typical crossbar structure is shown in Fig. 2.4. A memory cell is addressed by decoding a row and column line. In order to set a memory cell to LRS (logic 1), a positive bias greater than a particular threshold is applied. Similarly, a negative bias is applied for resetting the memory cell to HRS (logic 0). For read operation, a small non-destructive voltage is applied across the selected memory cell and the current is sensed using a sense amplifier to determine the stored data.

RRAM crossbars suffer from the current leakage through the unselected cells. Leakage currents through unselected cells are well known as sneak path currents. Unlike a transistor, there is no selector (gate) terminal that can control the current flow through the device. Therefore, an applied bias across a selected memory cell also causes current flow in the paths consisting of unselected cells. Current flow in the sneak path of a RRAM crossbar is shown in Fig. 2.5. Sneak path currents cause write disturbances in a crossbar RRAM. Different writing schemes have been proposed for mitigating the write disturbances caused by the sneak path currents [8, 91]. These schemes mainly focus on the biasing of selected and unselected lines in the crossbar in such a way that minimizes the impact of sneak path currents. Sneak path currents also affect the read operation. Due to the sneak paths, read current is contributed by the selected memory cell as well as unselected cells. Contribution of the unselected cells to the read current becomes overwhelming with the increase of the array size [55]. Reliability of read operation is therefore highly affected by the sneak path currents.

Different array structures have been proposed in order to mitigate the sneak path problems in a crossbar RRAM array [58]. The most popular idea for minimizing the effect of sneak path currents is to use a selector device such as diode, transistor in series with the RRAM cell. Diode only works for uni-polar devices while transistor works for bipolar devices too. The transistors in the unselected cells can be turned off by controlling the gate voltage which helps mitigating the sneak path currents. Memory cell consisting of a resistive switching element and a transistor is known as 1T1R structure. A typical 1T1R

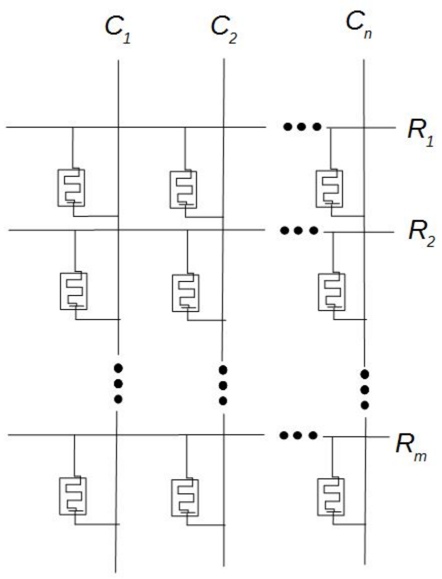


Figure 2.4: Typical RRAM crossbar structure.

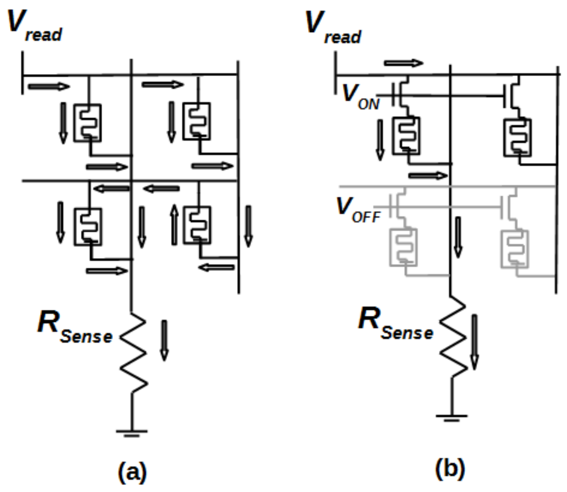


Figure 2.5: Sneak path currents in a crossbar.

crossbar RRAM is shown in Fig. 2.6. Gates of all select transistors in a row are controlled by a common word line (WL) signal. Bias voltage for reading or writing from a particular row is applied to the corresponding select line (SL) connecting the drain terminals of the transistors. Memristors placed in a column are connected using the bit lines (BL). Bit lines can be connected to respective circuitries for read and write operations.

Security Applications of RRAM

In an 1T1R structure, sneak path currents can be controlled by the gate voltage of selector transistors. Sneak path currents can be leveraged to build security applications for RRAM. Due to sneak path currents, write to a particular memory cell affects other unselected cells by changing their resistive states. This method has been proposed to encrypt the data stored in an RRAM crossbar array [39]. Similarly, a single read operation contains information about multiple memory cells due to the sneak path currents. A fault testing method for RRAM memory has been proposed that leverages sneak path currents [40]. Similarly, sneak path currents can be used in order to generate data tag from an RRAM memory [56, 55]. This technique is leveraged in this dissertation for developing a secure memory integrity checking scheme used in a RISC-V processor.

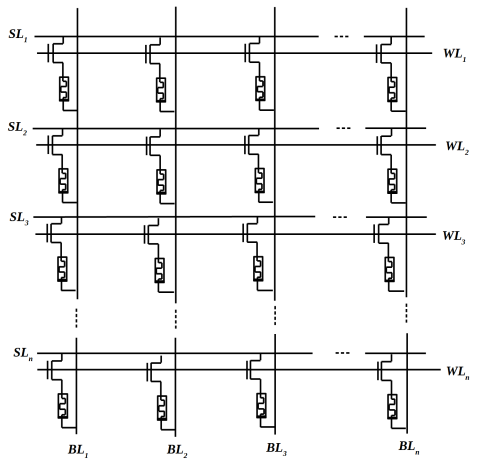


Figure 2.6: 1T1R model for Memristor based Crossbar RRAM.

2.3.2 Chaos Computing

Chaos theory studies the behavior of nonlinear dynamical system that exhibits difficult to predict deterministic behaviors. A chaotic system generates infinitely large aperiodic patterns highly sensitive to the initial condition. Use of a chaotic system in computation is an exciting field of research that can bring totally out-of-the-box solutions to the limitations of conventional computing system. A chaotic system starting from an initial condition evolves into aperiodic patterns over time. Based on the dynamics of the underlying system, chaotic evolution can be of two types: continuous time domain and discrete time domain. An example of chaos in continuous time domain is Chua's circuit [10]. On the other hand, the logistic map and tent map are some of the examples of discrete time chaotic systems [41].

Chaotic Oscillator

Dynamics of continuous time domain chaotic systems are based on time differential equations. Chua's circuit is one of the prominent continuous time chaotic oscillators that have been explored for chaos based computation [59, 7, 65, 65]. Fig. 2.7 shows a Chua's oscillator which can produce chaotic oscillation at the output of the circuit. Chua's oscillator is a 3rd order circuit consisting of two capacitors, one inductor, one resistor and one nonlinear circuit element known as chua's diode which exhibits negative resistance. State equations describing the Chua's circuit are as follows:

$$\begin{aligned} C_1\tfrac{dv_1}{dt} &= G(v_2 - v_1) - g(v_1) \\ C_2\tfrac{dv_2}{dt} &= G(v_1 - v_2) + i_L \\ L\tfrac{di_L}{dt} &= -v_2 \end{aligned} \tag{2.9}$$

There are a wide variety of discrete time domain chaotic oscillators known as chaotic maps. Logistic map, tent map, circle map are some of the examples of chaotic maps. Logistic map is the most well known chaotic maps which has been explored to be used in chaos computing. The logistic map function is expressed as follows:

$$f(x_{n+1}) = rx_n(1 - x_n) \tag{2.10}$$

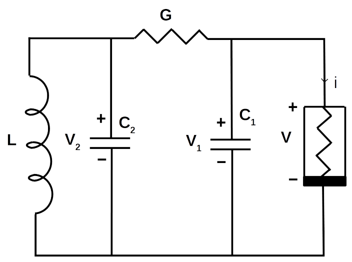


Figure 2.7: Chua's circuit used as a continuous time chaotic oscillator.

Behavior of the logistic map depends on the parameter, r. The function shows chaotic behavior for $3.57 \leq r \leq 4$. For $3 \leq r < 3.57$, it shows a periodic behavior and for $r < 3$, the function eventually converge to a fixed point after few iterations.

Logic Operation from Chaos

The generic architecture used for implementing logic gates from a chaotic circuit is shown in Fig. 2.8 [65, 64]. Input, x to this circuit is a summation of inputs I_1, I_2 of the logic gate and an initial condition, x_0. The output of the circuit, $f(x)$ is converted to a binary value using a parametric threshold value, x^*.

$$x = x_0 + I_1 + I_2 \tag{2.11}$$

$$I_1(I_2) = \begin{cases} 0, & \text{for logic `0'} \\ \delta, & \text{for logic `1'} \end{cases} \tag{2.12}$$

$$y = \begin{cases} 1, & \text{if } f(x) \geq x^* \\ 0, & \text{otherwise} \end{cases} \tag{2.13}$$

Logic functionality of this chaotic gate depends on the underlying dynamics of the circuit and value of parameters, x_0, δ and x^* satisfying the truth table of a particular Boolean function. Conditions need to be satisfied in order to implement different logic functions from a chaotic map are demonstrated in Table 2.4.

A key point regarding the scheme is that the digital inputs are summed to map to an analog value for the input of a chaotic circuit. Here, $I_1 = 0$, $I_2 = 1$ and $I_1 = 1$, $I_2 = 0$ both map to the same input and produce the same output accordingly. Thus, only commutative

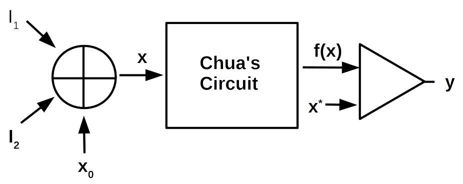


Figure 2.8: Architecture 1 for building logic gate from chaos gate

Boolean functions can be implemented using this method of chaotic gate implementation. Theoretically, maximum number of Boolean functions that can be achieved from a 2-bit input and 1-bit output chaotic gate using the above scheme is $2^3 = 8$. This problem can be resolved using the architecture shown in Fig. 2.9 where a DAC is used instead of addition in order to map the digital input to an analog value. Unlike the other method, all Boolean functions that are theoretically possible in a 2 bit input and 1-bit output can be obtained using this architecture. In this method, digital inputs are mapped to an analog value using an encoder which is fed to a chaotic nonlinear circuit. The nonlinear circuit initialized by the input generates chaotic sequence of voltages. Output of the circuit is decoded into a digital value. This circuit evolves into unpredictable sequence of Boolean functions over time. The encoder can be a conventional digital to analog converter (DAC) where each combination of the digital inputs is mapped to a unique analog value. Size of functionality space in a chaos based logic depends on various parameters controlling the characteristics of chaotic operations. Functionality space increases exponentially with the size of control parameters.

2.4 Conclusion

RRAM and chaos logic are two promising technologies for computing. RRAM technology has been explored by researchers for more than a decade in order to develop next generation memory devices. On the other hand, chaos computing is a fundamental computing idea

Table 2.4: Conditions for realizing different logic functions from a chaotic oscillator using the method shown in Fig. 2.8 [64].

Operations	AND	OR	XOR	NAND	NOT
Conditions	$f(x_0) \le x^*$ $f(x_0+\delta) \le x^*$ $f(x_0+2\delta) - x^* = \delta$	$f(x_0) \le x^*$ $f(x_0+\delta) - x^* \le \delta$ $f(x_0+2\delta) - x^* = \delta$	$f(x_0) \le x^*$ $f(x_0+\delta) - x^* \le \delta$ $\mathrm{f}(\mathrm{x}_0+2\delta) \le x^*$	$f(x_0) = \delta$ $f(x_0+\delta) = \delta$ $\mathrm{f}(\mathrm{x}_0+2\delta) - x^* \le x^*$	$f(x_0) - x^* = \delta$ $f(x_0+\delta) \le x^*$

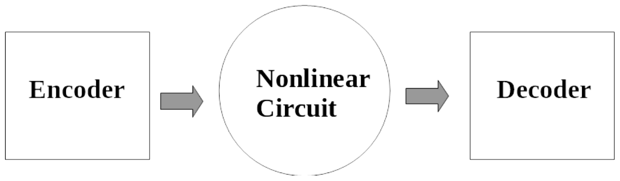


Figure 2.9: Architecture 2 for building logic gate from chaos circuit [45].

based on the mathematical chaos theory for nonlinear systems. These two technologies have also shown significant opportunities for designing security enhanced computing system. Secure design based on RRAM technology can be very lightweight and applicable to resource constrained systems. On the other hand, chaos based designs can provide comprehensive solutions by utilizing a large and complex functionality space. Design for an embedded computers can accommodate these technologies in order to provide security against various threats.

Chapter 3

Secure Memory Design

RRAM is one of the prominent emerging memory technologies. RRAM has shown its promise as the non volatile main memory for a processor due to its high integration, low power and CMOS compatibility. In addition, RRAM exhibits some characteristics that can be leveraged in order to implement security applications for a memory. In this work, we design a crossbar RRAM with a lightweight integrity checking mechanism using the sneak path currents in the crossbar array.

3.0.1 Motivation

Computer memory is vulnerable to various attacks that affect the security of a computing system. Several attack scenarios have been described in Chapter 2 where computer memory is subject to unauthorized modification [13, 66, 74]. Existing techniques use a separate cryptographic tag generation unit for data integrity checking as shown in Fig. 3.1. In a general integrity verification structure, a tag is generated from the memory data using a separate tag generating unit during memory write. Tag represents the memory status in a compressed way. Tag can be generated by using various cryptographic primitives such as hash function, Message Authentication Code (MAC), block cipher etc. Generated tag can either be stored in the memory itself or in some secure dedicated storage inside the processor based on the type of tag generation mechanism. Before accessing a memory location, tag is generated on the existing data and compared against the tag saved in the last

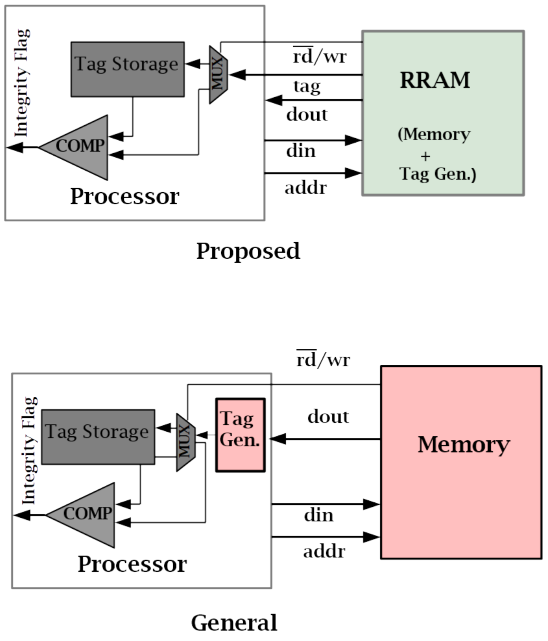


Figure 3.1: Integrity verification protocol for a general method and the proposed method.

write operation. Any mismatch between these two tags raise an exception which indicates unauthorized modification of the memory. In the proposed method, tag is generated by in-memory computation of RRAM crossbar leveraging sneak path currents. Other mechanisms of saving and comparing tag for integrity verification is the same as in a general approach. Due to the in-memory computation based tag generation, this method enables integrity verification with significantly lower overhead.

3.0.2 Security Goals

Unauthorized modification can be unnoticed by the integrity checking system if an adversary is able to modify data in such a way that the modified data generates the same tag. This phenomenon is known as collision. The most important criteria for a secure integrity verification system is collision resistance. Collision resistance refers to the scenario where finding two different data generating the same tag is computationally infeasible. Collision resistance can be both targeted and untargeted.

Targeted collision resistance is also known as second pre-image resistance in cryptographic literatures [48]. For a sample data space, x where the tag of x_1 is t_1, an attacker should find another x_2 with a negligible probability such that $x_2 \neq x_1$ and the tag of x_2 is also t_1. Targeted collision resistance for N-bit tag having an uniform probability distribution is 2^{-N}.

The term collision resistance in cryptographic hash function usually refers to the resistance against untargeted collision. For a secure tag generation, probability of finding any data pair, (x_1, x_2) such that $x_1 \neq x_2$ and both generate the same tag, should be negligible. An ideal tag generation method has a collision resistance of $2^{-N/2}$ for an N-bit tag.

Collision resistance is the primary security goal for a integrity checking system. However, the tag generation must meet several other properties in order to achieve desired security against collision attacks. Three properties namely Uniformity, Avalanche and Diffusion are used here to analyze the security of the proposed integrity checking system.

1. **Uniformity:**

   For a secure integrity verification system, generated tag should follow a uniform distribution. Collision resistance decreases as the probability distribution of tag value lacks uniformity. Collision resistance for an N-bit tag can be expressed as $2^{-\alpha N}$, where α is the measure of uniformity [5].

2. **Avalanche:**

   Avalanche effect is a cryptographic property first used by Fiestel [23]. Generally, a significant change in the output due to a small change in the input is known as avalanche. In an integrity checking system, generated tag should flip around 50% of its bits due to a small change in data. If the tag generation lacks avalanche, resulting tag space due to a small change would be smaller. Hence, chances of collision would be increased.

3. **Diffusion:**

   Diffusion measures the sensitivity of each tag bit to a change in data independently. A cryptographic function is considered well diffused if each bit of the output flips with a 50% probability due to a change in data [90]. This property is similar to the avalanche effect except it measures each bit individually. Diffusion is therefore also known as strict avalanche criteria. Both of these properties are important. We assume a case where the system meets avalanche but lacks diffusion. In this case, it might be possible that the bit flips are restricted to only a subset of the total bits which essentially shortens the attack space. An adversary may observe a certain amount of data-tag pair and find out the responding bits of the tag due to a small change in the data.

3.1 Design of Tag generation

In this work, a novel tag generation method is developed using sneak path currents in the crossbar array of RRAM. Current read from a selected column in a crossbar RRAM is a function of each memory cell given that the sneak path currents are enabled. This data dependency of sneak path current can be leveraged to generate authenticating tag from the RRAM. In this tag generation architecture, a non-destructive voltage is applied to a specific row for reading the memory through multiple columns in order to generate a tag. We assume a $m \times n$ crossbar RRAM as shown in Fig. 3.2. Read voltage, V_R is applied to a row for tag generation. k columns are read using a load resistor, R_L in each column to generate the tag. Load voltage in each of the k selected columns is converted to a l-bit digital value using an ADC which comprises the $k \times l$ bit tag.

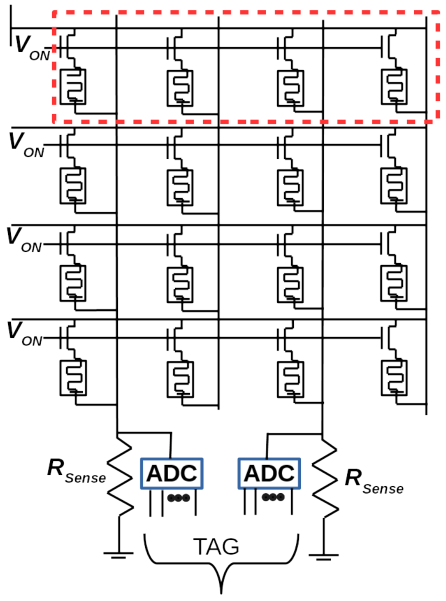


Figure 3.2: Reading multiple columns from a $1T1R$ crossbar RRAM array for tag generation. All select transistors are turned ON in order to enable sneak path currents. Reserved row used for tag generation is indicated by the dotted box [56].

The selected row has a bias on the tag generation process which can be inferred from the tag generation circuit. In order to design a secure integrity checking system, tag generation should not have any bias to a particular memory cell. Those specific memory bits can be a target for integrity attacks. The selected row is therefore reserved only for tag generation in this design and cannot be a part of the regular read write operation. Fig. 3.2 shows the concept of reserved row in a crossbar array of RRAM. The reserved row is randomly configured before each memory write operation. There are two security benefits in using the selected row for tag generation as a reserved row. First one is that the tag generation is not biased towards a particular memory cell. It also improves the avalanche property which refers to the sensitivity of the tag due to a small change in the data. Regardless of the change made in the original memory, reserved row is always changed in each tag generation instant.

3.2 Integrity Protocol

The tag generation is performed on individual memory sections. A memory section consists of multiple memory blocks where each block has multiple words. The same reserved row is used while generating tag from multiple sections of the memory. It is also possible to use individual reserved row for each section which would possibly increase the implementation overhead. Tag is generated after each write operation and saved to a tag storage. Memory access is verified at each read operation by generating the tag from the existing data and comparing with the saved tag. Reserved row is not changed during the tag regeneration before a read operation. Steps of the sneak path current based tag generation can be divided into read and write phases as follows:

Write and Tag Generation:

- Memory cells in the reserved row are written with random bits.
- Regular write operation is performed by keeping the respective select transistors on.
- Tag is generated from the respective memory section and saved in a secure tag storage.

The tag can be stored in the similar way of storing tags in the memory authentication scheme based on a hash function described in [20].

Read and Verification:

- Tag is generated from the current memory section prior to a read operation. Reserved row is unchanged during tag regeneration
- Regenerated tag from a section is compared with the tag saved during the last write
- An exception is thrown if the tags do not match. Regular read operation is performed otherwise.

3.3 Performance Results

3.3.1 Security

Integrity verification protocols are dependent on the security of tag generation. An adversary may try to leverage inadequate collision resistance of a protocol in order to modify the memory. In such cases, the same tag is generated as it was for the previous memory status. Collision resistance is an important requirement for any tag generation protocol [1]. The design goal of a secure integrity checking protocol is maximizing required number of trials in order to find a collision using brute force. This is analogous to the birthday problem of finding two or more students in a classroom having the same birthday [5]. Security against collision based on the birthday attack can be evaluated in terms of collision rate. Collision rate is the minimum number of trials required for a collision in the tags of two arbitrary data.

A MATLAB model for solving the crossbar is used here in order to evaluate the security of this tag generator leveraging sneak path currents. The MATLAB code for the RRAM solver is given in Appendix A.1. The select transistor of the 1T1R cell is modeled as an ideal switch. Nominal resistance values considered for the RRAM cells are $57M\Omega$ and $58K\Omega$, respectively for the OFF (HRS) and ON (LRS) state. For simulation, 20% and 10% standard deviations are considered for the OFF and ON state resistances of the memory cells. Memory is read using a voltage of 600 mV and a load resistor of $58K\Omega$. A 4-bit ADC is considered with a

10 mV resolution for the tag generation. Nominal HRS and LRS values for the RRAM cells used in the simulation are collected from an earlier work [62].

Tag generation using the method designed for this work is dependent on the crossbar dimensions. Crossbars of different sizes have been investigated in order to find an optimal dimension for securing the protocol against the birthday attack. Probability of collision with a particular number of trial is estimated for each of these observations. Crossbar dimension that gives the lowest probability is considered as the optimal design for security.

Fig. 3.3 shows that probability of collision increases with the number of rows and decreases with the number of columns in the crossbar. After a certain point, collision probability increases with the number of columns in a crossbar. An optimal dimension for crossbar can be found based on the analysis of collision probability.

A crossbar RRAM is simulated with the optimal dimension in order to find the collision rate of the tag generator. Different number of trials are considered for finding a collision. Number of trials required on an average for having a collision with 50% probability is the collision rate [5]. Collision rates are estimated for different tag sizes considered in the simulation. Collision resistance can be measured as the inverse of collision rate. Fig. 3.4 shows the plot of collision resistances for an RRAM tag generator for various tag sizes. An ideal tag generator has a collision resistance of $2^{-\frac{r}{2}}$, where r represents the total number of bits in the tag [5]. In order to find the deviation from ideal value, simulated results of collision rates are fitted with the equation: $2^{-\alpha . r}$, where α is a fitting parameter. The best fitting requires an α of 0.40 implying that the collision rate is $2^{-\frac{r}{2.5}}$. Comparison between the two equations for collision rate suggests that the tag generation method considered in this work requires 25% more tag bits in order to achieve the same level of collision resistances as expected from an ideal tag generating function. However, the overhead associated with additional tag bits is compensated by the lightweight implementation of tag generation. In this method, tag is generated by the memory itself while an individual tag generator is used in conventional implementations.

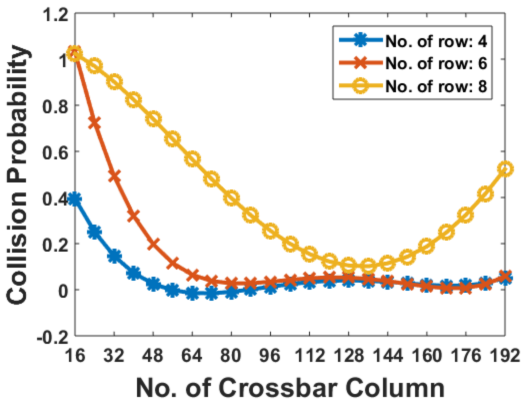


Figure 3.3: Collision probability of a 32 bit tag for different dimension of crossbar. Number of trials considered for finding the collision probability in this experiment is 256. [56]

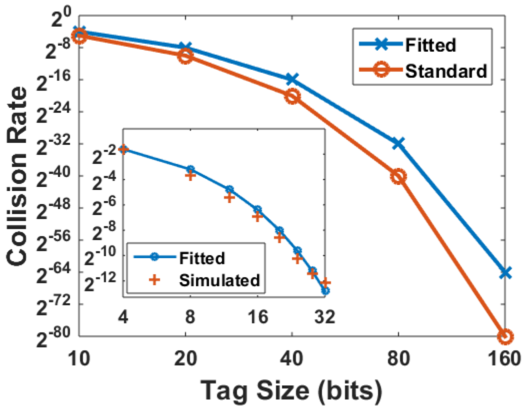


Figure 3.4: Comparison of collision rate with respect to tag sizes for a standard tag generator and the one designed this work. Simulated results for tag sizes up to 32 bits are fitted and interpolated for predicting the collision rate of higher tag sizes [56].

3.4 Analytical model

The tag generation method shown in this work requires more tag bits in order to provide ideal security against collision attacks. This requirement for extra bits is caused by the biased contribution of reserved row memory cells to the sneak path currents. A revised design has been developed towards more secure design with the same number of tag bits as required by an ideal tag generator. In this revised design, only the unselected cells in the reserved row are randomly configured instead of the full reserved row. Selected cells in the reserved row are kept at the OFF states in order to minimize their contribution in sneak path currents.

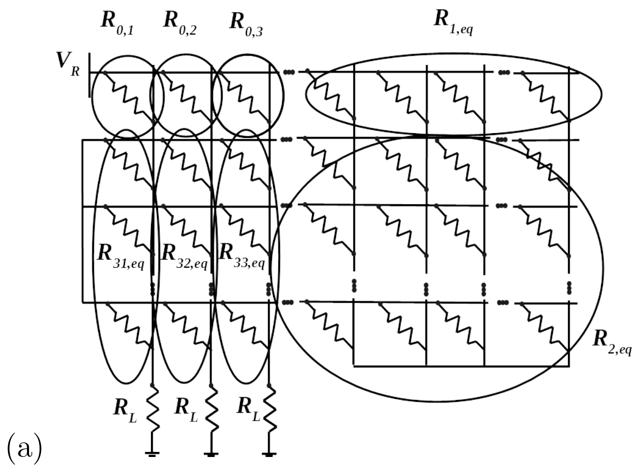
An analytical model is also developed with a view to having design guidelines for more robust tag generation against integrity attacks. Additional architectural modifications are made to the basic tag generation circuit described in section 3.1.

Let's consider the simplified crossbar network shown in 3.5(a) represented in terms of the memory cell resistances. RRAM memory cells can be categorized into 4 types based on their position in the crossbar array. Type 0 and 1 are the memory cells in the selected row connected to the selected and unselected columns, respectively. Similarly, memory cells in the unselected rows connected to the unselected and selected columns are type 2 and 3, respectively. These 4 types are labelled as R_0, R_1, R_2 and R_3, respectively.

In order to completely get rid of the bias of the selected cells in the reserved row, they are always kept OFF. Instead of the whole reserved row, only the unselected cells are reconfigured in this modified design. A Memory cell in the OFF state can be represented as an open circuit if the ON/OFF ratio is sufficiently higher. The selected cells $R_{0,1}$ $R_{0,1}$, $R_{0,2}$,..$R_{0,k}$ in the reserved row are removed from the equivalent circuit of Fig. 3.5(b) since they are always in the OFF state. Now, we can obtain the load voltage, V_i in the i^{th} sampled column using fundamental circuit analysis.

$$V_i = \frac{V_R.G_i.R_L}{1 + (R_{1,eq} + R_{2,eq}).\sum_1^k G_i}, \tag{3.1}$$

where $G_i = \frac{1}{R_{3i,eq}+R_L}$. V_i is converted to a binary value, τ_i by comparing with a reference voltage, $V_R/2$. τ_i is the i^{th} bit of the overall tag. In this section, an analytical model is developed for analyzing the statistical properties of the generated tag required for a secure



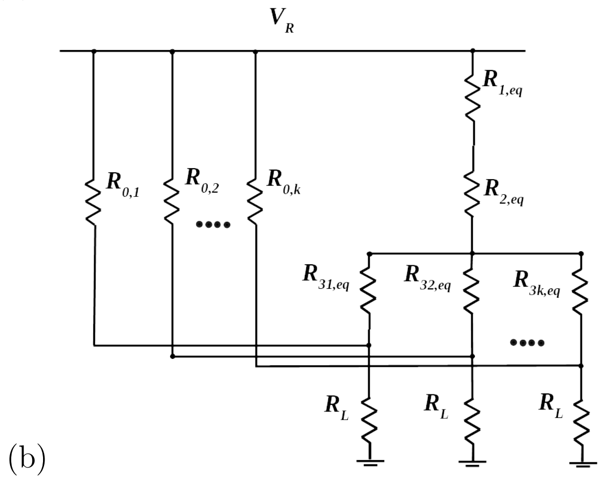


Figure 3.5: (a) Crossbar RRAM architecture for tag generation using multiple columns sampling. Unselected rows and columns are shorted in this architecture. (b) Redrawn crossbar network of (a) considering equivalent resistances for memory cells of the same type [55].

data integrity checking. The model predicts the probability of τ_i being 1 (0) in relation to different parameters such as crossbar size, tag size, load resistance value. Probability of τ_i being 1 can be expressed as follows:

$$Pr(\tau_i = 1) = Pr(V_i \geq \frac{V_R}{2}). \tag{3.2}$$

Substituting for V_i, the inequality $V_i \geq \frac{V_R}{2}$ can be rewritten as:

$$\frac{V_R.G_i.R_L}{1 + (R_{1,eq} + R_{2,eq}).\sum_1^k G_i} \geq \frac{V_R}{2}, \tag{3.3}$$

which reduces to

$$R_{1,eq} + R_{2,eq} \leq \frac{2.G_i.R_L - 1}{\sum_{j=1}^k G_j}. \tag{3.4}$$

$R_{1,eq}$, $R_{2,eq}$, $R_{3i,eq}$ are the equivalent resistances of different cell types defined earlier. Equivalent resistances for each category can be represented approximately as the inverse of total number of ON cells. The load resistance is also expressed as a fraction or multiple of ON resistance.

$$R_{1,eq} = \frac{R_{ON}}{N_1}, R_{2,eq} = \frac{R_{ON}}{N_2}, R_{3i,eq} = \frac{R_{ON}}{N_{3,i}}, \tag{3.5}$$

where N_1, N_2, N_3 represents the number of ON cells in type 1, 2 and 3 memory cells defined earlier; N_L represents the ratio of the LRS and load resistance. A uniform probability density function (p.d.f) for memory data is considered here for calculating the p.d.f of N_1, N_2 and N_3 random variables. Set definitions of N_1, N_2, N_3 are as follows:

$$N_1 = \{x \epsilon Z : 0 \leq x \leq n - k\} \tag{3.6}$$

$$N_2 = \{x \epsilon Z : 0 \leq x \leq (m - 1) * (n - k)\} \tag{3.7}$$

$$N_{3,i} = \{x \epsilon Z : 0 \leq x \leq m - 1\} \tag{3.8}$$

Eq. 3.4 can be rewritten in terms of N_L, N_1, N_2 and N_3.

$$\frac{1}{\frac{1}{N_1}+\frac{1}{N_2}} \geq (\frac{1}{\frac{1}{N_{3,i}}+\frac{1}{N_L}} + \sum_{j=1,j\neq i}^{k} \frac{1}{\frac{1}{N_{3,j}}+\frac{1}{N_L}}).(\frac{N_{3,i}+N_L}{N_{3,i}-N_L}) \tag{3.9}$$

Let X and Y represents the random variables of the left and right hand side, respectively of the inequality shown in Eq. 3.9. P.D.F of X and Y are represented as f_X and f_Y, respectively. From the properties of random variables we get,

$$Pr(X > Y) = \int_{-\infty}^{+\infty} \int_{y=-\infty}^{x} f_X(x) f_Y(y) dx dy. \tag{3.10}$$

$$Pr(\tau_i = 1) = Pr(X > Y) = \int_{-\infty}^{+\infty} \int_{y=-\infty}^{x} f_X(x) f_Y(y) dx dy. \tag{3.11}$$

Now we can find f_X and f_Y in order to calculate $Pr(X > Y)$ using Eq. 3.11.

3.4.1 Evaluating f_X

The range of random variable, X is estimated to be $[0, \frac{(n-k)(m-1)}{m}]$. The upper boundary can be approximated as $n-k$ for a considerably large m. Now, $f_X(p-1 \leq X < p)$ represents the probability of X having a value between $p-1$ and p, where p is an integer between 1 and $n-k$. The condition $p-1 \leq X < p$ reduces to,

$$\frac{N_1 * (p-1)}{N1 - (p-1)} \leq N_2 < \frac{N_1 * p}{N_1 - p}. \tag{3.12}$$

The lower limit of N_2 from Eq. 3.12 can not be larger than the maximum value of N_2, i.e.

$$\frac{N_1 * (p-1)}{N1 - (p-1)} \leq N_{2,max}. \tag{3.13}$$

Eq. 3.13 is simplified as

$$N_1 \geq \frac{N_{2,max} * (p-1)}{N_{2,max} - (p-1)}, \tag{3.14}$$

which eventually reduces to

$$N_1 \geq \frac{(m-1)*(n-k)*(p-1)}{(m-1)*(n-k)-(p-1)}. \tag{3.15}$$

Now, $f_X(p-1 \leq X < p)$ can be estimated as follows:

$$f_X(p-1 \leq X < p) = \sum_{i=r}^{n-k} Pr(N_1 = i) \sum_{u_{1,i}}^{u_{2,i}} Pr(N_2 = j), \tag{3.16}$$

where r is the minimum value of N_1 found from Eq. 3.15.

$$r = ceil(\frac{(n-k)*(m-1)*(p-1)}{(n-k)*(m-1)-(p-1)}), \tag{3.17}$$

$u_{1,i}$ and $u_{2,i}$ can be found from the lower and upper limit of N_2 given in Eq. 3.12.

$$u_{1,i} = ceil(\frac{i*(p-1)}{i-p+1}) \tag{3.18}$$

$$u_{2,i} = \begin{cases} floor(\frac{i*p}{i-p}), \text{ if } floor(\frac{i*p}{i-p}) \leq (m-1)*(n-k) \\ (m-1)*(n-k), \text{ otherwise} \end{cases} \tag{3.19}$$

3.4.2 Evaluating f_Y

Random variable, Y is a function random variables $N_{3,1}$, $N_{3,2}$,..., $N_{3,k}$.

$$Y = (\frac{1}{\frac{1}{N_{3,i}} + \frac{1}{N_L}} + \sum_{j=1,j\neq i}^{k} \frac{1}{\frac{1}{N_{3,j}} + \frac{1}{N_L}}).(\frac{N_{3,i}+N_L}{N_{3,i}-N_L}) \tag{3.20}$$

Y can be represented in terms of 3 separate random variables Y_1, Y_2 and Y_3.

$$Y = (Y_1 + Y_2).Y_3, \tag{3.21}$$

where

$$Y_1 = \frac{1}{\frac{1}{N_{3,i}} + \frac{1}{N_L}} \tag{3.22}$$

$$Y_2 = \sum_{j=1, j\neq i}^{k} \frac{1}{\frac{1}{N_{3,j}} + \frac{1}{N_L}} \tag{3.23}$$

$$Y_3 = \frac{N_{3,i} + N_L}{N_{3,i} - N_L} \tag{3.24}$$

Let f_{Y_1}, f_{Y_2} and f_{Y_3} are p.d.f of Y_1, Y_2 and Y_3, respectively. It can be seen from Eq. 3.22 and 3.23 that Y_1 and Y_2 are functions of $N_{3,i}$ which represents the number of ON cells in the i^{th} sampled column in the RRAM crossbar. $N_{3,i}$ follows a binomial p.d.f which can be evaluated using the set definition of $N_{3,i}$ provided in Eq. 3.8.

$$f_N(x) = \frac{\binom{m-1}{x}}{2^{m-1}} \tag{3.25}$$

f_{Y_1} can be calculated using f_N. Since Y_2 is a summation of Y_1, f_{Y_2} can be estimated from f_{Y_1} using central limit theorem. Let, $\mu = \text{mean}(f_{Y_1})$, $\sigma^2 = \text{variance}(f_{Y_1})$. According to the central limit theorem:

$$f_{Y_2}(x) = \frac{1}{\sqrt{(k-1)\mu}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right). \tag{3.26}$$

Now, we can evaluate f_Y by applying basic algebra of random variables on Eq. 3.21.

$$f_Y(z) = \int_{x=1}^{m-1} f_N(x) f_{Y_2}\left(\frac{z}{Y_3(x)} - Y_1(x)\right) \frac{1}{Y_3(x)} dY_1(x) \tag{3.27}$$

As f_X and f_Y are known, $P(\tau_i = 1(0))$ can be found using Eq. 3.11.

3.4.3 Security based Design Choice

According to the analytical model, the probability distribution of each tag bit depends on 3 factors: i) ratio of total and sampled columns, n/k, ii) number of rows in the crossbar array, m and iii) load resistance, R_L. Probability curves of a tag bit with respect to these factors are illustrated in Fig. 3.6. It can be observed that for a fixed R_L, the $Pr(\tau_i = 1)$ increases with m and n/k. Again, for a fixed crossbar dimension (m, n) and tag size (k), the probability increases with R_L. The rate of change in the $Pr(\tau_i = 1)$ with respect to n/k increases with R_L .

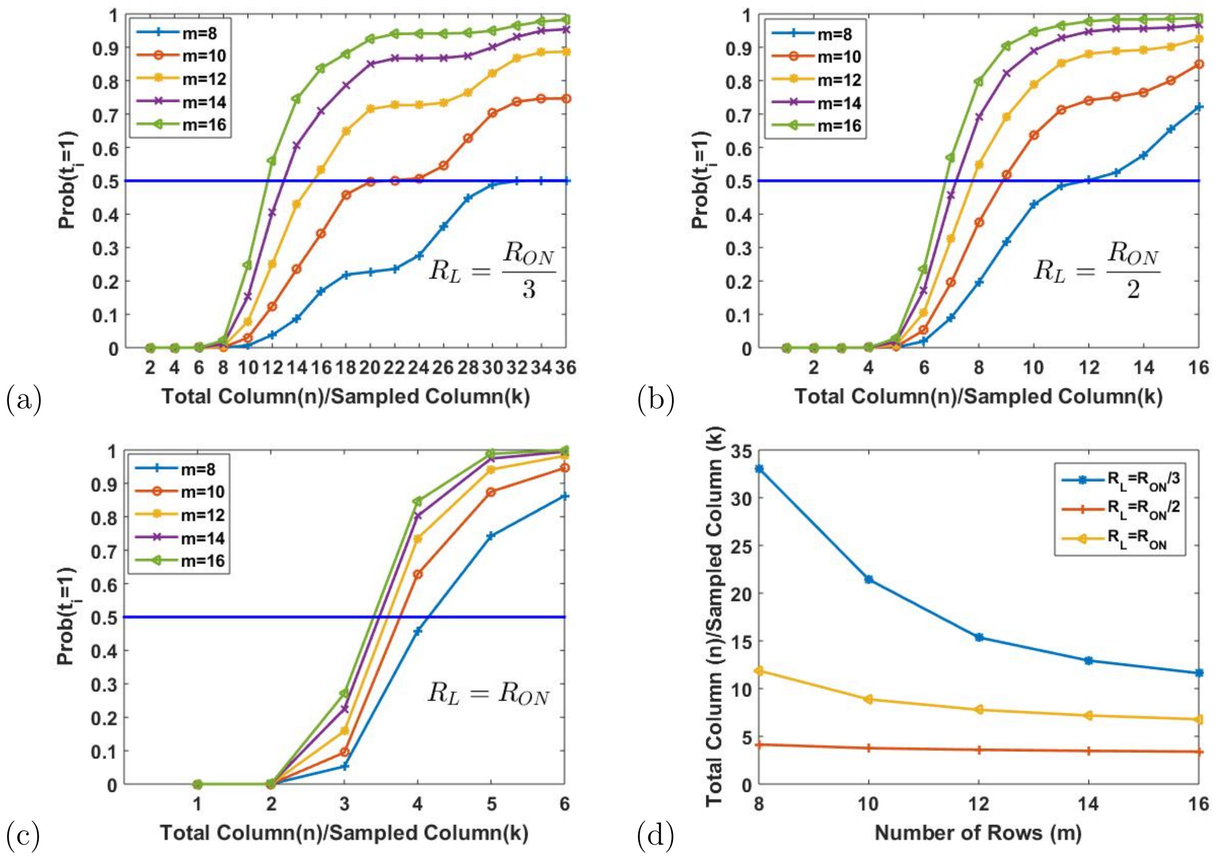


Figure 3.6: (a)-(c): Probability of τ_i being 1 for various dimensions of crossbar array represented as the number of rows in the crossbar and ratio of the number of total columns and sampled columns. Load resistances used in these plots are $\frac{R_{ON}}{3}$, $\frac{R_{ON}}{2}$, and R_{ON} for the sub figure (a), (b) and (c), respectively. (d): Optimal crossbar dimensions for various load resistances required for a uniform probability of τ_i [55].

The necessary condition for an uniform tag distribution is that $Pr(\tau_i = 1) = Pr(\tau_i = 0) = 0.5$. It can be obtained by choosing appropriate values for these 3 factors. Assuming the tag bits are independent in addition to these assumptions, it can be shown that the proposed method meets the security properties discussed earlier. Required parameters for security corresponds to the horizontal line at the 0.5 unit of y-axis in the Fig. 3.6. For better demonstration of the desired design choices, n/k with respect to m for different R_L is plotted in Fig. 3.6(d) corresponding to the 0.5 probability line. Security is analyzed here based on the basic properties: uniformity, diffusion and avalanche instead of the collision rate since the collision is dependent on those properties.

Uniformity

The probability of each tag outcome in a k-bit uniformly distributed tag value is:

$$Pr(\tau) = \frac{1}{2^k}. \tag{3.28}$$

The necessary condition, $Pr(\tau_i = 1(0) = 0.5)$ would also be sufficient in order to having a uniform distribution for the tag if each tag bit is independent of each other. In Eq. 3.20, the random variable Y is a bit specific term, $N_{3,i}$ which represents the number of ON cells in the i^{th} selected column. $P_r(\tau_i == 1)$ is independent due to the term Y. However, there is exception when the independent term, $N_{3,i}$ becomes negligible as compared to N_L in Eq. 3.20.

Diffusion

By choosing appropriate design parameters predicted by the analytical model, it can be ensured that each tag bit flips due to a random change in the data. However, the analytical model does not assume a small change in the data specifically. Modifications are made in the basic tag generation scheme of this work so that each change in data is accompanied by some extra randomness. Added randomness comes from two modifications to the basic protocol during each write operation: i) unselected cells in the reserved row are written randomly ii) set of k columns are selected randomly among the n total columns. Due to these added randomness, data seen by the tag generator becomes scrambled and random even if only a single bit of the data is flipped. Regardless of number of bits flipped in the input, data seen by the tag generator is random which causes each bit of the tag flip with a probability of 0.5.

Let, the tag generated before and after a change made in the data be τ and $\tau^{'}$, respectively. If $\Delta\tau$ represents the hamming distance between τ and $\tau^{'}$, the probability of a tag bit being flipped can be formulated as follows:

$$Pr(\Delta\tau_i = 1) = Pr(\tau_i = 1)Pr(\tau_i' = 0) + Pr(\tau_i = 0)Pr(\tau_i' = 1) \tag{3.29}$$

According to the analytical model, we can achieve $Pr(\tau_i = 1(0)) = 0.5$ by choosing appropriate parameters. Due to the added randomnesses, data is seen as random to the tag generator regardless of the number of bit flips which leads to $Pr(\tau_i = 1(0)) = \frac{1}{2}$. We get $Pr(\Delta\tau_i = 1) = \frac{1}{2}$ by plugging these number in Eq. 3.29. There is exception to the general case discussed above. The second source of randomness is basically data shuffling which does not work for corner cases of data having unbalanced proportion of 0 and 1. For example, tag generation for a data having all 0's or all 1's is affected by only the random reconfiguration of reserved row. Diffusion results may deviate from the expected value for such corner cases. However, this problem can be resolved with a cost of higher overhead by adding multiple reserved rows.

Avalanche Effect

Significant change in the output due to small change in the input is generally known as the avalanche effect. Quantitatively, a system exhibits avalanche effect if half of the total bits flip on an average due to the flip of a bit in the input. A coefficient, σ_h is used here to measure the avalanche effect. It can be calculated from the expected hamming distance (HD) value between the tags generated from two data differing by only a bit flip.

$$\text{HD} = \sum_{i=1}^{k} \Delta\tau_i \tag{3.30}$$

According to the diffusion property, a bit flip in the data flips each tag bit independently with a probability of 0.5. Let f_h be the p.d.f of hamming distance between the tags generated from two data separated by a bit flip. For a k-bit tag, f_h can be found using a binomial distribution.

$$f_h(HD = x) = \frac{1}{2^k}\binom{k}{x} \tag{3.31}$$

The avalanche coefficient, σ_h can be found from the expected value of the hamming distance.

$$\sigma_h = E(x) = \sum_{x=0}^{k} x.f_h(x) = \frac{k}{2}; \tag{3.32}$$

3.5 Circuit Design and Operation

Integrity checking system of this work leverages in-memory tag computation from an RRAM crossbar array. The memory with some peripheral circuitry can be designed in order to facilitate tag generation with a lower overhead. Detailed design and operation of the tag generator is described here.

3.5.1 Design

Circuit design of an 1T1R RRAM with control circuitry for tag generation is shown in Fig. 3.7. Additional control blocks for row and columns are multiplexed with the regular decoders in order to facilitate tag generation. The memory can switch between the regular operation and tag generation by selecting the selector of the multiplexer. The reserved row is connected to a read-write control circuit. Unselected rows (columns) are tied to a high impedance bus in order to implement the tag generation structure described in section 3.1. The transistor in each memory cell is controlled by a transistor control unit which facilities transistor enabling and disabling required for both regular read-write operation and tag generation.

A sampling decoder is used in order to implement the random sampling of k columns out of n total columns. The sampling decoder is fed by a pseudo random number generator (PRNG) to select k of the n outputs. The PRNG is triggered by the regular write signal to generate a random numbers at each regular write operation. Each column is demultiplexed

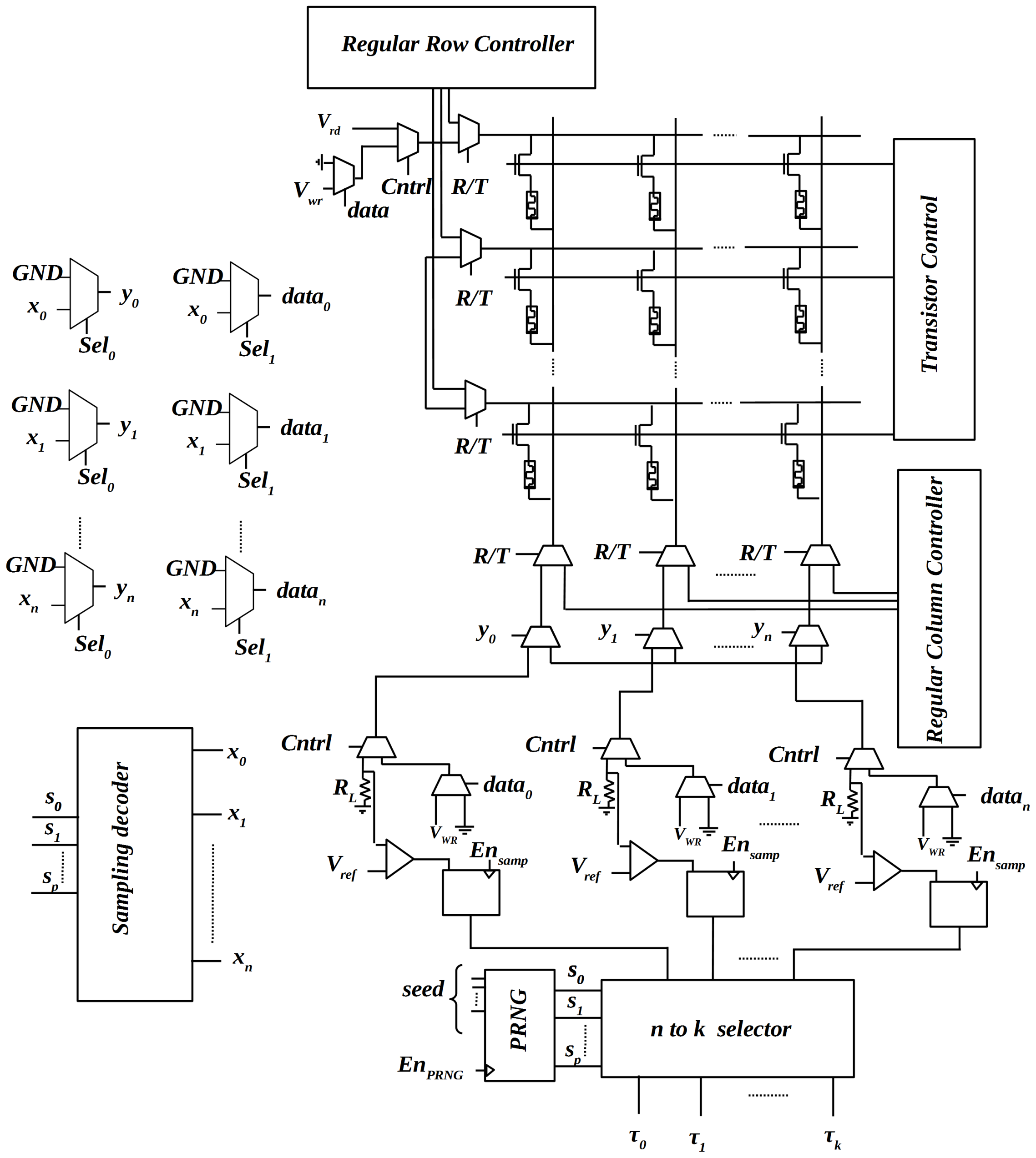


Figure 3.7: Circuit design of a crossbar RRAM with peripheral circuitry facilitating in-memory tag generation. [55].

using the output of the sampling decoder. One output of the demultiplexers goes to individual read-write circuits. The remaining outputs are connected to a common floating node as considered in our previously described tag generation architecture. The column read-write circuit has a load resistor which is used to read the selected columns during tag generation. The binary tag is found by comparing the analog load voltage with a reference voltage. Output of each comparator is saved by an n-bit register. Finally, an $n \times k$ selector is used to choose a k-bit tag from the n outputs of the register.

3.5.2 Operation

The tag generation method developed in this work can be divided into two phases. At first, the PRNG generates a random number which decodes k columns out of n columns. In this phase, type 0 memory cells i.e. selected cells in the reserved row are reset in order to remove added bias on the generated tag as described earlier. The unselected cells (type 1) of the reserved row are written randomly using a two cycle write method proposed for writing a crossbar row efficiently [92]. All cells in a row are reset in the first cycle of the two cycle write method. Only the desired cells are set in the second cycle. RRAM cells such as memristor exhibits stochastic switching behavior [26, 43]. The switching probability in a binary memristor is a function of the width of the applied write pulse. Probability of switching increases with the write pulse width. There is a particular window for the pulse width around which the memristor switches with a 50% probability. In the first cycle of the reserved row write, all reserved row cells are reset with a regular write pulse width. A regular pulse width is chosen sufficiently large that the memory cells switch with a 100% probability certainty. In the second cycle, a set pulse causing a stochastic switching is applied to the unselected cells of the reserved row. The reserved row is thus configured in such a way that the selected cells are written with all 0's and unselected cells are written with random combination of 0's and 1's. This stochastic method of writing is only valid for memristors that exhibit adequate amount of stochasticity. For devices that do not have a stochastic switching behavior, a PRNG can be used to write a random number in the reserved row.

In the second phase of the tag generation, the crossbar array is read through k columns by enabling the select transistors. In order to generate a tag, the load voltage is digitized using a comparator. Half of the supply voltage is used as the reference for the comparator. Comparator outputs are stored in a register at the edge of each tag sampling pulse. k outputs are chosen as the final tag among the n outputs of the register using $n \times k$ selector.

For integrity checking using this method, whenever the interacting processor writes something into the memory, the reserved row is also written with random bits except the selected cells which are all reset. During the write operation, the processor also sends a tag generation signal. The generated tag is then saved in a dedicated storage inside the processor. Whenever the processor reads from the memory after a write operation, it sends the tag generation signal without causing any change in the reserved row or other parts of the memory. Integrity is verified by comparing the regenerated tag with the saved tag. The processor raises an exception if a mismatch is found which indicates that the data has been modified and are not safe for using in the requesting application.

3.6 Security Analysis

Security properties for this tag generation method are evaluated empirically using a MATLAB based resistive crossbar array solver. The solver calculates all node voltages in the crossbar. The solution can be used to calculate the load voltages and hence, the tag. For explaining the model, a $m \times n$ crossbar is considered as shown in Fig. 3.8, where k columns are selected for tag generation. The read voltage, V_{Rd} is applied in the first row for tag generation. V_{R1}, V_{Rcom} are the node voltages corresponding to the selected row and the tied node of the unselected rows, respectively. Similarly, node voltages of the selected columns

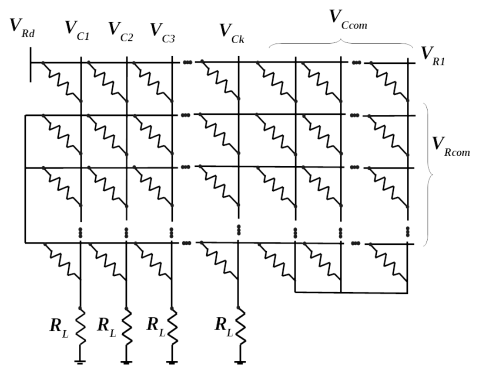


Figure 3.8: Crossbar RRAM considered in nodal analysis based MATLAB model.

are represented as V_{c1}, V_{c2}, V_{c3},...,V_{ck}. The tied floating node of the unselected columns is labelled as V_{Ccom}. G_{ij} represents the conductance of a memory cell connected between row, Ri and column, Cj. The nodal equations can developed from the crossbar circuit by applying Kirchhoff's current law (K.C.L) at each node. The nodal equations can be written as follows using Matrix representation.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{V_R} \\ \mathbf{V_C} \end{bmatrix} = \mathbf{F} \tag{3.33}$$

Description of the matrices used in Eq. 3.33 is as follows:

$\mathbf{V_R : 2 \times 1}$

$$\mathbf{V_R} = \begin{bmatrix} V_{R1} \\ V_{Rcom} \end{bmatrix} \tag{3.34}$$

$\mathbf{V_C : (k+1) \times 1}$

$$\mathbf{V_C} = \begin{bmatrix} V_{C1} \\ V_{C2} \\ . \\ . \\ . \\ V_{Ck} \\ V_{Ccom} \end{bmatrix} \tag{3.35}$$

$\mathbf{A} : (\mathbf{2 \times 2})$

$$A_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = 1 \\ \sum_{x=2}^{m} \sum_{y=1}^{n} G_{xy}, & \text{otherwise.} \end{cases} \tag{3.36}$$

$\mathbf{B} : (\mathbf{k+1 \times 2})$

$$B_{ij} = \begin{cases} -G_{ji}, & \text{if } j = 1, i < k+1 \\ \sum_{x=2}^{m} G_{xi}, & \text{if } j = 2, i =< k+1 \\ \sum_{y=k+1}^{n} G_j y, & \text{if } j = 1, i = k+1 \\ \sum_{x=2}^{m} \sum_{y=k+1}^{n} G_{xy}, & j = 2, i = k+1. \end{cases} \tag{3.37}$$

$\mathbf{C} : (\mathbf{2 \times k = 1})$

$$C_{ij} = \begin{cases} -G_{ij}, & \text{if } i = 1, j < k+1 \\ \sum_{y=k+1}^{n} G_{iy}, & \text{if } i = 1, j = k+1 \\ \sum_{x=2}^{m} G_x j, & \text{if } i = 2, j < k+1 \\ \sum_{x=2}^{m} \sum_{y=k+1}^{n} G_{xy}, & i = 2, j = k+1. \end{cases} \tag{3.38}$$

$\mathbf{D} : (\mathbf{k+1 \times k+1})$

$$D_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ \sum_{y=1}^{n} G_{iy}, & \text{if } i < k+1 \\ \sum_{x=1}^{m} \sum_{y=1}^{n} G_{xy}, & i = k+1. \end{cases} \tag{3.39}$$

$\mathbf{F} : (\mathbf{k+3}) \times \mathbf{1}$ $\mathrm{F}_i = \begin{cases} V_{Rd}, & \text{if } i = 1 \\ 0, & \text{otherwise.} \end{cases}$

Solution of node voltages can be used to find the voltage across the load resistor which eventually determines the tag. Solution of the node voltages of Eq. 3.33 is:

$$\begin{bmatrix} \mathbf{V_R} \\ \mathbf{V_C} \end{bmatrix} = \mathbf{inv}\left(\begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{B} & \mathbf{D} \end{bmatrix} \right) \times \mathbf{F} \tag{3.40}$$

Temperature variability is added to the crossbar solver in order to find how the security properties of generated tags are affected by temperature variation. Temperature coefficient for LRS and HRS variation considered for simulation are $0.004/^{o}C/$ and $-0.008/^{o}C$, respectively [81]. Nominal HRS and LRS are $57M$ and $58K$, respectively [62]. Read voltage is chosen to be $600mV$ for regular read and tag generation. In order to account for variation, 10% and 20% standard deviations are considered for the LRS and HRS, respectively from their nominal values. Several tests are performed to evaluate the security properties of this tag generation method.

3.6.1 Uniformity Test

Uniformity is the measure of balance in a probability distribution. In order to test the uniformity of the statistical distribution of generated tags, k tags are generated from k randomly chosen data. Generated tags are categorized into t equally spaced bins b_1, b_2,.., b_t with frequency nb_1, nb_2,.., nb_t, respectively. Uniformity metric can be defined as follows [5]:

$$uniformity, \alpha = log_t \left[\frac{k^2}{{nb_1}^2 + {nb_2}^2 + ... + {nb_t}^2} \right] \tag{3.41}$$

If the distribution of tag is fully balanced, number of tags in each bin would be equal i.e. $nb_1 = nb_2 = ... = nb_t = \frac{k}{t}$ which results in a uniformity of 1 according to the Eq. 3.41. The uniformity of the proposed tag generation is calculated using a k of 1000 and t of 256.

3.6.2 Avalanche Test

Avalanche test measures the hamming distance between the tags of a random data and and its 1 bit flipped variants. A number (k) of such data pairs are generated randomly and corresponding tag pairs are calculated. Let, H_1, H_2,..., H_k are hamming distances between the generated tag pairs. The hamming distance is represented as the percentage of bit change between the two tags. Average hamming distance is calculated to measure the avalanche effect. The term avalanche coefficient (Av) is used here in order to measure the avalanche effect quantitatively. The avalanche coefficient is 0.5 for a tag generation system having the perfect avalanche effect.

$$Av = \frac{\sum_{i=1}^{k} H_i}{K} \tag{3.42}$$

3.6.3 Diffusion Test

Diffusion test measures the probability of a tag bit being flipped in response to a bit flip in the data. In this test, the same tag pairs generated for the avalanche test are used. However, the average of individual bit distance is measured here unlike the avalanche test. Let, d_1, d_2,..., d_k represents the average bit distances of each bit across k pairs of tags. Diffusion coefficient ($diff$) is evaluated by taking the geometric mean of the the average flip in each bit. Desired diffusion coefficient is 50% for a secure tag generation system.

$$diff = \sqrt[k]{d_1.d_2....d_k} \tag{3.43}$$

3.6.4 Security Results

Optimal design choices for desired security of the proposed tag generation method can be found from the developed analytical model. Such design choices in terms of crossbar sizes, load resistance value, sampled columns to total columns ratio are described in section 3.4.3. In order to validate the analytical model, the proposed RRAM circuit with integrity checking protocol is simulated using the MATLAB based simulation model. Optimal design parameters in terms of crossbar dimensions and load resistance values are considered in the simulation.

According to the analytical model, the integrity checking security properties of the proposed memory is a function of the number of total columns to sampled columns ratio in the crossbar. The number of sampled columns is essentially the tag size because each sampled columns produces 1 bit of the tag from respective load voltages. Optimal design choices can be found in terms of the ratio of the tag size and total columns. Therefore, these findings apply to arbitrary tag sizes given that the number of crossbar columns scale accordingly. For example, the number of total columns to sampled columns (tag size) ratio is 12 for the desired security when the number of row is 8 and load resistance is $\frac{R_{ON}}{2}$. Therefore, the security results hold for any pairs of column size and tag size such as (96,8), (192,16), (384,32) that meets a ratio of 12.

In the design flow of the proposed RRAM, the number of rows, m is chosen first. Corresponding ratio of number of total columns and the tag size (n/k) is found based on the optimal design choices suggested by the analytical model. Finally, desired number of columns is found by multiplying the ratio with the tag size.

Security results are shown in Table 3.1 for all model suggested security driven design choices with a tag size of 8. These results are generated using room temperature.

From the results demonstrated in Table 3.1, it can be seen that the avalanche and diffusion coefficients are close to 0.50 for all of the optimal design choices. However, uniformity deviates from its ideal value 1 when the total number of rows in the crossbar increases for a fixed load resistance value. Uniformity deterioration rate increases with respect to the row size as larger load resistance is used.

This deviation of uniformity for the distribution of generated tag can be explained by the analytical model. The analytical model evaluates the probability of each tag bit being 1(0) individually. It has already been described that for a uniform distribution each tag bit needs to be independent of each other while having a 0.5 probability of being 1(0). The right hand side of the inequality shown in Eq. 3.9 has the term $N_{3,i}$ and N_L. The definition of the terms $N_{3,i}$ and N_L can be found in section 3.4. The term $N_{3,i}$ is the term associated with individual columns and causes each tag bit being independent of each other. However, when the number of rows, m increases, the number of cells in each column increases too and so does the term, $N_{3,i}$. For some row size, the term $\frac{1}{N_L}$ which is a constant dominates over the term $\frac{1}{N_{3,i}}$ and hence Eq. 3.9 lacks independence across the sampled columns. Sample space for the tag is therefore reduced which results in a lower uniformity in spite of each tag bit has the uniform probability of being 1 or 0. This condition arises also when the ratio (N_L) of ON resistance and load resistance (R_L) decreases which consequently makes the term $\frac{1}{N_L}$ dominate over the term $\frac{1}{N_{3,i}}$ in Eq. 3.9.

The results demonstrated in Table 3.1 are generated by considering room temperature in the MATLAB based simulator. The effect of temperature variation on the security properties of the tag generator is estimated by considering two corner temperatures 0^oC and 100^oC. Security results with the temperatures 0^oC and 100^oC is shown in Table 3.2 by considering a load resistance of $\frac{R_{ON}}{3}$.

Table 3.1: Security results represented as uniformity, avalanche and diffusion for optimal crossbar dimensions with different load resistance values. An 8-bit tag (number of sampled columns) is considered in this analysis [55].

R_{LOAD}	Crossbar Size	Uniformity	Avalanche	Diffusion
$\frac{R_{ON}}{3}$	8×264	0.9596	0.5035	0.4870
	10×176	0.9561	0.5011	0.4930
	12×120	0.9602	0.4983	0.5080
	14×104	0.9612	0.4946	0.4860
	16×92	0.9056	0.4948	0.4930
$\frac{R_{ON}}{2}$	8×96	0.9619	0.4983	0.4833
	10×72	0.9583	0.4913	0.4860
	12×60	0.8931	0.4853	0.4900
	14×58	0.7395	0.4945	0.4810
	16×55	0.6264	0.4989	0.4860
R_{ON}	8×34	0.7150	0.5009	0.4760
	10×32	0.5821	0.4951	0.4540
	12×30	0.4751	0.5052	0.4690
	14×29	0.3507	0.5145	0.4610
	16×28	0.3241	0.4931	0.4720

Table 3.2: Effect of temperature variability on the uniformity, avalanche and diffusion properties of the RRAM tag generator [55].

Crossbar Size	Uniformity		Avalanche		Diffusion	
	0^0C	100^0C	0^0C	100^0C	0^0C	100^0C
8×264	0.9562	0.9584	0.4959	0.4988	0.5159	0.4854
10×176	0.9590	0.9681	0.4905	0.5008	0.4994	0.4904
12×120	0.9581	0.9583	0.4984	0.4946	0.4944	0.4931
14×104	0.9577	0.9584	0.4890	0.4996	0.4997	0.4873
16×92	0.8911	0.8963	0.4999	0.5012	0.4994	0.4899

All the security results discussed so far are evaluated for an 8-bit tag. The results are also valid for higher tag sizes if n is scaled up accordingly maintaining a constant n/k. The scalability results are demonstrated in Table 3.3 where $R_L = \frac{R_{ON}}{2}$ and $n/k = 12$. 8, 16, 32 and 64-bit tags are considered for demonstrating the scalability of the proposed tag generation method.

The analytical model can be validated by the simulation results presented in Table 3.3. Model suggested design choices exhibit the desired security in most of the cases. However, few cases yields a security level slightly deviated from the general prediction of the analytical model. Again, such deviations can be explained based on additional assumptions considered in the model. Therefore, design choices made using the analytical model provides a sufficient level of confidence.

3.7 Reliability Analysis

Reliability is an important characteristics of a system to be usable. In this work, tag generation must be reliable in the consecutive reads after a write operation. However, memristor based RRAM cells suffers from a number of different variations that hinders the reliability. Variation sources considered in this work are supply voltage, temperature, load resistance and cycle to cycle variation. In this section, these variability sources and how they affect the reliability of tag generation are discussed.

3.7.1 Variability Sources

Cycle to cycle variation is generally one of the major reliability deterrents in memristor based systems. Memristor exhibits variation in different parameters across set/reset cycles. Since the read operation is performed using a non-destructive voltage, the variation is dominant in only the write operation. A reliable rag generation method requires that generated tags are exactly same over multiple read operations on the same data. Memristor exhibits major variation in one write cycle to another due to the switching between high and low resistance levels. A read operation is less affected by variation as memristor does not switch during a read due to the non-destructive read voltage. Tag generation during a read from the memory

Table 3.3: Scalability analysis of the tag generator by increasing the tag size (number of selected columns) while keeping the same ratio between the number of total columns and selected columns [55].

Tag size	Crossbar Size	Uniformity	Avalanche	Diffusion
8	8×96	0.95	0.49	0.4850
16	8×192	0.95	0.49	0.48
32	8×384	0.96	0.51	0.475
64	8×768	0.96	0.48	0.50

does not involve any switching of memristors. The impact of cycle to cycle variation is therefore negligible.

Memristor shows temperature dependency in its operation like most electronic devices. The specific impact of temperature on device parameters depends on the materials used in the device. For example, with the increase of temperature in a HfO_x memristors, HRS increases and set/reset voltages decreases [22]. Tag generation can be affected by the temperature variation in the proposed integrity checking protocol. Voltage sources exhibit statistical variation in the output. Read voltage fluctuation is another source of variation that is considered in the reliability analysis. From Eq. 3.1, it can be seen that the voltage across each load resistor, V_i is directly proportional to the read voltage, V_R. Therefore, fluctuation in read voltage has a direct impact on the generated tags.

Load resistance is another critical design parameters for the proposed tag generation method. The analytical model described earlier shows that the statistical properties of the generated tag exhibit significant changes for three different load resistances. Thus, variation in load resistance can be a potential factor affecting the load voltage and hence, the tag generation.

3.7.2 Reliability Test and Results

The RRAM tag generator is simulated using a transistor level circuit simulator, Cadence Spectre in order to investigate the reliability. In this simulation, tag from a random data stored in the RRAM is generated for a number of observations. The simulator is provided with a random value within respective distribution for each of the design parameters exhibiting variation. 2% cycle to cycle variation for memristor, 0°C to 100°C temperature range, and 5% variation for load resistance [81] are considered for this variability analysis. The read voltage considered in the simulation is modelled with a supply with 20 mV variation [75]. Generated tag during the write operation is saved as the reference. During reading from the memory, the reference tag is compared with the regenerated tag for integrity verification. Regenerated tags during each read operations before a new write operation is required to be the same for a reliable integrity checking. In the reliability simulation, the percentage of tag mismatch among regenerated tags is used as the measure of reliability. Reliability results are averaged over generated tags corresponding to a number of random data in the RRAM. Table 3.4 demonstrates reliability results for three different sizes of RRAM exhibiting the best security corresponding to each of the load resistance values considered. Results indicate that a load resistance of $\frac{R_{ON}}{2}$ results in higher reliability than designs with other two choices.

In the proposed tag generation architecture, tags are generated by converting the analog load voltage into a binary value using a comparator. It can be inferred that memory states providing less margin for the load voltage as compared to the comparator reference are less reliable. For tag generation, these memory states show high susceptibility to variation. Analog load voltage can be interpreted as the tag bit differently due to the variability of different parameters. However, the reliability of the system can be increased by identifying these states correctly and applying standard error correction schemes. Error correction is often used for other nanoelectronic systems such as physically unclonable functions (PUF) in order to improve reliability [12]. Developing a detailed reliability model relating the impact of design parameters on the reliability of tag generation can be a great future direction based on this work. Analytical model developed here can be modified by including the variability of design parameters. This will enable the designers choose optimal designs with a goal for maximizing both reliability and security.

Table 3.4: Reliability results for the tag generator having an optimal crossbar sizes for three different load resistance values[55].

R_{LOAD}	Crossbar Size	Reliability (%)
$R_{ON}/3$	12×120	83.11%
$R_{ON}/2$	8×96	91.44%
R_{ON}	8×34	83.11%

3.8 Overhead Analysis

Existing memory integrity checking works require operations that are computationally extensive. On the other hand, the method developed for this work uses a single read operation for tag generation leveraging the sneak path currents in the crossbar RRAM. Few comparables to our work proposed in different literatures are chosen here in order to perform a comparative study of overheads incurred by the design implementations [33, 73, 94]. A prototype implementation of the proposed work is compared with these works. For a fair comparison, a 64 bit memory and 8 bit tag is considered for all of these implementations. Table 3.5 shows the implementation cost for the proposed design and other 3 comparables. Cost metrics used in the comparisons of tag generation circuits are energy consumption, transistor counts and delay. A custom prototype is implemented for the design of Hong *et al.*'s work [33] for estimating its overheads. Overhead results for other two comparables [73, 94] are collected from the comparative study presented in [33]. According to the results in Table 3.5, the proposed design shows significant improvement over other comparables considered here. The design of Hong *et al.* is considered as the reference for overhead comparison since it exhibits the least overhead among 3 comparble designs. Proposed design shows nearly a 10× improvement in energy consumption. The improvement in transistor counts and the delay is nearly 2.5× .

The improvements of the proposed design over other existing works can be analyzed by various factors. Data processing for tag generation in the proposed design requires only a simple read operation. On the contrary, multiple rounds of operations such as shuffling, rotation and XOR are performed in the design by Hong. *et al.*. Some extra processing is needed for the proposed design too. However, it is performed only on the columns of the crossbar RRAM instead of the whole memory. In contrast, in other methods, operations

Table 3.5: Cost comparison of the RRAM based tag generator with three existing lightweight tag generation schemes[55].

Overhead	Hong *et al.*	Yan *et al.*	Roger *et al.*	Proposed
Energy (pJ)	96	408	455	9.75
Delay (ns)	50	104	138	20
Transistor count	9358	15340	15340	3456

are performed on each data. Therefore, the proposed design requires significantly less overhead than other existing tag generating circuits considered here for comparison. The delay improvement of the proposed design is also significant. Delay for the tag generation in the proposed method includes only the time of a single read operation and propagation delay of peripheral read write circuitry.

The proposed design has some additional overheads in memory write as the write operation requires additional write in reserved row. In order to minimize this overhead, a cost effective method is proposed for writing the reserved row using only a reset-set cycle of memristor. In addition, this extra overhead is involved only in the tag generation during write to the memory. The step of writing the reserved row is skipped in tag generation during reading. Therefore, the average case delay shows a significant reduction over the delay of other existing designs considered in this comparative study.

3.9 Conclusion

Integrity checking protocol is important in order to design a secure memory system. Most existing solutions use expensive crypto primitives which can not be afforded by many resource constrained systems. Proposed RRAM memory design uses sneak path currents in order to develop a lightweight tag generation scheme. An analytical model has been developed from the proposed design to help designers choose design choices optimal for security. Security properties have been investigated from the circuit level implementation which also comply with the results predicted by the analytical model. This RRAM based memory design with an integrated tag generation protocol facilitates a very cost effective way for verifying integrity of the memory data. This design can easily be incorporated in any microarchitectures in order to develop a secure microprocessor with special load/store instruction that will have default integrity checking features.

Chapter 4

Obfuscated Computing using Chaos Logic

Security has become an indispensable part of the design of modern computing systems. Microprocessors are the most ubiquitous computing element in today's digital world. One has to count for different vulnerabilities while designing a microprocessor. Most existing solutions are specific to vulnerabilities. Mitigating each vulnerability individually requires more resources. In this dissertation, the application of chaotic behavior in a nonlinear system is explored in order to find more comprehensive solutions to existing security vulnerabilities.

4.1 Motivation for Obfuscated Computing

Various security threats in the processing unit of an embedded computer are described in Chapter 2. Unauthorized code execution and side channel power based instruction reverse engineering are two of the major vulnerabilities in existing embedded computers. An adversary can execute malicious code on the target processor affecting confidentiality and the integrity of data processing. Reconfigurability provides a general class of solutions that has been proposed to obfuscate a design in order to prevent unauthorized code execution. On the other hand, various dynamic logic families have been proposed to prevent power analysis attack by reducing correlation between data and the power profiles. Existing mitigation techniques of both types require a significant amount of overhead. Moreover, a single design cannot mitigate both vulnerabilities. The possibility of using chaos based emerging computing technology are explored here in order to provide design obfuscation using a single scheme that can mitigate both vulnerabilities. In this chapter, the design methodology of chaos based reconfigurable logic gate and an obfuscated arithmetic logic unit (ALU) are described. The effectiveness of the proposed design is shown against unauthorized execution and power analysis based instruction classification attack.

4.2 Digital Logic Design using Chaos

A basic design methodology of digital logic using both continuous and discrete time chaotic oscillator has been described in chapter 2. Chua's Oscillator is a continuous time domain chaos generators [59, 7, 65, 65]. On the other hand, logistic map, tent map, circle map etc. are few of the most common examples of discrete time chaos.

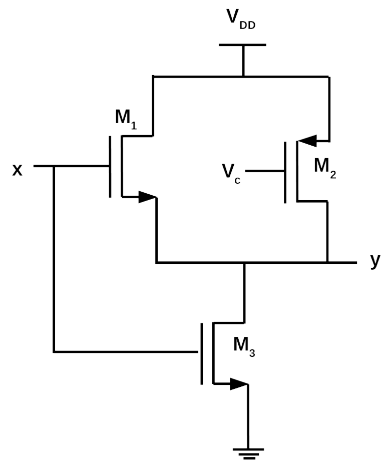
4.2.1 Design Basics

In this work, a 3-transistor circuit proposed by Dudek *et al.* is used as the chaos generator to implement reconfigurable logic functions [18]. The original circuit was designed and fabricated in $0.5\mu m$ process [37]. The circuit is redesigned for this work by scaling down to 65 nm technology as shown in Fig. 4.1(a). The circuit has an input, x, a control, V_c, and an output, y. Fig. 4.1(b) plots the transfer characteristics of the circuit for different control voltages. It can be seen that the transfer characteristics is a V-shaped curve which makes the circuit capable of generating chaos in discrete time. Control voltage, V_c determines the shape of the transfer curve which eventually controls chaotic characteristics. Therefore, V_c plays a major role for ensuring chaotic operation from this circuit.

The Discrete time chaos generators are basically iterated chaotic map functions expressed as follows:

$$f(x_{n+1}) = f(x_n), \tag{4.1}$$

where f is the chaotic map function, x is the input and n is the iteration number. The present output of the map function is used as the input to generate the next output. The

(a)


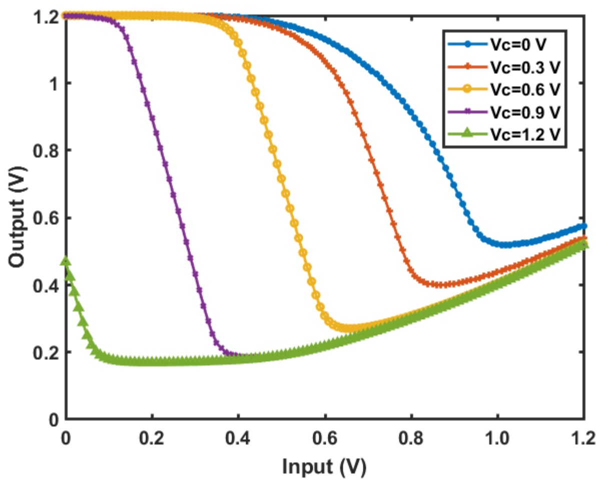
(b)


Figure 4.1: (a): 3-transistor map circuit. (b): Tranfer characteristics of the map circuit.

map function thus generates a discrete sequence of outputs. The circuit structure shown in Fig. 4.2 implements an iterated map using the 3-transistor circuit in order to generate chaotic behavior in discrete time domain. Two analog switches driven by the non-overlapping clocks ϕ_1 and ϕ_2 facilitate the iterative output generation. The output of the map circuit, x_{n+1} is sampled by the capacitor, C_1 at the active phase of ϕ_1. Voltage stored onto C_1 is transferred to the input of the map circuit at the active clock edge of ϕ_2. The current output thus becomes the input for the next iteration and generates a discrete time pattern.

The quality of a chaos generator can be analyzed using bifurcation diagram. The term bifurcation comes from the fact that the period of the output pattern generated from a chaos circuit doubles with the change of a controlling parameter. This period doubling is known as bifurcation. The control parameter controlling this period doubling is called bifurcation parameter. Bias voltage V_c works as the bifurcation parameter for the 3-transistor based chaotic map circuit used in this work. The bifurcation diagram plots all values in the generated output pattern with respect to the bifurcation parameter considering a sufficiently large number of iterations.

The bifurcation diagram for the chaos circuit used in this work is shown in Fig. 4.3. It can be seen from the bifurcation diagram that the periodicity of the output patterns generated by iterated chaotic map doubles gradually with the increase of V_c until the pattern becomes chaotic for some range of V_c. The bifurcation diagram of Fig. 4.3 has two separate chaotic

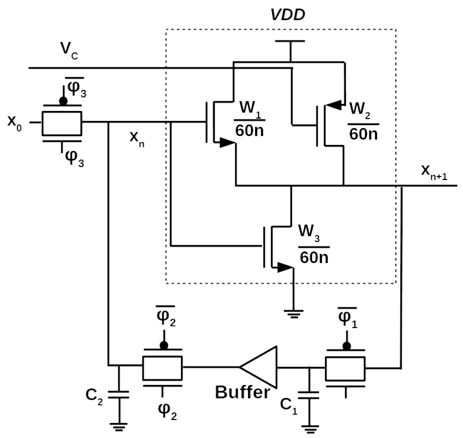


Figure 4.2: Chaos generator using iterated map circuit.

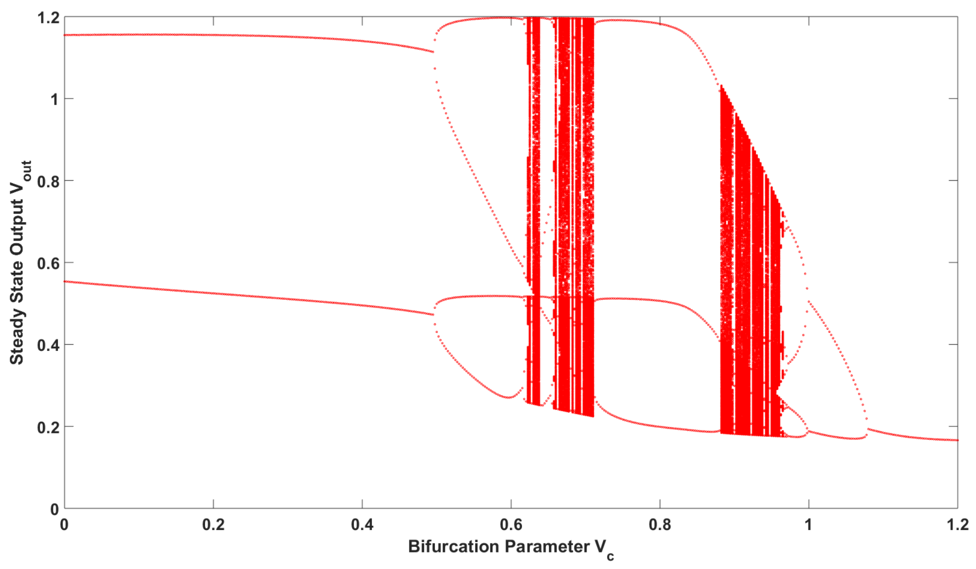


Figure 4.3: Illustration of state bifurcation in the chaos generator circuit considering a geometry of $W_1 = 4.8\mu m$, $W_2 = 0.48\mu m$, $W_3 = 0.48\mu m$.

regions ranging between the periodic zones. The two voltage ranges of V_c that leads to chaotic behavior is found to be 0.62 V-0.73 V and 0.88 V-0.96 V. Range of the outputs from the chaos generator circuit is wider in the first region than the other. The chaos generator must be tuned into the chaotic regions using the bifurcation parameter in order to utilize the complex space of chaotic behaviors.

4.2.2 Design Optimization

Performance of the chaos generator considered in this paper depends on the transistor sizing in the 3-transistor map circuit shown earlier. The design is optimized based on an empirical analysis where performance of the map circuit for different transistor sizing is evaluated and ranked based on a performance metric. The goal of the optimal design is to achieve higher range of V_c allowing chaotic behavior with a lower cost for area, power and delay overhead. The performance metric developed for evaluating different design choices is as follows:

$$PM = \frac{C^{w_c}}{A^{w_a}.P^{w_p}.D^{w_d}}, \tag{4.2}$$

where C represents the range of control voltage for chaotic operation and P, A , and D are the power, area and delay of the 3-transistor based chaotic map circuit, respectively. Individual weight vectors w_c, w_a, w_p and w_d are used in order to choose priority of each factor for evaluating the performance rank using Eq. 4.2. In this work, equal weights of 1 is considered for each factor. Table 4.1 shows the performance metric for 7 intuitively chosen transistor sizing options for the map circuit considered in this work. Minimum length (65 nm) of the process is chosen as the length of each transistor in the design. Optimal sizing of the map circuit based on the performance rank of the chaos generator is found for the chosen width of 1.2 μm for transistor M_1 and 0.12 μm for both M_2 and M_3.

The bifurcation diagram for the optimized design geometry is shown in Fig. 4.4. The range of V_c that allows chaotic behaviors are 0.62 V to 0.72 V as can be seen from the bifurcation diagram. We can generate chaotic patterns for the chaos generator circuit by applying a control voltage, V_c from these two regions. However, from the investigation of the circuit, it is found that the worst case delay of the chaotic map circuit increases with

Table 4.1: Performance evaluation for different geometries of transistors considered in the circuit design of the chaotic oscillator [54].

Index	Geometry			Metric
	$W_1(\mu m)$	$W_2(\mu m)$	$W_3(\mu m)$	$\frac{mV}{fJ.\mu m^2}$
1	4.8	0.48	0.48	0.85
2	4	0.15	0.15	5.1
3	4	0.12	0.12	7.1
4	6	0.15	0.15	3.5
5	2	0.15	0.15	4.7
6	1.2	0.12	0.12	7.2
7	2.4	0.24	0.24	2.1

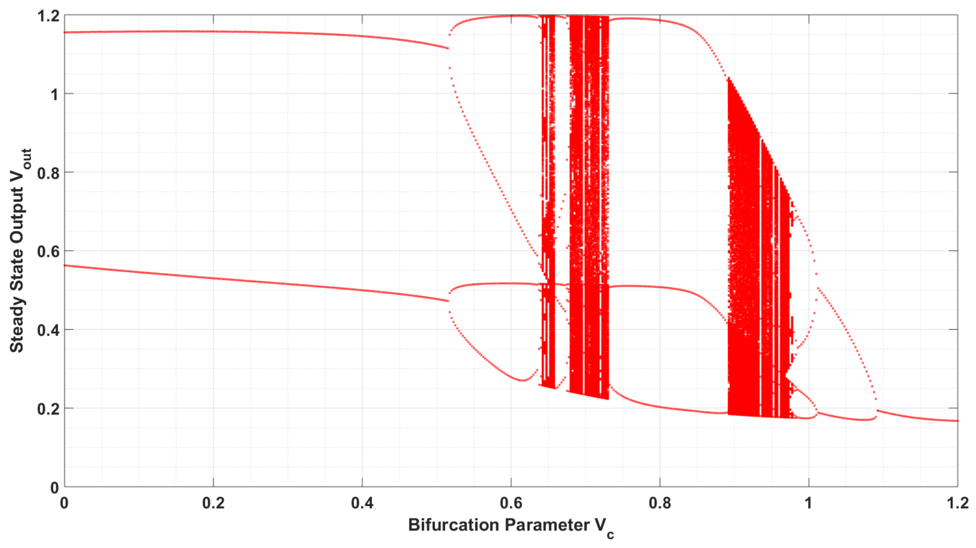


Figure 4.4: Illustration of state bifurcation diagram in the chaos generator circuit with the optimized design: $W_1 = 1.2\mu m$, $W_2 = 0.12\mu m$, $W_3 = 0.12\mu m$ [54].

the increase of V_c as demonstrated. The V_c is constrained within the lower values, 0.62 V-0.72 V in order to further optimize the design by reducing delay. It can be seen that this region in the bifurcation diagram is not a continuous chaotic region. Rather, it has a non-chaotic region which is excluded from the design consideration in order to ensure chaotic functionality.

4.2.3 Boolean Function from Chaos

As can be seem in earlier works, a chaos generator circuit can produce arbitrary Boolean function using an encoder and decoder [41, 42]. Fig. 4.5 shows such a circuit structure for Boolean functions from chaos generator circuit where the digital to analog converters (DAC) are used to drive the analog inputs and a comparator is used to digitize the analog outputs. The circuit in Fig. 4.5 thus works as a digital logic gate. In this work, generating 2-input 1-output Boolean functions from the chaos gate are considered. The chaos gate generates different Boolean functions over a number of iterations. Iteration number can be infinitely large which provides an infinite space for the functionality obtained from a chaos gate. The iteration number is encoded by a 3 bit iteration control input, K_t which allows for 8 iterations of the chaos generator. Additional control bits can be used with the primary data inputs in the DAC in order to have more flexibility for generating different Boolean functions. Data and control inputs are labelled as X and K_c in Fig. 4.5. A data input can be mapped into different analog values depending on the control bits. A single bit control input is considered for this work. The bias voltage, V_c can also be encoded by a number of bits to generate multiple discrete levels. Four discrete levels are considered for the bias voltage , V_c each 25 mV apart which require another 2-bit (K_b) DAC. The reference voltage of the output comparator can be used as another controlling parameter for additional tunability. For this work, a fixed reference is used in the comparator in order to reduce design complexity. The control bits are combined into a single configuration key for the reconfigurable chaos gate. The key bits are formed by appending individual control input bits as follows:

$$K = K_c, K_b, K_t$$

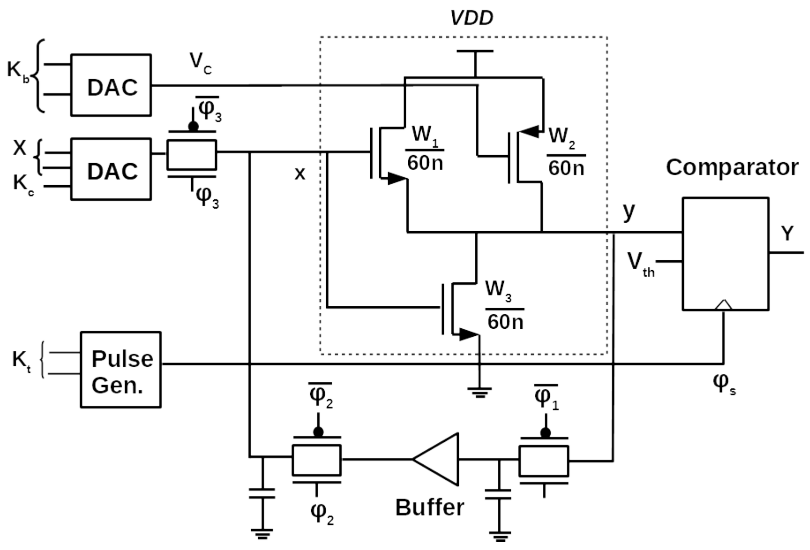


Figure 4.5: Design of chaos gate using the chaos generator circuit.

Table 4.2 shows a truth table of the chaos gate for 8 iterations with a control bit of 1 and bias voltage of 0.65 V. Binary output from each iteration of chaotic evolution is shown in the parenthesis. It can be seen that the outputs sampled from the 1*st* iteration of the chaos gate implements an NOR operation with the given initial condition. Similarly, an XOR operation is obtained by sampling the output from 2*nd* iteration. Based on the truth table for each combination of control parameters, we can generate a functionality table for the chaos gate for different keys. A part of the whole functionality table is shown in Table 4.3 which corresponds to the chaotic sequences shown in Table 4.2. Possible number of unique functionality that can be achieved from a 2-input 1-output logic gate is $2^{2^4} = 16$. The functionality is therefore labelled as an integer between 0 to 15 decoded by binary outputs from the truth table. For example, output from a truth table of an AND gate is 0,0,0,1 and can be represented as 8 in the functionality table. It can be noted that the corner functions such as ZERO and ONE are obtained with a higher frequency than the other functions from this chaos gate. Since these functions are not used in any applications, higher frequency of them increases the difficulty of finding the right functionality from the functionality space for an unauthorized user.

Table 4.2: Evolution of chaotic output for up to iterations with V_C=0.65 V, $K_c = 11$, V_{th}=0.6 V.

X_0 (V)	X_{n+1} **(V)**							
	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$
0(00)	1.199(1)	0.522(0)	0.523(0)	0.521(0)	0.523(0)	0.497(0)	0.644(1)	0.251(0)
0.32(01)	0.34(0)	1.182(1)	0.511(0)	0.573(0)	0.298(0)	1.194(1)	0.519(0)	0.537(0)
0.64(10)	0.339(0)	1.183(1)	0.512(0)	0.571(0)	0.302(0)	1.193(1)	0.5183(0)	0.539(0)
0.96(11)	0.523(0)	0.517(0)	0.543(0)	0.416(0)	1.032(1)	0.421(0)	1.011(1)	0.409(0)

Table 4.3: Logic functions from Table 4.2.

key	**Control**			**function**	**comment**
	V_c **(V)**	I_c	n		
011000	0.65	11	0	1	NOR
011001	0.65	11	1	6	XOR
011010	0.65	11	2	0	ZERO
011011	0.65	11	3	0	ZERO
011100	0.65	11	4	8	AND
011101	0.65	11	5	6	XOR
011110	0.65	11	6	9	XNOR
011111	0.65	11	7	0	ZERO

4.2.4 Impact of Variability

Process variation is a major concern for lower technology nodes. Transistor characteristics differ from one chip to another due to non negligible process variations. Process variation for the 3-transistor based map circuit used is investigated by Monte Carlo analysis using Cadence Spectre simulator. The impact of variation on the functionality of the map circuit amplifies as the chaotic behavior evolves over time. Chaotic oscillators are known for their sensitivity to the initial condition. This sensitivity comes from their dynamic behaviors. Therefore, the functionality from a chaotic oscillator is affected significantly due to small process variations across multiple chips. Chaotic sequences from an initial condition in 5 different chips are illustrated in Fig. 4.6. Since the chaotic sequence for a particular initial conditional is different in different chips due to process variation, logic functions for a particular configuration also vary from chip to chip.

In addition to the process variation, Monte Carlo analysis is also performed considering mismatch between transistors. Mismatch induced variation is found to be negligible for the map circuit used in this work. Therefore, each chaos gate in the same chip shows the same function for a particular configuration.

The functionality of chaos gate is analyzed based on the Monte Carlo simulations considering 100 chips. The new functionality table is demonstrated in Table 4.4 which shows the functionality variation of the chaos gate across 3 different chips. It can be seen that the functionality due to a key across various chips vary due to the process variation. It can be noted that this table is a small portion of the overall functinality space. Functionality for only 3 chips have been shown here for the sake of readability. However, considering the whole functionality space and large number of chips, it can be inferred that functionality varies significantly from one chip to another due to process variation.

4.2.5 Characterization and Enrollment

As chaos gates exhibit variation in their functionality from one chip to another, they need to be characterized after fabrication to configure into a particular logic function. Post fabrication characterization can be performed using a standard scan chain mechanism. The

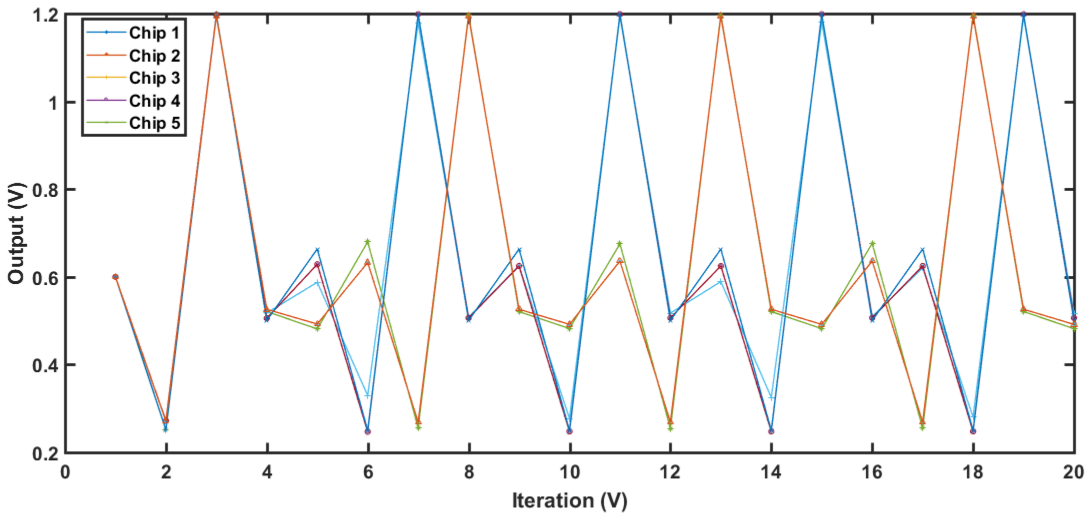


Figure 4.6: Sequence generated from the chaos generator in 5 different chips considering the same initial condition.

Table 4.4: Logic functions from the truth table of a chaos gate for various keys across different chips exhibiting process variations.

key	**Control**			**function**		
	V_c **(V)**	I_c	n	**Chip1**	**Chip2**	**Chip3**
01111000	0.65	11	0	1	1	1
01111001	0.65	11	1	6	14	14
01111010	0.65	11	2	0	1	1
01111011	0.65	11	3	0	14	10
01111100	0.65	11	4	8	14	1
01111101	0.65	11	5	6	14	0
01111110	0.65	11	6	9	14	0
01111111	0.65	11	7	0	14	4

inputs and outputs of reconfigurable chaos blocks must be accessible by the scan chain circuit. Variation analysis on the chaos map circuit shows that the functionality does not differ between chaos gate instances in the same chip. Therefore, it is sufficient to characterize only a randomly chosen chaos block per chip. During characterization, functionality found for each configuration key is enrolled into a database. Using the enrollment database, a designer can find the correct unlocking key for the circuit and provide it to the user. The unlocking key needs to be kept secret by the user. Without the key, one cannot achieve the desired functionality using the circuit.

A testing interface is required in order to perform the post fabrication characterization of a chaos gate in a chip. Testing interface enables an internal chaos gates to be accessible externally. All possible combinations of configuration key is provided to the input of the gate to be characterized and corresponding output is recorded. Since mismatch between transistors in a chip has been found negligible for this chaos circuit, functionality of each gate in the same chip is identical. Therefore, a test interface in order to characterize the chaos gates can be implemented without a complex design for test (DFT) methods such as scan chaining. A gate is chosen arbitrarily at the design phase in order to multiplex its inputs and output between the regular connections and the test interface. After fabrication, this gate can be characterized easily in order to find the desired configurations for the Boolean functionalities expected from different gates in the chip.

4.3 Design Obfuscation

Reconfigurable chaos logic gates can be used to design an obfuscated computing unit which can provide more inclusive security solutions. The design of chaos gates has two primary characteristics that makes it suitable for the two target security vulnerabilities considered in this work. The first one is that chaos gates are reconfigurable to arbitrary Boolean functions as shown in the previous section. Reconfigurability is featured with flexibility in a chaos gate. A single chaos gate can be configured into various Boolean functions which is defined as the reconfigurablity. In addition, a particular logic operation can be obtained using different configurations of a chaos gate due to its flexibility. Each of these configurations generate device specific unique power traces for each operation from the chaos gate. Besides, the chaos gate considered in this work exhibits process variation as demonstrated in the previous section. Due to process variation, the set of configuration keys for a particular logic function is unique. Therefore, the power traces of particular logic operations from the chaos gates in two different chips are significantly different. The uniqueness in power traces across multiple chips is another characteristics of chaos gate that can leveraged for security.

In a general computer architecture, data is processed in an execution unit commonly known as ALU that performs several logical, arithmetic, and shift instructions. Each instruction has a specific opcode by which the instruction is decoded and executed. If a computing unit such as an ALU is designed using chaos logic, correct functionality depends on the correct configuration key of each component. This functionality obfuscation can be achieved using any reconfigurable logic. However, a chaos gate also provides power obfuscation due to its flexibility in terms of implementation of a function. The obfuscation technique using chaos logic thus provides a comprehensive security solution to multiple security vulnerabilities in computing. The chaotic ALU is designed in a way that the configuration key and the functional opcode comprises the overall opcode for executing a particular instruction. The functional opcode decides the corresponding instruction's data path for the input to propagate to the output. Further, the configuration opcode decides the functionality of the reconfigurable chaos block on that datapath. A 4-bit ALU is considered here that can execute 7 instructions, AND, OR, XOR, ADD, SUB, $SHIFT$ and $ROTATE$. The implementation of the ALU using Boolean logic gates are shown in Fig. 4.7. Separate blocks are used in the ALU for respective instruction types. $Chao_{logical}$, $Chao_{add/sub}$ and $Chao_{shf/rot}$ are the chaos logic based implementation of the datapath used for logical, arithmetic, and shift instructions, respectively. The input operands are a and b. The configuration key for unlocking the desired functionality from the ALU is $\bar{K}$. Selectors s_0 and s_1 are used to choose a particular type of instruction for execution. Logical instructions are decoded by the configuration key among them.

A single reconfigurable block is used to implement any logical instructions in the ALU as shown in Fig. 4.8. Since 3 logical instructions are considered for the design of ALU in this work, there are $3\times$ saving in this implementation in terms of number of gates used.

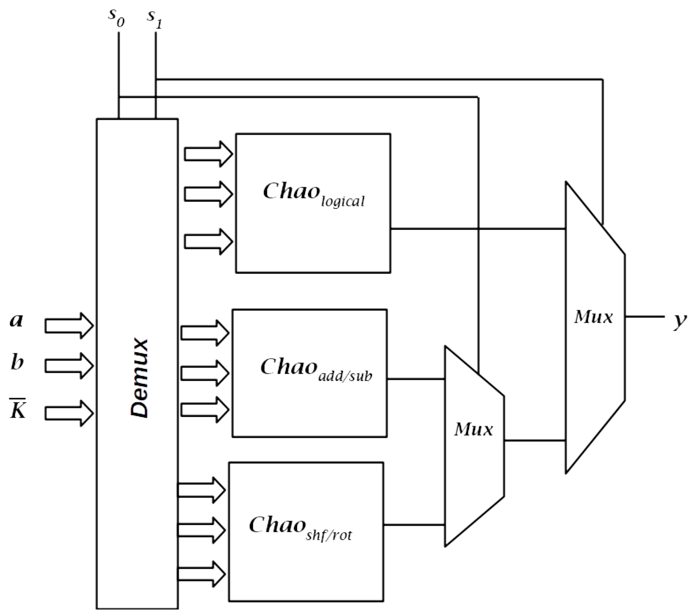


Figure 4.7: Obfuscated design of the ALU using chaos based data path for logical, arithmetic and shift instructions.

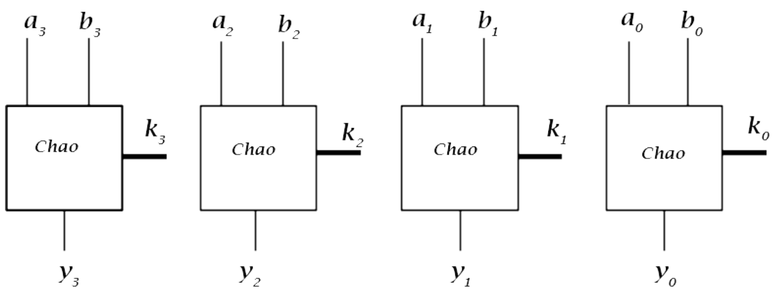


Figure 4.8: Data path for logical instructions (AND, OR, XOR) in the ALU. Four bit logical instructions can be designed using 4 instances of respective logic gates.

Arithmetic instructions are implemented using the circuit shown in Fig. 4.9. This is the conventional adder/subtractor circuit implemented using the reconfigurable chaos logic gates. The adder/subtractor circuit uses a full adder as the basic building element. The circuit implements the 2's complement of one of the operands using XOR gates in order to perform subtraction. Similarly, the conventional barrel shifter circuit is used for the shift and rotate instructions. The building block of the barrel shifter is a MUX circuit implemented using chaos logic. The barrel shifter circuit implemented using chaos blocks is shown in Fig. 4.10. For a 4-bit shift/rotate operation, two stages each having 4 MUXes are used. Three additional MUXes are used for choosing between shift or rotate operation.

4.4 Functional Obfuscation

The functionality of the chaos based logic unit depends on the configuration key. As already described, configuration keys for a particular Boolean function for a chaos unit varies from device to device due to process variation. Post fabrication characterization is therefore a part of the chaos based digital design flow. Configuration keys for each instruction are found based on the enrolled functionality data of the chaos logic blocks.

Let K_{AND}, K_{OR} and K_{XOR} be the key set for the chaos gate exhibiting the functionality of Boolean AND, OR, and XOR, respectively. Configuration key for a particular circuit is the combination of the individual keys of the logic blocks used in the circuit. For example, the full adder circuit can be implemented using 2 XOR, 2 AND and 1 OR block. Now, the valid configuration key, K_{valid} for full adder functionality from this circuit can be defined as follows:

$$K_{valid} = \{k_1, k_2, k_3, k_4, k_5\} : k_1, k_2 \in K_{XOR}, k_3, k_4 \in K_{AND}, k_5 \in K_{OR} \tag{4.3}$$

The circuit produces incorrect outputs for the application of an invalid key. Proposed functionality obfuscation using chaos based logic is evaluated based on hamming distance between the outputs corresponding to incorrect and correct key as suggested in [72]. 50% hamming distance is desired in order to achieve an ideal obfuscated behavior from the

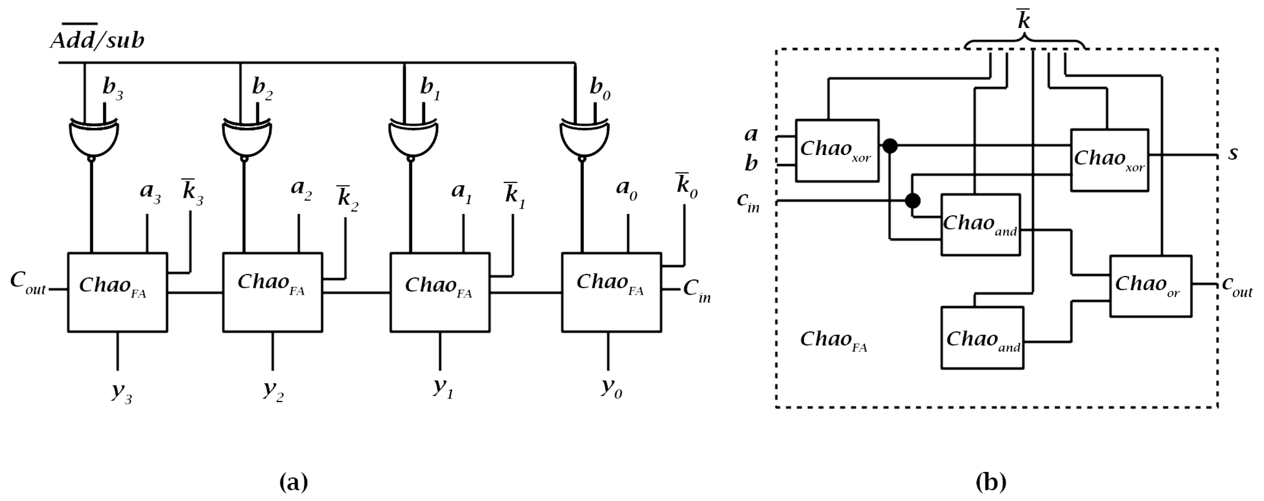


Figure 4.9: (a) Data path for arithmetic instructions designed using a ripple carry adder-subtractor circuit (b) Full adder circuit used in the ripple carry adder-subtractor.

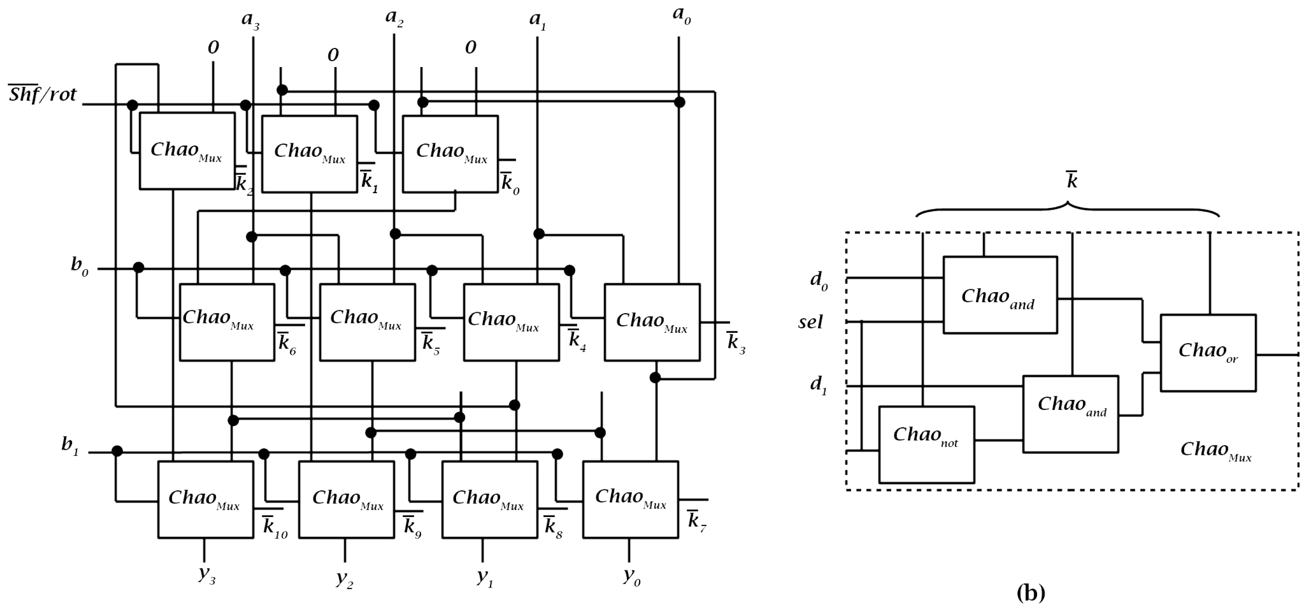


Figure 4.10: (a) Data path for shift and rotate instructions designed using a barrel shifter circuit (b) Mux circuit used in the barrel shifter.

operations. Table 4.5 shows the functionality obfuscation of each ALU instruction considered in this work. It can be seen that each instruction executed in the obfuscated ALU exhibits an obfuscation level of nearly 50% hamming distance.

4.4.1 Chaos-CMOS Hybrid Design

Desired functionality obfuscation can also be obtained by intelligently replacing a fraction of total gates with chaos gates requiring significantly lower overhead as compared to the fully chaos based design. This replacement based logic obfuscation is a popular technique in order to prevent reverse engineering, piracy, counterfeit of IC [4, 38].

The most common technique for replacement based logic obfuscation is based on the observability and controllability of the candidate gates [4]. Observability defines the impact of a node in the circuit to the output. The observability of a node depends on other logic gates in its propagation path to the primary outputs of the circuit. For example, the propagation of a node will be blocked by an AND gate that has a ’0’ in its other input. Similarly, an OR gate blocks the propagation of a node when it has a ’1’ in the other input. On the other hand, controllability refers to the flexibility of setting a particular value to an internal node by controlling the primary inputs of the circuit. In this work, controllability of a gate is measured as the combined controllability of its input nodes.

In this work, the logic of an ALU is obfuscated by replacing just a few logic gates with chaos based reconfigurable logic blocks. Replacement locations are chosen based on the observability of the output and controllability of the inputs. There are 3 individual

Table 4.5: Functionality obfuscation for each instruction measured as the hamming distance between the outputs of correct and incorrect keys.

Instruction	HD
AND	50.58%
OR	50%
XOR	50%
Add	49.95%
Sub	50.36 %
Shf	44.82 %
Rot	50.25 %

datapaths for different types of instructions in the ALU design considered in this work. For bitwise logical instructions, a single chaotic logic gate is used for each bit. The chaotic gate can implement the desired Boolean function with an appropriate configuration key. The second datapath is the adder/subtractor circuit which is used to execute ADD and SUB instructions, respectively. The shift and rotate instructions are executed by a barrel shifter circuit which is another data path in the ALU. Each data path is considered individually for choosing the replacement location of the chaos gate.

The circuit for logical instructions implement bitwise operations where every input and output is independent of each other. Replacing the logic gate of a particular bit location affects only that output bit. However the adder/subtractor and the barrel shifter circuits have diffusability which makes it possible to obfuscate the output unpredictably using only a fraction of total gates replaced with chaos gate. Each gate in the circuit are ranked based on their testability which is the product of the observability and controllability. Quantitatively, observability is measured by the percentage of the total input of the circuit that allows the node under test to propagate to the primary output of the circuit. On the other hand, controllability is measured by the uniformity of the logic value at the inputs of the candidate gate. If the probability of '00', '01', '10' and '11' in the input of the candidate gate is equal upon all possible primary input combinations, the gate has a controllability of 1.

Fig. 4.11 shows the CMOS-Chaos hybrid implementations of the full adder and multiplexer circuits used in the adder/subtractor and the barrel shifter circuit, respectively.

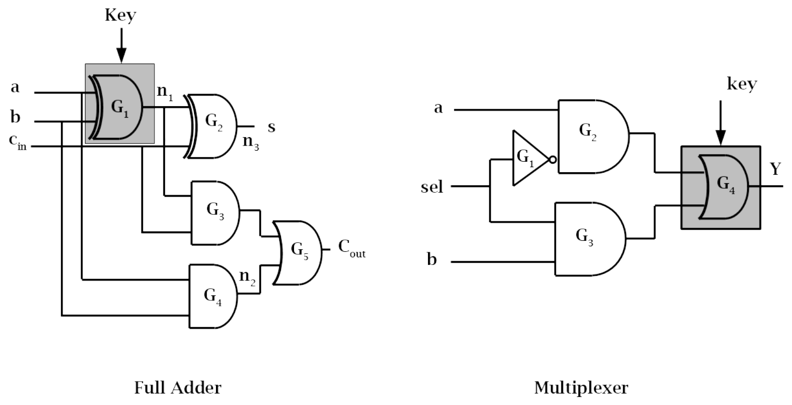


Figure 4.11: Hybrid implementation of a full adder and mux circuit used as the building block of adder/subtractor and barrel shifter, respectively.

The first XOR gate is found to be the most testable block in the full adder circuit based on the testability analysis. In the hybrid implementation, this gate is replaced with a reconfigurable chaos gate. Using similar analysis, the OR gate in the multiplexer circuit is found to be the gate with highest testability and replaced with chaos gate for the hybrid implementation.

Based on the testability ranks of the candidate gates, various proportions of chaos gates are used to replace the logic gates in the ALU. Here, hamming distance between the outputs corresponds to the difference between the correct and incorrect key combinations of the chaos gates. In order to create the maximum ambiguity for an attacker, 50% hamming distance is desired between the correct and obfuscated outputs. Table 4.6 shows the hamming distance value for different ratio of chaos gates to the total number of gates in the ALU design. The trend of increasing hamming distance with respect to number of chaos gate insertion is shown in Fig. 4.12. After a certain number of chaos gate used in the circuit, hamming distance does not increase much and becomes saturated at 50%.

4.5 Side Channel Power Obfuscation

The side channel attack is one of the major security concerns for building embedded computing hardware. This type of attack is based on information leakage through different physical quantities such as power consumption, timing information, electromagnetic emanation, etc. associated with the actual computational operation. To make a computing system secure, it is necessary to find the possible threats from the perspective of hardware implementation even though the underlying algorithm of computation may be considered mathematically secure.

Power consumption analysis is one of the well known side channel techniques used to gain secret information from the execution of a computing device. The key of a hardware-based encryption engine can be extracted based on differential power analysis attacks [44]. Power analysis can also be used in order to reverse engineer the machine instructions executing in a processor [11, 82, 63]. In this work, the power analysis assisted instruction reverse engineering attack is considered for demonstrating the resiliency of the chaos based design.

Table 4.6: Functionality obfuscation represented in terms of hamming distance between outputs corresponding to correct and incorrect key for various number of chaos gates used in the circuit.

No. of chaos gate				Overhead (%)	HD(%)
logical	arithmetic	shift	**total (out of 64)**		
2	2	2	6	9.3	24.71
4	2	2	8	12.5	35.28
4	4	4	12	18.75	44.29
4	4	6	14	21.88	45.57
4	4	8	16	25	48.14
4	4	10	18	30	48.71
4	4	11	19	31.67	49.14

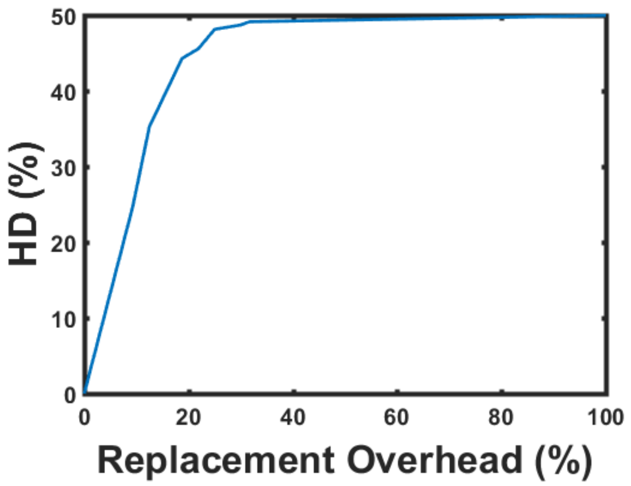


Figure 4.12: Functionality obfuscation for different proportion of chaos gates in the 4 bit ALU in terms of hamming distance.

Chaos based logic is used in an ALU design in order to obfuscate the power profile in such a way that it can prevent instruction reverse engineering.

4.5.1 Instruction Classification Attack

An attack scenario is considered here for recognizing different instructions of an ALU based on power profiling. Profile of each instruction is developed from their power signatures from a reference device. Instructions in a device under attack is then classified based on the developed instruction profiles. Steps of this classification attack are as follows:

Data Collection

Sufficient amount of sample power traces are collected for each instruction in order to perform profiling attack [19, 63]. Power traces of each instruction varies due to the operand and noise level. The ALU design is simulated using a transistor level circuit simulator, Cadence Spectre. 500 power traces are collected for each instruction for building the instruction profiles. 5-fold cross validation is performed on the collected power data set where samples used for training and testing are 80% and 20%, respectively.

Dimensionality Reduction

Dimensionality reduction is required for making the classification computationally more efficient. Dimensionality reduction techniques compress data into less number of features while maximizing the variance. Various dimensionality reduction algorithms have been described in the literature [89, 88, 63]. Principal Components Analysis (PCA) and Fishers Linear Discriminant Analysis (FLDA) are two popular dimensionality reduction techniques. In this work, these two techniques as well as two other methods namely Sum of Difference of Means (SDM) and Means-Variance (MV) are considered.

Principal Components Analysis (PCA) Principal Components Analysis (PCA) [89, 63] is a method to extract the features with higher variance in a data set. Original multi dimensional data is projected onto a lower dimensional subspace that maximizes the projected variance [78]. Fig. 4.13 shows the variance plot in order to find the principal

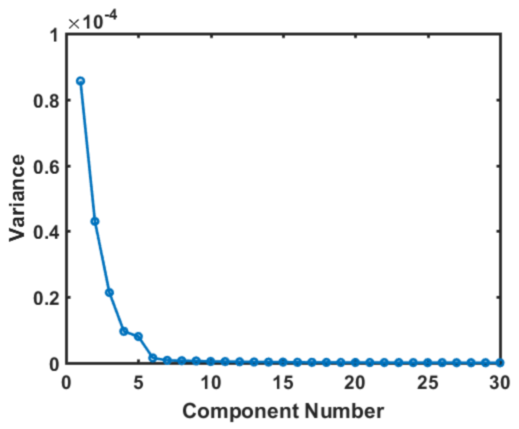


Figure 4.13: Illustration of variance of different components in the power data. Principal components are extracted by considering the first few components that preserve the most variance.

components from a typical data. It can be seen that the variance drops with the number of components in the data set and becomes saturated at some point. Principal components are selected up to a point where the rate of change in the variance with respect to the feature components becomes negligible.

Sum of Difference of Means (SDM) The method of computing a reduced dimension, D from an L-dimensional data using SDM is a method similar to the PCA. Here, the absolute difference between each pair of mean vector is calculated first. These differences are then added and first D points among the highest peaks are chosen.

Means-Variance(MV) Means-variance is another method motivated by the same underlying principle as SDM described earlier. Reasonable choice for a multi-class classification is to consider features providing the maximum variance across the classes. In order to find such features, the mean of each class for each dimension is put in a matrix. The row of the matrix indicates class and the column indicates means across each dimension. The variance of the columns are the feature wise inter-class variance. The first D columns exhibiting the highest variance is considered as the reduced dimension.

Fishers Linear Discriminant Analysis (FLDA) Fishers Linear Discriminant Analysis (FLDA) is another prominent technique used for reducing the dimensionality of data [24, 17]. In a multi-class classification, inter-class variance and intra-class variance are the helping and deterring factors, respectively. FLDA analyzes the intra and inter-class variance of a data set in order to maximize the ratio of inter-class and intra-class variance. The original data is projected into such a plane that maximizes this ratio. The size of the reduced dimension is $N-1$ in FLDA for an N-class classification.

Classification Algorithm

The next step in the classification is to use the data template built from the training data set with a classification algorithm in order to classify a test set. The training data set has the format of x, y for a supervised learning where x represents a sample and y represents which class it belongs to. A class is assigned by a classifier for an arbitrary instance x based on the training data set. Many different classifiers are used in machine learning problems and their relative superiority depends on its speed, implementation cost, accuracy and most importantly, the nature of the problem. In this subsection, we briefly discuss several classification algorithms used in this work.

K-Nearest Neighbor (K-NN) The k-NN is one of the simple and fundamental classification algorithms [25]. The full training data set needs to be stored for this classification which makes it computationally intensive. A sample is recognized based on its distance between the training sets. The distance between a test sample and each of the training samples are measured. The most common class among the K closest training samples is assigned to the test samples. Different values of K are chosen based on an application. In this work, K=1, 3 and 5 are considered for the classifier. Different distance measurement techniques can be applied for measuring distances between multi dimensional data samples. Some of the algorithms used for measuring distances in K-NN are Euclidean, Correlation, and Cosine distance functions [83, 14, 63].

Support Vector Machine Support vector machine (SVM) is an widely used [36] classification algorithm. It performs binary classification by assigning a training set into two categories where one category belongs to a particular class and the other combines all other classes. It is a non-probabilistic classifier where a test sample is labelled as one of two classes. An SVM classifier basically draws a line onto the 2-D space of containing training samples by clearly partitioning them. New samples are classified based on which sides of the line it falls into. SVM can be extended to classify multi-class data sets by choosing suitable kernel functions such as Gaussian radial basis function [34]. There are various types of binary classifier used in SVM such as 'onevsone', 'onevsall', 'binary complete', 'denser random', etc. Among these classifiers, 'onevsone' and 'onevsall' are considered in this work.

Multivariate Gaussian Probability Density Function In a Multivariate Gaussian classification, a template is built from the training set for each class based on the mean, $\mu_{\mathbf{k}}$ and covariance, σ_k of the set by assuming that it follows a multivariate Gaussian distribution [31, 63]. Now, the probability of a test sample falling into the distribution of each class is measured. Class with the highest probable distribution for this test sample is assigned as its label by the classifier.

4.5.2 Attack on Conventional Design

The classification attack is performed on the all CMOS gate based ALU design using the K-NN, SVM and MVG classification algorithm. Three different neighbors 1, 3 and 5 and 3 different distance measurement techniques Euclidean, Correlation and Cosine are considered for the K-NN algorithm used in this work. Table 4.7 shows the classification results for all these various classification techniques. It can be seen that ALU instructions implemented using traditional CMOS logic gates are classifiable with a high accuracy. The best case classification result is found to be 93% using the K(=1)-NN with PCA and Euclidean distance. It was also shown in an earlier work that instructions in a microcontroller can be classified with a high accuracy using K(=1)-NN combined with PCA [63].

Table 4.8 shows the confusion matrix for each instruction class. Rows of Table 4.8 represents the training set of each instruction and columns represent the percentage of match to a particular instruction class. Among the instructions, XOR, ADD and SUB can be recognized with 100% accuracy. The classifier recognizes AND correctly with a 94 times (out of 100) and confuses with the OR and XOR for the remaining 6 test cases. Similarly, the OR instruction was recognized properly for 91 times and was otherwise recognized as AND. Similarly, ROT and SHF instructions were recognized 86 and 78 times, respectively while confused among themselves otherwise. Overall classification accuracy can be found to be 93% which is the average value of the individual recognition rate across all instructions.

Table 4.7: Classification accuracy among instructions for different classifier and dimensionality reduction algorithm for a CMOS only ALU implementation. Static CMOS logic gates are used for implementing different instructions of the ALU

category	**sub-category**	**Accuracy with different data reduction**				
		no reduction	*PCA*	*SDM*	*MV*	*FLDA*
K-NN	1NN(euc)	93	93	93	93	88
	1-NN(corr)	76	90	68	69	88
	1-NN(cos)	90	90	90	90	86
	3-NN(euc)	90	90	90	90	85
	3-NN(corr)	82	87	86	86	85
	3-NN(cos)	86	86	86	86	85
	5-NN(euc)	88	88	88	88	83
	5-NN(corr)	85	85	85	85	82
	5-NN(cos)	86	86	86	86	84
SVM	onevsall	82.7	80.66	58.3	57	82.47
	allpairs	90.55	89.14	78.23	74.61	85.86
MVG	N/A	73	79	79	79	85

Table 4.8: Classification of ALU instructions for CMOS gate based implementation. Best case accuracy is found for K(=1)-NN with Euclidean distance and PCA as the dimensionality reduction technique

Instruction	**Matched Class (%)**						
	AND	**OR**	**XOR**	**ADD**	**SUB**	**ROT**	**SHF**
AND	94	5	1	0	0	0	0
OR	9	91	0	0	0	0	0
XOR	0	0	100	0	0	0	0
ADD	0	0	0	100	0	0	0
SUB	0	0	0	0	100	0	0
ROT	0	0	0	0	0	86	14
SHF	0	0	1	0	0	21	78

4.5.3 Attack on Chaos based Design

Proposed chaos based reconfigurable logic can implement a particular Boolean function using a number of different chaos configurations. Each of these configurations causes the chaos gate to exhibit unique power profiles. Besides, due to process variation, a chaos gates produces different Boolean functions for the same configuration key in different devices. Valid key sets for a particular function from a chaos gate in different chips are unique.

For the classification attack on the hybrid ALU implementation, a design is chosen with 25% chaos gates as suggested by the analysis described in section 4.4.1. It can be seen that this design has a 48% hamming distance between the outputs corresponding to the correct and incorrect configuration key. For power profile obfuscation, the design with equal power level for each instruction is preferable which is obtained from the design with 4 chaos gates in each data path of the ALU. However, that design has a hamming distance of 44% between the outputs of the correct and randomly keys. The chosen design with 4, 4 and 8 number of chaos gates in different data paths is the optimal one which exhibits 48% hamming distance. This design is also more uniform in terms of the power level for different instructions as compared to the other design choices having higher hamming distance.

Four different ALU chips are considered for performing inter chip classification attack. Each chip requires unique set of configuration keys for the expected functionality. Unique configuration generates unique power profile for instructions across different chips. An adversary collects training data from the chip she is authorized to use. She cannot run arbitrary instructions on a random chip as each chip needs an activation key which the adversary supposedly does not have. Therefore, the adversary cannot easily collect training data from the chip under attack. The attack is performed just based on the training data from a different chip. Since each chip exhibits unique power profiles for the instructions, the classification accuracy is significantly lower than the CMOS based design. The attack is performed using various training-testing combinations among the 4 different chips.

For the classification of instructions in the chaos based ALU implementation, training and testing set corresponds to two different chips. Therefore, a whole set of 500 observations can be used for the testing unlike the CMOS based implementation where only 20% of the whole data set is used for testing. Table 4.9 shows the classification accuracy of the attacks performed on those training testing combinations. The best case average classification accuracy is found for the multivariate Gaussian algorithm with Fisher's linear discriminant analysis as the dimensionality reduction. Detailed recognition rate is shown in Table 4.10 where percentage match of each test instructions with the instruction classes are averaged over all chip combinations. The results indicate that chaos based design shows significantly higher resiliency against the instruction classification attack as compared to the conventional design.

4.6 Overhead Analysis

The proposed chaos based reconfigurable logic provides both functional and side channel obfuscation. Functional obfuscation can also be obtained by look up table (LUT) based design. However, they cannot provide power obfuscation. LUT based designs are basically memory where the configuration bits are stored for the logic implementation. Significant amount of overhead in terms of area, power and delay are incurred by the LUT based design.

In this section, the overhead incurred by the proposed chaos based design is compared with a spin transfer torque (STT) LUT based reconfigurable design used for functionality obfuscation [86, 87] and an power analysis resistant improved delay based dual rail pre-charge logic (iDDPL) [6]. Static CMOS based logic gates are considered as the baseline for overhead comparison among these logic technologies. Both gate level and system level overhead are estimated for the comparison. The design used for the system level comparison is the ALU circuit where 25% of the total gates are reconfigurable.

There are total 64 logic gates in the 4 bit ALU circuit consisting of 4 in the logical, 24 in the arithmetic and 32 in the shift instruction datapath. For reconfigurable logic based implementation, the logical instructions are executed by a single type of gate. Therefore, 4 reconfigurable gates are sufficient for executing all 4-bit logical instructions. On the other hand, individual logic gates are required for each logical instruction in a CMOS only implementation. Since, 3 logical instructions are considered in the ALU example considered

Table 4.9: Classification accuracy of instruction set among different chaos based machine with several classification and data reduction algorithms.

$category$	$data\ reduction$	$config\ pair$						
		$1,2$	$1,3$	$1,4$	$2,3$	$2,4$	$3,4$	$average$
$K-NN(=1)$	$no\ reduction$	39	19	17	20	16	37	24
	PCA	39	19	17	19	16	37	24.67
	SDM	38	17	11	17	16	35	24.5
	MV	39	17	11	17	16	35	22.33
	$FLDA$	39	21	20	22	24	24	22.5
SVM	$no\ reduction$	31	30	26	25	28	30	28.33
	PCA	33	29	21	23	28	33	27.83
	SDM	30	21	26	23	28	32	26.67
	MV	34	22	26	24	25	34	27.5
	$FLDA$	14	14	14	14	14	14	14
MVG	$no\ reduction$	28	28	28	14	14	28	23.33
	PCA	24	25	23	14	14	16	19.33
	SDM	14	14	14	11	14	14	13.5
	MV	14	14	14	11	14	14	13.5
	$FLDA$	53	36	26	38	32	34	36.5

Table 4.10: Best case average classification results for Chaos gate based implementation. Best results found for Multivariate Gaussian algorithm with Fisher's linear discriminant analysis

Instruction	**Matched Class (%)**						
	AND	**OR**	**XOR**	**ADD**	**SUB**	**ROT**	**SHF**
AND	48	19	14	3	1	0	0
OR	8	37	27	11	2	0	0
XOR	34	19	22	8	4	0	0
ADD	13	0	9	11	10	20	22
SUB	10	01	8	10	14	22	21
ROT	0	0	0	0	0	58	27
SHF	0	0	0	0	0	56	30

here, 8 more gates are required in the CMOS based implementation. Number of total gates are 72 for that case. Taking all these factors into consideration, the system level overhead for the chaos and STT LUT based reconfigurable logic can be estimated. On the other hand, iDDPL targets at counter-measuring the power analysis attack by a current balancing technique where the the circuit consumes a constant power for all input values. Balancing power profile requires the full circuit designed with iDDPL. The gate level and the system level overhead are therefore the same for iDDPL. For demonstrating the effectiveness of the design proposed in this work, the overhead is estimated for a hypothetical design where both STT LUT and iDDPL are used in order to provide the functional and power obfuscation, respectively. In the hypothetical design, 25% of the gates are considered to be replaced with the STT LUT and the remaining 75% with the iDDPL.

$$OV_{sys} = \frac{(0.75G_D + 0.25G_C)X64}{72G_D} \tag{4.4}$$

where G_C and G_D represents the cost associated with each chaos and standard CMOS gate, respectively. Eq. 4.4 can be rewritten as follows:

$$OV_{sys} = \frac{2}{9}(OV_{gate} + 1) \tag{4.5}$$

where $OV_{gate} = \frac{G_C}{G_D}$.

Table 4.11 shows the overhead comparison between the chaos and STT LUT based logic design in terms of area and power-delay product. It can be seen that for the 4 bit ALU design, area overheads are $1.5x$, $2.8x$ and $4x$ and PDP overheads are $6x$, $23x$ and $2.3x$, respectively for the chaos, STT LUT and iDDPL. The chaos based design exhibits significant improvement over the other two existing design methods. In addition, the STT LUT and iDDPL based design only provides functional and power obfuscation, respectively whereas the proposed chaos based design obfuscates both. The hypothetical implementation could provide both functional and power obfuscation by incorporating the two individual mitigation techniques on the same circuit. Overheads associated with the hypothetical implementation are found to be $27x$, $5.4x$ in terms of PDP and area, respectively. The improvements of the chaos based design of the 4 bit ALU is therefore $4.6x$ and $3.6x$, respectively for the PDP and area overhead

Table 4.11: Overhead estimates for chaotic logic in gate level and system level. 4 bit ALU with a proportion factor of 25% are considered for the system level overhead estimates.

Obfuscation Method	Power-Delay product (PDP)			Area			Security	
	value(fJ)	OV$_{gate}$	OV$_{sys}$	value(μm^2)	OV$_{gate}$	OV$_{sys}$	func. obfus.	power obfus.
Static CMOS gate	6	-	-	0.29	-	-	No	No
STT-RAM LUT [86]	624	104x	23x	2.8	9.6x	2.8x	Yes	No
iDDPL [6]	14	2.3x	2.3x	1.15	4x	4x	No	Yes
STT-RAM LUT+iDDPL (hypo.)	-	-	27x	-	-	5.4x	Yes	Yes
Chaos	152	25x	6x	1.22	4x	1.5	Yes	Yes

as compared to the hypothetical implementation considered for providing security against both attacks. Thus, chaos based design is capable of providing comprehensive solutions to the security problems of a computing system.

4.7 Conclusion

Proposed design obfuscation using chaos based logic provides multiple security benefits in computing as compared to other conventional methods. The functionality of a chaos based design is locked. Only the authorized user of a chip has the valid key to unlock the design and perform desired task on the device. Thus, an adversary cannot get the desired functionality from the circuit in order to extract a secret. In addition, due to the reconfigurability and process variation of the chaotic circuit, power profiles of each operation in a particular chip are significantly different than another chip. The power template built from an identical device cannot be applied to distinguish operations in an another device by using statistical learning. Security of conventional computing devices can be improved significantly in this way by applying the chaos based logic in the design.

Chapter 5

Security Enhanced RISC-V Microarchitecture

Security based logic and memory design proposed in this dissertation is applied on a RISC-V microarchitecture in order to enhance security of computation. The microarchitecture chosen for this security enhanced implementation is capable of executing a subset of RISC-V 32I instruction set. This implementation is named as MiniRISC-V in this work for future references. Designed microarchitecture has 5 pipeline stages for executing an instruction sequence. We consider a Harvard architecture where the instruction and data memory are separate. The data memory is built using the RRAM proposed in Chapter 3. Memory instructions are secured by the integrated integrity verification feature of the proposed RRAM. The execution unit is obfuscated by replacing a number of logic gates with reconfigurable chaos blocks. Thus, a program executed on this processor cannot function correctly unless the correct key is applied to the chaos gates. As we have seen in Chapter 4 that the chaos logic exhibits unique power signature from one chip to another due to process variation and chaotic reconfigurability. The side channel power based instruction reverse engineering can not be performed in this implementation. In this chapter, the detailed microarchitectural implementation of the MiniRISC-V is described with the security enhancing features.

5.1 MiniRISC-V Microarchitecture

5.1.1 ISA

MiniRISC-V microarchitecture implements the user level subset of original RISC-V (RV)32I instruction set [84]. RV32I supports only integer operations. As described in the background section, instructions of RV32I are classified into 6 different categories: R-type, I-type, S-type, B-type, U-type and J-type. Detailed description of the instruction subset used in the MiniRISC-V is provided in chapter 2.

5.1.2 Pipeline

Execution of an instruction in MiniRISC-V is performed in 5 pipeline stages. An instruction is partially processed in each stage and the results are saved in pipeline registers in order to make it accessible to the next stage in the next cycle. Similar to many other RISC-V processors, pipeline stages of MiniRISC-V are Fetch, Decode, Execute, Memory and Writeback as can be seen from the block diagram of the MiniRISC-V microarchitecture shown in Fig. 5.1. General naming convention followed here in order to refer a pipeline register is to use the abbreviated names of shared stages separated by a '/'. For example, pipeline register at the interface of Fetch and Decode stage is IF/ID (instruction fetch/instruction decode). Similarly, other pipeline registers are ID/EX, EX/MEM, and MEM/WB. Brief descriptions of the pipeline stages of MiniRISC-V are as follows:

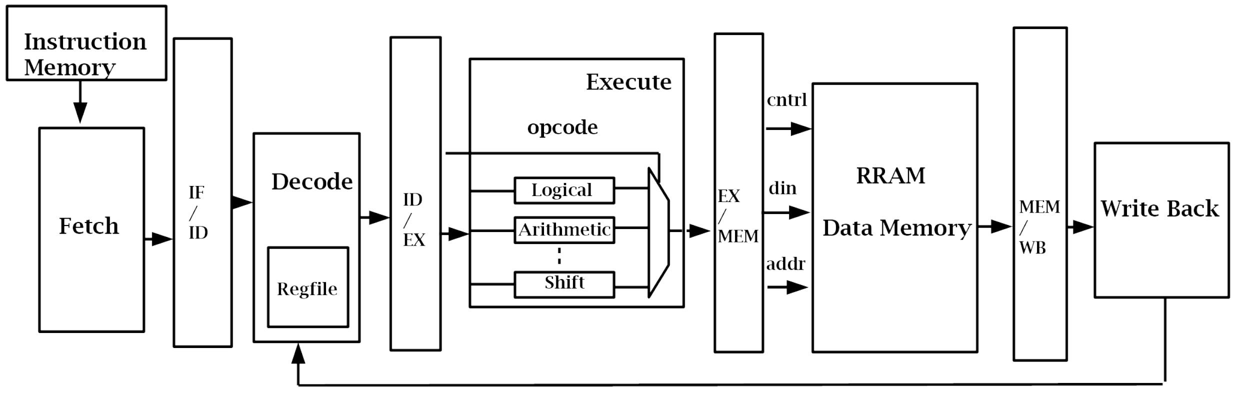


Figure 5.1: MiniRISC-V microarchitecture for implementing the user level subset of RV 32I.

Fetch

The function of the fetch unit is to read the instructions from the instruction memory sequentially and pass it to the execute stage. A counter called a program counter (PC) generates the address for the instruction memory. All memory blocks in this architecture are considered as byte addressable. A 32-bit instruction therefore takes 4 consecutive addresses in the memory. Each instruction in a RV 32I is of similar length. During normal execution, PC is incremented by 4 to fetch the next instruction. For branch instructions, PC is incremented by the offset of the branch address. PC is loaded with the destination address during the execution of control flow transfer instructions, JAL (CALL) and JALR (RET).

Decode

The primary function of the decode stage is to read the instruction from the IF/ID register and assert the control signals in different datapaths with the appropriate values for execution. Register values for source addresses rs1, rs2 and destination address rd are also fetched in the decode stage. In the MiniRISC-V microarchitecture, branch decisions are also made in the decode stage instead of the execute stage which causes the pipeline to stall for only one cycle if the branch condition is false. It is worth noting that the branch is considered as not taken by default until the real branch decision is known. If the branch is taken, the fetched instruction is discarded and the PC is loaded with the branch address. In order to avoid a control hazard that arises from the pipeline data dependency from partially executed instructions, a branch forward unit is used in the decode stage. The branch forward unit forwards the result of the execute and memory units to the decode stage. A hazard detection unit stops fetching new instruction by the PC for a required number of cycles in order to let few existing instructions finish and the hazard is cleared.

Execute

The execute stage consists of an ALU and a forwarding unit. The ALU performs different logical, arithmetic, and shift operations based on the opcode and extended opcode fields, funct3 and funct7. The first operand of the ALU comes from the register file output corresponding to rs1. The second operation can be either register data corresponding to rs2 or an immediate data from the immediate field of the instruction. A hazard detection unit is used to detect data hazards created by the dependency on previous executing instructions in the pipeline. A forwarding unit forwards the ALU or Memory data based on the type of detected hazard. Execute unit also computes the address for load and store instructions which basically performs an addition between a base address and an offset.

Memory

Data memory is separate from the instruction memory in the MiniRISC-V as it is built based on a Harvard architecture. Data memory is also byte addressable. RV 32I allows memory read-write operations using data of different byte sizes. LB, LHW, and LW represent the instructions for loading a byte, half-word and full word, respectively from the data memory to the register file. There are two more load instructions LHU and LBU which interpret the half-word and a byte data as an unsigned 32-bit value by filling the remaining most significant bits with 0's. Similar to load instructions, SB, SH and SW represent instructions for storing a byte, half-word and a full word into the data memory.

Write Back

The final execution result of an instruction is written back to the destination register address, rd. The write back operation selects the final result between the memory output and the ALU output depending on the instruction type. The address of the destination register is propagated to the write back stage through each pipeline registers. Destination register is updated with the results of the operation in the next clock cycle. Write back is required in the R-type and few I-type instructions where the results of an operation between two operands are written in a destination register. For JUMP instructions (jal, jalr), address of the next instruction of the caller procedure is written in the destination register.

5.2 Security Enhancing Modification

The secure memory and execution unit design described in chapter 3 and 4, respectively are used for the data memory and the execution unit of the security enhaced design of the MiniRISC-V. The modified design is shown in Fig. 5.2 where the security enhancing modifications are highlighted.

5.2.1 Chaos based Execution Unit

The execution unit of the MiniRISC-V microarchitecture considered in this work is obfuscated by replacing some of the logic gates with chaos logic in the most testable locations of the circuit. A common methodology for choosing the testable locations is described in Chapter 4. Individual datapaths in the execution unit for executing different types of instructions are obfuscated using chaos based logic. The MiniRISC-V microarchitecture has datapath for logical, arithmetic, and shift instructions in the execution unit. A number of chaos based reconfigurable logic blocks replace the original logic gates used in the datapath implementation of ALU instructions. The execution becomes dependent on the configuration key of the chaos gates. Incorrect key results in an incorrect output from the execution result of an instruction. Incorrect results from individual instructions without the correct key obfuscates the program behavior significantly. Obfuscation is not only limited to the ALU instructions though only the execution unit is implemented using chaos based logic gates. Results of the ALU instructions determines the value of different registers in the register file which affects the decision making of branch instructions. Memory instructions are also affected by the ALU obfuscation since the address to be accessed by a memory instruction is evaluated by the ALU. Therefore, obfuscating only the execution unit is sufficient in order to obfuscate the program behavior significantly. Required proportion of chaos logic in order to obfuscate the functionality of various instructions depend on the instruction datapath.
Bitwise logical instructions are implemented using multiple copies of respective logic gates as shown in Fig. 5.3. The path from each input to output consists of a single gate. There is no diffusion of data between the inputs and outputs. Each of the gates therefore need to be replaced with reconfigurable chaos gates in order to have optimal obfuscation. However, the

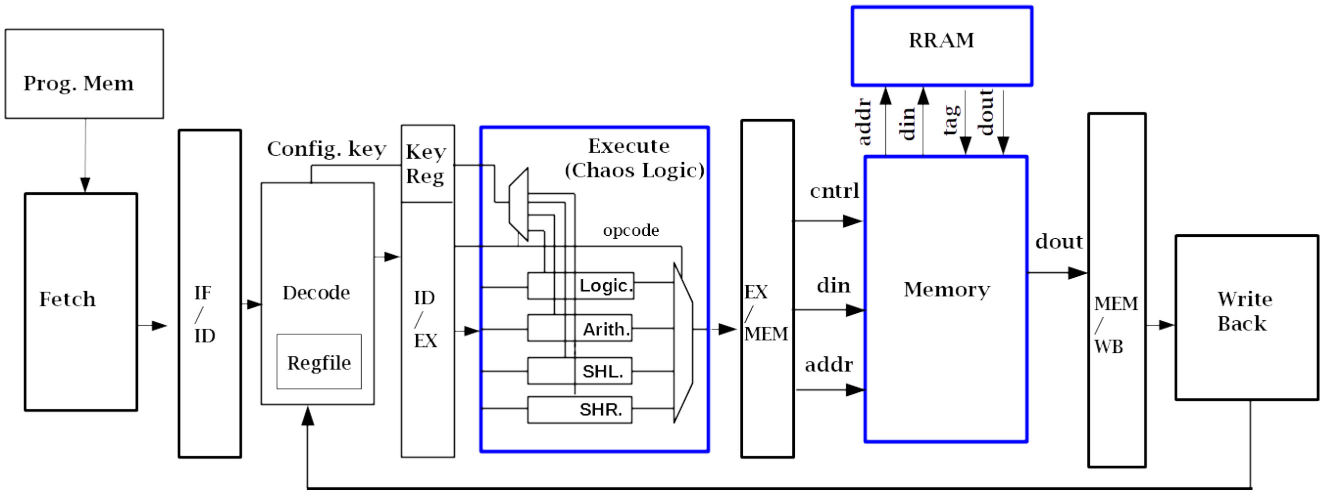


Figure 5.2: MiniRISC-V microarchitecture with security extension using chaos logic in the execution unit and RRAM memory with integrity checking as the data memory.

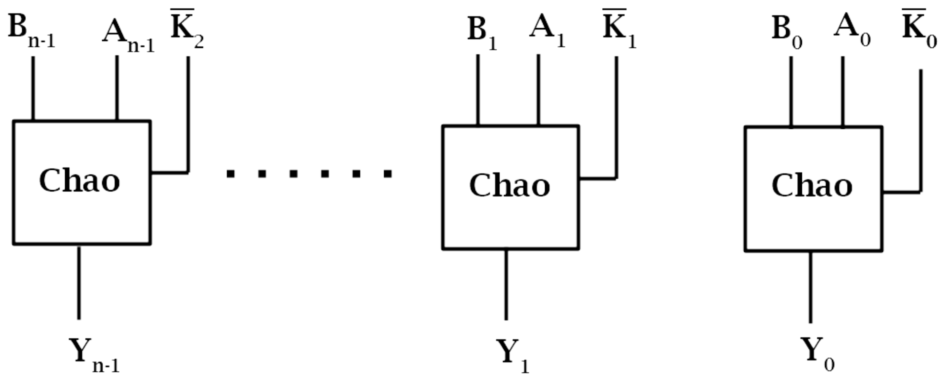


Figure 5.3: Obfuscated data path for 32 bit logical instruction in MiniRISC-V.

same hardware can be used to implement all logical instructions due to the reconfigurable nature of chaos gates.

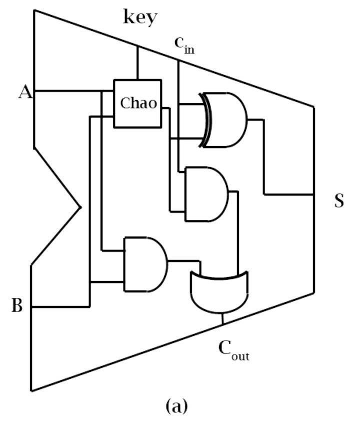
Arithmetic instructions are implemented using a ripple carry adder-subtractor circuit as shown in Fig. 5.4(b). The building block of the adder-subtractor circuit is a full adder which can be implemented using two XOR gates, two AND gates and an OR gate. From the testability analysis of the full adder circuit the first XOR gate is found optimal to be replaced with reconfigurable chaos gates. The obfuscated full adder shown in Fig. 5.4(a) is used in the ripple carry adder-subtractor circuit in order to implement 32-bit arithmetic instructions.

A barrel shifter circuit is used for implementing shift instructions in the MiniRISC-V microarchitecture. A part of the whole barrel shifter circuit is shown in Fig. 5.5 (b). Multiplexer is the building block of a barrel shifter circuit. For an N-bit barrel shifter, there are a $log_2 N \times N$ array of multiplexers. The OR gate in the multiplexer circuit is found to be the optimal candidate to be replaced with the chaos gate. A fraction of the total $log_2 N$ chunks of N multiplexers are obfuscated using chaos logic. For example, there are 5 chunks of 32 multiplexers for a 32-bit shift operation. From the testability analysis, obfuscating only the 3^{rd} and 4^{th} chunks are found to be sufficient for provide optimal level of obfuscation. The obfuscated multiplexer circuit is shown in Fig. 5.5 (a). Based on the testability analysis of the whole barrel shifter circuit, the 3^{rd} and 4^{th} rows of the multiplexer array are chosen to be implemented by the obfuscated multiplexer arrays.

5.2.2 Configuration Key Management

Each obfuscated datapath requires a valid configuration key for correct execution of respective instructions. The configuration key is demultiplexed in the execute stage and applied to the respective datapath for the ALU instructions. The selector of the demultiplexer is the original opcode of the instruction. For an instruction to be executed correctly, the configuration key has to be among the valid key sets for a particular datapath implementation.

For the MiniRISC-V, each chaos gate is designed to be reconfigured using a 6-bit key as described in Chapter 4. Since each gate in the data path of a particular instruction is of



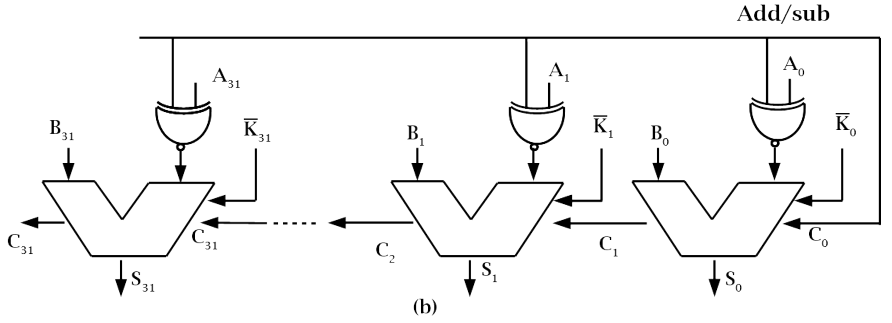
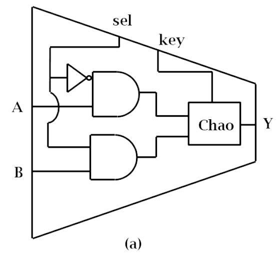


Figure 5.4: (a) Chaos obfuscated adder block used in arithmetic instructions (b) Obfuscated datapath of 32bit arithmetic instruction in MiniRISCV.



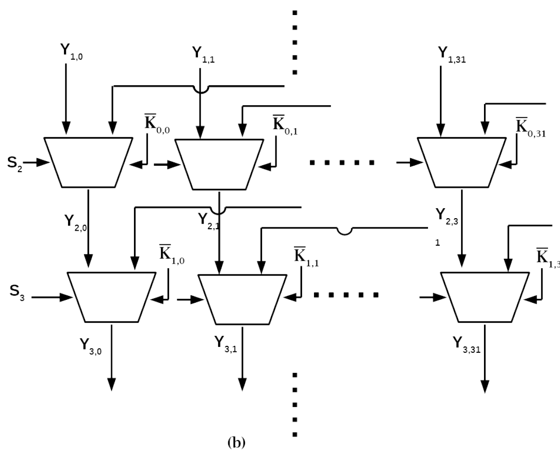


Figure 5.5: (a) Chaos obfuscated Mux block used in the barret shifter circuit (b) Obfuscated datapath of 32-bit shift instruction in MiniRISCV

the same type, a single 6-bit key is sufficient to use for each of them. However, chaos gates in each instruction datapath of instructions use multiple sets of keys in order to increase the functional ambiguity from an attacker's perspective. For example, 32 chaos gates in a logical instruction use 2 independent key sets for obfuscation. The distribution of key sets among the gates can be chosen by the designer. In this work, gates corresponding to the first 16 gates use the first key set while the rest use the second one. The same key to the both key sets would unlock the instruction. However, an attacker does not know which bit position in the overall key goes to which instruction and is forced to use brute force when attacking. Similarly, arithmetic instructions use two sets of 6-bit key and each of the logical shift and arithmetic shift instructions uses 4 sets. The total key size is 96 bits for this design. However, a designer can easily make a larger key by either increasing the bits in each key set or increasing the number of sets used in each datapath's circuit. Increasing the number of bits in each key set requires changing the resolution of encoding between digital and analog conversion of the chaos circuit. Key distribution in MiniRISC-V is illustrated in Fig. 5.6.

Configuration keys are stored in a key register. Opcode of each instruction is decoded to fetch the stored key of the corresponding datapath from the key register. The fetched key is distributed to the chaos gates used for obfuscation. key propagation from the key register to the instruction datapaths of the execution unit is illustrated in Fig. 5.7. The key register is write only to user in order to prevent exposure of the key. Therefore, an adversary cannot find the key without invasive attack such as micro-probing. Moreover, the configuration key is machine specific. Implementation of the same microarchitecture on different physical chips have unique key combination for the obfuscated instruction set. If a configuration key can be revealed from a particular processor chip which should be difficult, it does not affect the security of the other chips. Configuration key of every processor chip has to be extracted individually. The MiniRISC-V thus has an extra layer of security for code execution in addition to the basic logic encryption technique. Machine specific keys for an logic obfuscated hardware can also be achieved by mapping the original key to a PUF's challenge. In this case, required challenge would be unique in order to map to the same key in different chips. However, the methodology developed in this work has intrinsically unique keys for the obfuscated hardware due to process variation of chaos based logic gates.

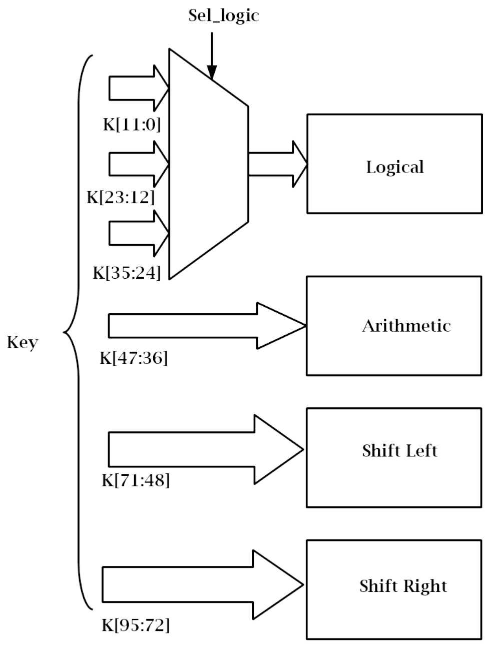


Figure 5.6: Distribution of configuration key among chaos based data path of different ALU instructions in the MiniRISC microarchitecture.

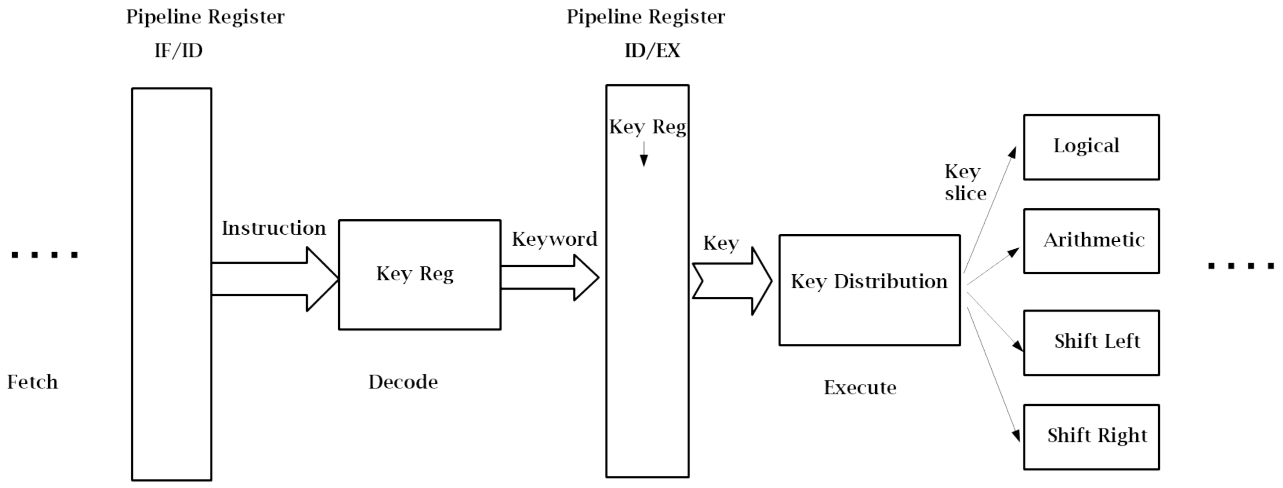


Figure 5.7: Illustration of key decode and propagation from fetched instruction to the execute unit.

5.2.3 RRAM Data Memory

RRAM with the integrity checking protocol proposed in Chapter 3 is used as the data memory of the MiniRISC-V microarchitecture. Security of tag generation has been analyzed from the perspective of various cryptographic properties and found to meet the requirements. This memory design facilitates providing data integrity with significantly less overhead as compared to conventional designs for tag generation. Memory instructions in the MiniRISC-V microarchitecture are designed in such a way that integrity checking is a part of instruction execution. There are 3 memory write instructions, sw,sh,sb and 5 read instructions, lb,lh,lw,lbu,lhu in the RV32I ISA.

The integrity checking protocol is divided into two phases. The first step is tag generation and store. The RRAM generates a tag on the existing data prior to each memory access. During the execution of a memory write instruction, a new tag is generated from the updated memory and saved in a secured tag register only accessible to internal logic. In the second phase, tag is generated from the existing memory and compared with the saved tag from the tag register prior to the execution of both load and store instructions. In a byte accessible memory, it is necessary to perform the verification in both load and store instructions. The tag is generated on a block of data in the memory which is a multiple of bytes. But the memory can be written with just a single byte for a byte accessible memory. If tag comparison is not performed during store instruction, an authorized write between two authorized write instructions can not be detected because the tag register will be updated considering the modified data. However, if integrity verification is performed on a cache memory where write is performed on a whole block instead of a byte, tag comparison during only load instruction would be sufficient. Integrity of memory data is verified upon a match between the generated and stored tag. An exception is raised indicating that the memory data has been compromised if a mismatch occurs between the two tags. Comparison is performed in the write back stage after the regenerated tag is available at the MEM/WB pipeline register. Processor-memory interaction during the execution of store and load instructions are illustrated in Fig. 5.8.

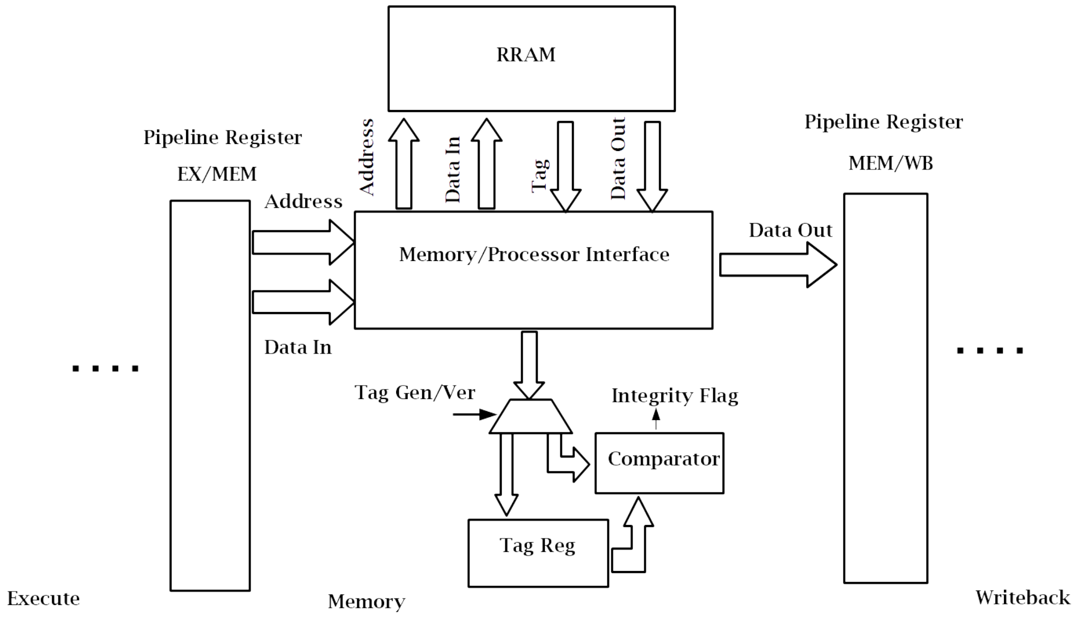


Figure 5.8: Datapath for interaction between MiniRISCV processor core and RRAM data memory with integrity checking protocol.

The tag generation and regular read-write operations are performed at a different clock cycles used in the RRAM memory circuit as described in Chapter 3. The internal clock used in the memory circuit is faster than the processor clock in order to accommodate the memory operations with tag computation in a single cycle of the pipeline.

5.3 Program Execution

5.3.1 Translating from Assembly

Assembly language is the symbolic form of machine code that a computer can understand. A translator, or assembler, translates assembly code into binary machine code which is ready to be executed. In this work, a Xilinx block RAM memory IP is used as the program memory. This memory IP facilitates initialization using a coefficient file where the machine instructions are written according to a format specified by the radix number used in the file header. A python program is developed as the assembler in order to translate the assembly into the coefficient file of block RAM. The assembler code is provided in the Appendix C.

The coefficient file can also be used in order to load a new program on to the processor during run time.

Traditional RISC-V assembly code conventions are followed in order to write assembly programs for this MiniRISC-V processor [70]. An user may follow the standard RISC-V software conventions in order to use registers for writing assembly programs for MiniRISC-V. Registers are symbolized as x followed by a number between 0 and 31 to indicate different registers in the register file. Register x0 always has a value of 0. Register x1 is used for holding the return address of a procedure. Similarly, registers x2 and x8 are used as the stack pointer and frame pointer, respectively. Registers x10-x17 are used to send arguments or hold results of a function. x5-x7 and x28-x31 are used as temporary registers. All other registers are used as general purpose registers. The values of general purpose registers are saved to the stack memory before modifying them inside a procedure. An example assembly and the machine code for division operation is shown in the List. 5.1 and 5.2, respectively.

Listing 5.1: Assembly code

```
addi x3,x0,102
addi x4,x0,7
xor x5,x5,x5
xor x6,x6,x6
addi x6,x3,0
blt x6,x4,loop_s
loop_div:
addi x5,x5,1
sub x6,x6,x4
bge x6,x4,loop_div
loop_s:
jal x1,loop_s
```

Listing 5.2: Machine code

```
06600193
00700213
0052c2b3
00634333
00018313
00434663
00128293
40430333
fe435ae3
ffdff0ef
;
```

5.3.2 Compiling from Higher Level Language

MiniRISC-V programs can also be translated from higher level language programming languages such as C. The GCC compiler for RISC-V RV32I instruction set compiles a C program into assembly code. This assembly contains pseudo-instructions which are slightly different in their symbols and syntax than the regular assembly instructions. Pseudo-instructions provide better readability for further processing steps used in the translation procedure. For example, the mv pseudo instruction used to copy a register content into another is equivalent to the addi instruction where one of the source operands is zero. Similarly, the jump and link instruction jal x0, Label is replaced with only j in the pseudo-instruction set. The assembler converts the pseudo instructions into an object file consisting of machine instructions, data and other information necessary for placing the instructions properly in the memory. For branching to different places in a program, the assembler needs to know the address of all the labels in the program. This task is accomplished using a symbol table where the labels and their corresponding addresses are stored as references for the assembler [70]. An object file contains the file header, text segment, data segment, relocation information, symbol table and debugging information [70]. Each procedure used in a program is compiled and assembled into individual objects files. A link editor also known as linker combines them into a single executable file ready to run on the computer. A custom linker is developed and associated with the RISC-V toolchain to generate machine code from a C code.

5.4 Security Analysis

The MiniRISC-V provides security for three different scenarios: functional obfuscation using chaos based reconfigurable logic prevents unauthorized code execution in the processor; functional diversification of chaos based logic along with the process variation makes it secure against instruction reverse engineering based on power analysis; RRAM data memory with embedded integrity checking protocol enables detection of unauthorized memory writes and thus provides integrity.

Three different test programs have been used in order to evaluate various security features of the MiniRISC-V microarchitecture. All programs are written in the assembly language of RV32I instruction set. Assembly programs are translated to MiniRISC-V machine code using the assembler tool shown in Appendix C.

5.4.1 Code Execution

The execution datapath of MiniRISC-V is logic-locked. A valid key is required in order to execute code correctly on this processor. An invalid key is expected to produce an incorrect output. In order to evaluate the effectiveness of this execution locking method, outputs of the test programs for randomly generated keys are compared against the desired output. The data memory is initialized with random data in order to mimic the run time condition. Due to use of an invalid key, each ALU instruction produces incorrect responses. These incorrect responses from each instruction propagate through other obfuscated instructions and produce the final outputs of the program which are not the desired output of the program. In this way, an adversary cannot execute a code on the processor without knowing the valid key.

Three test programs used here in order to test the security properties are bubble sort, modular exponentiation, and SIMON encryption. Bubble sort takes an input array and returns the sorted array. For testing purpose, an integer array of size 5 is used for the bubble sort program. The modular exponentiation takes a base (x), exponent (y), and modulus(n) as the input and returns $x^y \bmod n$. The modulus n is restricted to 8 bits for faster execution of the program. The SIMON encryption program implements the round function of a 64 bit version of SIMON cipher where both the plain text and the cipher text are 64 bit wide. Table 5.1 shows the output of each test program for the valid key and 3 other randomly generated keys. It can be seen that program generates corrupted outputs for all invalid keys which prevents unauthorized code execution on the MiniRISC-V.

In this obfuscation method, only ALU instructions are logic-locked using reconfigurable chaos gates. However, output corruption of ALU instructions due to an invalid key affects the whole program. Obfuscation does not only affect functionality but also various other characteristics of program execution such as program control flow and memory traces. Control flow is the order in which a program executes its various instructions. Control flow

Table 5.1: Results of obfuscated execution of several programs. Input-Output for the programs are shown for both valid key and several randomly chosen key.

Program	Input	Output			
		key_{valid}	key_{rand1}	key_{rand2}	key_{rand3}
Bubble sort	a2,22,2d,54,66	22,2d,5a,66,a2	6e,7c,8a,98,a6	2f,30,31,32,33	f6,07,18,29,3a
	fd,ff,31,4b,c0	31,4b,c0,fd,ff	28,36,44,52,5e	51,52,53,54,55	12,23,34,45,55
	8b,3f,c8,dc,12,3f	12,3f,8b,c8,dc	e2,f0,fc,0a,18	72,73,74,76,77	2e,3f,4f,60,71
Mod. exponent.	x=d5,y=c8,n=80	61	fe	0	0
	x=b3,y=1f4,n=40	11	fe	0	0
	x=80,y=200,n=3c	10	fe	0	0
SIMON encrypt.	802c4400802d5200	52006c02	adff9fff	00230000	d0009fff
	804dfe00804f0c00	0c00d402	f3ffe5ff	00000000	0000e5ff
	806fb8008070c600	c6007402	39ff2bff	00000000	000002bff

and branch instructions are not explicitly obfuscated in the MiniRISCV microarchitecture. However, the corruption in the ALU result due to obfuscated ALU instructions corrupts the register file which in turn affects the execution of branch and control flow instructions. Similarly, memory read and write operations are not modified in the microarchitecture. However, the memory address comes from the ALU result which is obfuscated by chaos logic gates. Therefore, the memory trace of a program execution is also corrupted due to an invalid key input.

Control flow of a test program, modular exponentiation with the valid key and 3 random keys are illustrated in Fig. 5.9 where the x and y-axes represent control flow break points and corresponding instruction addresses, respectively. It can be noted that each point in the control flow curve indicates a discontinuity in the program execution. It may be either a procedure call, branch instruction or return from a procedure. Odd segments in the control flow curve represents the continuous execution without any break in the control flow and the even segments indicate a discontinuity in the program execution due to a control flow or branch instruction. However, this is not illustrated in the curve in order to make the curve more readable. The modular exponentiation program is chosen for this illustration since it is the program with the most branch and control flow instructions among the three programs. It can be seen that the processor follows a unique sequence for the execution of its instruction due to the invalid keys. Control flow is shown here for a particular portion of the overall execution time from the beginning of the program.

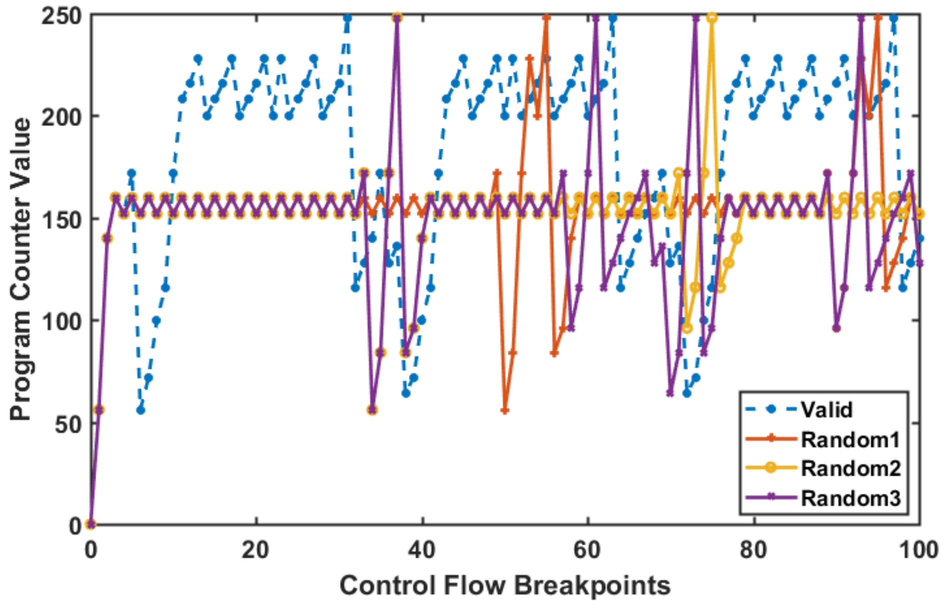


Figure 5.9: Control flow of the modular exponentiation program for the valid key and three random keys.

Memory trace is another program behavior considered here in order to analyze the effect of incorrect key on the program execution. Memory trace is a record of accessed memory locations during a program execution. Fig. 5.10 shows memory traces for the bubble sort program with the valid key and three other random keys. The memory trace curve indicates the accessed memory addresses in the y-axis and the index of memory instruction in the x-axis. It is clearly seen that due to the logic locking of ALU instructions the memory access behavior is completely different from one another during the execution of a program in MiniRISC-V. Both the number of memory access events and the accessed addresses vary due to invalid keys being applied to the obfuscated execution unit.

5.4.2 Power Analysis

Power analysis attacks for extracting sensitive information has been a persistent threat to the security of embedded computing. One of the more significant security features of MiniRISC-V is that it is resilient to power profiling attacks for instruction reverse engineering. The execution unit of MiniRISC-V is built using a combination of chaos and CMOS logic gates. Each copy of MiniRISC-V is unique in terms of its power profiles due to the process variation

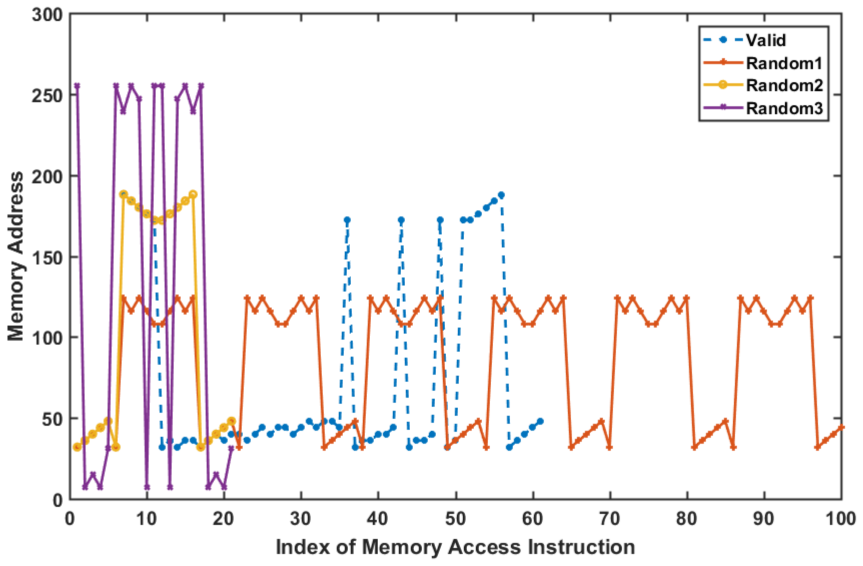


Figure 5.10: Memory trace of the bubble sort program for the valid key and three random keys.

of chaos gate as shown in chapter 4. In traditional microprocessors, instructions can be profiled based on their power traces for various operand data. Generated power profiles can be used in order to classify instructions on any processor of the same model since each copy is identical. The power based instruction classification attack is demonstrated on a 4-bit ALU in chapter 4 where it can be seen that the chaos based implementation mitigates the attack to a great extent. In this chapter, a power attack on the 32-bit MiniRISC-V is performed for both chaos based and CMOS only implementations. The first phase of the attack is to collect power traces for each ALU instructions in order to build profiles.

Instantaneous power traces are required for the instruction classification attack. Transistor level simulation provides instantaneous power traces with a cost of high simulation time, specially for a large system. Therefore, it is inconvenient to simulate the whole system in transistor level simulator in order to collect power profiles for each instruction. A simplified high level power model is developed here for generating power traces used in the power analysis attack. Description of the power model used to generate power profiles are given in Appendix B.

At any given time, multiple instructions are in various phases of execution in a multi-staged pipelined processor like MiniRISC-V. In order to classify instructions, a particular stage is targeted. In this work, the execution unit is used as the target since it is the most instruction dependent unit of the pipeline. Power consumption by this unit is the primary contributing factor to classification. Power consumption by the execution unit is modeled based on the transistor level power trace of each building block of the datapath. For instance, power consumption of a 32-bit AND operation can be generated by simply adding the power consumption of each 2 input single bit AND gate. It is sufficient to simulate a 2 input AND gate in transistor level for all possible input transitions in order to generate the power profile for a larger bitwise AND operation. Similarly, power consumption for ADD operation can be developed by adding the power consumption of each full adder circuit sequentially for respective input transitions. Block diagram for a generalized ALU instruction is shown in Fig. 5.11. Generalized power model for this instruction can be expressed as follows:

$$P_{ins} = P_{init} + \sum_{i=1}^{m-1} \left(\sum_{j=0}^{n-1} P(\Delta A_{i,j}, \Delta B_{i,j}) \right) u(d(i-1,i)) \tag{5.1}$$

where $P(\Delta A_{i,j}, \Delta B_{i,j})$, represents the power trace for an input transition of $(\Delta A_{i,j}, \Delta B_{i,j})$; P_{init} represents the power consumption at the first time step; $d(i-1,i)$ represents the delay between the i^{th} and j^{th} sequential operations; n is the number of sequential stages of operations; m is the number of parallel operations in each sequential stage; u is the unit step function.

For logical instructions, each of the 32 gates operates in parallel which implies $m = 32$ and $n = 1$ in Eq. 5.1. Similarly, $m = 1$ and 5 for the arithmetic and shift instructions, respectively and $n = 32$ for both.

Power consumption contributed by other pipeline stages contributes as noises to the classification of instructions since the target stage is execution. An instruction in the execution stage with specific operand values can exist in various combination of other 4 instructions being executed in other pipeline stages.

The remaining analysis is the same as that described in chapter 4. There are 7 ALU instructions as categorized into 3 categories: logical, arithmetic and shift. AND (and,andi), OR (or,ori) and XOR (xor,xori) are the logical instructions, ADD (addi,add) and SUB (sub) are the arithmetic instructions, SHL(sll,srl,slli,srli) and SHA(SRA) are the shift instructions

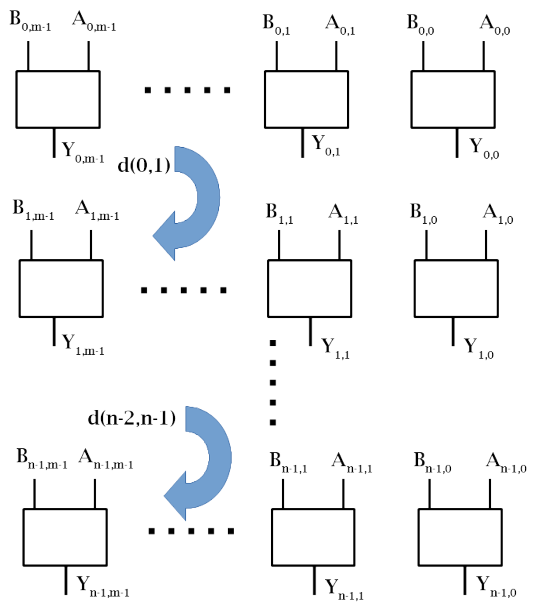


Figure 5.11: Generalized block diagram of an instruction datapath in the ALU used for the power model. n stage of sequential operation with m parallel operation in each stage.

in the RV32I instruction set. 5000 power traces are collected for each instruction. 1000 samples are collected for each trace in CMOS only instructions. Since the delay of a chaos gate is higher than the static CMOS gate, 1600 samples are collected for each power trace in the chaos based implementation.

The classification attack is performed on MiniRISC-V using the KNN, SVM and MVG classification algorithm. The nearest neighbor (K=1) is considered for the KNN method with Euclidean distance measurement. Classification is performed without reduction in dimensionality as well as with reduction using PCA, SDM, MV and FLDA methods.

Results for a power analysis attack on the ALU instructions of the CMOS only implementation of 32-bit MiniRISC-V are demonstrated in Table 5.2. The classification accuracy is found to be nearly 94% in the best case. The best case results are found for the combination of SVM algorithm and FLDA. Results of the best case scenario is presented in Table 5.3 as an inter-instruction confusion matrix. It can be seen that both logical and arithmetic instructions can be classified with very high accuracy. The shift instructions show less accuracy than other instructions due to their similarity in power profiles among themselves. The overall accuracy for this case is 94%.

Table 5.2: Classification accuracy among ALU instructions for different classifier and dimensionality reduction algorithm for CMOS implementation of MiniRISC-V .

category	**Accuracy with different data reduction**			
	PCA	SDM	MV	$FLDA$
K-NN	93	92	92	31
SVM	86	84	84	94
MVG	71	73	72	94

Table 5.3: Best Case Classification results of ALU instructions for CMOS gate based implementation of the MiniRISC-V. Best case accuracy is found for SVM with FLDA as the dimensionality reduction technique.

Instruction	**Matched Class (%)**						
	AND	**OR**	**XOR**	**ADD**	**SUB**	**SHL**	**SHA**
AND	99	0	1	0	0	0	0
OR	0	99	1	0	0	0	0
XOR	1	2	97	0	0	0	0
ADD	0	0	0	98	2	0	0
SUB	0	0	0	1	99	0	0
SHL	1	0	0	0	0	90	9
SHA	0	0	1	0	0	20	80

Similary, a power analysis attack is performed on the chaos based implementation of the MiniRISC-V in order to classify the 32-bit ALU instructions. In a profiling attack, power traces are collected from a reference processor. Instructions executing on the processor under test is then classified based on the power profiles generated from the reference. Different classification algorithms look at different features of the samples and classify accordingly. The advantage of the chaos based implementation is that each copy of the processor requires a unique set of functional key. Therefore a chaos gate exhibits unique power profile across different chips since the key in a chaos gate controls the physical parameters such as bias voltage, V_c, initial condition, x and the sampling iteration, n as shown in chapter 4. Power trace collected from the test processor is ambiguous to classify based on the classifier trained by the reference traces. Diversified power profile along with process dependent variation of chaos gates helps mitigating instruction reverse engineering by power analysis. Four chaos based MiniRISC-V chips are considered for demonstrating the attack. Functionality tables of chaos blocks in each chip can be built using the results of Monte Carlo simulation. Various combinations among these chips are considered for the reference and target processors for classification. Table 5.4 shows the classification results for all these combinations of chaos based MiniRISC-V chips. The same classification algorithms and dimensionality reduction techniques are used as before. It is found that the MinIRISC-V instructions on chaos based implementations can be classified with only 36% accuracy at best which is significantly lower than that of the CMOS only implementation. Best case results are achieved by the K(=1)NN with dimensionality reduced by FLDA. The confusion matrix for inter-instruction classification is shown in the Table 5.5. The instructions are confused among themselves due to diversity in power traces of an instruction across different processor chips.

Assumption made in this analysis is that an adversary can not collect power samples of known instructions on the processor under test. MiniRISC-V is a logic-locked processor where only an authorized user who knows the valid key for unlocking the microarchitecture, can execute code successfully. A random key results in an invalid outcome for a given instruction. Therefore, it is not possible to train a classifier with power traces for known instructions from the target processor. An adversary can only collect power profiles from the processor she is authorized to use. Even if the adversary is able to extract the key from

Table 5.4: Classification accuracy of instruction set among different copy of chaos based MiniRISC-V with several classification and data reduction algorithms.

$category$	$data\ reduction$	$config\ pair$						
		$1,2$	$1,3$	$1,4$	$2,3$	$2,4$	$3,4$	$average$
$K(=1)NN$	PCA	19	17	3	41	31	45	26
	SDM	22	30	8	42	49	31	30
	MV	23	30	10	42	47	30	30
	$FLDA$	29	30	29	45	55	27	36
SVM	PCA	31	24	15	43	16	55	31
	SDM	20	32	8	31	34	25	25
	MV	20	31	9	31	35	24	25
	$FLDA$	14	14	14	14	14	14	14
MVG	PCA	14	14	14	14	17	28	17
	SDM	26	36	29	25	28	26	28
	MV	26	37	30	25	28	22	28
	$FLDA$	23	14	14	22	15	14	17

Table 5.5: Best case average classification results for Chaos gate based implementation of MiniRISC-V. Best results found for K(=1)NN classifier with FLDA.

Instruction	**Matched Class (%)**						
	AND	**OR**	**XOR**	**ADD**	**SUB**	**SHL**	**SHA**
AND	22	50	14	5	8	0	0
OR	10	73	6	5	6	0	0
XOR	18	43	33	5	2	0	0
ADD	16	49	6	9	15	1	4
SUB	15	49	7	9	14	1	4
SHL	0	0	0	0	0	65	34
SHA	0	0	0	0	0	65	34

a processor by applying a large number of computational resources, the extracted key does not apply to other copies. Therefore, reverse engineering instructions for the chaos based implementation of MiniRISC-V using power analysis attack is extremely difficult.

The hypothetical scenario where two chaos based MiniRISC-V chips do not have process variation is also studied here with regards to the power analysis attack. In this case, two different processor chips of the same model can be unlocked using the same key. Since the power profiles of a chaos gate depends on the functional key, both chips would have a similar power profiles for each instruction and can be classified with higher accuracy. Results of classification for such a case is shown in the Table 5.6. It can be seen that the instructions can be classified with an accuracy of 72% in this case. This analysis indicates that process variation helps in a chaos based computation to mitigate profiling attacks based on power analysis. Nevertheless, defense against such attack using a chaos gate would not be effective without the intrinsic chaos property. Unlike the conventional static CMOS logic operations, process variation is amplified in a chaos gate due to the chaotic evolution described in Chapter 4. Such amplification leads to a unique power profile for each instruction across different chip. However, amplified process variation would be considered a security advantage until the functionality of the chip can be ensured. Due to the large functionality space of chaos logic, a chaos based chip can be fully functional (with a different key) enduring large process variation.

The classification accuracy is also measured for the ALU instructions used in various benchmark applications developed for MiniRISC-V. Best case classification algorithms found from earlier analysis are used here for CMOS and chaos based implementation, respectively.

Table 5.6: Best case average classification results for the hypothetical case where no process variation is considered between two chaos based implementation of MiniRISC-V processors.

Instruction	**Matched Class (%)**						
	AND	**OR**	**XOR**	**ADD**	**SUB**	**SHL**	**SHA**
AND	99	0	1	0	0	0	0
OR	0	100	0	0	0	0	0
XOR	1	0	99	0	0	0	0
ADD	1	0	1	47	47	2	2
SUB	1	0	1	47	50	1	0
SHL	1	0	0	1	0	57	41
SHA	1	0	0	0	1	47	52

The purpose of the classification attack considered in this work is to extract the information of ALU execution. The instructions that do not require the ALU for execution are not classified in this work. The variants of an instruction where the ALU operation is the same is considered as the same instruction. For instance, the classifier does not differentiate add and addi which is often unnecessary. Memory instructions are identified as add since the underlying ALU operation for a memory instructions use add when calculating the address of memory access.

The overall classification accuracy of the test program depends on the ratio of different types of ALU instructions used in the program. From an earlier analysis, it is found that the shift instructions show lower accuracy than others. Therefore, the overall accuracy of a program would be higher if fewer shift instructions are used. Fig 5.12 shows the overall accuracy for classification of the ALU instructions used in several benchmark programs developed for the MiniRISC-V.

5.4.3 Memory Integrity

The MiniRISC-V is designed to use the RRAM described in chapter 3. The RRAM is designed in such a way that it can generate data tags upon read and write. During the execution of memory write (sb, shw, sw) instructions, the memory generates a tag on the updated status of the memory and saves it in a tag register. The tag register is an additional storage unit used only for storing the tag and is itself not directly accessible. It is assumed that an unauthroized party cannot access the tag register while modifying the memory contents. Value of the tag register is updated only if the write is authorized. Prior to each memory access (load/store), the memory performs a tag generation on the existing data. Without any adversarial modification in the memory, the newly generated tag upon each memory access will be identical to the tag register during the last write instruction.

Detailed implementation and performance of the integrity checking scheme using RRAM is described in chapter 3 where quality of the tag is evaluated based on various cryptographic properties such as uniformity, diffusion, and avalanche. Security against a birthday attack is also evaluated by measuring the collision rate which is a function of uniformity. It is found

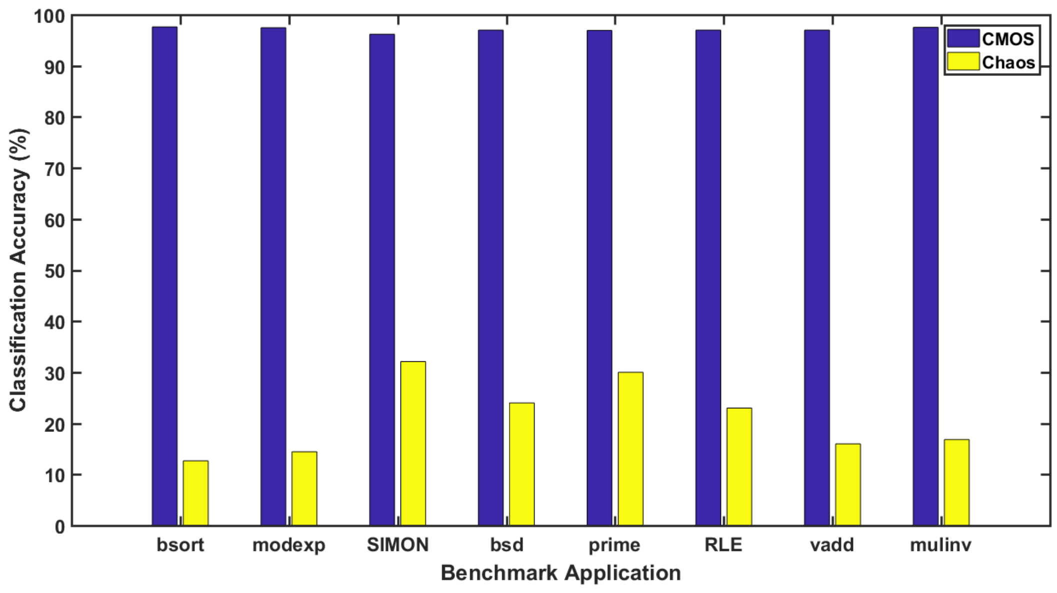


Figure 5.12: Instruction classification accuracy in different benchmark applications for both CMOS and chaos based implementations of MiniRISC-V processor.

that the integrity checking system can be secure by choosing proper design parameters suggested by the developed analytical model.

Memory integrity of MiniRISC-V is examined by emulating a direct memory access attack as a representation of unauthorized memory modification. The 3 test programs are also used here in order to examine the memory integrity of MiniRISC-V. The memory prototype used for testing data integrity in this work is 256 bytes long. In order to emulate direct memory access (DMA) behavior, the memory is given the access to an external port. The port can directly access the memory without any supervision of the processor. A DMA memory modification is required to be detected by the processor in order to ensure data integrity. A simple authorization scheme is also developed to support the integrity verification protocol. In this case, if memory is accessed by the processor itself with regular store instructions, the generated tag will be stored in the tag register. On the other hand, prior to each memory access, the processor receives the tag generated from the current memory and compares it to the tag register. An integrity flag then shows the integrity status while encountering a mismatch between the new and previously saved tag.

The detailed circuit level implementation of the RRAM with data integrity checking is shown in chapter 3. For emulating this attack, the MATLAB based memory model is used to generate tag from a given memory status in order to emulate the DMA attack. A MATLAB based ISA simulator is also developed for faster investigations of MiniRISC-V behavior. The simulator calls the memory model function at each memory access in order to generate tag. The memory saves tag to the tag register unless the memory is access without the authorization of the processor. In that case, the new tag generated in each memory access would not be identical to the one saved in the tag register in the previous write operation.

During the execution of each test programs, a DMA event is created at a regular interval throughout the whole execution window. The integrity flag of the processor is observed at every memory read operation in order to measure the success rate of the integrity verification protocol. Table 5.7 indicates that the MiniRISC-V can detect an unauthorized modification during the execution of the test programs with a success rate of 100%. This also can be inferred from the fact that the tag generating function has a high collision resistance of nearly 2^{-N} for a targeted collision in $N-$ bit tag. An adversary requires 2^N trial in order to choose data randomly that generates the same tag as the present tag register value. The time difference between the execution of two memory instructions are very low as compared to the time required for 2^N trial for unauthorized write event for a moderately large value of N. Therefore, it is very unlikely that an attacker can be able to remain unnoticed after modifying the memory with an unauthorized access.

Table 5.7: Success rate of the RRAM based integrity verification method in MiniRISC-V processor.

Program	Memory Instruction (%)		DMA event	Success Rate
	Store	Load		
Bubble sort	25	21	91	100
Mod. exponent.	11	11	54	100
SIMON encrypt.	10	10	128	100

5.5 Cost Analysis

Security comes with an implementation cost. Manufacturers often analyze the cost of damage done by different attacks with respect to the cost of implementing countermeasures. However, resource constrained IoT edge devices can hardly afford expensive countermeasures. In a practical scenario, the countermeasures need to be more comprehensive and lightweight in order to apply to the modern embedded computers used in edge devices.

The RISC-V embedded computer designed in this work provides security to both the processing core and memory. It provides security against a very broad range of vulnerabilities. The execution unit is logic-locked by using chaos gates which provides secure execution of programs. The very same design also provides security against side channels, specifically power analysis. Memory security is provided by an RRAM design used to implement the data memory, which can inherently generate authenticating tag for verifying memory integrity. Since the tag is generated by using the sneak path current based in-memory computing of RRAM, very minimal additional hardware is required for tag generation and integrity checking. The overhead is reduced significantly in this design as compared to other cost effective designs used for same purpose.

Cost analysis for the complete design of this work is performed. In earlier chapters, detailed descriptions of hardware designs and security analyses of individual security components are presented. Table 3.5 and Table 4.11 demonstrate the implementation cost of comparable methods in order to provide memory and processor security, respectively. The design cost for MiniRISC-V from the perspective of area, power and energy is estimated and presented in Table 5.8. For a side by side comparison, Table 5.8 also presents the cost of MinIRISC-V with a hypothetical design that includes comparable security techniques for each of the vulnerabilities considered in this work. A base line implementation of MiniRISC-V without any countermeasures is used to estimate the overhead for the secure implementations. The cost of the CMOS only implementation of MiniRISC-V is estimated from the Synopsys Design Compiler.

Table 5.8: Cost comparison of MiniRISCV for different implementations with and without security measures.

MiniRISCV Design	Area		Power		Energy	
	value(um^2)	OV	value(mW)	OV	value(pJ)	OV
Static CMOS gate	1413	1x	7.13	1x	40	1x
STT-RAM LUT [86]+iDDPL[6]+CETD [33]	2384	1.68x	40	5.6x	292	7.3x
Chaos+RRAM	1639	1.2x	10	1.4x	80	2x

5.6 Conclusion

The proposed security extensions of the MiniRISC-V microarchitecture using RRAM and chaos based logic can mitigate a number of security threats using significantly less overhead. These security oriented design techniques help verifying memory integrity, mitigating unauthorized code execution and side channel power attacks. In order to correctly execute a program in this platform, one needs to have the knowledge of the valid configuration key. Besides security of the processing unit, the memory instructions have a built-in integrity verification feature in this modified architecture. This addition of integrity to the memory instruction can be very helpful in order to detect unauthorized data modification and avoid memory based attack hazards.

Chapter 6

Conclusions and Future Prospects

6.1 Summary

The goal of this dissertation work is to design a secure computing microarchitecture for embedded system applications using emerging technologies. Security measures have been developed for both memory and processing unit which are the two basic blocks in a traditional Von Neuman or Harvard computer architecture. The security measures are effective against unauthorized memory data modification, arbitrary code execution and instruction reverse engineering by side channel power attack. Performed works are summarized as follows:

An RRAM design is developed with integrated tag generation mechanism for integrity checking purpose. Sneak path currents in the RRAM crossbar array are leveraged in order to generate tag from memory data. Randomnesses are added to the design for security robustness by keeping a reserved memory row and shuffled column sampling. Both circuit level and MATLAB based high level designs are performed for the RRAM in order to perform thorough security and performance analysis by simulation.

An analytical model is developed for guiding a designer to choose optimal design parameters for ensuring the security of tag generation. Design parameters in terms of crossbar sizes, value of load resistance, tag sizes can be found from the probability distribution of each tag bit. Desired security properties are validated based on the optimal design parameters suggested by the analytical model. The analytical model is itself validated against the results from circuit level simulation of the design.

A chaos based logic design methodology has been developed in order to design a secure processing unit for embedded system applications. Designed chaos gates are reconfigurable and has a very large functionality space. Reconfigurability facilitates implementing all basic 2-input Boolean functions from the chaos gate. Each Boolean function also can be implemented using multiple chaos configurations providing more flexibility and security. The chaos gate design exhibits device to device variation in functionality as well as power traces due to variation in process.

A technique for obfuscated execution using reconfigurable chaos gate is developed. An ALU circuit has been logic-locked by replacing some of the logic gates using the chaos logic. The replacement locations are selected based on testability analysis. Optimal number of chaos gates required for obtaining the desired level of obfuscation is found using the replacement heuristic. The obfuscated circuit provides correct functionality when a correct key is applied. Functionality obfuscation is evaluated in terms of hamming distance between the circuit output corresponding to the correct and random key. This obfuscated computing technique prevents unauthorized execution of micro instructions on a computing unit. The optimal design provides a functionality obfuscation of nearly 48% hamming distance which is very close to the desired level of obfuscation, 50%

Side channel security of the chaos based design obfuscation is evaluated against power analysis based instruction reverse engineering attack. Instructions are classified using different classification algorithms on pre-generated power template. K nearest neighbor, support vector machine and multivariate Gaussian, these 3 algorithms are used for classification in this work. Best case instruction recognition rate is 36% for the chaos based obfuscated design where the instructions based on static CMOS gates can be recognized with an accuracy of as high as 94%.

A microarchitecture, MiniRISC-V is developed for a subset of RISC-V 32I instruction set. Security enhancing modifications are performed by obfuscating the execution unit using chaos gates. The RRAM based memory design with integrated tag generation protocol is used as the data memory for the MiniRISC-V microarchitecture.

Additional logic is developed for facilitating load and store instruction with built in integrity checking. New tag is generated from the data memory during store instructions and saved to the tag storage. Before each memory access, tag is regenerated on the existing memory and compared with the previously stored tag. A status flag is used to show the integrity status upon mismatch of the regenerated tag with the saved tag.

Various test programs have been developed in RISC-V assembly for executing on MiniRISCV. Security properties are evaluated from the perspective of different security features of the processor designed in this work. Unauthorized code execution is evaluated using various program behaviors such as output, control flow and memory trace, corresponding to a valid key and few randomly generated keys. Resiliency toward side channel based instruction classification attacks is tested by the recognition rate of the instructions used in the test programs based on their power profiles. A DMA attack is modelled using the MATLAB based custom emulator of the MinIRISC-V in order to test the success rate of detecting unauthorized modifications in the memory.

6.2 Conclusions and Future Prospects

The embedded processor developed in this work addresses three significant inter-disciplinary security issues by using emerging technologies in computing such as chaos logic and RRAM memory. Inclusion of chaos logic basically locks the datapath of execution unit in order to provide security against unauthorized code execution. The very same design can also be used against power analysis based instruction classification attack. Resiliency to classification attack leverages the amplified process variation of a chaotic evolution and large space of functionality in chaos based logic gates. These properties of chaos based computation provides efficient schemes for designing secure system against multiple attacks with a single design technique. Besides the security of processing unit, the MiniRISC-V design also provides security for the memory by integrating a RRAM with in-memory tag computation capability. Integrity verification can be a part of each memory access in the MiniRISC-V with significantly less computational resources as compared to other existing methods. Overall, the MiniRISC-V design provides execution security, memory integrity and resiliency to side channel attack.

This work shows a great prospect of research in designing secure microarchitecture for RISC-V as well as other ISA. Exploring the key management in both user and the privileged mode of execution can be an exciting direction. Depending on the features of a particular system, the options may have unique advantages over one another.

This work considers a single core processor and single process execution scenario for studying various security vulnerabilities. Multi core and multi process execution brings interesting security problems. Future works on supporting these features on the MiniRISC-V in order to mitigate difficult security problems in an efficient and cost effective way. New found vulnerabilities can be studied in order to mitigate them using the idea used in this work with necessary architectural modifications.

The overhead of the chaos based logic is one of major concern in building larger systems. Exploration of novel devices in order to build chaos logic would be a great direction in order to reduce the overhead and make the chaos based computation more viable to be applied in versatile and larger systems. Maximizing the reuse of each chaos logic for implementing different logic operations would be a great way of fully utilizing the advantage of chaos. Effort in this direction seems highly promising.

Bibliography

[1] Alomair, B. and Poovendran, R. (2010). Efficient authentication for mobile and pervasive computing. In *International Conference on Information and Communications Security*, pages 186–202. Springer. 29

[2] Altun, M. (2016). Computing with emerging nanotechnologies. In *Low-Dimensional and Nanostructured Materials and Devices*, pages 635–660. Springer. 1

[3] Amer, S., Sayyaparaju, S., Rose, G. S., Beckmann, K., and Cady, N. C. (2017). A practical hafnium-oxide memristor model suitable for circuit design and simulation. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE. 16

[4] Baumgarten, A., Tyagi, A., and Zambreno, J. (2010). Preventing ic piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers*, 27(1):66–75. 72

[5] Bellare, M. and Kohno, T. (2004). Hash function balance and its impact on birthday attacks. In *Advances in Cryptology EUROCRYPT 04, Lecture Notes in Computer Science*, pages 401–418. Springer-Verlag. 26, 29, 30, 47

[6] Bellizia, D., Scotti, G., and Trifiletti, A. (2018). Tel logic style as a countermeasure against side-channel attacks: Secure cells library in 65nm cmos and experimental results. *IEEE Transactions on Circuits and Systems I: Regular Papers*, (99):1–11. 82, 85, 114

[7] Cafagna, D. and Grassi, G. (2005). Chaos-based computation via chua's circuit: Parallel computing with application to the sr flip-flop. In *International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.*, volume 2, pages 749–752. IEEE. 20, 57

[8] Cassuto, Y., Kvatinsky, S., and Yaakobi, E. (2016). Write sneak-path constraints avoiding disturbs in memristor crossbar arrays. In *Information Theory (ISIT), 2016 IEEE International Symposium on*, pages 950–954. IEEE. 17

[9] Chua, L. (2011). Resistance switching memories are memristors. *Applied Physics A*, 102(4):765–783. 14, 15

[10] Chua, L. O. and Lin, G.-N. (1990). Canonical realization of chua's circuit family. *IEEE transactions on Circuits and Systems*, 37(7):885–902. 20

[11] Clavier, C. (2004). Side channel analysis for reverse engineering (scare)-an improved attack against a secret a3/a8 gsm algorithm. 74

[12] Colombier, B., Bossuet, L., Fischer, V., and Hély, D. (2017). Key reconciliation protocols for error correction of silicon puf responses. *IEEE Transactions on Information Forensics and Security*, 12(8):1988–2002. 53

[13] Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE. 10, 24

[14] Deza, M. M. and Deza, E. (2009). Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer. 78

[15] Ditto, W. L., Miliotis, A., Murali, K., Sinha, S., and Spano, M. L. (2010). Chaogates: Morphing logic gates that exploit dynamical patterns. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 20(3):037107. 2

[16] Ditto, W. L., Murali, K., and Sinha, S. (2008). Chaos computing: ideas and implementations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1865):653–664. 2

[17] Duda, R. O., Hart, P. E., and Stork, D. G. (2012). *Pattern classification.* John Wiley & Sons. 77

[18] Dudek, P. and Juncu, V. (2003). Compact discrete-time chaos generator circuit. *Electronics Letters*, 39(20):1431–1432. 57

[19] Eisenbarth, T., Paar, C., and Weghenkel, B. (2010). Building a side channel based disassembler. In *Transactions on computational science X*, pages 78–99. Springer. 76

[20] Elbaz, R., Champagne, D., Gebotys, C., Lee, R. B., Potlapally, N., and Torres, L. (2009). Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*, pages 1–22. Springer. 28

[21] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., and Martinez, A. (2010). Transactions on computational science x. chapter Block-level Added Redundancy Explicit Authentication for Parallelized Encryption and Integrity Checking of Processor-memory Transactions, pages 231–260. Springer-Verlag, Berlin, Heidelberg. xiii, 11, 12

[22] Fang, Z., Yu, H. Y., Liu, W. J., Wang, Z. R., Tran, X. A., Gao, B., and Kang, J. F. (2010). Temperature instability of resistive switching on $hboxHfO_x$-based rram devices. *IEEE Electron Device Letters*, 31(5):476–478. 52

[23] Feistel, H. (1973). Cryptography and computer privacy. *Scientific american*, 228(5):15–23. 26

[24] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188. 77

[25] Fix, E. and Hodges Jr, J. L. (1951). Discriminatory analysis-nonparametric discrimination: consistency properties. Technical report, California Univ Berkeley. 78

[26] Gaba, S., Sheridan, P., Zhou, J., Choi, S., and Lu, W. (2013). Stochastic memristive devices for computing and neuromorphic applications. *Nanoscale*, 5(13):5872–5878. 43

[27] Gassend, B., Suh, G. E., Clarke, D., Van Dijk, M., and Devadas, S. (2003). Caches and hash trees for efficient memory integrity verification. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 295–306. IEEE. 11

[28] Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F. K., and Benini, L. (2017). Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713. 6

[29] Gibbons, J. and Beadle, W. (1964). Switching properties of thin nio films. *Solid-State Electronics*, 7(11):785–790. 15

[30] Gschwind, M., Salapura, V., and Maurer, D. (2001). Fpga prototyping of a risc processor core for embedded applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):241–250. 6

[31] Gut, A. (2013). *Probability: a graduate course*, volume 75. Springer Science & Business Media. 79

[32] Haifeng, M., Chengjie, and Zhenguo, G. (2014). Memory integrity protection method based on asymmetric hash tree. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2014 13th International Symposium on*, pages 263–267. 11

[33] Hong, M., Guo, H., and Hu, S. X. (2012). A cost-effective tag design for memory data authentication in embedded systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 17–26. ACM. 11, 54, 114

[34] Hsu, C.-W. and Lin, C.-J. (2002). A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks*, 13(2):415–425. 79

[35] Hund, R., Holz, T., and Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX security symposium*, pages 383–398. 11

[36] Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer. 78

[37] Juncu, V., Rafiei-Naeini, M., and Dudek, P. (2006). Integrated circuit implementation of a compact discrete-time chaos generator. *Analog Integrated Circuits and Signal Processing*, 46(3):275–280. 57

[38] Kamali, H. M., Azar, K. Z., Gaj, K., Homayoun, H., and Sasan, A. (2018). Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 405–410. IEEE. 72

[39] Kannan, S., Karimi, N., Sinanoglu, O., and Karri, R. (2015). Security vulnerabilities of emerging nonvolatile main memories and countermeasures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(1):2–15. 19

[40] Kannan, S., Rajendran, J., Karri, R., and Sinanoglu, O. (2013). Sneak-path testing of crossbar-based nonvolatile random access memories. *IEEE Transactions on Nanotechnology*, 12(3):413–426. 19

[41] Kia, B., Lindner, J. F., and Ditto, W. L. (2016). A simple nonlinear circuit contains an infinite number of functions. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(10):944–948. 20, 62

[42] Kia, B., Mobley, K., and Ditto, W. L. (2017). An integrated circuit design for a dynamics-based reconfigurable logic block. *IEEE Transactions on Circuits and Systems II: Express Briefs*. 62

[43] Knag, P., Lu, W., and Zhang, Z. (2014). A native stochastic computing architecture enabled by memristors. *IEEE Transactions on Nanotechnology*, 13(2):283–293. 43

[44] Kocher, P., Jaffe, J., Jun, B., and Rohatgi, P. (2011). Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27. 74

[45] Kohar, V., Kia, B., Lindner, J. F., and Ditto, W. L. (2017). Implementing boolean functions in hybrid digital-analog systems. *Physical Review Applied*, 7(4):044006. xiii, 23

[46] Lee, M.-J., Park, Y., Kang, B.-S., Ahn, S.-E., Lee, C., Kim, K., Xianyu, W., Stefanovich, G., Lee, J.-H., Chung, S.-J., et al. (2007). 2-stack 1d-1r cross-point structure with oxide diodes as switch elements for high density resistance ram applications. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pages 771–774. IEEE. 6

[47] Leemaster, J., Vai, M., Whelihan, D., Whitman, H., and Khazan, R. (2018). Functionality and security co-design environment for embedded systems. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5. IEEE. 6

[48] Leurent, G. (2008). Md4 is not one-way. In *International Workshop on Fast Software Encryption*, pages 412–428. Springer. 25

[49] Liu, B. and Qu, G. (2016). Vlsi supply chain security risks and mitigation techniques: A survey. *Integration*, 55:438–448. 1, 13

[50] Liu, B. and Wang, B. (2014). Embedded reconfigurable logic for asic design obfuscation against supply chain attacks. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE. 13

[51] Liu, T., Guo, H., Parameswaran, S., and Hu, X. S. (2016). Improving tag generation for memory data authentication in embedded processor systems. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 50–55. 11

[52] Lu, C., Zhang, T., Shi, W., and Lee, H.-H. S. (2006). M-tree: a high efficiency security architecture for protecting integrity and privacy of software. *Journal of Parallel and Distributed Computing*, 66(9):1116–1128. 11

[53] Macé, F., Standaert, F.-X., and Quisquater, J.-J. (2007). Information theoretic evaluation of side-channel resistant logic styles. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 427–442. Springer. 13

[54] Majumder, M. B., Hasan, M. S., Shanta, A., Uddin, M., and Rose, G. (2019). Design for eliminating operation specific power signatures from digital logic. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 111–116. ACM. x, xiv, 61

[55] Majumder, M. B., Uddin, M., and Rose, G. S. (2018). *Security from Sneak Paths in Crossbar Memory Architectures*, pages 344–359. CRC Press. x, xiv, 17, 19, 33, 38, 42, 50, 52, 53, 54

[56] Majumder, M. B., Uddin, M., Rose, G. S., and Rajendran, J. (2016). Sneak path enabled authentication for memristive crossbar memories. In *Hardware-Oriented Security and Trust (AsianHOST), IEEE Asian*, pages 1–6. IEEE. xiii, 19, 27, 31

[57] Manem, H., Rajendran, J., and Rose, G. S. (2012a). Design considerations for multilevel cmos/nano memristive memory. *J. Emerg. Technol. Comput. Syst.*, 8(1):6:1–6:22. 2

[58] Manem, H., Rajendran, J., and Rose, G. S. (2012b). Design considerations for multilevel cmos/nano memristive memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 8(1):6. 15, 17

[59] Matsumoto, T. (1984). A chaotic attractor from chua's circuit. *IEEE Transactions on Circuits and Systems*, 31(12):1055–1058. 20, 57

[60] McDonald, N. R., Bishop, S. M., Briggs, B. D., Van Nostrand, J. E., and Cady, N. C. (2012). Influence of the plasma oxidation power on the switching properties of Al/Cu_xO/Cu memristive devices. *Solid-State Electronics*, 78:46–50. 15

[61] Merkle, R. C. (1980). Protocols for public key cryptosystems. In *IEEE Symposium on Security and privacy*, volume 122. 11

[62] Mohammad, B., Homouz, D., and Elgabra, H. (2013). Robust hybrid memristor-cmos memory: modeling and design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(11):2069–2079. 30, 47

[63] Msgna, M., Markantonakis, K., and Mayes, K. (2014). Precise instruction-level side channel profiling of embedded processors. In *International Conference on Information Security Practice and Experience*, pages 129–143. Springer. 13, 74, 76, 78, 79

[64] Munakata, T., Sinha, S., and Ditto, W. L. (2002). Chaos computing: implementation of fundamental logical gates by chaotic elements. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(11):1629–1633. x, 21, 22

[65] Murali, K., Sinha, S., and Ditto, W. L. (2003). Implementation of nor gate by a chaotic chua's circuit. *International Journal of Bifurcation and Chaos*, 13(09):2669–2672. 20, 21, 57

[66] Mller, T. and Freiling, F. C. (2015). A systematic assessment of the security of full disk encryption. *IEEE Transactions on Dependable and Secure Computing*, 12(5):491–503. 10, 24

[67] Okamoto, T. (2015). Seconddep: Resilient computing that prevents shellcode execution in cyber-attacks. *Procedia Computer Science*, 60:691–699. 13

[68] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent {ROP} exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 447–462. 11

[69] Park, J. and Tyagi, A. (2017). Using power clues to hack iot devices: The power side channel provides for instruction-level disassembly. *IEEE Consumer Electronics Magazine*, 6(3):92–102. 13

[70] Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition. 98, 99

[71] Rajendran, J., Karri, R., Wendt, J. B., Potkonjak, M., McDonald, N., Rose, G. S., and Wysocki, B. (2015a). Nano meets security: Exploring nanoelectronic devices for security applications. *Proceedings of the IEEE*, 103(5):829–849. 2

[72] Rajendran, J., Zhang, H., Zhang, C., Rose, G. S., Pino, Y., Sinanoglu, O., and Karri, R. (2015b). Fault analysis-based logic encryption. *IEEE Transactions on computers*, 64(2):410–424. 70

[73] Rogers, A. and Milenković, A. (2009). Security extensions for integrity and confidentiality in embedded processors. *Microprocessors and Microsystems*, 33(5):398–414. 11, 54

[74] Ruan, X. (2014). *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkely, CA, USA, 1st edition. 10, 24

[75] Sahu, B. and Rincon-Mora, G. A. (2007). An accurate, low-voltage, cmos switching power supply with adaptive on-time pulse-frequency modulation (pfm) control. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(2):312–321. 53

[76] Saito, T., Kondo, S., Miyazaki, H., Bing, W., Watanabe, R., Sugawara, R., and Yokoyama, M. (2017). Mitigating use-after-free attack with application program loader. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 919–924. IEEE. 11

[77] Simmons, J. (1971). Conduction in thin dielectric films. *Journal of Physics D: Applied Physics*, 4(5):613. 15

[78] Strang, G. (1993). *Introduction to linear algebra*, volume 3. Wellesley-Cambridge Press Wellesley, MA. 76

[79] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. (2008). The missing memristor found. *nature*, 453(7191):80. xiii, 15, 16

[80] Tamimi, S., Ebrahimi, Z., Khaleghi, B., and Asadi, H. (2019). An efficient sram-based reconfigurable architecture for embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(3):466–479. 6

[81] Uddin, M., Majumder, M., Beckmann, K., Manem, H., Alamgir, Z., Cady, N. C., and Rose, G. S. (2017). Design considerations for memristive crossbar physical unclonable functions. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 14(1):2. 47, 53

[82] Vermoen, D., Witteman, M., and Gaydadjiev, G. (2007). Reverse engineering java card applets using power analysis. *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, pages 138–149. 13, 74

[83] Wang, L., Zhang, Y., and Feng, J. (2005). On the euclidean distance of images. *IEEE transactions on pattern analysis and machine intelligence*, 27(8):1334–1339. 78

[84] Waterman, A., Lee, Y., Patterson, D. A., and Asanovi, K. (2014). The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES. x, 7, 8, 9, 87

[85] Werner, J., Baltas, G., Dallara, R., Otterness, N., Snow, K. Z., Monrose, F., and Polychronakis, M. (2016). No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 35–46. ACM. 13

[86] Winograd, T., Salmani, H., Mahmoodi, H., Gaj, K., and Homayoun, H. (2016a). Hybrid stt-cmos designs for reverse-engineering prevention. In *Proceedings of the 53rd Annual Design Automation Conference*, page 88. ACM. 82, 85, 114

[87] Winograd, T., Salmani, H., Mahmoodi, H., and Homayoun, H. (2016b). Preventing design reverse engineering with reconfigurable spin transfer torque lut gates. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pages 242–247. IEEE. 82

[88] Witten, D. M. and Tibshirani, R. (2011). Penalized classification using fisher's linear discriminant. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(5):753–772. 76

[89] Wold, S., Esbensen, K., and Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52. 76

[90] Xiao, D., Liao, X., and Deng, S. (2005). One-way hash function construction based on the chaotic map with changeable-parameter. *Chaos, Solitons & Fractals*, 24(1):65–71. 26

[91] Xu, C., Niu, D., Muralimanohar, N., Balasubramonian, R., Zhang, T., Yu, S., and Xie, Y. (2015a). Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 476–488. IEEE. 17

[92] Xu, C., Niu, D., Muralimanohar, N., Balasubramonian, R., Zhang, T., Yu, S., and Xie, Y. (2015b). Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 476–488. IEEE. 43

[93] Yakopcic, C., Taha, T. M., and Hasan, R. (2014). Hybrid crossbar architecture for a memristor based memory. In *Aerospace and Electronics Conference, NAECON 2014-IEEE National*, pages 237–242. IEEE. 2

[94] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y. (2006). Improving cost, performance, and security of memory encryption and authentication. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 179–190. IEEE Computer Society. 11, 54

Appendices

A RRAM Integrity Checking

A.1 RRAM Crossbar Matlab Model

```
% code for solving resistive crossbar array
function y=solver_shorted_row_column_crossbar(R1,R23,R4,Rload
   ,deln,VR)
%R1-Equivalent resistance of type1 cell
%R23-Summation of equivalent resistance of type 2 and type3
   cell
%R4-Array of equivalent resistance of each selected column
%Rload-Load resistance
%deln-Tag bits i.e. No. of selected columns
%VR-Read voltage
mdel=zeros(2+deln,2+deln);
mdel(1,1)=1;
mdel(2,1)=-G23;
mdel(2,2)=G23+sum(G4);
mdel(2,3:end)=-G4;

for i=3:length(mdel)
    for j=1:length(mdel)
        if(j==1)
            mdel(i,j)=-G1(i-2);
        elseif (j==2)
            mdel(i,j)=-G4(i-2);
        elseif (j==i)
            mdel(i,j)=G1(i-2)+G4(i-2)+Gload;
        else
            mdel(i,j)=0;
        end
    end
end

Vright=zeros(2+deln,1);
Vright(1)=VR;
result=mdel\Vright;
y=result(3:end);
```

A.2 Uniformity Measurement

```
%Script for measuring uniformity of the generated tag
function prob_high=uniformity(m,n,deln,LOAD_nom)
% m=No. of row
% n=No. of column
% deln=No. of tag;
%LOAD_nom=ratio of load resistance and ON resistance

index=1:m*n;
R=zeros(m,n);

%% Analog to digital conversion
vmin=0;
vmax=.6;
nb=1; %ADC bits
nlev=2^nb;
res=(vmax-vmin)/nlev;
del=n/deln;% del-th no. of sampling column
w=1; % no. of reserved row combinations to be iterated
sprd=zeros(1,w);% spreadness vector

HRS=100e3;
```

```
LRS=10e3;
Rload=LRS*LOAD_nom*ones(1,deln);
R_lrs=LRS*ones(m,n);
R_hrs=HRS*ones(m,n);
ind_Rload=1:deln;
k=1000;
y=zeros(k,deln);
z=zeros(k,nb*deln);
output=zeros(k,1);% final decimal equivalent of the binary load
   voltage

for i=1:k
    b_num=randi([0 1],m,n);
    b_num(1,1:deln)=zeros(1,deln);
    f0=find(b_num==0);
    f1=find(b_num==1);
    R(f0)=R_hrs(f0);
    R(f1)=R_lrs(f1);
    R1=R(1,1:deln);
    R2=1/sum(1./R(1,deln+1:end));
    R2p=1/sum(sum(1./R(2:end,deln+1:end)));
    R23=R2+R2p;
    R4=1./sum(1./R(2:end,1:deln));
    y(i,:)=solver_shorted_row_column_crossbar(R1,R23,R4,LRS*
       LOAD_nom,deln,vmax);

 %analog to digital conversion of load voltage
 for v=1:deln
   z(i,(nb*(v-1)+1):(nb*v))=ADC(res,y(i,v),nb,vmin);
 end
output(i,1)=bin_to_dec(z(i,:));
end
prob_high=sum(z(:,1))/k
end
```

A.3 Diffusion Measurement

```
%Script for measuring diffusion of the tag distribution
function diff_coeff=diffusion(m,n,deln,LOAD_nom)
% m=No. of row
% n=No. of column
% deln=No. of tag;

index=1:m*n;
R=zeros(m,n);
%% Analog to digital conversion
vmin=0;
vmax=.6;
nb=1; %ADC bits
nlev=2^nb;
res=(vmax-vmin)/nlev;
del=n/deln;% del-th no. of sampling column
w=1; % no. of reserved row combinations to be iterated
sprd=zeros(1,w);% spreadness vector

HRS=300e3;
LRS=10e3;
LOAD=LRS/2;
Rload=LOAD*ones(1,deln);
R_lrs=LRS*ones(m,n);
```

```
R_hrs=HRS*ones(m,n);

ind_Rload=1:deln;
k=1000;
y=zeros(k,deln);
HD=zeros(deln,k);
av_coeff=zeros(1,deln);
z1=zeros(1,nb*deln);
z2=zeros(1,nb*deln);
output=zeros(k,1);% final decimal equivalent of the binary load
   voltage
b_num=zeros(m,n);
for i=1:k
    b_num(1,:)=randi([0 1],1,n);
    hw1=sum(b_num(1,:));
    b_num(1,1:deln)=zeros(1,deln);
    temp=randi([0 1],m-1,n);
    size(temp);
    b_num(2:end,:)=temp;
    f0=find(b_num==0);
    f1=find(b_num==1);
    R(f0)=R_hrs(f0);
    R(f1)=R_lrs(f1);
    R1=R(1,1:deln);
    R2=1/sum(1./R(1,deln+1:end));
    R2p=1/sum(sum(1./R(2:end,deln+1:end)));
    R23=R2+R2p;
    R4=1./sum(1./R(2:end,1:deln));
    y(i,:)=solver_shorted_row_column_crossbar(R1,R23,R4,LOAD,deln
       ,vmax);
```

```
%analog to digital conversion of load voltage
for v=1:deln
  z1(1,(nb*(v-1)+1):(nb*v))=ADC(res,y(i,v),nb,vmin);

end
   b_num(1,:)=randi([0 1],1,n);
   b_num(1,1:deln)=zeros(1,deln);
   rw=randi([2 m],1,1);
   cl=randi([1 n],1,1);
   b_num(rw,cl)=mod((b_num(rw,cl)+1),2);
   b_num;
   b_num(2:end,:)=swap_column(b_num(2:end,:));
   b_num;

   f0=find(b_num==0);
   f1=find(b_num==1);
   R(f0)=R_hrs(f0);
   R(f1)=R_lrs(f1);
   R1=R(1,1:deln);
   R2=1/sum(1./R(1,deln+1:end));
   R2p=1/sum(sum(1./R(2:end,deln+1:end)));
   R23=R2+R2p;
   R4=1./sum(1./R(2:end,1:deln));
   y(i,:)=solver_shorted_row_column_crossbar(R1,R23,R4,LOAD,deln
      ,vmax);
%analog to digital conversion of load voltage
for v=1:deln
  z2(1,(nb*(v-1)+1):(nb*v))=ADC(res,y(i,v),nb,vmin);
end
for hh=1:deln
```

```
 HD(hh,i)=(mod((z2(hh)+z1(hh)),2));
 end
end

for hh=1:deln
av_coeff(1,hh)=mean(HD(hh,:));
end
diff_coeff=geomean(av_coeff)
end
```

A.4 Avalanche measurement

```
%Script for measuring avalanche property of the tag distribution
function av_coeff=avalanche(m,n,deln)
% m=No. of row
% n=No. of column
% deln=No. of tag;
index=1:m*n;
R=zeros(m,n);
%% Analog to digital conversion
vmin=0;
vmax=.6;
nb=1; %ADC bits
nlev=2^nb;
res=(vmax-vmin)/nlev;
del=n/deln;% del-th no. of sampling column
w=1; % no. of reserved row combinations to be iterated
sprd=zeros(1,w);% spreadness vector
HRS=57e6;
LRS=58e3;
LOAD=LRS/2;
```

```
Rload=LOAD*ones(1,deln);
R_lrs=LRS*ones(m,n);
R_hrs=HRS*ones(m,n);
ind_Rload=1:deln;
k=1000;
y=zeros(k,deln);
HD=zeros(1,k);
z1=zeros(1,nb*deln);
z2=zeros(1,nb*deln);
output=zeros(k,1);% final decimal equivalent of the binary load
   voltage
b_num=zeros(m,n);
for i=1:k
    b_num(1,:)=randi([0 1],1,n);
    hw1=sum(b_num(1,:));
    b_num(1,1:deln)=zeros(1,deln);
    temp=randi([0 1],m-1,n);
    size(temp);
    b_num(2:end,:)=temp;
    f0=find(b_num==0);
    f1=find(b_num==1);
    R(f0)=R_hrs(f0);
    R(f1)=R_lrs(f1);
    R1=R(1,1:deln);
    R2=1/sum(1./R(1,deln+1:end));
    R2p=1/sum(sum(1./R(2:end,deln+1:end)));
    R23=R2+R2p;
    R4=1./sum(1./R(2:end,1:deln));
    y(i,:)=solver_shorted_row_column_crossbar(R1,R23,R4,LOAD,deln
       ,vmax);
```

```
 %analog to digital conversion of load voltage
 for v=1:deln
   z1(1,(nb*(v-1)+1):(nb*v))=ADC(res,y(i,v),nb,vmin);
 end
    b_num(1,:)=randi([0 1],1,n);
    b_num(1,1:deln)=zeros(1,deln);
    rw=randi([2 m],1,1);
    cl=randi([1 n],1,1);
    b_num(rw,cl)=mod((b_num(rw,cl)+1),2);
    b_num(2:end,:)=swap_column(b_num(2:end,:));
    f0=find(b_num==0);
    f1=find(b_num==1);
    R(f0)=R_hrs(f0);
    R(f1)=R_lrs(f1);
    R1=R(1,1:deln);
    R2=1/sum(1./R(1,deln+1:end));
    R2p=1/sum(sum(1./R(2:end,deln+1:end)));
    R23=R2+R2p;
    R4=1./sum(1./R(2:end,1:deln));
    y(i,:)=solver_shorted_row_column_crossbar(R1,R23,R4,LOAD,deln
       ,vmax);
 %analog to digital conversion of load voltage
 for v=1:deln
   z2(1,(nb*(v-1)+1):(nb*v))=ADC(res,y(i,v),nb,vmin);
 end
 HD(1,i)=sum(mod((z2+z1),2));
end
av_coeff=mean(HD(1,:))/deln
end
```

B Power Model

For faster simulation of side channel power analysis,power model for different type of instructions has been developed. The main idea of this power model is to simulate the basic circuit block of an instruction using transistor level circuit simulator for the delay and instantaneous power estimation. Power traces are generated for that block for all possible input transitions. The power traces along with the delay information of this block can be used to estimate the overall power trace of an instruction since the same block is repeated in the hardware for implementing the datapath of that instruction. The model for power estimation in different types of instructions are explained as follows.

B.1 Logical Instruction

Single bit logic gates operate in parallel in order to implement the hardware of logical instructions as shown in Fig. B.1. For a 32 bit logical instruction 32 copy of a logic gate works. The overall power trace can be calculated by adding the power trace of the single bit logic gate due to the corresponding input transitions.

$$P_{logic} = \sum_{i=0}^{n-1} P_(\Delta A_i, \Delta B_i) \tag{B.1}$$

where $P_(\Delta A_i, \Delta B_i)$ is the power trace of a single logic gate for an input transition, $(A_i, \Delta B_i)$.

B.2 Arithmetic Instruction

Arithmetic instruction ADD and SUB are built using ripple carry adder where a full adder circuit is repeated over the bit size of the instructions. A simple ripple carry adder is shown in Fig. B.2. In each repetition, the carry out from the previous stage is propagated to the current stage as the carry in. Clearly the basic building block for this type of instructions are the full adder. The power traces for all possible input transition in a full adder circuit is pre calculated using Spectre or any other transistor level circuit simulator. The power trace for each stage due to their respective input transitions are added together in order to estimate the overall power trace. However, unlike the logical instructions, the blocks operate

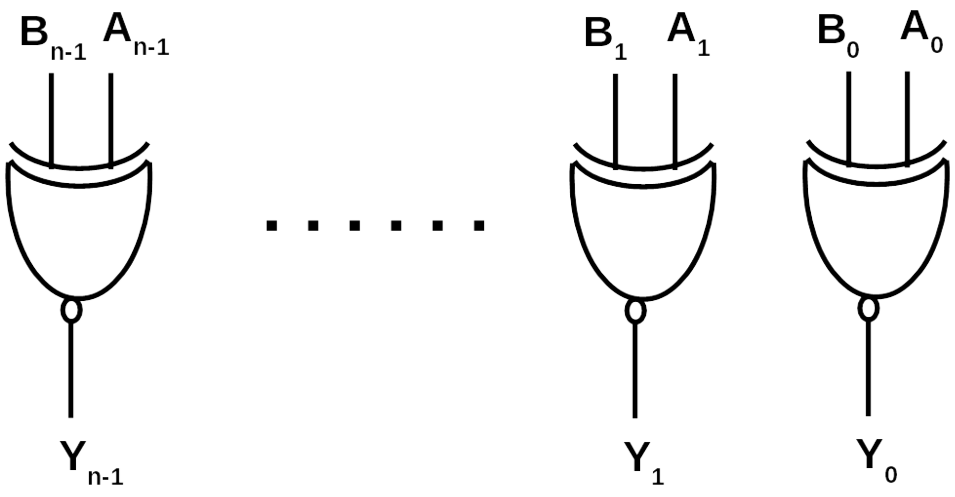


Figure B.1: Hardware implementation of bit wise logical instruction.

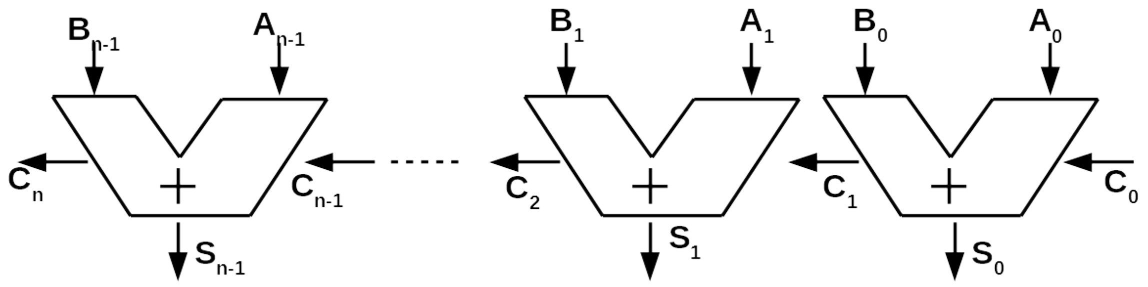


Figure B.2: Hardware implementation of arithmetic instructions.

sequentially except for the initial time slot where all blocks operate on the operands A and B. Later, only one block operates in each time slot once the previous one finishes operation on the propagated carry in signal. The time slots are basically the delay required for calculating the carry out by each full adder block. The delay of each block for all input transition is estimated by the circuit simulator. Overall power consumption is contributed by the parallel operations of all blocks at the first time slot and sequential operations of the blocks at later slots. During the parallel operation, carry in signal, C_i remains constant while the signals, A_i and B_i are constant during the sequential operations. Individual power traces separated by their respective delay are added in order to estimate the overall power trace. This power estimation can be expressed mathematically as follows:

$$P_{arith} = \sum_{i=0}^{n-1} P_{(\Delta A_i, \Delta B_i, C_i)} + \sum_{i=1}^{n-1} P_{(A_i, B_i, \Delta C_i)} u(t - d_{i-1,i}) \tag{B.2}$$

where P is power trace of each full adder block due to a transition in its inputs, A_i, B_i and C_i; $d_(i-1,i)$ is the delay between two sequential operations and u is the unit step function.

Shift Instruction

Shift instructions are implemented using barrel shift circuit for this work. A conceptual diagram of a barrel shifter circuit is shown in Fig. B.3. In this circuit, the basic building block is a multiplexer. This circuit also operates in $log_2 n$ stage where each stage consists of n number of multiplexer. Unlike the adder circuit, each block in a stage operates here in parallel in order to generate results for the next stage. At the first time slot, all multiplexer blocks operate on their new operands similar to the logical and arithmetic instructions. In later time slots, n blocks from a particular stage operates in parallel on the propagated operands from the previous stage. During these operations, the selector input of the multiplexer block remains constant. The overall shift is performed by incremental shift operations in each stage. The power model for this type instructions are combination of the models for logical and arithmetic instructions. Power traces for each stage is estimated by simply adding the corresponding block power for respective input transitions. Now, the overall power trace is estimated by adding the power trace of each stage with a time delay. The mathematical expression is as follows:

$$\begin{aligned} P_{shift} &= P_{init} + \sum_{i=1}^{log_2 n-1} \left(\sum_{j=0}^{n-1} P(\Delta A_{i,j}, f(\Delta A_{i,j}), B_i) \right) u(t - d(i-1,i)) \\ P_{init} &= \sum_{j=0}^{n-1} P(\Delta A_{0,j}, f(\Delta A_{0,j}), \Delta B_0) + \sum_{i=1}^{log_2 n-1} \left(\sum_{j=0}^{n-1} P(A_{i,j}, f(A_{i,j}), \Delta B_i) \right) \end{aligned} \tag{B.3}$$

where P is the power trace of each multiplexer, $d(i-1,i)$ is the delay of a sequential operation and u is the unit step function.

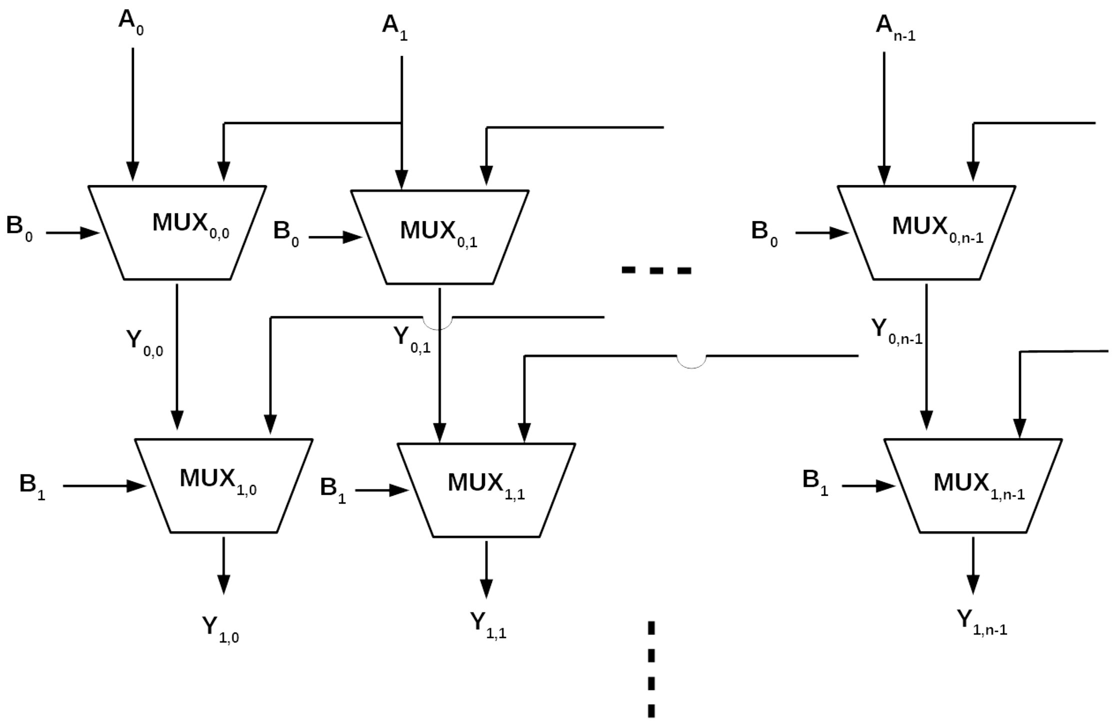


Figure B.3: Conceptual diagram of barrel shifter used to implementation shift instructions.

C MiniRISC-V Assembler

```
import re
import sys

inputfile=sys.argv[1]
outputfile=sys.argv[2]
fw=open(outputfile,'w')
fw.write('memory_initialization_radix=16;\n'+'
   memory_initialization_vector=\n')
Rtype=['add','sub','and','or','xor','sll','srl','sra','slt']
Itype_1=['addi','subi','andi','ori','xori','slli','srli','srai','
   slti']
Itype_2=['lb','lh','lw','lbu','lhu']
Stype=['sb','sh','sw']
Btype=['beq','bne','blt','bge','bltu','bgeu']
Call=['jal']
Ret=['jalr']
LUI=['lui']
KEY=['shk']

funct3={'add':'000',
        'sub':'000',
        'and':'111',
        'or' :'110',
        'xor':'100',
        'sll':'001',
        'srl':'101',
        'sra':'101',
        'slt':'010',
        'addi':'000',
        'andi':'111',
        'ori' :'110',
        'xori':'100',
        'slli':'001',
        'srli':'101',
        'srai':'101',
        'slti':'010',
        'beq' :'000',
        'bne' :'001',
        'blt' :'100',
        'bge' :'101',
        'bltu':'110',
        'bgeu':'111',
        'lb'  :'000',
        'lh'  :'001',
        'lw'  :'010',
        'lbu' :'100',
        'lhu' :'101',
        'sb'  :'000',
        'sh'  :'001',
        'sw'  :'010'}

ins_bin=''
def line_num(phrase,inputfile):
        with open(inputfile) as f:
                count=0
                for line in f:
                        count=count+1
                        if line==phrase:
```

```
                                        return count
a=line_num('start:\n',inputfile)

with open(inputfile) as f:
        labeldict={}
        count=0
        for line in f:
                count=count+1
                line=line.strip('\n')
                line=line.strip(']')
                line=line.strip(':')
                ins=re.split(',|\[| ',line)
                if ins[0] not in Rtype+Itype_1+Itype_2+Stype+
                   Btype+Call+Ret+LUI+KEY:
                        labeldict[ins[0]]=count
                        count=count-1
print(labeldict)
with open(inputfile) as f:
        lcount=0
        truecount=0
        for line in f:
                lcount=lcount+1
                truecount=truecount+1
                line=line.strip('\n')
                line=line.strip(']')
                ins=re.split(',|\[| ',line)
                flag=1
                print(ins[0])
                #opcode field
                if  ins[0] in Rtype:
```

```
        opcode='0110011'
        rd=format(int(ins[1][1:]),'05b')
        rs1=format(int(ins[2][1:]),'05b')
        rs2=format(int(ins[3][1:]),'05b')
        if (ins[0]=='sub' or ins[0]=='sra'):
                op30='1'
        else:
                op30='0'

        ins_bin='0'+op30+'00000'+rs2+rs1+funct3[
           ins[0]]+rd+opcode

elif ins[0] in Itype_1:
        opcode='0010011'
        rd=format(int(ins[1][1:]),'05b')
        rs1=format(int(ins[2][1:]),'05b')
        imm=int(ins[3])
        print(imm)
        if imm<0:
                imm=2**12+imm
        imm=format(imm,'012b')
        print(imm)
        ins_bin=imm+rs1+funct3[ins[0]]+rd+opcode
elif ins[0] in Itype_2:
        opcode='0000011'
        rd=format(int(ins[1][1:]),'05b')
        rs1=format(int(ins[2][1:]),'05b')
        imm=int(ins[3])
        print(imm)
        if imm<0:
                imm=2**12+imm
        imm=format(imm,'012b')
        ins_bin=imm+rs1+funct3[ins[0]]+rd+opcode
elif ins[0] in Stype:
        opcode='0100011'
        rs2=format(int(ins[1][1:]),'05b')
        rs1=format(int(ins[2][1:]),'05b')
        imm=int(ins[3])
        if imm<0:
                imm=2**12+imm
        imm=format(imm,'012b')
        ins_bin=imm[0:7]+rs2+rs1+funct3[ins[0]]+
           imm[7:]+opcode
elif ins[0] in Btype:
        opcode='1100011'
        rs1=format(int(ins[1][1:]),'05b')
        rs2=format(int(ins[2][1:]),'05b')
        jump_label=ins[3]
        print(jump_label)
        line_jump=labeldict[jump_label]
        imm=line_jump-truecount-1
        imm=imm*2
        if imm<0:
                imm=2**12+imm
        imm=format(imm,'012b')
        ins_bin=imm[0]+imm[2:8]+rs2+rs1+funct3[
           ins[0]]+imm[8:]+imm[1]+opcode
        print(ins_bin)
elif ins[0] in Call:
        opcode='1101111'
        rd=format(int(ins[1][1:]),'05b')
        jump_label=ins[2]
        line_jump=labeldict[jump_label]
        print(line_jump)
        print (truecount)
        imm=line_jump-truecount-1
        imm=imm*2
        print(imm)
        if imm<0:
                imm=2**20+imm
        imm=format(imm,'020b')
        ins_bin=imm[0]+imm[10:20]+imm[9]+imm
           [1:9]+rd+opcode
elif ins[0] in Ret:
        opcode='1100111'
        rd=format(int(ins[1][1:]),'05b')
        rs1=format(int(ins[2][1:]),'05b')
        imm=format(int(ins[3]),'012b')
        ins_bin=imm+rs1+'000'+rd+opcode
elif ins[0] in LUI:
        opcode='0110111'
        rd=format(int(ins[1][1:]),'05b')
        imm=(int(ins[2]))
        print(imm)
        if imm<0:
                imm=2**20+imm
        imm=format(imm,'020b')
        ins_bin=imm+rd+opcode
elif ins[0] in KEY:
        opcode='1111111'
```

```
                        keysel=format(int(ins[1]),'03b')
                        keyword=format(int(ins[2]),'016b')
                        ins_bin=keyword+'000000'+keysel+opcode

                else:
                        flag=0
                        truecount=truecount-1
                if flag==1:
                        ins_bin_hex=format(int(ins_bin,2),'08x')
                        fw.write(ins_bin_hex+'\n')
                        print(ins_bin)
fw.write(';')
fw.close()
```

Vita

Md Badruddoja Majumder received his B.Sc. degree in Electrical and Electronic Engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 2013. He has been a research assistant and a PhD candidate in the department of EECS at the University of Tennessee, Knoxville, TN, USA. His research interests include developing secure computing system using emerging technologies.