Bar-Ilan University

Computer Engineering

Code and Control Flow Integrity

For Embedded Systems

Avihay Grigiac **Itai Riven**

Academic Advisor: Prof. Osnat Keren

Instructor: Gilad Dar

October 2020

Table of Contents

Abstract

Code and Control Flow Integrity is a technique for detecting an error in the control flow of the program and the instructions. It does not aim to prevent the attacks from happening, but instead it rely on monitoring a program at runtime to detect abnormal behavior of the program on real time.

In this project we introduce a hardware-based implementation of Code and Control Flow Integrity for embedded systems. The implementations is based on linear error detecting code and capable to handle variable basic blocks' length while detecting errors in real-time, and by simulating different type of attacks we test its strength and limitations.

1. Introduction

1.1 Motivation

The use of technology has reached almost every aspect of our life, such as medicine, transportation, and home appliances, with products like heart-pacemaker, autonomous vehicle and IoT's. Hence, the quality of life and even life itself depends on the reliability of those products. Since we highly relay on those products it must be firmly resistant to errors that may cause it malfunction.

In our work we aim to detect errors that caused by malicious attacker who has a physical access to the system and is able to sabotage its operation. We focus on embedded system; a microcontroller-based system with a dedicated function and pre-loaded code. Under the assumption that the code is written, complied, and loaded properly and safely.

1.2 Threat Model

The attacker is an insider who is familiar with the code and the system. We assume the system is reliable and the attacker is able to inject faults that cause bit flips. The error e is defined as an additive error; it is the difference between the original vector v and the tampered vector v'; that is, $v' = v \oplus e$. The attacker has physical access to the instruction's path solely, from the memory to the processor as shown in figure 1.1. To avoid architecture changes in the processor we assume that the adversary can change the instruction only before its execution.

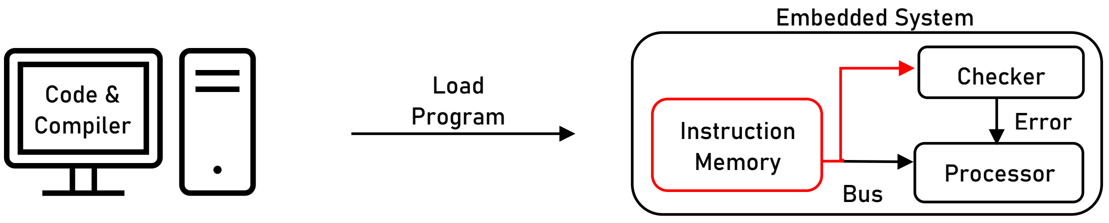


Figure 1.1: Attacker access(marked in red) and the checker module

1.3 Objective

In this project we design a hardware module for embedded systems that detects malicious errors in a program’s instructions. This module needs to fulfil the following requirements:

1. Minimal effect on system's performance and architectural changes
2. Minimal hardware cost
3. Maximal error detection

The module is add-on, i.e. it connects to the system's microprocessor without interfering the pipeline stages and we assume it is not accessible to the attacker. When the module detects an error, it notify the system's controller that the original code has been tempered, and the controller decides what to do with that information for example, terminate it or restart the system.

2. Background

2.1 Code and Control Flow Integrity

The Control Flow[1] of a program refers to the order in which its instructions, branches, loops, and function-calls have to be executed. The program is typically structured into code fragments, such fragments are called basic blocks, and for that matter we have to consider two types of instructions:

1. Sequential instructions, like arithmetic and memory operations, which are executed in direct sequential order.
2. Control-flow instructions, like branches and function calls, which alter the execution sequence and might have more than one subsequent instruction to select as next.

Basic block is a set of arbitrary sequential instructions which can only be entered at the first instruction and exited after the last instruction.

The label is an indication for the beginning of a basic block because it translated to address that the code can jump to. The branch instruction is an indication for the basic block's end since it can redirect the path of the program.

We considered that this is not necessarily always the case, meaning a basic block can begin without label in case of conditional branch that does not meet conditions, and a basic block might end with sequential instruction in case of subsequent instruction that has label attached to it.

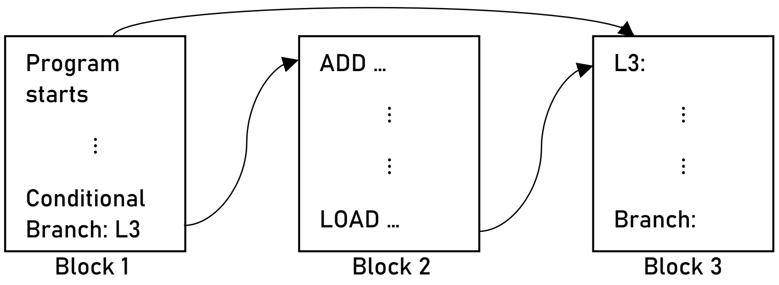


Figure 2.1: Types of Basic Blocks

For example, as shown in figure 2.1, the first block begins with the first instruction of the program and ends with a conditional branch, while between first and last instructions there are only sequential instructions and no labels.

The second block begins with a sequential instruction since the branch at the end of block 1 may not be taken and block 2 is the subsequent block. It also ends with sequential instruction since after this instruction there is a label that the program might jump to.

The third block is the trivial basic block, starts with a label and ends with a branch instruction.

The control flow of a program can be modeled by **Control Flow Graph**(CFG). The CFG is a directed graph where each node is a basic block and each edge denote a valid transition between basic blocks. The edges caused by jumps and calls referred as forward edges, while the edges that are caused by returns referred as backward edges. When creating the CFG there are three cases to consider:

1. If the last instruction is direct branch, then there is specific block in the flow to move to which defined by the label that appears in the branch instruction.
2. If the last instruction is conditional branch, then the flow of the program splits, either the branch taken or not taken, to cover all possible paths in the program.
3. If the last instruction is not branch instruction, then the block ends because label appears, which means there is specific block in the flow to move to, that defined by the label at the beginning of the next block

Protecting the control flow of a program alone is not enough for keeping the system secured from tampered instructions. It is also necessary to verify that no tampered instruction within the basic block was executed. Ensuring code and control flow integrity means that all instruction in a basic block are untampered and executed correctly, and the program flow allowed only through valid edge of the control flow graph.

2.2 Related Work

Defensive attempt against attackers who uses memory leak exploitation led to arm-race between attackers and defenders, where each defensive attempt led to a new attack that bypass it, for example:

Stack buffer overflow attack occures when data written to a buffer exceed its size limit. It can be used to inject code into the stack and manipulate the existing code by overwriting the return address. To defend against it stack canaries were used, by placing a canary value between the return address and the local function variables. However, **format string vulnerabilities** attack allows overwriting arbitrary addresses and can be used to overwrite addresses without changing the canary value and inject code to the program. Non-executable memory introduced as a defense against code injection attacks such as this one. But, it got circumvented by **code reuse attacks(CRA)** which do not require code to be injected but uses existing one instead for malicious purpose. **Code and control flow integrity** mechanism developed to overcome CRA and ensure that tampered software will be detected.

We reviewed various of approaches to enforce CFI:

Labels[2] is an architecture that is used to enforce forward-edge policy. A unique label is inserted at the beginning of each basic block and before each indirect branch, the destination basic block's label is compared to a label identifier which is stored inside the program. The control flow checks are performed using code checks which are inserted at the end of each block.

Shadow Call Stack(SCS)[3] is an architecture that is used to enforce a backward-edge policy. An additional stack is used to store an extra copy of the return address for each function call and before returning from a function call, the integrity of the return address is verified by comparing the return address stored on the two stacks. If there is a mismatch, an exception is raised. Similar to the canary values mentioned above, the SCS secure the return addresses but it has high overhead due the use of 2 execution cycles for each function call and return.

Table[4] based approach uses a table of allowed branches with a single entry for each direct branch, and possibly multiple entries for each indirect branch. At runtime, each branch is verified by checking for the existence of an entry inside the table.

Finite State Machine[5] is used to enforce a valid sequence of events. While a program is executing, a state machine tracks the events, with each event representing a node in the FSM. As long as the events inside an executing program follows the correct sequence, the state transitions will be valid. However, if an invalid state transition occurs, an attack is assumed, and appropriate action is taken.

Branch Limitation[6] limit branch targets to basic block entries. This restriction ensures that control can only flow from the last instruction of one basic block to the first instruction of another basic block. When a branch targets the middle of a basic block it violates the basic block definition, implying that the system is under attack. It used to enforce forward-edge policy for indirect branches but can be combined with an SCS to check backward edges.

Instruction Set Randomization[7] encrypt each instruction's bytes using control flow information from a CFG. Each instruction's bytes are decrypted using a combination of the current and previous program counters. Therefore, any invalid control flow between two instructions will lead to a decryption error. Instruction bytes in each basic block decrypted with a secret key that is unique for each device. This approach check the validity of control flow between basic blocks and verifies the control flow between every two instructions. However since the encryption is not systematic, the decryption might creates large overhead on the system

Signature Modeling periodically calculates checksum on the executed instructions and compare it with stored reference values, that is computed offline.

Continuous Signature Monitoring[8] is based on signature modeling with the addition of update function. The signature depends on the current instruction and the previously executed instructions, and the update function ensures that signature value at a given location is always the same, regardless of the program's flow. At run time, it verifies and update the signature of each executing instruction, which allows us to enforce not only code integrity but also control flow integrity due to the update function addtion.

2.3 Hardware versus Software

Implementing code and control flow integrity on systems can be done by either software or hardware. Each method has its own advantages and disadvantages.

2.3.1 Software Implementation

Software implementation rely on CCFI code insertions into the original program. CFG is created during compilation time and at run time, with a dedicated function, it verifies the instructions before the execution at pre-arranged spots along the code.

Compared to a hardware solution, this method is easy and cheap to implement, but it has its flaws. If the compiler is unaware of the security aspect of the CCFI checks, it might cause sensitive data leak to the stack making the protection vulnerable. Software-based CCFI also creates high overhead to the program execution time, since the program has to pause its main purpose, check validity, and then return to it. But the major disadvantage of software implementation is the ineffectiveness against physical attacker, assuming the attacker has physical access to the system, he can choose to manipulate the securing code itself.

2.3.2 Hardware Implementation

Hardware can provide strong isolation to the CCFI implementation, but using this method requires addition of new hardware modules to existing system. It may lead to architecture changes if there is not enough free space, and the limited space is also a major constrain on the amount of data the hardware module can store.

However, this method allow us to detect errors even from strong attacker, who is familiar with the system and the code and has physical access. In addition to that, hardware implementation allows parallelism, i.e. it can verify the validity of the program with a minimal effect on the execution time and provides low overhead. Therefore, in our work we focus on hardware implementation and aim to minimize its drawbacks.

3. Preliminary

In our project we will implement the solution presented in [9]. The authors present an efficient hardware-supported technique to enforce strong and secure code and control flow integrity for embedded system against fault attacks.

3.1 Derived Signatures

The derived signature[9] is a computing technique that is used to detect violation in the code integrity, by adding a small piece of hardware to the system's processor. Upon the execution of each instruction the hardware calculates a signature value that depends on the previous signature and the current instruction.

In order to check the validity of the signature it compares the result to a saved reference value. The reference value is calculated during compilation when the program is created and stored in a dedicated memory. Checking a reference value for each instruction is inefficient since it requires large size of memory, which leads to high hardware costs. However, we can utilize the fact that all the instructions in a basic block are executed consecutively, regardless of the program flow, in order to work in a basic block resolution and check the signature at the end of each basic block.

3.2 Generalized Path Signature Analysis (GPSA)

The derived signature alone does not provide error detection for the transitions between basic blocks, and only ensure that each basic block is executed properly.

The GPSA[9] is used to check the integrity along paths in the CFG. The idea is to insert a signature update to a block to make its signature the same as another block's signature that leads to the same block. The signature updates allow us to monitor the flow of blocks with multiple entrances.

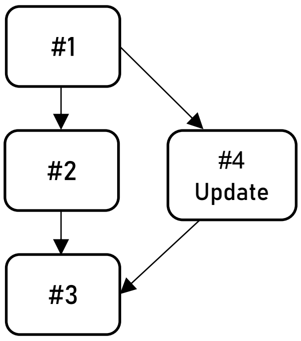


Figure 3.1: Example of Update Function Usage

For example, as shown in figure 3.1, assuming the signature of block #3 is based on the signature of block #2 and the program flow is #1 → #4 → #3. The update function makes the signature of block #4 the same as block #2. That way the signature of block #3 will be same in both cases, either it reached from block #2 or block #4

3.3 Signature Function Selection

The signature function is calculated by the derived signature module, denoted by $S_j = f(S_{j-1}, I_j)$. The next signature is based on preceding signature S_{j-1}, and current instruction I_j. We denote the additive error in the instruction I_j by ΔI_j and the faulty instruction is $I_j \oplus \Delta I_j$.

In order to make a faulty instruction detectable with certainty, independent of the actual error, the function has to fulfill the following requirements [9]:

Reliability: every error, $\Delta I_j \neq 0$, must result in a signature error, $\Delta S_J \neq 0$, given the original signature was correct. only if: $|S| \geq |I|$

$$S_J \oplus \Delta S_J = f(S_{j-1}, I_j \oplus \Delta I_j), \forall\, \Delta \mathrm{I_j} \neq 0 \rightarrow \Delta S_J \neq 0$$

Error in the instruction will derive an error to the signature.

Error Preservation: a signature error will force an error in all the following sequential signatures.

$$S_J \oplus \Delta S_J = f(S_{j-1} \oplus \Delta S_{j-1}, I_j), \forall\, \Delta \mathrm{S_{j-1}} \neq 0 \rightarrow \Delta S_J \neq 0$$

This requirement allows to delay the checking of a signature and can reduce the number of necessary signature checks.

Non associativity: the order of the instructions that is calculated by f must have an influence on the signature value.

$$\forall\, I_j \neq I_k \rightarrow f(f(S_{j-1}, I_j), I_k) \neq f(f(S_{j-1}, I_k), I_j)$$

This will allow us to detect swap of different instructions in the same basic block.

3.3.1 Cyclic Redundancy checks(CRC)

Cyclic Redundancy Checks function fulfills those requirements; however, it cannot guarantee errors detection certainty against highly controlled injected faults. Therefore, we need to have constrained on the number of bit-flips to be able to detect some errors with certainty.

All arithmetic are done in $\mathbb{F}_2$, the normal rules of polynomial addition, division, multiplication, and division apply, except that all coefficients are either 0 or 1 and the coefficients add and multiply using the $\mathbb{F}_2$ rules. Algebraic representation of a codeword: $c(x) = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$

$$[\,0\ 1\ 0\ 1\,] \rightarrow x^2 + 1$$

Figure 3.2: Code word represented as the polynomial

Given a message $m(x)$ of size k bits, CRC function produces systematic encoding with a signature of size r bits, where r is constant that defined by the degree of the generator polynomial $g(x)$, and together creates a code word of length $k + r = n$

Encoding a message is done by the formula: $c = x^r m(x) + x^r m(x)\ mod\ g(x)$, but since it does not represent the additive property of the CRC and the dependance on the preceding signature, we had to reform it. Given a sequence of messages $m_1 \dots m_n$, each of size k, the signature of each message is:

$$s_1 = s_0 + x^r m_1(x) mod\ g(x)$$

$$s_2 = s_1 + x^k \cdot x^r m_2(x) mod\ g(x)$$

$$\vdots$$

$$s_i = s_{i-1} + x^{r+(i-1)k} \cdot m_i(x)\ mod\ g(x)$$

While $s_0 = 0$ represent the initial value of the signature.

However, since the signature is depended on x^{ik}, it causes the calculation time to increase with each iteration and it cannot be simulated as hardware that has limited amount of registers.

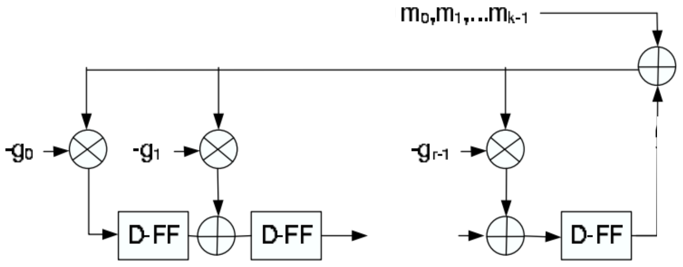


Figure 3.3: Hardware CRC computation using division circuit

Modulus operated by division circuit, figure 3.3, allows us to visualize the signature's dependency of the current instruction on the signature of the preceding instruction. When first message enters the circuit, the DFFs are set to '0' but when the second message enters the circuit, the DFFs are set with the value of the first message's signature and shifted each clock cycle.

<u>Theorem</u>: cyclically shifted code word equals to multiple the polynomial by $x(\text{mod } x^n - 1)$.

Using this theorem, we can multiple the signature by constant x^k each iteration and keep the calculation time the same:

$$s_1 = x^r m_1(x) mod\ g(x)$$

$$s_2 = s_1 x^k + x^r m_2(x) mod\ g(x)$$

$$\vdots$$

$$s_i = s_{i-1} x^k + x^r m_i(x) mod\ g(x), i \geq 1$$

This form represent the dependency of each signature on the previous signature and the current instruction.

3.3.2 Generator Polynomial Selection

The selection of the generator polynomial is important part of the CRC implementation, it needs to involve the size of the data word we want to protect and the polynomial must be chosen to maximize the error-detecting capabilities.

Every code word has a certain minimum hamming distance, d, from every other code word. The value of d is the minimum number of bit flips that required to change one code word to another code word.

The hamming distance determined by the selection of the generator polynomial and effected directly by the polynomial degree, meaning a higher degree leads to larger hamming distance.

The selection of the generator polynomial also has direct impact on hardware costs. Higher degree leads to more reduncdancy bits which requires larger memory space, and larger amount of elements leads to more XOR gates in the CRC implementation.

CRC is able to detect any error vector of weight $d - 1$ or less, and capable to detect any burst errors of length $n - k = r$ or less. Burst error is a sequence of bits, where the first and last bits are '1' and others bits between them could be either '0' or '1'.

3.4 Update Function Selection

The update function balances the various paths through the CFG and denote by $S_i = u(S_j, J_j)$. It updates the siganture of block S_j with a value J_j to be the signature of block S_i if both blocks lead to the same next block.

The update function has to fulfill the following requirements[9]:

Full Control: for all possible values of S_i and S_j there is a value J_j that sustains:

$$S_{\mathrm{i}} = u\big(S_j, J_j\big), \quad \forall S_i, S_j, \exists\, J_j$$

Error Preservation: a signature error will force an error in all the following sequential signatures, and must not be eliminated by $\Delta J_j = 0$

$$S_i \oplus \Delta S_i = u\big(S_j \oplus \Delta S_j, J_j\big), \forall\, \Delta S_j \neq 0 \rightarrow\ \Delta S_i \neq 0$$

Invertibility: Given S_i and S_j the reverse function u^{-1} should return the value J_j

$$J_j = u^{-1}\big(S_j, S_{\mathrm{i}}\big), \forall\, S_j, S_i$$

It allows us to find the J_j that match the S_j which is used to update the signature.

3.4.1 XOR Function

The binary XOR function fulfills those requirements. Updaing a signature is done by the formula: $S_i = S_j \oplus J_j$ and the J value is calculated by: $J_j = S_j \oplus S_i$

For example, as shown in figure 3.1, assuming the signature of block #3 is based on block #2. The block #4 need to sustain: $S_4 \oplus J_4 = S_2$ in order to get the signature of block #3 properly, the J_4 value is: $J_4 = S_4 \oplus S_2$

3.5 Attacker Profile

Under the constrains of the signature and update functions we have to define the capabilites of the attacker. Since the CRC is a linear code, the type of errors that can be detected is limited and the attacker have to be restricted to them.

Code word defined as: $c_i = (b_i, s_i)$, b is a binary vector represent a basic block, and $s_i = s_{i-1}x^k + x^r b_i(x) mod\ g(x)$ is the CRC.

The signatures are pre-calculated and stored in a protected memory the attacker cannot access, hence:

Errors defined as: $e = (\Delta b, 0)$, Δb is a binary vector represent an error that is limited to maximum weight of $d-1$ or burst error with maximum length of r

Faulty basic block defined as: $c + e = (b + \Delta b, s)$

Based on the attacker we defined, the CRC form we developed fulfills the derived signature function requirements:

Reliability: $s_{i-1}x^k + x^r(m_i \oplus \Delta m_i)\ mod\ g(x) = s_i \oplus \Delta s_i$

This property is valid only if $|m| \leq |s|$, but since $|s| = r$ and and the $|m| = k$, where $k > r$, there are errors $\Delta m_i \neq 0$ that will result with no signature errors. Therefore, this is the reason we have to limit the attacker to certain type of errors.

Error Preservation: $(s_{i-1} \oplus \Delta s_{i-1})x^k + x^r m_i\ mod\ g(x) = s_i \oplus \Delta s_i$

In this case $|s_i| = |s_{i-1}|$ and since the attacker is limited to insert erros only to the message. In a sequence of messages: $m_1 \dots m_n$ where some m_i is tampered, such that $\Delta m_i \neq 0 \rightarrow \Delta s_i \neq 0$, the final signature will be $s_n \oplus \Delta s_n$, where $\Delta s_n \neq 0$.

Non associativity:

$$\left(s_{j-1}x^k + x^r m_j\ mod\ g(x)\right)x^k + x^r m_k\ mod\ g(x)$$
$$\neq \left(s_{j-1}x^k + x^r m_k\ mod\ g(x)\right)x^k + x^r m_j\ mod\ g(x)$$

$$\left(s_{j-1}x^{2k} + x^{r+k}m_j + x^r m_k\right) mod\ g(x) \neq \left(s_{j-1}x^{2k} + x^{r+k}m_k + x^r m_j\right) mod\ g(x)$$

$$\left(x^{r+k}m_j + x^r m_k\right) mod\ g(x) \neq \left(x^{r+k}m_k + x^r m_j\right) mod\ g(x)$$

4. Solution Implementation

The implementation includes offline preparations in order to divide the program into basic blocks, determine program's valid flow and pre-calculation of the signature and update values. As well as real-time program run on hardware simulation while encoding and checking for errors.

4.1 Benchmark Analysis

As part of the offline preparations we have to decide the size of the basic block, which determine the dimension of the code and effect the selection of the generator polynomial. If the basic block's size will be too small, the program will have high number of blocks that results in higher hardware cost, since we have to store large number of signatures in the protected memory.

On the other hand, if the basic block size will be too big it might take too long to detect an error and a critical damage may occur. For that matter we analyzed various programs to find the ideal basic block size.

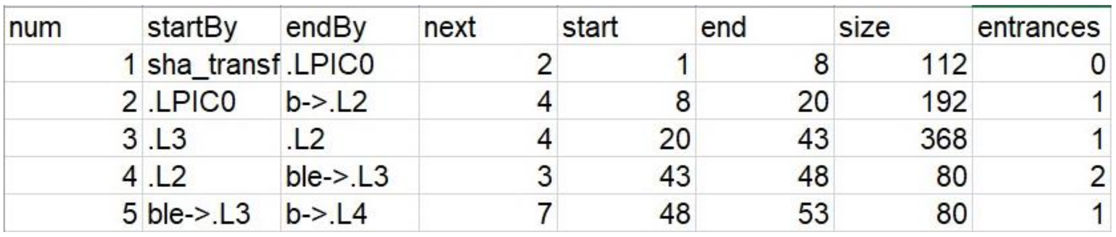
We used ARMv7 microprocessor architecture that is simulated over a virtual machine using QEMU emulator. Using the emulator, we compiled different C files programs and created an assembly file and opcodes file for each program with the following commands:

Create assembly file: arm- linux -gnueabihf-gcc -S program.c
Create binary file: arm- linux -gnueabihf-as program.s -o program.o
Create opcode file: arm- linux -gnueabihf-objdump -d program.o > opcodes.txt

We analyzed the assembly file with Python code(Appendix A) to divide the program into basic blocks and to create the control flow of the program.

For the basic block division, we use the labels and branch-instructions as indicators. The code reads the assembly file line-by-line and it identify labels and branch instructions to ensure that basic block does not contains a label or branch instruction in the middle of it.

In order to create the control flow graph of the program, the script keep track of the first and last instruction of each basic block, the labels of the branch instructions jump to and the number of each block.



num	startBy	endBy	next	start	end	size	entrances
1	sha_transf	.LPIC0	2	1	8	112	0
2	.LPIC0	b->.L2	4	8	20	192	1
3	.L3	.L2	4	20	43	368	1
4	.L2	ble->.L3	3	43	48	80	2
5	ble->.L3	b->.L4	7	48	53	80	1

Figure 4.1: partly output of the Python code

The Python code output for each program a csv file, as shown in figure 4.1, with the following information:

- Column 1: block number
- Column 2+3: instruction or label that start and end the block
- Column 4: subsequent basic block, if branch is taken the next block is in column4, if branch is not taken the next block is the next row
- Column 5+6: number of row the block start and end in the opcode file
- Column 7: basic block size
- Column 8: number of enterance for each block

We summarized the information of various programs to get the statistic of basic block size, which defined by the number of instructions it contains multiple by 16 since ARMv7 microprocessor architecture is 16 bits. To simplify the statistic, the size of blocks is rounded up to power of 2.

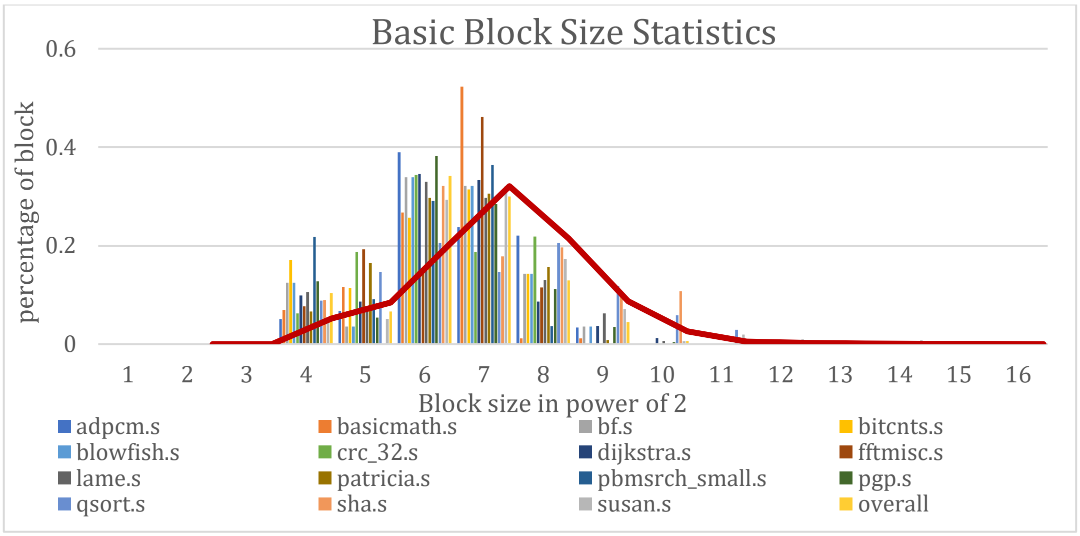


Figure 4.2: basic block size distribution of several programs and the average size of a basic block (marked in red)

Based on our analysis and the limition of the attacker, the generator polynomial selection must include the following:

1. As resulted from our analysis that is shown in figure 4.2, the average data word size is 2^8 bits.
2. We want to detect a single instruction substitution. In this architecture, the instructions are of size 16 bits, meaning we have to detect burst error of length 16. Therefore, the generator polynomial must have minimal degree of 16.
3. Under the constrains of the previous parameters we also want to achieve maximum hamming distance to detect errors of weight $d-1$ or less.

In [11] the authors suggests generator polynomials that provides largest hamming distance for specific data word size sorted by the polynomial degree. Ideally we would select a polynomial of degree 16 that is suitable for data word of size 2^8 but since the suggested polynomial of that degree is not perferctly matched, we have to consider trade-off between block size and hamming distance size.

We examined couple of possible generator polynomials:

- $x^{16} + x^{15} + x^{13} + x^9 + x^7 + x^6 + x^5 + x^3 + x + 1$
  Provides $d = 4$, suitable for data word of size $k = 32{,}751$ bits, and in particular for data word of size $k = 256$ bits.

- $x^{16} + x^{14} + x^{12} + x^{11} + x^8 + x^5 + x^4 + x^2 + 1$
  Prodives $d = 5$, but only suitable for data word of size 241 bits that is smaller than the average block size. Which means we will have to divide the program to larger amout of blocks resulting in more signatures to store and higher hardware cost.

- $x^{19} + x^{17} + x^{15} + x^{14} + x^{12} + x^7 + x^5 + x^4 + x^2 + 1$
  Prodvides $d = 6$, suitable for data word of size $k = 494$ bits, but requires to store 3 extra bits for each signature, which also results in higher hardware cost.

Higher hamming distance value requires selection of polynomial with a degree that is nearly double the scale we need, which is highly inefficient. We decided to select the first polynomial, since the primary threat is substitute of single instruction and this polynomial allow us to keep minimal hardware cost.

Each basic block will have maximum size of 256 bits, this allows us to detect error at most every 16 instructions. Therefore, we have to divide larger basic blocks into that size and create a new program info file with re-sized basic blocks.

Althought the maximum size of a basic block is allowed to be 256 bit, we decided to split the bigger blocks to their mean size in order to make the validty check rate more efficient. For example, assuming we have a basic block of size 528 bits, instead of spliting it to smaller basic blocks of size 256,256,16 bits and cause the validty check to occure after 16, 16, 1 instructions, we first detected the amount of blocks it will be splited to and set each block to have average amount of instructions, meaning the block is splited to blocks of sizes 176,176,176 bits and the validty check will occure after 11,11,11 instructions.

4.2 Pre-Calculation of Signatures and Update Values

Based on the new program information and opcodes file, we created a MATLAB function(Appendix B) that calculate the signature value and the update function value for each basic block. This information will be part of the protected memory in our module.

We created a recursive function that finds all possible paths of the program and creates vector of all signatures and vector of all update function values. Basic blocks that does not require update value has the value '0' in their vector position.

The path with all conditional branches that are not taken is consider 'main path' and each block's signature on this path is based on the previous block's signature of the same path. In figure 4.3 the main path is $2 \rightarrow 4 \rightarrow 5$ and the signature of block 4 is based on the signature of block 2.

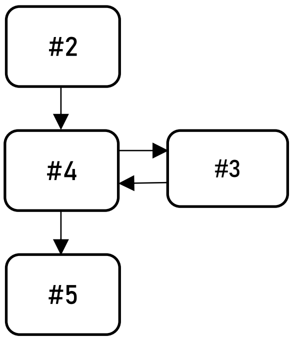
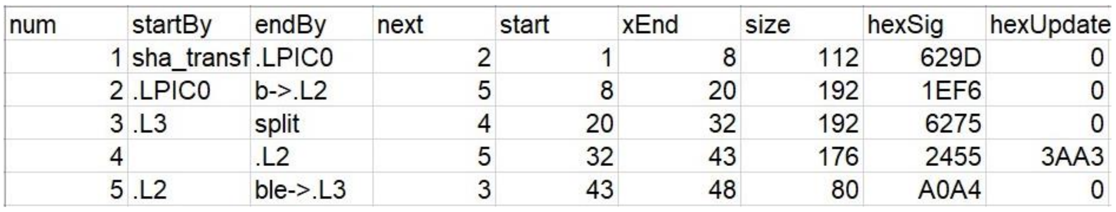


Figure 4.3: Part of the program flow graph that appears in figure 4.1

When the recursion splits into different paths, if the next block already has a signature then an update function value is required and calculated. For example, when the script reaches block 3 it checks if block 4 has signature, and since block 4 is on main path it already have a signature and an update value is calculated.



num	startBy	endBy	next	start	xEnd	size	hexSig	hexUpdate
1	sha_transf	.LPIC0	2	1	8	112	629D	0
2	.LPIC0	b->.L2	5	8	20	192	1EF6	0
3	.L3	split	4	20	32	192	6275	0
4		.L2	5	32	43	176	2455	3AA3
5	.L2	ble->.L3	3	43	48	80	A0A4	0

Figure 4.4: updated csv file with the addition of signature and update values

4.3 Hardware Implementation

For the real-time hardware simulation, we created the following system, as shown in figure 4.5, on MATLAB environment. The MATLAB code uses the program info file and the opcode file to simulate the program run, track the program flow, encode, update the signature and check for errors.

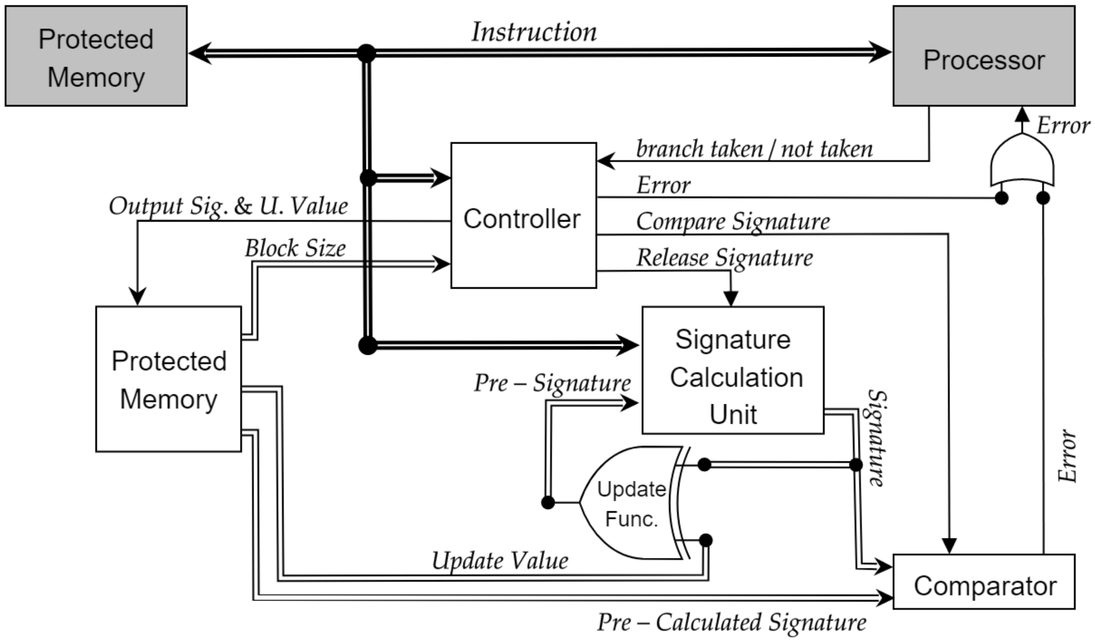


Figure 4.5: System Scheme

4.3.1 System Simulation

While the instruction transfer on the system's bus, from the instruction memory to the processor, each module performs the following:

The comparator checks the validity of the signature by comparing it to a stored value it receives from the protected memory. If it does not match then it signal the processor an error occurred.

The update function updates the signature every time it released from the signature calculation unit. It XOR the signature with a stored value it receives from the protected memory that can be either update value or '0'. If the block ends with a conditional branch and the branch is not taken, it will be XORed with '0' regardless of its stored value.

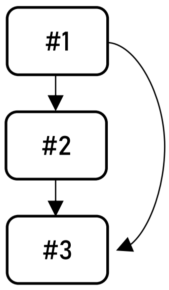


Figure 4.6: example of skipped update value

For example, as shown in figure 4.6, the block #1 ends with conditional branch. In case branch is taken, it XOR the block's signature with its stored update value but if the branch is not taken, it must not used it stored value, and it XORed with '0'

The controller outputs the signature from the signature calculation unit if it either reaches the end of the block by counting the instructions and comapre it to the block's size, or detect a branch instruction. When the signature released it signal the protected memory to output the matching CRC value and signal the comparator to perform a validity check. According to the basic block last instruction and whether the branch was taken or not, the controller also signal the protected memory to output matching update value.

The protected memory stores the CRC values, the update values and the sizes for each basic block. If a block does not require an update, then the respected part of the signature holds the value '0'. Each CRC value and update value is size of 16 bits, and the size of the block is size of 8 bits.

CRC value (16 bits)	Update value (16 bits)	Basic block size (8 bits)

Figure: 4.7: protected memory data for each basic block

The addition of block size to the protected memory allows the controller to be used as first line of error detection. If the attacker deletes a branch instructions he may cause the execution of instructions beyond the basic block limits and result in delaying the validty check. But, by counting the number of instructions the controller can ensure the validty check will occure approximately every 16 instructions, even if the block was tampered.

Another use of the block size is to prevent manipulation on instructions within the basic block size limit. By changing single instruction to branch the attacker can redirect the program to sequence of instructions of his choice that will be executed according to the basic block size. But we can prevent that by checking when a branch instruction appear if its the last instruction in the block, if its not it means that the instruction was tampered and an error occurred.

The MATLAB code uses the program information file to extract the data needed for each module. The opcode file is used as instruction memory, and the code go through opcodes in the file according to the block number.

The following code simulate the controller role for the signature calculation unit and the comparator module. It extract the basic block size and calculate the signature for each instruction in the block, from the beginning of the block until it reaches the end of the block, while checking every branch instruction if it is the last instruction within the block. Since the bus's processor is only 16 bits, in case of 32 bits instruction we split the instruction and execute an extra cycle for the signature calculation. When it reaches the end of the block the outputted signature is being compared to the pre-calculated value stored in the program info file.

```
for j = bbStart:bbEnd %calculate siganture
    try %handle 16 bits instruction
        instruction = hexToBinaryVector(opcodes{j,1},k);
        instCounter = instCounter + 1;
        if ismember(opcodes{j,1}{1}(1:2), branchOpcodes1)
            if instCounter*k > programInfo{bbNum,7}
                error = 1;
                break;
            end
        end
    catch %handle 32 bits instruction
        command = hexToBinaryVector(opcodes{j,1},2*k);
        instCounter = instCounter + 2;
        if ismember(opcodes{j,1}{1}(1:5), branchOpcodes2)
            if instCounter*k > programInfo{bbNum,7}
                error = 1;
                break;
            end
        end
        instruction = command(1:k);
        signature = hardwareCRC(instruction, signature,
remindersVector);
        instruction = command(17:2*k);
    end
    signature = hardwareCRC(instruction, signature,
remindersVector);
end
hexSig{bbNum} = binaryVectorToHex(signature);
checker
if error == 1 || ~strcmp(hexSig{bbNum},programInfo{bbNum,8})
    break;
end
```

To simulate the controller role for the update function module the simulation have to randomly decides the flow of the program and update the signature properly for each scenario. The next basic block is determined by the following cases:

1. If the basic block ends with conditional branch, we set a random number $p = [0\ 1]$ and in $P = 0.5$ probablity the branch is either taken or not taken.
   - If the branch is taken, the subsequent block number is taken from 'next block number' column of the current block number row. Since we might jump to block in the 'main path' it required to update the signature.
   - Else the branch is not taken; the subsequent block number is the consecutive number. No update is needed since we stay on 'main path', meaning we have to skip the use of update value.
2. If the basic block ends with direct branch or none branch instruction, the subsequent block number is taken from 'next block number' column of the current block number row. In this case we also might jump to block in the 'main path' and required to update the signature.

```
if ismember(programInfo{bbNum,3}{1}(1:3), cb) %path splits
   p = rand;
   if p > 0.5 %branch taken
      updateValue = hexToBinaryVector
         (programInfo{bbNum,10}{1});
      updateValue = [zeros(1, r-length(updateValue)),
         updateValue];
      signature = xor(updateValue, signature); %update func
      hexSig{bbNum} = binaryVectorToHex(signature);
      bbNum = str2double(programInfo{bbNum,4});
   else
      bbNum = bbNum+1; %branch not taken
   end
else %fixed path
   updateValue = hexToBinaryVector(programInfo{bbNum,10}{1});
   updateValue = [zeros(1, r-length(updateValue)),
     updateValue];
   signature = xor(updateValue, signature); %update func
   hexSig{bbNum} = binaryVectorToHex(signature);
   bbNum = str2double(programInfo{bbNum,4}); %
end
```

4.3.2 Encoding Simulation

The signature calculation unit is computing the signature for each instruction based on the formula: $s_i = \left(s_{i-1}x^k + x^r I_i(x)\right) mod\ g(x)$.

In order to simulate real-time hardware, the signature calculation unit must finish the computing for each instruction at the same clock cycle as system. Otherwise, the system will have to hold instructions until the signature calculation is done, which create high overhead, or the signature calculation unit has to compute 16 times faster than the system clock cycle, which requires architecture changes to the systems we want to avoid.

We have to multiple the previous signature by x^k and multiple the current instruction by x^r, which is equals to shift left the polynomials by k and r, respectively. Then we sum the outcomes and compute modulus to the result. Division circuit is a common hardware implementaion to operates modulus but it works bit by bit, therefore we have to develop a way to exectute the calculation for all bits at once in order to avoid overhead in the system.

The sum result is equal to the polynomial: $a_n x^n + \cdots + a_m x^m$ and by using the property: $(a_n x^n + \cdots + a_m x^m) mod\ g(x) = a_n x^n modg(x) + \cdots + a_m x^m modg(x)$, we can convert the calculation to the sum of the reminders of each x^i. Since the aritmethic done in $\mathbb{F}_2$, addtion is euqal to XOR, therefore given a polynomial we can calculate the reminder of each x^i and design a module that XOR all bits at once.

In the MATLAB implementation we created a matrix that simulates the hardware XOR gates design of the generator polynomial. For a given polynomial we calculate all the reminders of x^i, where $1 < i < n$, and store it in a $n \times r$ matrix at the i row. Since the code word is size of $n = k + r$ and the data word is being multiple by x^r as part of the CRC calculations, the relevant reminders are only the last k value of x^i, meaning the reminders with weight of 1 are excluded, and we get a matrix with size of $k \times r$, where each row is data word bit and each column is signautre bit.

```
function [ remindersMatrix ] = reminders(polynomial, k, r)
    n = k + r;
    remindersMatrix = zeros(n,r);
    for i = 1:n
        [q,s] = deconv([zeros(1,r-i), 1, zeros(1,i-1)],
polynomial);
        remindersMatrix(i,:) = mod(s(length(s)-r+1:end),2);
    end
        remindersMatrix = flip(remindersMatrix);
        remindersMatrix = remindersMatrix(1:k,:);
end
```

The CRC function zero pads the previous signature and the instruction and XOR them. Then it execute modulus by finding the set position in the resulted vector and XOR the matching rows of the reminder matrix.

```
function [ signature ] = hardwareCRC(instruction, preSig,
remindersVector)
  r = length(preSig);
  k = length(instruction);
% S_i = (s_i-1*x^k + I_i*x^r) mod g
  msg = xor([preSig, zeros(1,k)],[instruction, zeros(1,r)]);
  setPositions = find(msg);
  signature = zeros(1,r);
  for j = 1:length(setPositions) %mod g
 signature=xor(signature,remindersVector(setPositions(j),:));
  end
end
```

For example, given the generator polynomial: $g(x) = x^4 + x + 1$ and resulted vector of the first step in the CRC calculation: $s_{i-1}x^k + x^r I_i(x) = x^{10} + x^7 + x^5$, we want to compute: $x^{10} + x^7 + x^5 \; mod \; x^4 + x + 1 =$
$(x^{10} \; mod \; x^4 + x + 1) + (x^7 \; mod \; x^4 + x + 1) + (x^5 \; mod \; x^4 + x + 1)$

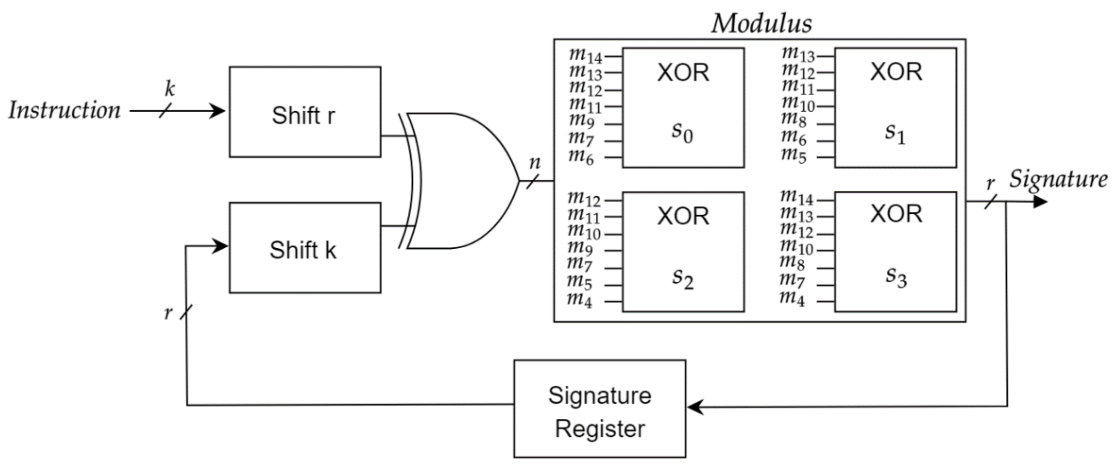
We first calcutate the reminder matrix, as shown in figure 4.7. The matrix is in reversed order for the MATLAB indexing.

	s_0	s_1	s_2	s_3
m_{14}	1	0	0	1
m_{13}	1	1	0	1
m_{12}	1	1	1	1
m_{11}	1	1	1	0
$\rightarrow m_{10}$	0	1	1	1
m_9	1	0	1	0
m_8	0	1	0	1
$\rightarrow m_7$	1	0	1	1
m_6	1	1	0	0
$\rightarrow m_5$	0	1	1	0
m_4	0	0	1	1

Figure 4.7: reminder matrix of the polynomial $x^4 + x + 1$

We find the set positions of the binary vector $x^{10} + x^7 + x^5$: {5, 8, 10}. Those positions matchs the rows of the reminder matrix (marked with arrow). The modulus is operated by XORing those rows of the reminder matrix and results the signature value: $(x^2 + x + 1) \oplus (x^3 + x + 1) \oplus (x^2 + x) = x^3 + x$

The hardware implementation for CRC with this generator polynomial shown in figure 4.8. each column of the matrix represents a signature bit and allow us to create a modulus unit that operate the computations for all bits at once.



Modulus
Instruction
k
Shift r
Shift k
r
n
m_{14} m_{13} m_{12} m_{11} m_9 m_7 m_6
XOR s_0
m_{13} m_{12} m_{11} m_{10} m_8 m_6 m_5
XOR s_1
m_{12} m_{11} m_{10} m_9 m_7 m_5 m_4
XOR s_2
m_{14} m_{13} m_{12} m_{10} m_8 m_7 m_4
XOR s_3
r
Signature
Signature Register

Figure 4.8: Hardware implementation of CRC unit with the polynomial $x^4 + x + 1$

4.4 Hardware Costs

The selected generator polynomial for our system adds $r = 16$ bits and has the following hardware cost impact on every unit(without considering the controller):

CRC Calculation unit

The CRC unit requires r XOR boxes, one for each bit, with the following input size:

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
9	7	8	7	7	7	8	9	9	5	9	9	9	7	8	10

In order to create n-bits XOR gate we have to concatenation $(n-1)$ 2-bits XOR gates. That creates a upper bound for the amount of reuquired XOR gates for the system: $\sum_{i=0}^{r}[(input\ size)_i - 1]$, But most likely we will use less XOR gate then the boundry, since some redundency bits uses the same XOR gates. In our case the modulus operation bounded by: $\sum_{i=0}^{r}[(input\ size)_i - 1] = \mathbf{112}$ **XOR** gates.

Update Function Unit

The update function requires to XOR between 2 values of r bits each, meaning the cost is $r = \mathbf{16}$ **XOR** gates.

Comparator

The comparator requires to XOR between 2 values of r bits each, meaning the cost is $r = \mathbf{16}$ **XOR** gates, than preform OR for r bits that cost $r - 1 = \mathbf{15}$ **OR** gates.

Protected Memory

In case of fixed size memory, the protected memory has to store 40 bits for each basic block, as described above in 4.3.1. Its size not only determined by the generator polynomial degree but also by the basic block maximum size, meaning total cost will be: $[r + r + \log_2(Max\ Block\ Size)] \cdot \#blocks$

But since most of the blocks do not have update value, we can reduce hardware cost by using flexible size memory. Instead of storing r bits of '0' for every block that do not have an update value, we can store it only once for all those kind of blocks by adding a single bit, $'update\ enable'$, that indicates if the update value is '0' or not. That way, we will have the following memory structures:

1. If a basic block do not have a update value, it has addition of $[1 + r + \log_2(Max\ Block\ Size)]$ bits: *(in our case 25 bits)*

Update enable(1bit)	CRC value (16 bits)	Basic block size (8 bits)

2. If a basic block have a update value, it has addition of $[1 + r + r + \log_2(Max\ Block\ Size)]$ bits: *(in our case 41 bits)*

Update enable (1bit)	CRC value (16 bits)	Update value (16 bits)	Basic block size (8 bits)

In this case, the total cost will be:
$[1 + r + \log_2(Max\ Block\ Size)] \cdot \#blocks + r \cdot (\#Total\ blocks\ enterances - \#blocks)$

For example, considering one of the bench-mark programs (sha.s) and the selected generator polynomial we get the following parameters:

$r = 16$,
Max Block Size = 256 bits,
$\#\text{blocks} = 48$,
$\#\text{Total\_blocks\_enterances} = 55$,
$\#\text{Total blocks enterances} - \#\text{blocks} = 55 - 48 = 7$, which means only 7 blocks requires update value.

In case of **fixed** size protected memory, the cost will be:
$[r + r + \log_2(\text{Max Block Size})] \cdot \text{blocks} = [16 + 16 + \log_2(256)] \cdot 48 =$ **1920 bits**

In case of **flexible** size protected memory cost will be:
$[1 + r + \log_2(\text{Max Block Size})] \cdot \#\text{blocks} + r \cdot (\#\text{Total blocks enterances} - \#\text{blocks})$
$= [1 + 16 + \log_2(256)] \cdot 48 + 16 \cdot 7 =$ **1312 bits** , that is 46% cheaper memory.

4.4.1 Cost Summerize

For the implementation of the hardware the costs are upper bounded as following:

- 16+16+112 = **144 XOR** gates or less
- **15 OR** gates

From memory aspect:

In case of fixed size memory **40 bit** per basic block

In case of flxible size memory it vary from **25 bits** to **41 bits** per basic block

As we can see the flexible size memory would save us significantly amout of memory *(about 46% for sha.s program).*

5. Attacker Simulation

Injection of errors to system during the program run is simulated by the following modifications on the system code, mark in bold. Those changes allows the attacker to tamper the instructions right after they are extracted from the meomry.

```
insertError = rand < errorRate;
i = 1;
for j = bbStart:bbEnd %calculate siganture
    deleteInst = rand < deleteRate;
    if deleteInst == 0
    try %handle 16 bits instruction
        instruction = hexToBinaryVector(opcodes{j,1},k);
        instruction = xor(instruction,
(insertError*errorVector{bbNum}((i-1)*k+1:i*k)));
        instCounter = instCounter + 1;
        i = i+1;
        inst = binaryVectorToHex(instruction);
        if ismember(inst(1:2), branchOpcodes1)
           if instCounter*k > programInfo{bbNum,7}
               break;
           end
        end
     catch %handle 32 bits instruction
         command = hexToBinaryVector(opcodes{j,1},2*k);
         command = xor(command,
(insertError*errorVector{bbNum}((i-1)*k+1:2*i*k)));
         instCounter = instCounter + 2;
         i = i+2;
         inst = binaryVectorToHex(command);
         if ismember(inst(1:5), branchOpcodes2)%check branch
            if instCounter*k > programInfo{bbNum,7}
               break;
            end
         end
         instruction = command(1:k);
         signature = hardwareCRC(instruction, signature,
symbolsMatrix);
         instruction = command(17:2*k);
     end
     signature = hardwareCRC(instruction, signature,
symbolsMatrix);
     addInst = rand < addRate;     %add instruction
     if addInst == 1
        R = [bbStart bbEnd];
        s = round(rand*range(R))+min(R);
        try %handle 16 bits instruction
           instruction = hexToBinaryVector(opcodes{s,1},k);
          …
end (full code in appendix )
```

Adding an instruction is done by selecting a random opcode of the same block from the opcode file and execute it by doing an extra cycle. Deleting an instruction is done by skipping the signature's calculation of a certain instruction.

Error addition to a basic block is done by adding parts of error vector to each instruction in the basic block. The error vector is created in adenvence to keep the weight of the error added to entire basic block under control.

5.1 Attacker Workspace

The attacker can control error rate, type of error, error weight and burst length. We simulate the system's operation while injecting random errors. If the system detects an error it terminates its operation and start again with a different error. If an error is masked, the run will stop and return the error.

```
R1 = [ 1 (d-1) ];
maxLen = 16;
errorRate = 0.5;
addRate = 0;
deleteRate = 0;
i = 0;
while flag == 0
  i = i+1 %print progress
  weight = round(rand*range(R1))+min(R1);
  errorVector = nWeightError(weight, blockSize);
  [flag, error, bbNum] = runSystem( errorVector, errorRate,
         addRate, deleteRate );
end
```

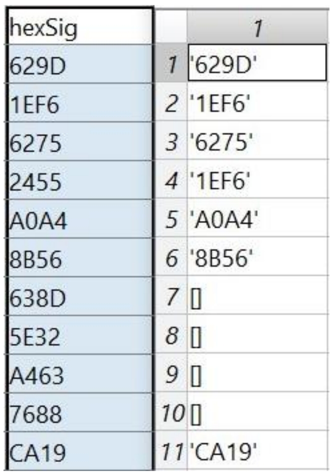
We create the errors in advance with the following functions. It create a vector of error for each block according to its size and the set parameters. First function creates erros of random weight from 1 to $d-1$, and second function creates random burst error with maximum length of r.

```
function [ errorVector ] = nWeightError( weight, blockSize )
    errorVector{size(blockSize,1),1} = [];
    for i = 1:size(blockSize,1)
        error = zeros(1,blockSize(i));
        error(1:weight) = 1;
        error = error(randperm(numel(error)));
        errorVector{i} = error;
    end
end
```

```
function [errorVector] = burstError(r,blockSize)
    errorVector{size(blockSize,1),1} = [];
    for i = 1:size(blockSize,1)
        R1 = [ 2 r ];
        m = round(rand*range(R1))+min(R1);
        ptrn = rand(1,r-m) < 0.5;
        burst = [ 1 ptrn 1 ];
        R2 = [ 1 (blockSize(i)-length(burst))];
        h = round(rand*range(R2))+min(R2);
        burst = [ burst zeros(1,blockSize(i)-length(burst))];
        errorVector{i} = circshift(burst,h);
    end
end
```

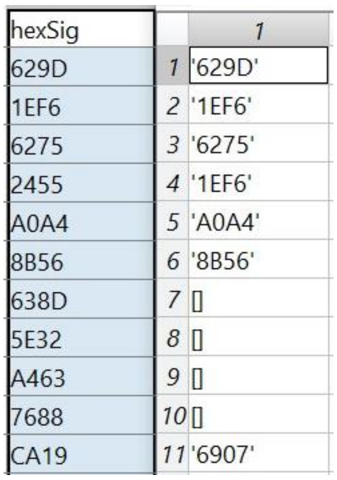
5.2 Attacks Analysis

We tested couple of benchmarks with vast number of runs. When the attacker is limited to errors of weight $d-1=3$ or less and burst errors of length $r=16$ or less, the errors are always detected, as we expected from the CRC.



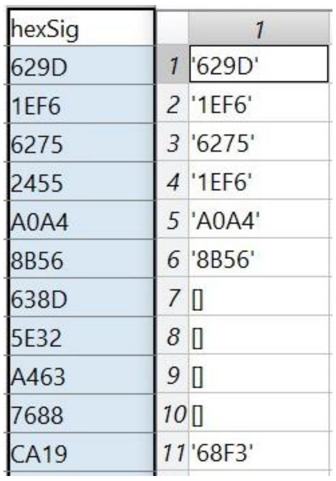
hexSig		1
629D	1	'629D'
1EF6	2	'1EF6'
6275	3	'6275'
2455	4	'1EF6'
A0A4	5	'A0A4'
8B56	6	'8B56'
638D	7	[]
5E32	8	[]
A463	9	[]
7688	10	[]
CA19	11	'CA19'

No errors



hexSig		1
629D	1	'629D'
1EF6	2	'1EF6'
6275	3	'6275'
2455	4	'1EF6'
A0A4	5	'A0A4'
8B56	6	'8B56'
638D	7	[]
5E32	8	[]
A463	9	[]
7688	10	[]
CA19	11	'6907'

$W(error) = 2$



hexSig		1
629D	1	'629D'
1EF6	2	'1EF6'
6275	3	'6275'
2455	4	'1EF6'
A0A4	5	'A0A4'
8B56	6	'8B56'
638D	7	[]
5E32	8	[]
A463	9	[]
7688	10	[]
CA19	11	'68F3'

$L(b.error) = 14$

Figure 5.1: sample of signature values

Figure 5.1 shows samples of the signatures values after the system run(left column) compred to the store signature value(right column) for 3 cases, including errors injections to basic block #11: $b = (F107\ 0308\ 681B\ 2B4F\ DDC5)_{HEX}$

1. No errors was injected.
2. Error of weight 2: $e = (0008\ 0000\ 0004\ 0000\ 0000)_{HEX}$
3. Burst error of length 14: $e = (0000\ 0000\ 0000\ 0076\ D600)_{HEX}$

When no error was added the output signatures are the same as the stored value. But when errors were added, the computed signature was the same for each block untill it reached block #11, then it output different signature from the stored value.

This allows us to observe the error preservation property of the CRC, the error added to the basic block is affecting 2 instruction within the basic block and the calculations of the second tampered instruction does not cancel the error, so the final signature is wrong and the error is detected when compared at the end of the basic block.

The empty cells are basic blocks that were skipped during the run since branch was not taken, and blocks that has the same signature, for exmaple basic blocks #2 and #4, means the signature was updated.(after the check for errors)

As we mentioned in the prelimenary chapter, the CRC cannot detect every error. While some errors are always detected, some errors will never be detected. In order to find those type error we simulated attacks on the system again, but without restricting the type of error, meaning errors weight can be larger than d and burst errors length can be longer than r, limited to the basic block size.

We tested couple of benchmarks with vast number of runs again, expecting to find error that will not be detect, but the simulator always detected the errors. Therefore, from this formula: $s_{i-1}x^k + x^r(m_i \oplus \Delta m_i) \ mod \ g(x) = s_i \oplus \Delta s_i$, we have to find $\Delta m_i \neq 0$ that leads to $\Delta s_i = 0$

$$[s_{i-1}x^k + x^r(m_i + \Delta m_i)] \ mod \ g(x) = s_i + 0$$

$$s_{i-1}x^k mod \ g(x) + x^r m_i mod \ g(x) + x^r \Delta m_i \ mod \ g(x) = s_i + 0$$

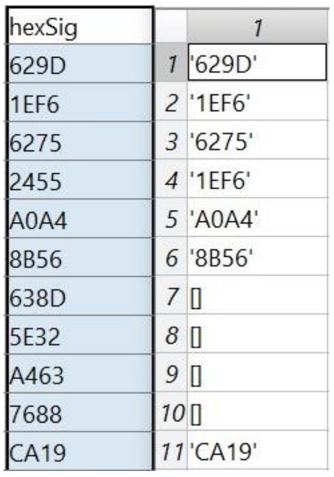
The above equation hold iff: $x^r \Delta m_i \ mod \ g(x) = 0$

Since $\Delta m_i \neq 0$, it must be multiple of $g(x)$

In order to test this type of errors we created the following functions that creates errors that are multiple of $g(x)$. We simulated the program execution with injection of such errors, and as we expected the errors never detected.

```
function [errorVector] = maskedErrors( polynomial, blockSize)
    r = length(polynomial)-1;
    errorVector{size(blockSize,1),1} = [];
    for i = 1:size(blockSize,1)
        error = round(rand(1,blockSize(i)-r));
        errorVector{i} = mod(conv(error,polynomial),2);
    end
end
```

For example, when error $e = (1D02\ 2C97\ 3A2E\ 7224\ 43FA)_{HEX}$, which is multiple of the generator polynomial, is added to the basic block the signature stays the same although error was added, as shown in figure 5.2.



hexSig		1
629D	1	'629D'
1EF6	2	'1EF6'
6275	3	'6275'
2455	4	'1EF6'
A0A4	5	'A0A4'
8B56	6	'8B56'
638D	7	[]
5E32	8	[]
A463	9	[]
7688	10	[]
CA19	11	'CA19'

Figure 5.2: sample of signature values with masked error

6. Conclusions

In this project, we represented the challenge of detecting an error that is caused maliciously by attacker who has pysical access to the system, while focusing on embedded system. We reviewed several approaches to implement code and control flow integrity, including software and hardware methods, which led us to prefer the hardware implementation.

We implemented the solution presented in [9], which suggest an efficient hardware-supported technique to enforce strong and secure code and control flow integrity for embedded system against fault attacks. Since the author doesn't represent all the steps in his solution, during the implemention stage we had to develop for ourself the signature function and find an efficient way to encode the data word on real-time hardware simulation.

We simulated attacks on the system while assuming the attacker is limited to inject either error of weight $d - 1$ or less, or burst error of length r or less and analyzed the results. This solution implement code and control flow integrity and provides the following:

- an add-on module that does not require architectural changes and offers little to none overhead on the system.
- For basic block size of 256 bits and generator polynomial with a degree size of 16, it uses small amount of addional memory, 40 bits for each basic block.
- With the selected basic block size and generator polynomial, errors of weight 3 or less and burst error of length 16 or less are always detected.

We also tested the solution against attacker that is not limited to the restrictions, which is able to inject errors of weight larger than d and burst errors longer than r. We concluded that errors which are multiple of the generator polynomial are never detected, meaing while some errors will always be detect, other errors will never be detected, and the the solution is detecministic due to the use of linear code.

In future work, we can set the initial value of the CRC, s_0, with random value instead of '0'. Assuming the attacker does not know the random value, he will not know the signatures values. Therefore, we can exclude the signatures from the protected memory and use this cost reduction to select a higher degree polynomial and increase the hamming distance of the code by using the same size of memory or simply reduce the hardware costs.

However, this improvement still can not detect errors that are multiple of the generator polynomial and in order to handle it, the encoding calculation must include a step with a random variable for each signature computation.

References

[1] A survey of Hardware-based Control Flow Integrity(CFI), RUAN DE CLERCQ and INGRID VERBAUWHEDE, KU Leuven, 2017

[2] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications.

[3] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. HCFI: Hardware-enforced Control-Flow Integrity.

[4] Z. Guo, R. Bhakta, and I. G. Harris. 2014. Control-flow checking for intrusion detection via a real-time debug interface.

[5] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2006. Hardware-assisted Run-time Monitoring for Secure Program Execution on Embedded Processors.

[6] Wenjian He, Sanjeev Das, Wei Zhang, and Yang Liu. 2017. No-Jump-into-Basic-Block: Enforce Basic Block CFI on the Fly for Real-world Binaries.

[7] Ruan de Clercq, Johannes Gotzfried, U"bler David, Pieter Maene, and Ingrid Verbauwhede. 2017. SOFIA: Software and Control Flow Integrity Architecture.

[8] K. Wilken and J. P. Shen. 1988. Continuous signature monitoring: efficient concurrent-detection of processor control errors.

[9] Protecting the Control Flow of Embedded Processors against Fault Attacks, Mario Werner, Erich Wenger, and Stefan Mangard, 2016

[10] Detecting Bit Errors, MIT Lecture Note:
http://web.mit.edu/6.02/www/f2010/handouts/lectures/L7.pdf

[11] Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks, Philip Koopman ECE Department & ICES Carnegie Mellon University and Tridib Chakravarty

List of all best general-purpose CRC polynomials with specific Hamming Distance Properties: https://users.ece.cmu.edu/~koopman/crc/

[12] The Theory of Error Correcting Codes, F.J. MacWilliams and N.J.A. Sloane, Lecture Notes by Prof.Osnat Keren

Appendix A

The Full code is on Github by the following link:

https://github.com/agrrr/Final-project

```
import csv
import sys
from dataclasses import dataclass
from pathlib import Path
import math
import networkx as nx
import matplotlib.pyplot as plt
import string
OPSIZE = 16
BB_MAX_BYTE_SIZE = pow(2, 8)
BB_MAX_COMMAND_COUNTER = BB_MAX_BYTE_SIZE/16
CON_BRANCHES_LIST = ['ble','beq','bne','bpl','bmi', 'bcc','bcs',
'bgt','bge', 'blt','bhi','bls']


@dataclass
class Basic_block:
    num: int
    num_for_matlab:int
    startBy: string
    endBy: string
    start_row_counter: int
    start_row_op: int
    start_location: int
    end_row_counter: int
    end_row_op:int
    end_location: int
    num_of_asm_rows: int
    num_of_long_commands: int
    num_of_op_rows: int
    is_it_con_branch: bool
    next1: int
    next1_location: int
    next2: int
    next2_location: int
    next_for_matlab: int
    byteSize: int
    enterances: int

def collect_labels_data(filename):
def collect_jump_data(lsd):  # lsd[file_name, file_size, labels]
def write_to_csv_file(*argv):
def write_to_file(*argv, separatore=','):
def analytic_part(*argv):
def make_CFG(list_of_data):
def draw_graph(G_directed):
def making_binary_and_assembly_files(source_name):
def need_to_be_splited(bb:Basic_block):
def optimizing_bb(asm_file_name:string, bb:Basic_block,
num_to_split_to:int):
def spliting_bb(asm_file_name, bb:Basic_block,
mean_command_for_bb,skip:bool):
def split_bb_to_min_size(CFG, list_of_data, flow_file,
bb_max=BB_MAX_BYTE_SIZE):
def is_a_con_branch(str):
def is_a_branch(str: string):
def num_of_long_commands(asm_name=string, start=int, end=int):
def convert_dict_to_classes(dict):
def covert_references(bb_dict: dict, assembly_file: string):
def print_label_file(list_of_data):
def print_branch_file(list_of_data):
def print_flow_file(list_of_data):
def print_dict(dict):
def print_bb_dict(dict:{'':Basic_block}):
def print_bb_dict_for_matlab(dict:{'':Basic_block}):
def main(*argv):   # *argv
    making_binary_and_assembly_files(*argv)
    analytic_part(*argv):
# End of program
```

Appendix B

MATLAB code for the offline pre-calculations of CRC and update values

```
opcodes = readtable('sha_bin.txt','Format','auto');
programInfo = readtable('sha_flow.csv');
polynomial = (hexToBinaryVector('1a2eb')); %generator polynomial
sigVector{size(programInfo,1),1} = []; %vector of all signature
preSigVector{size(programInfo,1),1} = []; %vector of all pre-
signature
preSigVector{1} = zeros(1,length(polynomial)-1); %initial first
signature
updateValue{size(programInfo,1),1} = []; %vector of update function
value
bbNum = 1; %first block
cb = {'ble',
'beq','bne','bpl','bmi','bcc','bcs','bgt','bge','blt','bhi','bls'};
%conditional branch commands type

[sigVector,updateValue,preSigVector] = sigAndUpdateValue(opcodes,
programInfo, polynomial, sigVector, preSigVector, updateValue, cb,
bbNum, 1);
hexSig{size(programInfo,1),1} = [];
hexUpdate{size(programInfo,1),1} = [];
for i = 1:size(programInfo,1)
    hexSig{i} = binaryVectorToHex(sigVector{i});
    hexUpdate{i} = binaryVectorToHex(updateValue{i});
end
%writing values to file
T1 = array2table(hexSig);
T2 = array2table(hexUpdate);
programInfo = [ programInfo T1 T2 ];
writetable(programInfo, 'sha_flow_compiled.csv');
```

CRC function for the offline calculations:

```
function [ signature ] = CRC_func( inst, poly, preSig, k )
% S_i = (s_i-1*x^k + I_i*x^r) mod g
msg = [preSig, zeros(1,k)] + [inst, zeros(1,(length(poly)-1))];
[q,reminder] = deconv(msg,poly);
signature = mod(reminder(k+1:end),2);
```

Recursive function that tracks all possible paths:

```
function [ sigVector,updateValue,preSigVector] =
sigAndUpdateValue(opcodes, programInfo, polynomial, sigVector,
preSigVector, updateValue, cb, bbNum, preBbNum)
    k = 16;
    if isnan(bbNum) %end of program
       updateValue{preBbNum} = zeros(1,length(polynomial)-1);
        return
    end
    if isempty(sigVector{bbNum}) %calculate basic block signaute
      updateValue{preBbNum} = zeros(1,length(polynomial)-1);
      signature = preSigVector{bbNum};
      % basic block bounds
      bbStart = programInfo{bbNum,5};
      bbEnd = programInfo{bbNum,6}-1;
      for j = bbStart:bbEnd %calculate siganture
            try %handle 16 bits instruction
                instruction = hexToBinaryVector(opcodes{j,1},k);
            catch %handle 32 bits instruction
                command = hexToBinaryVector(opcodes{j,1},2*k);
                instruction = command(1:k);
                signature = CRC_func(instruction,
polynomial,signature,k);
                instruction = command(17:2*k);
            end
            signature = CRC_func(instruction,
polynomial,signature,k);
        end
        sigVector{bbNum} = signature;
        %determine basic path
        if ismember(programInfo{bbNum,3}{1}(1:3), cb) %path splits
            bbNum1 = bbNum+1; %not taken
            bbNum2 = programInfo{bbNum,4}; %taken
            if ~isnan(bbNum1) && isempty(preSigVector{bbNum1})
                preSigVector{bbNum1} = signature;
            end
            [sigVector,updateValue,preSigVector] =
sigAndUpdateValue(opcodes, programInfo, polynomial, sigVector,
preSigVector, updateValue, cb, bbNum1, bbNum);
            if ~isnan(bbNum2)&& isempty(preSigVector{bbNum2})
                preSigVector{bbNum2} = signature;
            end
            [sigVector,updateValue,preSigVector] =
sigAndUpdateValue(opcodes, programInfo, polynomial, sigVector,
preSigVector, updateValue, cb, bbNum2, bbNum);
                return
        else %fixed path
            bbNum1 = programInfo{bbNum,4}; %the next block
            if ~isnan(bbNum1) && isempty(preSigVector{bbNum1})
                preSigVector{bbNum1} = signature;
            end
            [sigVector,updateValue,preSigVector] =
sigAndUpdateValue(opcodes, programInfo, polynomial, sigVector,
preSigVector, updateValue, cb, bbNum1, bbNum);
            return
        end
    else
updateValue{preBbNum} = xor(sigVector{preBbNum},preSigVector{bbNum});
        return
    end
end
```

Appendix C

System and encoding simulation code:

```
clear all;
opcodes = readtable('sha_bin.txt','Format','auto');
programInfo = readtable('sha_flow_compiled.csv','Format','auto');
polynomial = (hexToBinaryVector('1a2eb')); %generator polynomial
k = 16;
r = length(polynomial) - 1;
signature = zeros(1,length(polynomial)-1); %initial first signature
hexSig{size(programInfo,1),1} = []; %vector of all signature
bbNum = 1; %first block
error = 0;
cb = {'ble',
'beq','bne','bpl','bmi','bcc','bcs','bgt','bge','blt','bhi','bls'};
%conditional branch commands type
remindersVector = reminders(polynomial, k, r);
branchOpcodes1 =
{'d0','d1','d2','d3','d4','d5','d8','d9','da','db','dc','dd','e0'};
%16bits branchs
branchOpcodes2 =
{'f000b','f0008','f0408','f53fa','f73fa','f2808','f6ffa','f3408','f
2408'};%32bits branchs

while ~isnan(bbNum) %run program untill it finish
    % basic block bounds
    bbStart = programInfo{bbNum,5};
    bbEnd = programInfo{bbNum,6}-1;
    instCounter = 0;
    for j = bbStart:bbEnd %calculate siganture
        try %handle 16 bits instruction
            instruction = hexToBinaryVector(opcodes{j,1},k);
            instCounter = instCounter + 1;
            if ismember(opcodes{j,1}{1}(1:2), branchOpcodes1)%
                if instCounter*k > programInfo{bbNum,7} %size check
                   error = 1;
                   break;
                end
            end
        catch %handle 32 bits instruction by doing extra cycle
            command = hexToBinaryVector(opcodes{j,1},2*k);
            instCounter = instCounter + 2;
            if ismember(opcodes{j,1}{1}(1:5), branchOpcodes2)%
                if instCounter*k > programInfo{bbNum,7} %size check
                    error = 1;
                    break;
                end
            end
            instruction = command(1:k);
            signature = hardwareCRC(instruction, signature,
remindersVector);
            instruction = command(17:2*k);
        end
        signature = hardwareCRC(instruction, signature,
remindersVector);
    end
    hexSig{bbNum} = binaryVectorToHex(signature);
    %to do: checker
    if error == 1 || ~strcmp(hexSig{bbNum},programInfo{bbNum,8})
```

```
        break;
    end

    %determine path
    if ismember(programInfo{bbNum,3}{1}(1:3), cb) %path splits
        p = rand;
        if p > 0.5 %branch taken
            updateValue =
hexToBinaryVector(programInfo{bbNum,9}{1});
            updateValue = [zeros(1, r-length(updateValue)),
updateValue];
            signature = xor(updateValue, signature);
            hexSig{bbNum} = binaryVectorToHex(signature);
            bbNum = programInfo{bbNum,4};
        else
            bbNum = bbNum+1; %branch not taken
        end
    else %fixed path
        updateValue = hexToBinaryVector(programInfo{bbNum,9}{1});
        updateValue = [zeros(1, r-length(updateValue)),
updateValue];
        signature = xor(updateValue, signature);
        hexSig{bbNum} = binaryVectorToHex(signature);
        bbNum = programInfo{bbNum,4}; %update to the next block
    end
end
```

Appendix D

System and encoding simulation code with modifications to inject errors:

```
function [flag, error, bbNum, hexSig] = runSystem( errorVector,
errorRate, addRate, deleteRate )
  opcodes = readtable('sha_bin.txt','Format','auto');
  programInfo = readtable('sha_flow_compiled.csv','Format','auto');
  polynomial = (hexToBinaryVector('1a2eb')); %generator polynomial
  k = 16;
  r = length(polynomial) - 1;
  signature = zeros(1,r); %initial first signature
  hexSig{size(programInfo,1),1} = []; %vector of all signature
  bbNum = 1; %first block
  error = 0;
  cb = {'ble',
'beq','bne','bpl','bmi','bcc','bcs','bgt','bge','blt','bhi','bls'};
%conditional branch commands type
    branchOpcodes1 =
{'d0','d1','d2','d3','d4','d5','d8','d9','da','db','dc','dd','e0'};
%16bits branchs
    branchOpcodes2 =
{'f000b','f0008','f0408','f53fa','f73fa','f2808','f6ffa','f3408','f
2408'};%32bits branchs
  reminderMatrix = reminders(polynomial, k, r);
  flag = 0;
  while ~isnan(bbNum) %run program untill it finish
        error = errorVector{bbNum};
        % basic block bounds
        bbStart = programInfo{bbNum,5};
        bbEnd = programInfo{bbNum,6}-1;
        insertError = rand < errorRate;
        instCounter = 0;
        i = 1;
        for j = bbStart:bbEnd %calculate siganture
            deleteInst = rand < deleteRate;
            if deleteInst == 0
                try %handle 16 bits instruction
                  instruction = hexToBinaryVector(opcodes{j,1},k);
                  instruction = xor(instruction,
(insertError*errorVector{bbNum}((i-1)*k+1:i*k)));
                  instCounter = instCounter + 1;
                  i = i+1;
                  inst = binaryVectorToHex(instruction);
                  if ismember(inst(1:2), branchOpcodes1)
                    if instCounter*k > programInfo{bbNum,7}
                        error = 1;
                        break;
                    end
                  end
                catch %handle 32 bits instruction
                    command = hexToBinaryVector(opcodes{j,1},2*k);
                    command = xor(command,
(insertError*errorVector{bbNum}((i-1)*k+1:2*i*k)));
                    instCounter = instCounter + 2;
                    i = i+2;
                    inst = binaryVectorToHex(command);
                    if ismember(inst(1:5), branchOpcodes2)
                        if instCounter*k > programInfo{bbNum,7}
                            error = 1;
                            break;
                        end
                    end
                    instruction = command(1:k);
                    signature = hardwareCRC(instruction, signature, 
reminderMatrix);
                    instruction = command(17:2*k);
                end
                signature = hardwareCRC(instruction, signature, 
reminderMatrix);
                %add instruction
                addInst = rand < addRate;
                if addInst == 1
                    R = [bbStart bbEnd];
                    s = round(rand*range(R))+min(R);
                 try %handle 16 bits instruction
                    instruction = 
hexToBinaryVector(opcodes{s,1},k);
                    instruction = 
xor(instruction,(insertError*errorVector{bbNum}((i-1)*k+1:i*k)));
                    instCounter = instCounter + 1;
                    i = i+1;
                    inst = binaryVectorToHex(instruction);
                    if ismember(inst(1:2), branchOpcodes1)
                        if instCounter*k > programInfo{bbNum,7}
                            break;
                        end
                    end
                catch %handle 32 bits instruction
                    command = hexToBinaryVector(opcodes{s,1},2*k);
                    command = 
xor(command,(insertError*errorVector{bbNum}((i-1)*k+1:2*i*k)));
                    instCounter = instCounter + 2;
                    i = i+2;
                    inst = binaryVectorToHex(command);
                    if ismember(inst(1:5), branchOpcodes2)%check 
branch
                        if instCounter*k > programInfo{bbNum,7} 
%size check
                            break;
                        end
                    end
                    instruction = command(1:k);
                    signature = hardwareCRC(instruction, signature, 
reminderMatrix);
                    instruction = command(17:2*k);
                 end
                    signature = hardwareCRC(instruction, signature, 
reminderMatrix);
                end
            end
        end
        hexSig{bbNum} = binaryVectorToHex(signature);
        %checker
        if strcmp(hexSig{bbNum},programInfo{bbNum,8})  && 
(insertError == 1)
            flag = 1;
            return
        end
        if ~strcmp(hexSig{bbNum},programInfo{bbNum,8})
           flag =1;
           return
        end

%determine path
        if ismember(programInfo{bbNum,3}{1}(1:3), cb) %path splits
            p = rand;
            if p > 0.5 %branch taken
                updateValue =
hexToBinaryVector(programInfo{bbNum,9}{1});
                updateValue = [zeros(1, r-length(updateValue)),
updateValue];
                signature = xor(updateValue, signature);
                hexSig{bbNum} = binaryVectorToHex(signature);
                bbNum = programInfo{bbNum,4};
            else
                bbNum = bbNum+1; %branch not taken
            end
        else %fixed path
            updateValue =
hexToBinaryVector(programInfo{bbNum,9}{1});
            updateValue = [zeros(1, r-length(updateValue)),
updateValue];
            signature = xor(updateValue, signature);
            hexSig{bbNum} = binaryVectorToHex(signature);
            bbNum = programInfo{bbNum,4}; %update to the next block
        end
    end
end
```