

INJECTION ATTACK IMMUNITY USING REDUNDANT HETEROGENEOUS
PROCESSING CORES

by

Colter Ross Barney

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2023

©COPYRIGHT

by

Colter Ross Barney

2023

All Rights Reserved

ACKNOWLEDGEMENTS

First and foremost I would like to thank Dr. Brock LaMeres for the opportunity to work on a project I am passionate about, and for all the support he has given me over my four years as his research assistant. Secondly I would like to thank the other students who helped work on this specific project, Tristan Running Crane, Blake Stanger, Walker Ward, and Rory Myer. I would like to also thank the other students who I have had the absolute pleasure to work with over the years, Justin Williams, Chris Major, Zach Hull, and Hezekiah Austin. Finally I would like to thank my family who has always encouraged me to chase my dreams and work hard.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. MOTIVATION	4
Homogeneous Environments and The Danger They Present.....	4
Opcodes and Instruction Sets	5
Secure Computing Systems	7
Secure Embedded Systems	7
Buffer Overflow Attack	10
3. RELATED WORK	13
Opcode Encryption.....	13
4. OUR APPROACH.....	17
Obfuscation.....	17
5. SYSTEM DESIGN.....	20
Processor	20
Memory	23
Peripherals	25
Input Output Design	25
Timer Design	25
Interrupt Controller Design	29
Serial Link Design	30
Tool Chain	33
Recovery System	34
6. SYSTEM VERIFICATION	36
Processor Verification.....	36
Peripheral Verification	39
GPIO Verification	39
Timer and Interrupt Verification	40
Serial Link Verification	43
Recovery Verification	44
Full System Verification	45
System Application	46
Attack Vector and Method.....	46

TABLE OF CONTENTS – CONTINUED

Full System Results	47
Performance Evaluation	49
7. FUTURE WORK	50
Partial Reconfiguration	50
Processor Shift	50
Attack Prevention Techniques	51
ASLR	51
Stackguard.....	52
Non-executable Stack	52
Recovery Subroutines.....	52
8. CONCLUSION	54
REFERENCES CITED.....	55

LIST OF FIGURES

Figure	Page
2.1 Top Five Microprocessor Companies.....	4
2.2 MSP430 Opcode Table [14].....	6
2.3 Cyber Attack Types [4]	8
2.4 Vulnerabilities Classified [9]	10
2.5 Memory Layout	12
3.1 “Encoded instructions are decoded before being processed by the CPU”[5]	13
3.2 “ASIST Decryption Scheme” [7]	15
3.3 Polyglot Structure [11]	16
4.1 Assembly Program [3]	19
5.1 MSP430FR2355 CPU Block Diagram [14].....	21
5.2 Base Register in Simulink	22
5.3 Routes Through the Control Unit	23
5.4 MSP430FR2355 Memory Map [13]	24
5.5 MSP430FR2355 Block Diagram [13]	26
5.6 MSP430FR2355 Timer B diagram [14]	28
5.7 MSP430FR2355 Timer Capture[14]	29
5.8 MSP430FR2355 Interrupt Controller Module[14].....	30
5.9 MSP430FR2355 eUSCI Module[14]	31
5.10 MSP430FR2355 Baudrate Generator Settings[14]	32
5.11 CCS Project	34
6.1 Assembly Test Program.....	37
6.2 Matching Registers.....	38
6.3 Busy Loop PWM Simulation	40
6.4 Hardware Setup for Servo Test.....	40

LIST OF FIGURES – CONTINUED

Figure	Page
6.5 Busy Loop PWM Measurement	41
6.6 Code Comparison.....	42
6.7 Timer PWM Simulation	43
6.8 Timer PWM Measurement	43
6.9 UART Echo Simulation	44
6.10 UART Echo Measurement	44
6.11 Simulation of Recovery Procedure	45
6.12 Recovery Procedure on FPGA	45
6.13 Demo Block Diagram	46
6.14 Demo Physical Setups	48
6.15 Attack on Hardware	48
6.16 Single Core Resource Usage	49
6.17 Redundant Cores Resource Usage	49

ABSTRACT

Technology is an integral part of modern society. Devices such as smart lights, locks and appliances are becoming more commonplace. This class of devices are called embedded systems. Embedded systems can be targeted by malicious cyber attacks just as a normal computer can. Unfortunately, many techniques used to secure and protect normal computers do not work on embedded systems. New security techniques must be developed and designed to protect embedded systems. This paper investigates using physically diverse processing cores to defeat cyber attacks in real time. Diverse processing cores were implemented using reconfigurable hardware devices called FPGAs. The use of FPGAs allows diverse cores to be utilized, without losing the benefits gained from standardized processors. The cores implemented were based on a commercial processor made by Texas Instruments (TI). Modeling the diverse cores after a commercial processor enables the cores to utilize development tools created for TI's processor. A complete system was built using diverse processors to prove the feasibility and usability of secure embedded systems. The cores were used to control a realistic embedded system application. While operating, the cores were subjected to a cyber attack, and they were able to nullify the attack. An identical setup was created using the commercially available processor. Attacking the commercial processor compromised the application and reinforced the need for secure systems. The techniques investigated and utilized in this paper can be expanded to increase security in the many embedded systems that have become an essential part of modern lifestyles.

INTRODUCTION

Computers are one of the most revolutionary inventions of human history. Since their invention in the 1940's, the use of computers has spread to nearly every aspect of life. In 2009 manufacturers produced over nine billion microprocessors with approximately 99 percent of those being deployed to embedded systems [1]. Embedded systems are devices built to accomplish a limited set of tasks and use microprocessors to better accomplish those tasks. Many of these embedded systems interact with people or are connected to the internet. These interactions allow for embedded systems to be targeted by malicious parties. Attacks on computer architectures are commonly called cyber attacks.

Processors are one of the most prolific type of chips used in the modern, technology rich environment. Only a handful of companies design and manufacture the vast majority of the processing chips on the market. In order to make code development easier and more transferable, most of these manufactures use one architecture to cover many different processor models. These factors work to increase productivity, but they also add vulnerability. The same way development becomes more transferable, so does malware. Malware is code inserted into a system with the intent to damage, steal information, or otherwise compromise the system. Malware can be designed for a specific architecture and transfer to all processors using the same architecture. This transferability of malware is dangerous to all computers, but especially dangerous to embedded systems. Embedded systems cannot utilize many anti-malware techniques, and they are not designed to be updated after deployment.

Embedded systems accomplish a specific task using a computer as a main processing unit. The computer internal to an embedded system is not used to perform general

computing but exists only to better perform the specific task of the system. “Embedded systems are often characterized by what they are not: the antithesis of the embedded system is the desktop personal computer...” [6] Some common embedded systems would be the flight control unit in an airplane, the engine control unit in an automobile, or a digital thermostat in a house heating system. Embedded systems use microprocessors, processors with a limited memory and hardware peripherals all together on a single integrated circuit. Microprocessors have also benefited from the rapid increase in processor capabilities, becoming faster, smaller, and more powerful in only a couple decades. Just as computers have made rapid technological advancements, so has computer security.

As cyber attacks became more prevalent and more dangerous, people in the computing field began to develop methods to prevent and handle cyber attacks. The study of these methods and development of new ones is now its own field called cyber security. In a desktop computer, most cyber attacks leverage a vulnerability in the software being run by the computer. Once a vulnerability in software is detected, a fix can be quickly developed and deployed as part of an update to the software itself. If a vulnerability is detected in hardware, the mitigation strategy is significantly more complex and costly. In some cases, updates and operating system patches can make the vulnerability inaccessible. If the vulnerability cannot be fixed by updates, new hardware without the vulnerability must be developed, fabricated, and deployed. This process often takes years during which the systems in use are vulnerable. Embedded systems rarely have the ability to be updated. Appliances such as an oven or refrigerator, smart home lights, and locks are items that could contain a vulnerability. A vulnerability must be treated like a hardware vulnerability in these devices because they do not support software patches. As processors were developed, methods were implemented to help with the prevention of cyber attacks at the hardware level. Embedded systems did not make the same security advancements as desktop computers and are becoming increasingly vulnerable to malicious attacks.

Microprocessors generally do not run a conventional operating system like a desktop computer. This feature makes many security programs and techniques irrelevant for securing embedded systems. Embedded system security was not advanced as quickly because there was little need for it. A house thermostat with only control buttons is secure. The only user interaction is limited to pressing the buttons. Embedded security may not have been a major issue in the past, but recent technology trends have put embedded systems at risk. One trend in particular being the Internet of Things (IOT). IOT connects devices like automobiles, thermostats, home locks, lighting, etc, to the internet and allows them to be controlled remotely. While this technology is extremely convenient and useful, it allows previously safe devices to be attacked by malicious actors. This new vulnerability demands new security techniques be developed to protect embedded systems.

Researchers at Montana State University (MSU) have investigated implementing heterogeneous processing cores as a method to detect malware attacks. This paper is a continuation of research into heterogeneous processing cores allowing processing cores to detect and overcome malware attacks. The system designed for this paper had more processing power than previous systems investigated. The system also supported a commercial development environment and program development in C. Testing of the system was performed using a serial injection taking advantage of a buffer overflow vulnerability. While the system developed for this paper is nearly ready for commercial applications, further expansion of the concepts used is still beneficial.

MOTIVATION

Homogeneous Environments and The Danger They Present

During early development of processors, many different companies were fighting for market control. This capitalist competition lead to many different processing architectures being created. In a 2009 article, nine different processor architectures were deemed “world changing” [10]. Processor architectures were being improved quickly meaning each company was releasing their own “better” processor. This led to heterogeneity in the market. These older processors were programmed primarily with compiled languages like C and Fortran. The compilers converted relatively low level code into the binary instructions needed on a per architecture basis. Malicious executable programs were often written to target a single architecture. The diversity created through competition meant that a malicious program was only dangerous to a relatively small portion of the market processors. Today’s processor market is significantly different. There are two primary methodologies for designing computers, Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). A market survey from 2021 shows that the top five microprocessor companies account for 82% of total microprocessor sales [8]. These five companies all make

Leading MCU Suppliers (\$M)						
2021 Rank	Company	Headquarters	2020	2021	21/20 % Chg	2021 Marketshare
1	NXP	Europe	2,980	3,795	27%	18.8%
2	Microchip	U.S.	2,872	3,584	25%	17.8%
3	Renesas	Japan	2,748	3,420	24%	17.0%
4	ST	Europe	2,506	3,374	35%	16.7%
5	Infineon	Europe	1,953	2,378	22%	11.8%

Source: Company reports, IC Insights

Figure 2.1: Top Five Microprocessor Companies

their own processors, but the processors all utilize a common architecture named “ARM”.

Advanced RISC Machine (ARM) processors have become the industry standard. Using a standard processor across the industry enables developers to quickly transfer processor manufacturers. The standard also allows developer skills to remain relevant for a whole career. Unfortunately a homogeneous market also means malicious programs written for a specific architecture place much of the market at risk. This paper investigates a technique to mitigate the risk of a homogeneous market while maintaining the benefits.

Opcodes and Instruction Sets

Processors are a hardware circuit that manipulates and moves data. There are different paths through the circuit like different branches in a tree. Each path through the circuit handles data in unique ways. These paths are called instructions, and they define exactly what a processor can accomplish. There are three general categories of instructions; data flow, program flow, and arithmetic/logic operations. Data flow instructions are used to manage data movement between memory, system registers, and any other component that can store data. Program flow instructions are used to change the order of instruction execution. Arithmetic and logic operations are used to change data. Executing these instructions in different sequences is programming at the most basic level. CISC architectures create an instruction for any operation possible. Instructions dedicated to an operation can be completed faster than using multiple instructions. A disadvantage of a CISC architecture is hardware circuits must be generally larger and more power intensive. A RISC architecture has only the most basic operations needed. Simpler operations have simpler circuits, and take less time to execute. Chaining multiple simple operations allows even complex tasks to be achieved. Multiplication is one of the best examples to illustrate RISC vs CISC. A CISC architecture will have an instruction dedicated to multiplication, while most RISC architectures will instead implement only an addition instruction. Repeating the addition achieves multiplication. Both architecture ideologies have benefits and drawbacks, but CISC

computers have proven to be faster. This speed benefit has lead nearly all desktop processors to use CISC architectures. Embedded systems most commonly use RISC style architectures. The size and low power benefits of RISC architectures are more valuable the the speed benefits gained from CISC architectures.

Instructions allow a processor to move and manipulate numerical data. A processor loads numerical values from the program into system registers. The values loaded are called operation codes or opcodes. Operation codes are numerical values linked to the instructions a processor can execute. An example operation table can be seen in Figure 2.2

Table 4-20. Instruction Map of MSP430X

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx	MOVA, CMPA, ADDA, SUBA, RRCM, RRAM, RLAM, RRUM															
10xx	RRC	RRC. B	SWP B		RRA	RRA. B	SXT		PUS H	PUS H.B	CALL		RETI	CALL A		
14xx	PUSHM.A, POPM.A, PUSHM.W, POPM.W															
18xx	Extension word for Format I and Format II instructions															
1Cxx																
20xx	JNE, JNZ															
24xx	JEQ, JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Figure 2.2: MSP430 Opcode Table [14]

A processors set of instructions and their corresponding opcodes are almost always documented in the publicly available information about a processor. Developers designing systems use the publicly available information in their designs, but malicious actors also use the information to create malware.

Secure Computing Systems

Computers store astronomical amounts of sensitive information and control many important systems. Computers must have security measures in place to protect both the information they store and any systems they control. These security measures can be quite complex and must handle many different styles of security threats. Some common types of cyber attacks are malware, spyware, ransomware, denial of service, phishing, spoofing, database injection, and password compromising attacks. Each different style of attack requires a unique security measure. For example, password compromising attacks can be defeated by requiring passwords to be complex and difficult to guess. Strong passwords are hard to compromise, but they provide no security from malware, spyware, or ransomware. Computers run antivirus software to defeat malware attacks. Using these many layers of security computers can be built to withstand and defeat many cyber attacks. Embedded systems are a subset of computing systems in general. Just as embedded systems are a subset of computers, there are less cyber attack types that can target embedded systems. Attacks like phishing, spoofing, and database injection are almost never a threat to an embedded system. These attacks rely heavily on user interactions that rarely occur on embedded systems, for example surfing the internet. There are still cyber attacks that target embedded systems. Malware is one of the most common attacks and embedded systems do not have antivirus software to protect them. This vulnerability of embedded systems is extremely dangerous.

Secure Embedded Systems

As embedded systems are increasingly used in critical infrastructures, the need for them to be secure increases. The focus of this paper is to protect embedded systems from malware. According to the National Institute of Standards and Technology (NIST), malware

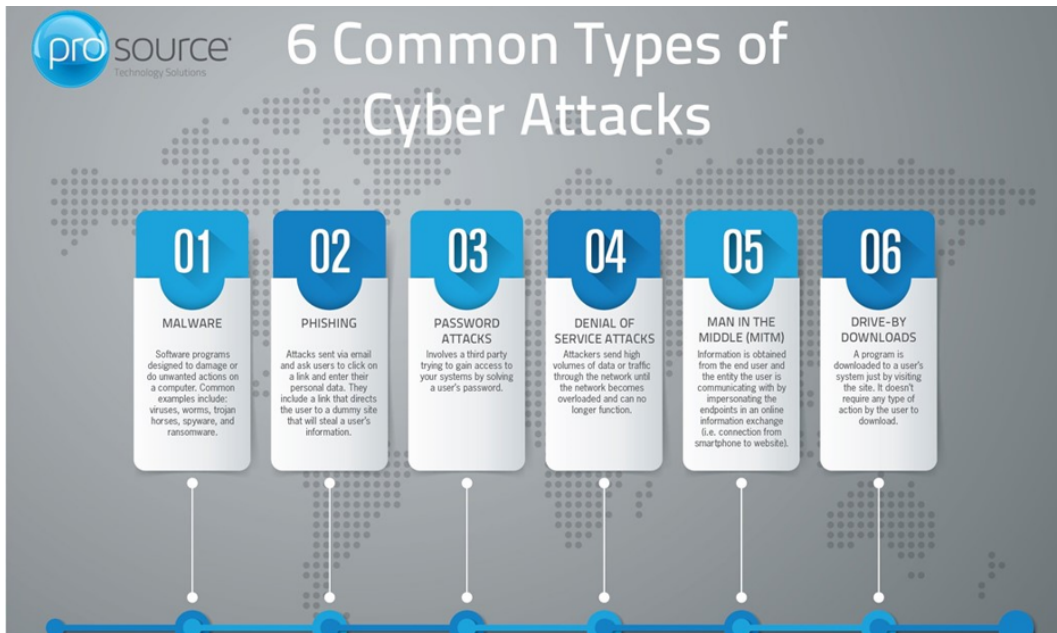


Figure 2.3: Cyber Attack Types [4]

is “a program that is covertly inserted into another program with the intent to destroy data, run destructive or intrusive programs, or otherwise compromise the confidentiality, integrity, or availability of the victim’s data, applications, or operating system.” [12] While many malware mitigation techniques already exist, most of these techniques do not apply to embedded systems.

User machines like desktops and laptops often use antivirus services to help protect themselves from malware. Antivirus services search for known forms of malware in system files as well as files sent to a system across a network. These scans are extremely effective at preventing and removing attacks from known malware, but antivirus applications often fail when presented with a new or highly customized malware attack. Furthering the shortcomings of antivirus software, embedded systems often have limited processing resources and no support for antivirus services to run in the background.

Another security measure to help prevent malware attacks is a technique called sandboxing. NIST states “Sandboxing refers to a security model where applications are run

within a sandbox- a controlled environment that restricts what operations the applications can perform and that isolates them from other applications running on the same host.” [12] Sandboxing is extremely effective as it limits the permissions and access that a compromised application has. Unfortunately, sandboxing is rarely applicable to embedded systems. Embedded systems typically run a single application and do not have multiple users with varying levels of permissions. A compromised embedded application will have access to the whole system.

Firewalls and network-based intrusion prevention systems (IPS) can also be used to prevent malware attacks. These systems can both be implemented on a host machine or at the network level. “A host-based firewall is a piece of software running on a single host that can restrict incoming and outgoing network activity for that host only” [12] Firewalls are configured with rules to block network traffic based on the information’s start or destination. An IPS functions similar to a firewall with one key difference. Instead of blocking network traffic based on source/destination rules, an IPS analyzes the data inside traffic and blocks traffic detected as potential malware. While a network level IPS or firewall will help to protect an embedded system, both systems cannot be run on the embedded system itself. Similar to antivirus software, most embedded platforms do not support or have surplus resources to be used for these protection measures.

Application whitelisting can also be used to prevent malware attacks on many systems. Application whitelisting stores a list of acceptable programs that can be run on a machine and blocks a program from executing if it is not on the list. Most embedded systems run a single application meaning the only task acceptable is the only task possible. This programming style makes application whitelisting irrelevant on embedded systems and thus it does not protect embedded systems from malware.

Malware is often delivered to a system through email or malicious websites. Due to this fact, NIST recommends exercising proper content filtering. Content filtering is the act

of blocking access to services that may carry malware. This is most commonly seen as not allowing potentially harmful websites to be visited and blocking spam emails on a work computer. While embedded systems do not have email or web browsing services the idea remains the same. Embedded systems must interact with the outside world at some point, and that access point is a potential malware delivery point. Due to the single threaded nature of these systems, content filtering is a resource expensive technique and is not commonly implemented. Furthering the issue, filters cannot be easily updated post system deployment. Embedded systems will need a different approach to protect their access points.

Buffer Overflow Attack

Any attack on a system by malware can be broken into two distinct parts, the attack vector and the executable malware. The executable malware is the malicious program that when executed causes damage to the system. The attack vector is the mechanism used to insert the executable into the system and trigger executable operation. The largest category of attack vectors are classified as vulnerabilities in both hardware and software. A very common attack vector is the buffer overflow vulnerability.

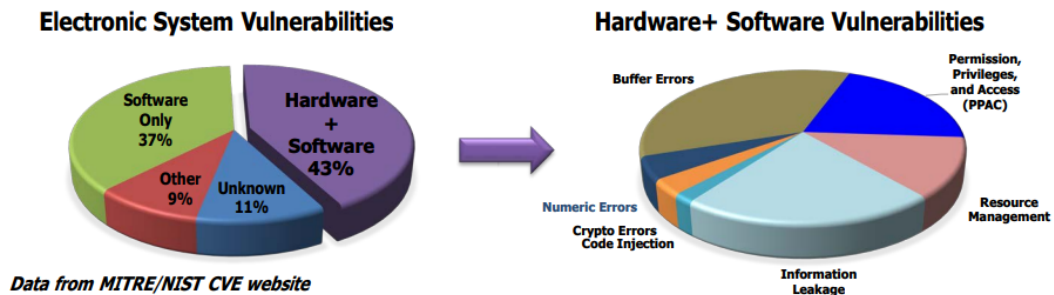


Figure 2.4: Vulnerabilities Classified [9]

A buffer overflow vulnerability is a broad category of issues with each implementation having unique dangers. An attack exploiting a buffer vulnerability can be extremely complex or can be quite simple. This paper focuses on a relatively simple buffer exploit, targeting

some of the core functions required for any exploit regardless of complexity. At the core of any buffer attack is the malware injection and the program counter redirection. The malware injection stage is sending a program disguised as normal user input. The inserted malware can be nearly any program. After the malware is injected, the system has to be forced into executing the inserted program. This stage is called the program counter redirection. Processors use a special register to track the location of current program execution in memory. This register is named the program counter. Program counter redirection is often done by changing the return address inside a function. To make programs more readable and compact, programmers write sections of code called functions. These functions then can be executed multiple times avoiding the need to copy and paste code for repeated operations. When a function is called, the program counter jumps to the location of the function in program code. A structure called the stack is used to store some important values about the state of a processor before executing a function. One of these values is the location of the program counter before the function, also known as the return address. If the return address on the stack were to be changed during function execution, the processor would finish the function and return to this incorrect address. This is the idea behind program counter redirection. The most simple buffer attack can insert malware and change the return address in one step.

Figure 2.5 shows a common memory layout and how a buffer overflow attack can be used to execute arbitrary malware. An attacker who finds a vulnerable buffer and knows the underlying architecture of a system can design an attack with relative ease.

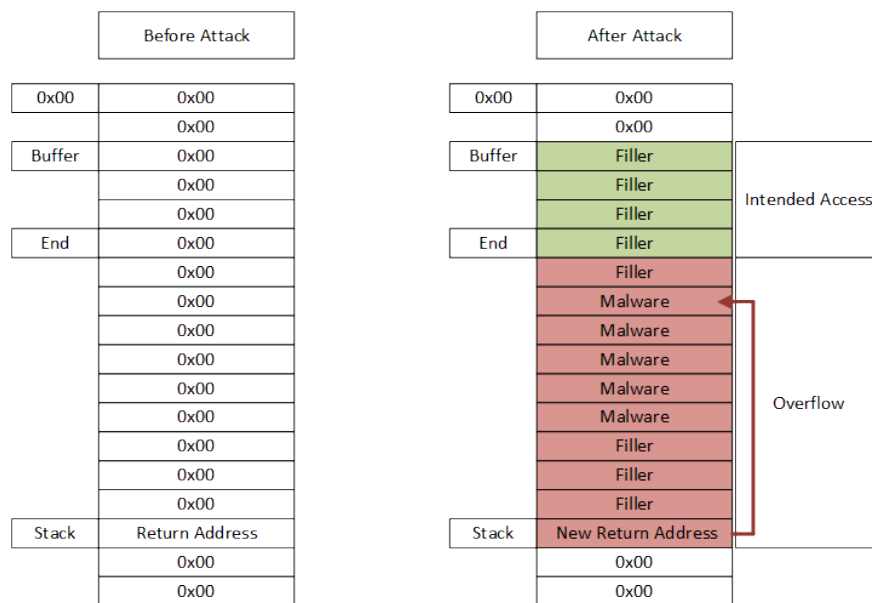


Figure 2.5: Memory Layout

RELATED WORK

Heterogeneous processing is not a new concept. When computer were in the early stages of development, there were many manufacturers leading to a naturally heterogeneous market. As computing systems matured, the benefits of homogeneous computing steered the market towards the state it is today.

Opcode Encryption

Opcode encryption is a technique to prevent malware execution. Conceptually, opcode encryption makes it more difficult for a malicious actor to design malware for a system. The instruction set of a system architecture is public knowledge but the opcode mapping is encrypted per process execution. This technique was initially presented by researchers at Columbia University [5] and University of New Mexico [2] in 2003. In these papers, systems used emulators to create an environment where opcode mappings could be encrypted before operation. The opcodes were then decrypted at program execution. Figure 3.1 shows how instructions were handled in an emulated system.

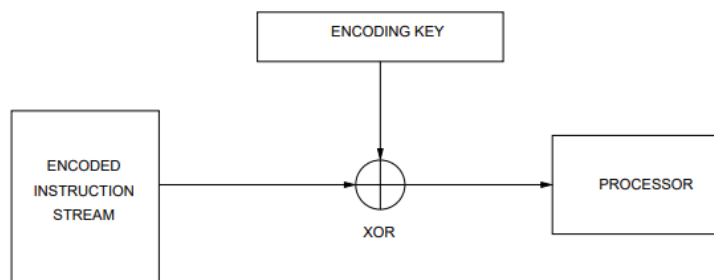


Figure 3.1: “Encoded instructions are decoded before being processed by the CPU” [5]

These systems were proven to be effective at defeating code injection attacks. Barrentes et al. even claims their system RISE “protects against all binary code injection attacks

regardless of the method by which the machine code is injected” [2]. These two systems were demonstrated on emulators resulting in large resource and speed overheads.

Opcode encrypted systems were revisited in 2013 in an attempt to progress past emulator implementation. Antonis et al. implemented a hardware based system on an Field Programmable Gate Array (FPGA) device. An FPGA is a chip consisting of logic blocks, memory, and some other systems. The logic blocks of an FPGA can be programmed to create hardware circuits after fabrication. This is significantly different than the standard Application Specific Integrated Circuit (ASIC) that most people are familiar with. A soft-core processor is a processor designed to be implemented on an FPGA. Implementing a soft-core on an FPGA allows the processor to change hardware configurations easily. Antonis et al. used an FPGA to implement a system they named ASIST (architectural support for instruction set randomization) [7]. The ASIST system used an open source soft-core called the Leon3 SPARC V8 with modifications to support opcode decryption. A modified Linux Kernel was used to take advantage of two new hardware registers which store encryption keys for user and kernel level processes. Figure 3.2 shows how applications decrypt instructions before executing them on the FPGA implemented processor. The ASIST system has significantly lower overhead than previous emulator implementations, but ASIST uses basic encryption methods. These basic methods could potentially be guessed or reverse engineered allowing malware to be injected.

In 2017, opcode encryption was investigated again by Sinha et al [11]. A system was created by the name of Polyglot. Polyglot used an FPGA device, the Leon3 soft-core, and a Linux kernel to create an injection attack resistant system. Polyglot used a much more sophisticated encryption system to mask the instructions. Polyglot traded more advanced encryption for a significant increase in resources needed. The base Leon3 used 13,986 Look-Up Tables (LUTs) inside the FPGA. Polyglot increased that value by 256 percent to a value of 49,724. A large portion of the LUT usage was the decryption circuitry meaning smaller

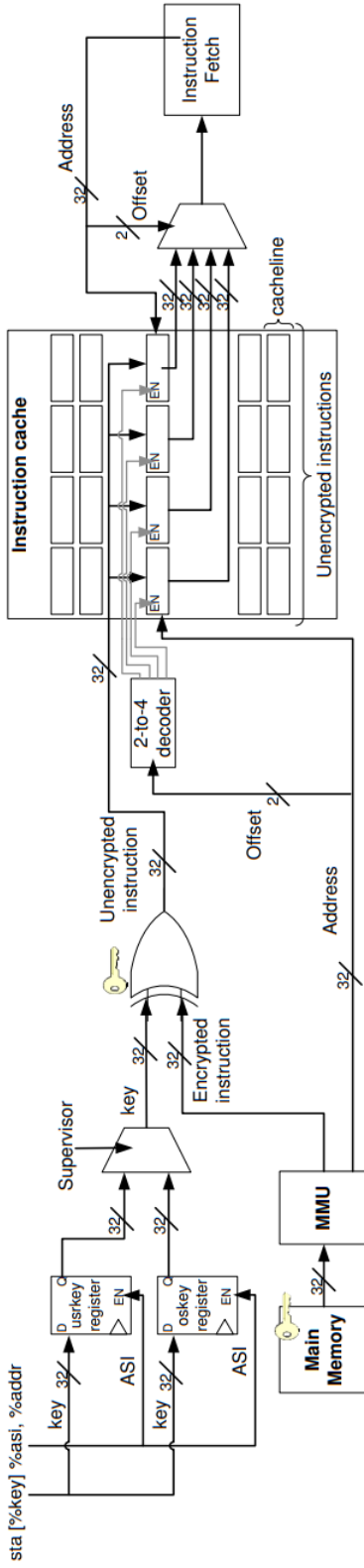


Figure 3.2: “ASIST Decryption Scheme” [7]

embedded processors would see a larger percentage of resource overhead.

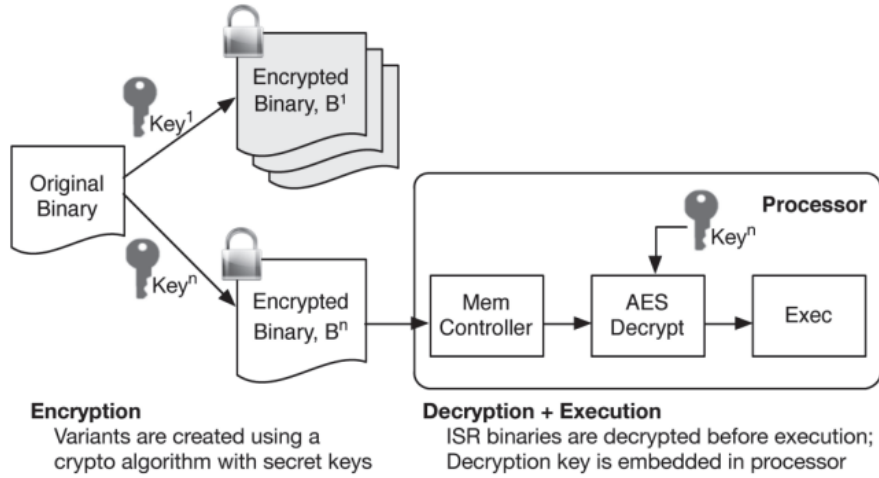


Figure 3.3: Polyglot Structure [11]

All of the above opcode encryption schemes share a fundamental flow. Programs are written with publicly known instructions. These programs are then encrypted using various methods and stored in memory. At run time, these encrypted instruction streams are loaded, decrypted and finally executed. The decryption process adds overhead to program execution either in the form of increased resources or decreased speed.

OUR APPROACH

Researchers at Montana State University have been investigating the use of redundant processors for various applications. The MSU team has used various soft-core architectures as they researched creating radiation resistant processors for aerospace applications. These processors were able to withstand single bit upsets when upsets were injected into the system. The upsets were detected by voting circuits inside the FPGA. If a processor was determined to be faulted, recovery methods were initiated. Recently the MicroBlaze soft-core by Xilinx has been the architecture used for these experiments. Unfortunately, the MicroBlaze is a proprietary system making opcode obfuscation or encryption impossible.

Obfuscation

In order to achieve the project goal of defeating injection attacks on an embedded system, a system was created with a layer of obfuscation between software and hardware. Adding this obfuscation layer and running redundant yet diverse subsystems allows injection attacks to be detected in real time. Obfuscation is accomplished by changing the instruction set to opcode linking in three functionally identical cores. Changing the instruction to opcode map means each core has a unique program binary. The instruction to opcode map of a core is built into the hardware of the core. Having obfuscated programs meant the three cores used are physically unique. Using an FPGA for this project allowed the changing of hardware to be a quick and efficient process, meaning that the cores could easily be changed to match any needed instruction to opcode mapping.

A previous research project at MSU has done experiments looking into opcode obfuscation. Previous experiments used a handwritten soft-core largely based on a core designed in MSU's digital logic design course. The core performed extremely well, being able to withstand an attack where unknown opcodes were injected across a serial port. This

experiment showed what was possible with opcode obfuscation but, as stated by a researcher, “This demonstrates a proof of concept, but falls short of most industry instruction sets” [3]. The core used did not have enough instructions to be used viably in a production environment. The core had only two processor data registers and an extremely limiting amount of memory, only 256 bytes. This memory was split between data memory, program memory, and the stack leading to a core with 96 bytes of program memory, 72 bytes of data memory, and 56 bytes for the stack. The core’s memory capabilities would need to be drastically increased for it to be a useful processor. The limiting factor of this core’s memory was the size of registers used in the processor. The processor used 8-bit registers and the small registers limited the addressing space available. Expanding the register size would allow the processor to access a much larger address space and perform more complex mathematical operations. A researcher stated that “a more robust compiler system will need to be implemented.” [3] A custom built assembler was used to create programs in the past. An example program can be seen in Figure 4.1. This rudimentary assembler searched for instruction names and replaced them with opcode values before exporting a VHDL file containing the program to be run. An assembler that supports features such as comments, preprocessor directives or even a C compiler would drastically increase the usability of an obfuscated core.

```
1  LDA_IMM -- Load A (Immediate Addressing)
2  x"AA"   -- Operand (Value)
3  STA_DIR -- Store A (Direct Addressing)
4  x"E0"   -- Operand (Memory Location)
5  LDA_IMM -- Load A (Immediate Addressing)
6  x"BB"   -- Operand (Value)
7  STA_DIR -- Store A (Direct Addressing)
8  x"E0"   -- Operand (Memory Location)
9  BRA     -- Branch Always
10 x"00"   -- Operand (Memory Location)
```

Figure 4.1: Assembly Program [3]

SYSTEM DESIGN

Processor

No embedded system is complete without a processing unit of some kind. The processor is tasked with fetching and executing operations quickly and in a certain order. This paper aimed to create a powerful enough processor for real world application. The processor should also be easily programmable and well documented. The processor design was based off of Texas Instrument's MSP430 microprocessor family. Specifically the MSP430FR2355 was the basis of the processor design due to the chip being easily accessible using TI's launchpad board.

The MSP430FR2355 is a sixteen-bit microprocessor with twenty-seven instructions. The MSP430FR2355 has twenty-bit registers and some instructions dedicated to performing operations using the full twenty bits. For simplicity, the soft-core processor designed does not implement any of the twenty-bit features. The MSP430FR2355 supports two source and four destination addressing modes for instructions. The twenty-seven base instructions can be separated into three distinct categories; Format I, Format II, and JMP instructions. Format I instructions take two operands, a source and a destination. Format I instructions are used for operations like adding, subtracting, logical operations, and comparing two values. Format II instructions take one destination operand. Format II instructions are used for bit wise rotations, byte swaps, and sign extensions. JMP instructions change the location in memory of program execution based on various conditions. The MSP430FR2355 has seven special registers dedicated to the processor operation. A Program Counter (PC) register tracks the current location in program memory of instructions being executed. A Stack Pointer (SP) is used to control and track the stack. A Status Register (SR) is used to monitor results flags from arithmetic operations and show the status of some other important functions. The SR is also used as a constant generator when an instruction requires certain constants

as the source operand. A second constant generator is also implemented, and is tied to the register named R3. The remaining twelve registers are general purpose and named R4-R15. An Instruction Register (IR) is used to keep track of the current instruction being executed. A Memory Address Register (MAR) is used to specify what data is retrieved from data memory. The IR and MAR are hidden from a user. A diagram of the MSP430 CPU can be seen in Figure 5.1.

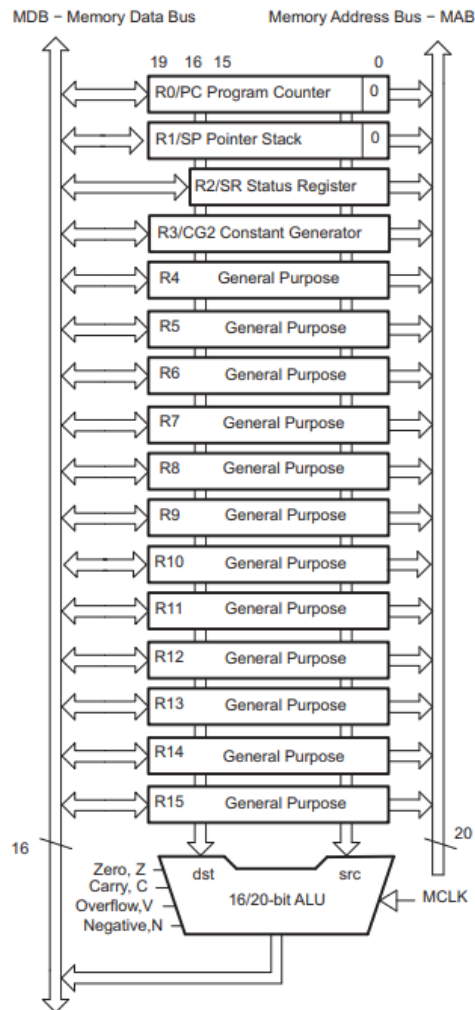


Figure 4-1. MSP430X CPU Block Diagram

Figure 5.1: MSP430FR2355 CPU Block Diagram [14]

Creating a VHDL soft-core version of the MSP430 CPU was mostly done using Simulink and the HDL Coder plugin. Simulink is a graphical software developed by Mathworks as a Matlab extension. Simulink is used for simulating a wide variety of systems. The HDL Coder plugin allows drawn diagrams to be translated into HDL code. The data paths were drawn first allowing information to be moved in between registers, memory, and the ALU. In the above diagram, these are shown as busses named MDB (Memory Data Bus) and MAB (Memory Address Bus). Not shown on the diagram are multiplexers allowing the busses to be only driven by a single source at a time. Included in the data path are the registers needed for operation. In Simulink, a register is implemented as a delay block of one clock cycle with a switch using feedback or the new value of the register. This can be seen in Figure 5.2.

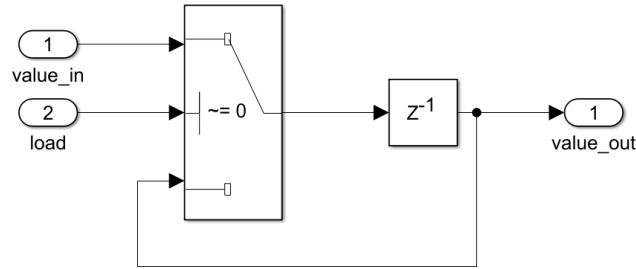


Figure 5.2: Base Register in Simulink

The Arithmetic Logic Unit (ALU) was created using Simulink. The ALU consists of combinational logic circuits built to execute a single operation. The output of each circuit is then passed to a multiplexer that determines the correct operation based on the IR. Also generated in the ALU are values for the Overflow (V), Negative (N), Zero (Z), and Carry (C) status bits stored in the SR. The multiplexers, registers, and ALU control signals are all managed by the control unit.

The control unit was a finite state machine that handles the execution of operations. One design goal of the control unit was to minimize the number of states as much as possible. This goal helped make the state diagram readable and allows a future researcher to easily understand the system’s functionality. States were reused based on the instruction format being executed, creating three distinct routes through the state machine. A fourth route allowed for interrupts to be handled. When the processor was started, a “Reset” interrupt was immediately serviced. In the interrupt, the PC was loaded with the address of the first instruction in the program to be executed. In Figure 5.3 the four routes through the control unit are highlighted and named.

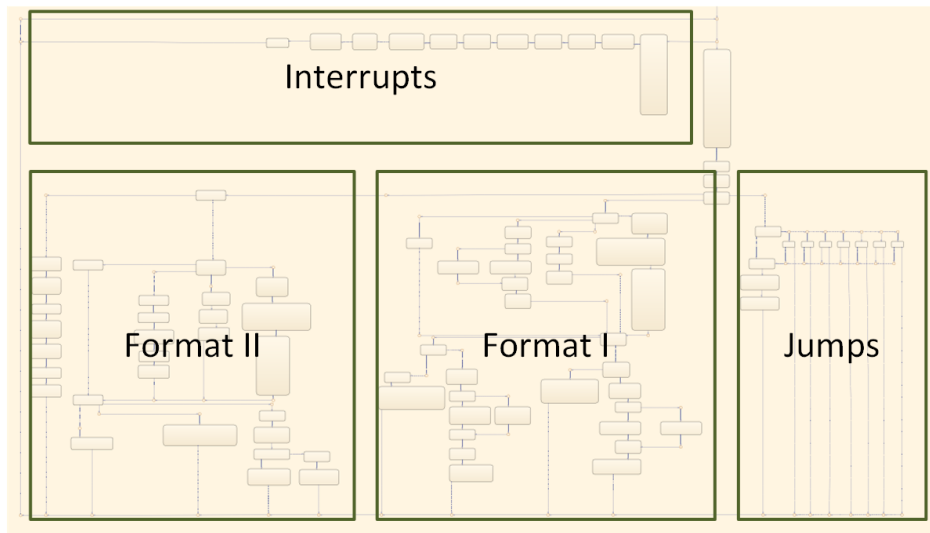


Figure 5.3: Routes Through the Control Unit

Memory

Computing power is directly tied to memory functionality. The size and speed of memory is often the constraint that limits computing capabilities. The MSP430FR2355 is no exception to this rule. The MSP430FR2355 has 32KB of read/write program memory, 4KB of read/write data memory, and 4KB dedicated to memory mapped peripherals registers.

	ACCESS	MSP430FR2355	MSP430FR2353
Memory (FRAM) Main: interrupt vectors and signatures Main: code memory	Read/Write (Optional Write Protect) ⁽¹⁾	32KB FFFFh to FF80h FFFFh to 8000h	16KB FFFFh to FF80h FFFFh to C000h
RAM	Read/Write	4KB 2FFFh to 2000h	2KB 27FFh to 2000h
Information memory (FRAM)	Read/Write ⁽²⁾	512 bytes 19FFh to 1800h	512 bytes 19FFh to 1800h
Driver library and FFT library (ROM)	Read only	20KB FAC00h to FFBFFh	20KB FAC00h to FFBFFh
Peripherals	Read/Write	4KB 0FFFh to 0020h	4KB 0FFFh to 0020h
Tiny RAM	Read/Write	26 bytes 001Fh to 0006h	26 bytes 001Fh to 0006h
Reserved ⁽³⁾	Read	6 bytes 0005h to 0000h	6 bytes 0005h to 0000h

Figure 5.4: MSP430FR2355 Memory Map [13]

This memory is accessed through the MAB and MDB as well as signals to designate reading versus writing and byte versus word access. Using the data path and control unit, instructions can be retrieved and executed from any location in memory, or data can be read and stored to any location. The memory map of the MSP430FR2355 is shown in Figure 5.4. The soft-core built for this project was required to match the memory map of the original processor. Matching the memory map allows code compiled in TI's tool chain to be directly loaded onto the soft-core without the need for remapping. In order to allow for obfuscation, the program and data memories are handwritten HDL as opposed to being built in Simulink. The program and data memory are dual port RAM blocks with one port dedicated to accessing data pointed to by MAB, and one port dedicated to address MAB+1. When the memory is accessed in word mode, the two ports are concatenated and the value is placed on the MDB. When in byte mode, the MAB port is concatenated with a blank byte and the data is placed onto the MDB. The HDL structure is largely based on a Quartus template for dual port RAM, allowing the memory to be synthesized into Block RAM instead of LUTs on an FPGA.

Peripherals

Peripherals allow microprocessors to perform a large array of functions, directly contributing to their versatility and widespread use. The peripherals of the MSP430 can be seen in Figure 5.5 from TI's documentation. The soft-core MSP does not implement all of the MSP peripherals. However, a select few were chosen to demonstrate the capabilities of an obfuscated system. The peripherals implemented allowed for system interrupts, general purpose input output, timer functions, and a serial link to be established. The peripherals implemented were designed using Simulink, and HDL was generated using the HDL coder plugin.

Input Output Design

The General Purpose Input Output (GPIO) peripheral is the fundamental way that processors interact with real world systems. On the MSP430 soft-core, a total of three ports with a width of sixteen bits were designed to match the embedded MSP430. The ports are accessed and controlled via registers that are memory mapped. Memory mapping the registers allowed the processor to read and write to the control registers as if they were memory locations. The GPIO ports mapped a single bit from the control registers to a single physical pin, allowing each pin to be individually configured with port direction and interrupt functionality. Registers that allow specific pins to be controlled by other peripherals were also created. These registers enabled the functionality that pins can be used for general purpose or dedicated to things such as serial communication.

Timer Design

Timers are one of the most useful peripherals in a microprocessor. Timers are hardware counters with the ability to specify input clocks and interrupt trigger settings. The MSP430 implements four separate timers for generating specific timing events. The timer peripheral

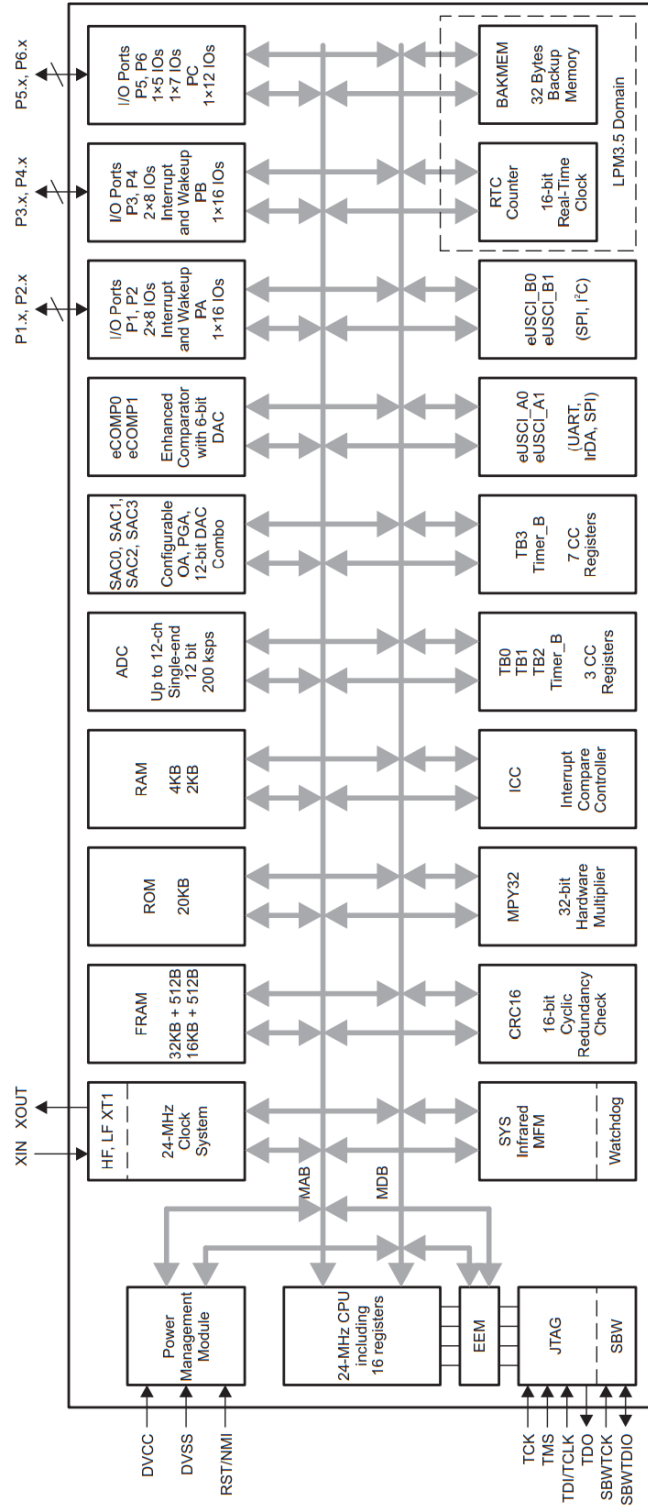


Figure 1-1. MSP430FR2355 Functional Block Diagram

Figure 5.5: MSP430FR2355 Block Diagram [13]

interfaces with the main processor using memory mapped registers, exactly like the GPIO module. Each timer module consists of a clock selector, two clock dividers, a counter, and capture compare logic. This can be seen in the diagram provided by TI, shown in Figure 5.6. The clock selector can choose between using the system main clock, typically 1MHz, an auxiliary clock, typically 32KHz, or an externally provided clock. The timer counter is a sixteen bit timer but can be configured to use eight, ten, twelve, or sixteen bits. The counter has four modes that control counting behavior; stop, up, continuous, up/down. In stop mode the counter is disabled and does not increase it's value. Up mode counts from zero to the value stored in a control register then restarts at zero. Continuous mode counts from zero to the max value then restarts counting at zero. The max value is set based on the number of counter bits used. Up/down mode counts from zero to the value stored in a control register, then counts back down to zero. Capture compare logic allows a processor to generate more specific timings using the compare mode, or to time events using the capture mode. The compare mode consists of registers which can be loaded by the processor with any sixteen bit value. When the timer count matches a compare register value, an interrupt flag is set allowing the processor to create very precise timings. The capture mode uses the same registers as compare mode, but the functionality is different. The timer module can be configured to store the current timer value based on a triggering signal. The triggering signal can be either an external or internal signal, and it can trigger on a rising or falling edge. After a trigger, the processor can read from the register that was updated and determine when the trigger occurred. The capture state diagram is shown in Figure 5.7. The timer module allows a processor to measure time accurately while performing other operations. The ability to interleave multiple tasks using timer modules is an invaluable resource required for any modern microprocessor.

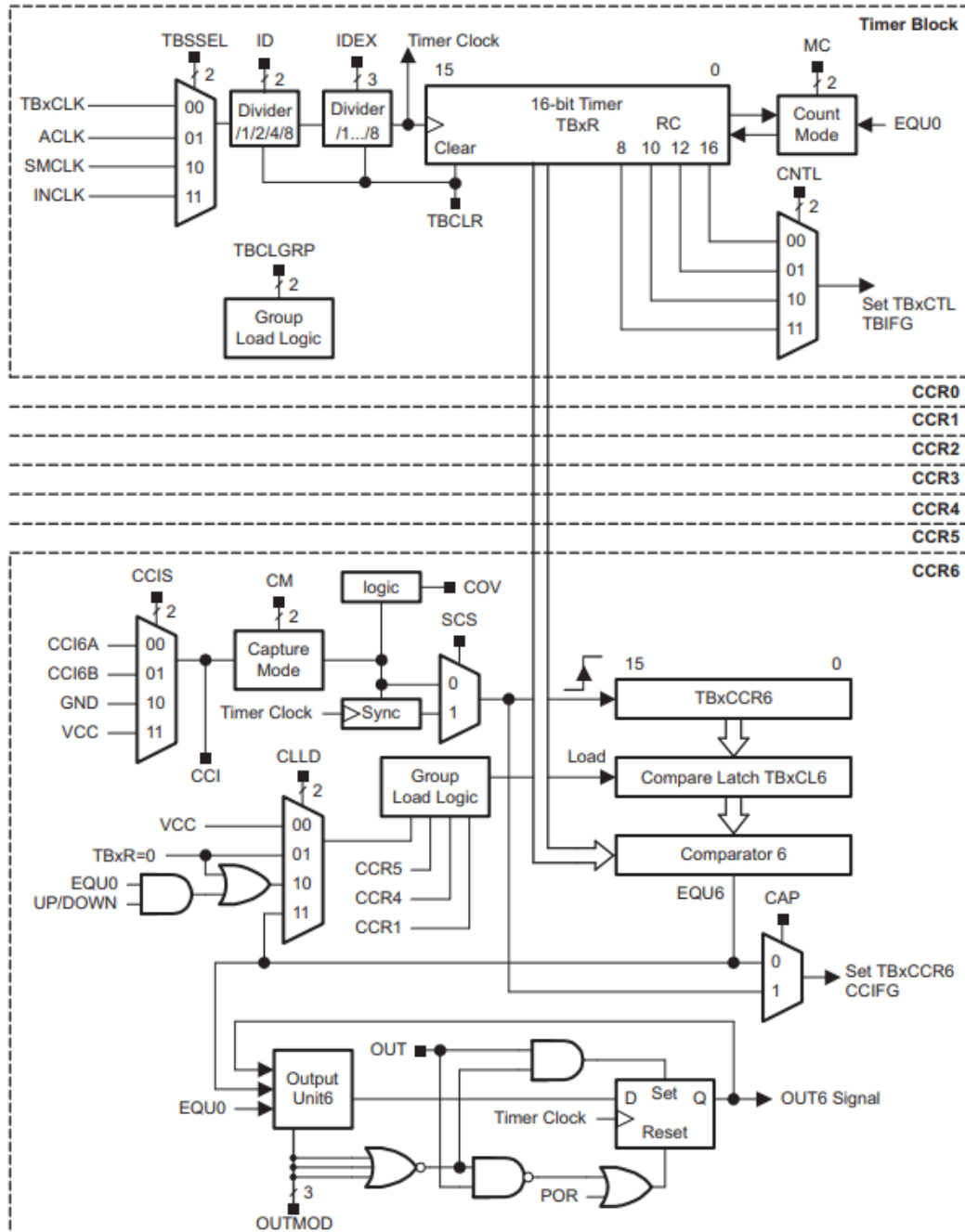


Figure 14-1. Timer_B Block Diagram

Figure 5.6: MSP430FR2355 Timer B diagram [14]

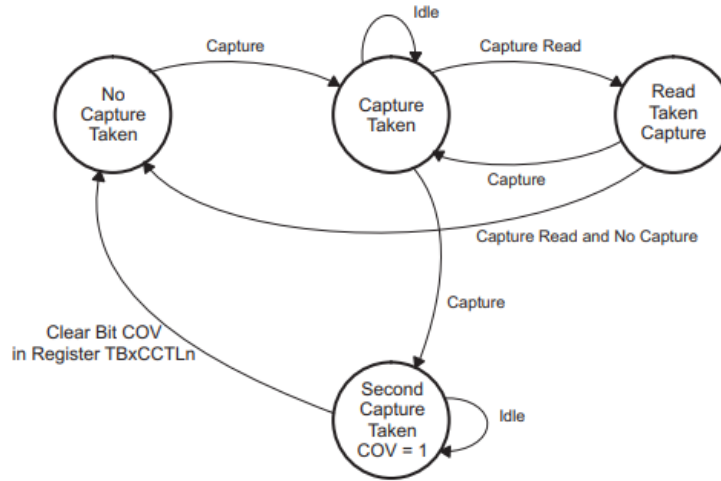


Figure 14-11. Capture Cycle

Figure 5.7: MSP430FR2355 Timer Capture[14]

Interrupt Controller Design

Interrupts are a crucial part of any microprocessor. Interrupts allow the processing unit to multi-task without having a separate processing thread. Figure 5.8 shows the block diagram of the Interrupt controller. Interrupts can have many sources. They are most commonly linked to peripherals such as the previously mentioned GPIO and timers. These peripherals throw interrupt flags based on certain conditions. Interrupt flags inform the processing unit that something important has occurred, and the current processing should be paused. Handling an interrupt flag is accomplished using an Interrupt Service Routine (ISR). ISRs are functions that perform needed tasks when an interrupt flag is thrown. Once an ISR has been handled and the corresponding interrupt flag has been cleared, the processor unit returns to the task it was processing before the interrupt.

The final peripheral designed for this soft-core was a serial communication link. The protocol chosen to be used was Universal Asynchronous Receiver-Transmitter (UART). UART was chosen due to its versatility and prevalence in electronics. The commercial MSP430 has a peripheral called an Enhanced Universal Serial Communication Interface (eUSCI). The eUSCI module uses the same registers with different configurations to handle various serial protocols such as Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and UART. The soft-core peripheral has the same registers as the commercial eUSCI, but the only mode implemented is UART. UART is the only implemented protocol because only a single protocol is needed to provide an attack vector. UART is also the only protocol capable of being easily connected to a desktop computer. The system has a

serial communication protocol which allows for external information to be inserted into the processor. The UART module is comprised of a baudrate generator, transmission state machine, and receive state machine. Figure 5.9 shows the eUSCI block diagram from TI.

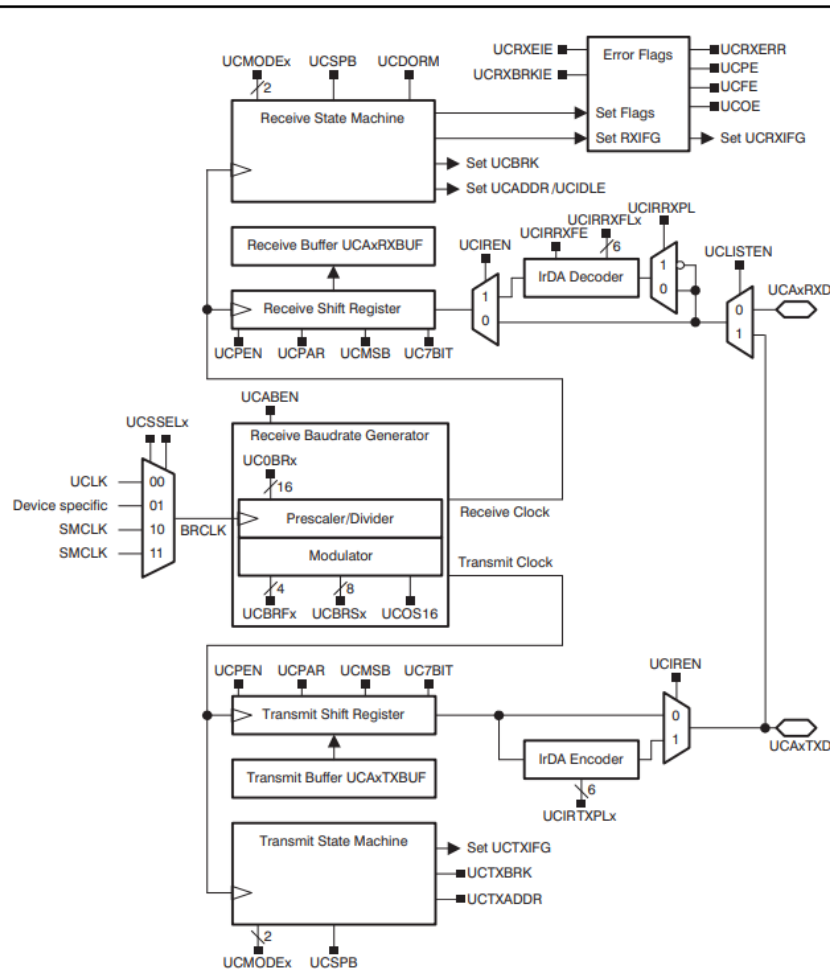


Figure 5.9: MSP430FR2355 eUSCI Module[14]

The baudrate generator consists of a clock multiplexer, clock divider, oversample divider, and two stages of clock modulation. The clock multiplexer allows a clock to be chosen from multiple sources. The chosen clock is called `BRCLK`. The divider is a programmable clock divider allowing any divisor from one to two-hundred and fifty five, and the oversample divider is a set divisor of sixteen. When not using oversample mode,

the oversample divider does nothing and BRCLK is divided by the normal divider. With oversample mode enabled, the normal divider generates Bit Clock sixteen (BITCLK16). The normal clock modulator extends the Bit Clock (BITCLK) by one cycle of either BRCLK or BITCLK16 based on oversampling mode and a configuration register that set the modulation pattern. The oversampling modulation extends BITCLK16 by one BRCLK cycle based on a configuration register. Using various combinations of dividers and modulation patterns allows the generation of many different baud rates from a broad range of source clocks. Figure 5.10 shows example divider and modulation settings to create baudrates from common clock frequencies.

BRCLK	Baud Rate	UCOS16	UCBRx	UCBRFx	UCBRSx
32768	1200	1	1	11	0x25
32768	2400	0	13	–	0xB6
32768	4800	0	6	–	0xEE
32768	9600	0	3	–	0x92
1000000	9600	1	6	8	0x20
1000000	19200	1	3	4	0x2
1000000	38400	1	1	10	0x0
1000000	57600	0	17	–	0x4A
1000000	115200	0	8	–	0xD6
4000000	9600	1	26	0	0xB6
4000000	19200	1	13	0	0x84
4000000	38400	1	6	8	0x20
4000000	57600	1	4	5	0x55
4000000	115200	1	2	2	0xBB
4000000	230400	0	17	–	0x4A
8000000	9600	1	52	1	0x49
8000000	19200	1	26	0	0xB6
8000000	38400	1	13	0	0x84
8000000	57600	1	8	10	0xF7
8000000	115200	1	4	5	0x55
8000000	230400	1	2	2	0xBB
8000000	460800	0	17	–	0x4A

Figure 5.10: MSP430FR2355 Baudrate Generator Settings[14]

The transmission state machine handles moving information from the memory mapped transmission register into the transmit shift register. Using BITCLK, the information is transmitted out of the shift register at the desired baud rate. The state machine also handles throwing flags to the interrupt controller. The reception state machine uses the BITCLK and a voting mechanism to determine if each bit is a logical zero or one. The bit is then moved into a shift register to create space for the next serial bit. After receiving a full byte of information, the data is transferred to a memory mapped register, and interrupt flags are thrown informing the processor that data is ready to be pulled from the buffer. These features allow the processor to transmit and receive serial data reducing the number of input/output pins needed.

Tool Chain

Every processor needs a tool chain to convert human readable code into binary information. Texas Instruments uses a program called Code Composer Studio (CCS) for their MSP430 family devices. CCS is a plugin built on top of the Eclipse framework. CCS supports project management, C code compilation, live debugging, and much more. A project in CCS is shown in Figure 5.11. Due to having the same memory map and instruction set, the soft-core can directly execute code developed for the MSP430FR2355. The code has to be loaded into the soft-core program memory. Loading is accomplished through the HDL files of the core. CCS allows for custom post build steps. For this project, one of those steps is to execute a custom built python script. The python script opens output files from CCS compilation and converts them into HDL memory files containing the binaries of the developed C code. The python script also handles obfuscation of the program. Using CCS output files, instructions are separated from data. In python, the instruction codes are modified. The python obfuscation scripts also output HDL package files specifying the instruction to opcode mapping used.

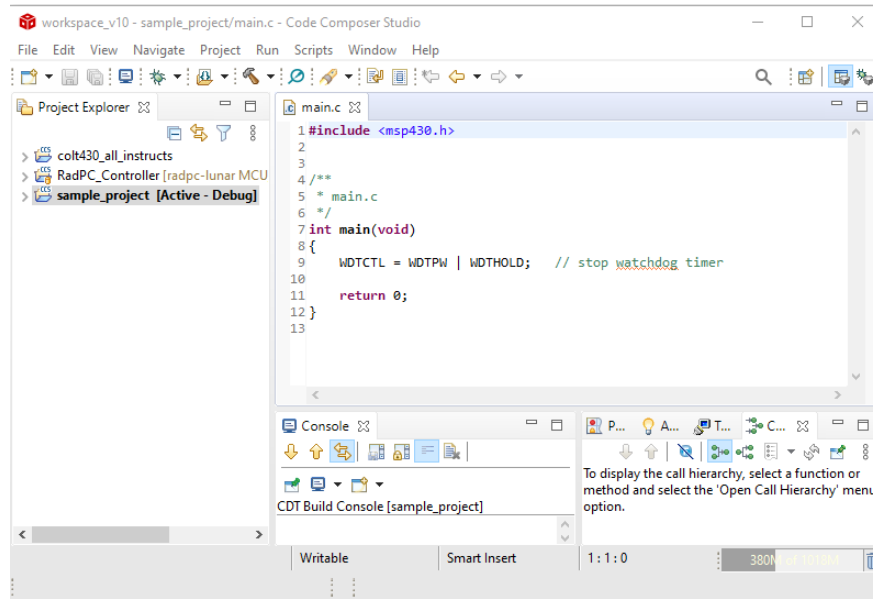


Figure 5.11: CCS Project

Recovery System

Some systems can safely be turned off or shut down when attacked by malicious code, but many systems must be active at all times. This type of system must be able to handle a cyber attack. The soft-core developed for this project was designed to quickly recover in the event of an attack. The two steps involved in the recovery process are detection and reset. Detection of malicious code being injected into the system is done through hardware obfuscation. The system as a whole is running three soft-cores, each with a unique opcode to instruction mapping. If the system is executing benign code written by the system designer, the instructions will be the same across all three cores, and the opcodes will be different at any point in time. If a malicious section of code is inserted into the design, it will be inserted into all three cores. When the system begins executing the malicious section, all the opcodes are now the same across the cores. A combinational logic circuit is constantly analyzing the opcodes from each core. When the opcodes are all the same, the combinational circuit

immediately pauses the processing section of the cores and triggers the recovery method. Once the processors are paused, the program memory is refreshed. This refresh is done by reloading the active program memory with a backup read only version containing the original code for each core. Once the active program memory has been fully refreshed, the processing core is reset to move the program counter away from malicious code location. The program memory is refreshed as the initial program is considered a safe state. Data memory is constantly changing during normal operation, so the data memory is not refreshed by the recovery system. The program will set important data memory fields back to the initial values when the processor is reset. Malware injected into data memory is not guaranteed to be removed by the recovery system, but the program counter redirection is neutralized.

SYSTEM VERIFICATION

No design is complete without verification. Components were individually tested as they were implemented. After significant progress was made on the system, a more complete test would be developed and demonstrated to the research sponsor. These comprehensive tests helped to ensure sections of the project were done and to direct what section would be worked on next.

Processor Verification

The processing section of the system was the first developed and the first verified. Initial verification of the processor was done in simulation. Simulation was used because external interface devices are peripherals. The peripherals had not been developed at the time of initial processor verification. To ensure full processor functionality, all twenty-seven supported instructions and the various addressing modes were tested. This was accomplished by constructing an assembly program that used all the instructions. Part of this program can be seen in Figure 6.1. This test bed program was first developed and run on the commercial MSP430. After monitoring the execution on the commercial processor, the binary code was obfuscated and simulated on the soft-cores. Viewing the signals over time in simulation allowed both the obfuscation code and processor execution to be verified. Initially the processor did not function as expected. Using simulations the processor was further developed until the expected behaviour was achieved. Once processor execution matched the expected execution, system development moved forward. A comparison of the registers before and after execution of the test program can be seen in Figure 6.2. The commercial processor registers are shown on the top and the obfuscated core on the bottom.

```

msp_ops_main.asm x
1;-----
2; MSP430 Assembler Code Template for use with TI Code Compo
3; Walker Ward
4; 09/16/2021
5;
6;-----
7; .cdecls C,LIST,"msp430.h" ; Include devic
8;-----
9;-----
10; .def RESET ; Export progra
11; ; make it known
12;-----
13; .text ; Assemble into
14; .retain ; Override ELF
15; ; and retain cu
16; .retainrefs ; And retain an
17; ; references to
18;-----
19;-----
20 RESET mov.w # _STACK_END,SP ; Initialize st
21 StopWDT mov.w #WDTW|WDTCTL ; Stop watchdog
22
23
24;-----
25; Main loop here
26;-----
27 init:
28 mov.w #02000h, R4 ; use immediate to
29 mov.w R4, R5 ; init R5 with val
30 mov.w #Var1, R6 ; init R6 with addr
31
32 bic.b #BIT1, &P4DIR ; set P4.1
33 bis.b #BIT1, &P4REN ; EN pull u
34 bis.b #BIT1, &P4OUT ; pull-up r
35 bic.b #11111111b, &P4IFG ; clear int
36 bis.b #BIT1, &P4IES ; set as fa
37 bis.b #BIT1, &P4IE ; assert lo
38
39 nop
40 bis.w #GIE, SR ; assert gl
41 nop
42 eint
43 nop
44 dint
45 nop ; clear global
46
47 bic.b #LOCKLPM5, &PM5CTL0 ; disable D
61
62; data manipulation instructions
63 manipulation:
64
65; add.w-----
66
67 mov.w #02000h, R4
68 add.w R4, Const2
69 add.w &02000h, R5
70 mov.w #02000h, R4
71 mov.w #Var1, R6
72 add.w 0(R4), 4(R6)
73 add.w #Con1, Var1
74 add.w @R5+, R6
75
76; addc.w-----
77
78 addc.w R4, R5
79 addc.w R5, Const2
80 addc.w &02000h, R5
81 mov.w #02000h, R4
82 mov.w #Var1, R6
83 addc.w 0(R4), 4(R6)
84 addc.w #Con1, Var1
85 addc.w @R5+, R6
86
87; addc.w-----
88
89 setc
90 adc.w R4, Const2
91 adc.w Const2
92 adc.w &02000h, R4
93 mov.w #02000h, Var1
94 mov.w #02000h, R4
95 setc
96 adc.w Var1
97 adc.w 4(R4)
98
99; sub.w-----
100
101 sub.w R4, R5
102 sub.w R4, Const2
103 sub.w &02000h, R5
104 mov.w #02000h, R4
105 mov.w #Var1, R6
106 sub.w 0(R4), 4(R6)
107 sub.w #Con1, Var1
108 sub.w @R5+, R6
109

```

Figure 6.1: Assembly Test Program

BEFORE		AFTER	
Core Registers		Core Registers	
PC	0x00803E	PC	0x00803E
SP	0x003000	SP	0x003000
SR	0x0000	SR	0x0000
R3	0x000000	R3	0x000000
R4	0x002000	R4	0x0084E4
R5	0x002000	R5	0x00000C
R6	0x002004	R6	0x000000
R7	0x000000	R7	0x000001
R8	0x000000	R8	0x000080
R9	0x000000	R9	0x00ACED
R10	0x000000	R10	0x002000
R11	0x000000	R11	0x00BEEF
R12	0x000000	R12	0x000000
R13	0x000000	R13	0x000000
R14	0x000000	R14	0x000000
R15	0x000000	R15	0x000000

Core Registers	
pc_out1[15:0]	=803E
sp_out1[15:0]	=3000
sr_out1[15:0]	=0000
reg3_out1[15:0]	=0000
reg4_out1[15:0]	=2000
reg5_out1[15:0]	=2000
reg6_out1[15:0]	=2004
reg7_out1[15:0]	=0000
reg8_out1[15:0]	=0000
reg9_out1[15:0]	=0000
reg10_out1[15:0]	=0000
reg11_out1[15:0]	=0000
reg12_out1[15:0]	=0000
reg13_out1[15:0]	=0000
reg14_out1[15:0]	=0000
reg15_out1[15:0]	=0000

Core Registers	
pc_out1[15:0]	=803E
sp_out1[15:0]	=3000
sr_out1[15:0]	=0000
reg3_out1[15:0]	=0000
reg4_out1[15:0]	=84E4
reg5_out1[15:0]	=000C
reg6_out1[15:0]	=0000
reg7_out1[15:0]	=0001
reg8_out1[15:0]	=0080
reg9_out1[15:0]	=ACED
reg10_out1[15:0]	=2000
reg11_out1[15:0]	=BEEF
reg12_out1[15:0]	=0000
reg13_out1[15:0]	=0000
reg14_out1[15:0]	=0000
reg15_out1[15:0]	=0000

Figure 6.2: Matching Registers

Peripheral Verification

The MSP430 has multiple hardware peripherals that are controlled by the processor. These peripherals allow the system to accomplish extremely powerful tasks and interface with the external world. Various system demonstrations were designed and created in order to verify peripheral functionality.

GPIO Verification

The first peripheral designed and tested was the GPIO interface. This peripheral was used to connect the processor with sensors and actuators to accomplish external tasks. Verification of this system was accomplished by driving servo motors using Pulse Width Modulated (PWM) control signals. PWM signals for servo motors specify a twenty millisecond period with the signal being logical high for one to two millisecond and low for the rest of the period. Non-continuous servos typically have one-hundred and eighty degrees of rotation with a one millisecond high cycle corresponding to position zero and a two millisecond high cycle corresponding to a position at one-hundred and eighty degrees. The PWM timing was accomplished in code using busy loops. This technique to create timing is not extremely precise but the timing is often good enough for a physical system such as motors. Another downside of busy loops is the processor spends time in between signal transitions doing computations to “measure” time. The servo driving program was written on the commercial MSP430 and tested with motors before being converted into obfuscated VDHL memory. Once the VDHL memory and an initial digital IO design were created, the peripheral was tested in simulation. The design was repeatedly simulated and modified until the simulations matched the expected behavior of the system. The simulation of a PWM signal can be seen in Figure 6.3. Only after successful simulations was the system placed on FPGA hardware for testing. Once on hardware, external logic analyzers and physical

motors, see Figure 6.4, were used to determine proper system functionality.

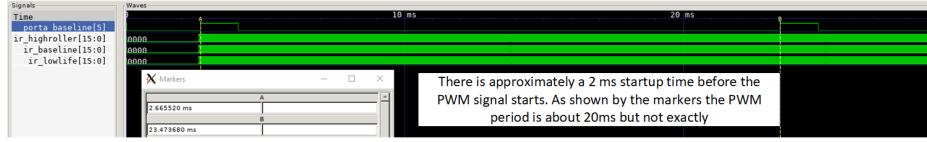


Figure 6.3: Busy Loop PWM Simulation

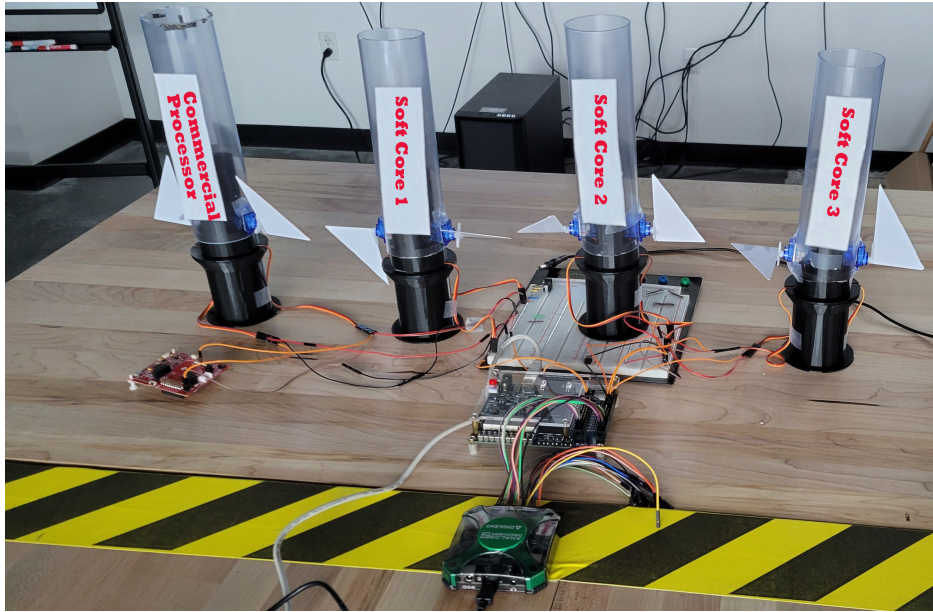


Figure 6.4: Hardware Setup for Servo Test

Figure 6.5 shows an external logic analyzer measuring two different pulse lengths corresponding to two different positions of the motors. The lower portion of the figure shows the current opcode being executed on each core. It can clearly be seen the opcodes are obfuscated on each core.

Timer and Interrupt Verification

The next implemented peripheral was a timer module. The timer module allows processors to precisely measure time and execute code based on specific timings. One

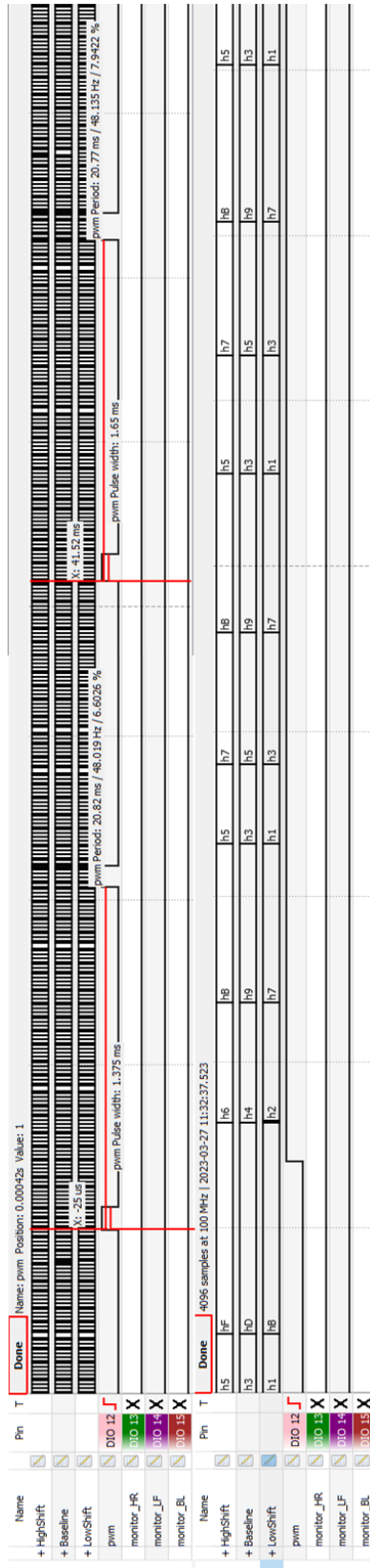


Figure 6.5: Busy Loop PWM Measurement

extremely common application of timer modules is creating PWM signals. Using timers for creating PWM signals allows processors to generate more precise timings than busy loops. Timer peripherals also offload the task of measuring time from the processor, allowing it to perform useful computations in between signal transitions. Timer modules track specific timings. Interrupts inform the processor when to interact with the timer. Interrupts are special subroutines that can be triggered by various subsystems, and they take priority over main thread execution. When an interrupt is triggered, the processor pauses the main thread and handles the interrupt before proceeding with the main thread. A timer peripheral and an interrupt peripheral were created based off of diagrams from the MSP430 documentation. A program to control servo motors using timers and interrupts was developed and verified on the commercial MSP430 processor. Figure 6.6 shows a comparison of busy loop and timer PWM generation code. The code was then obfuscated and translated to VHDL program

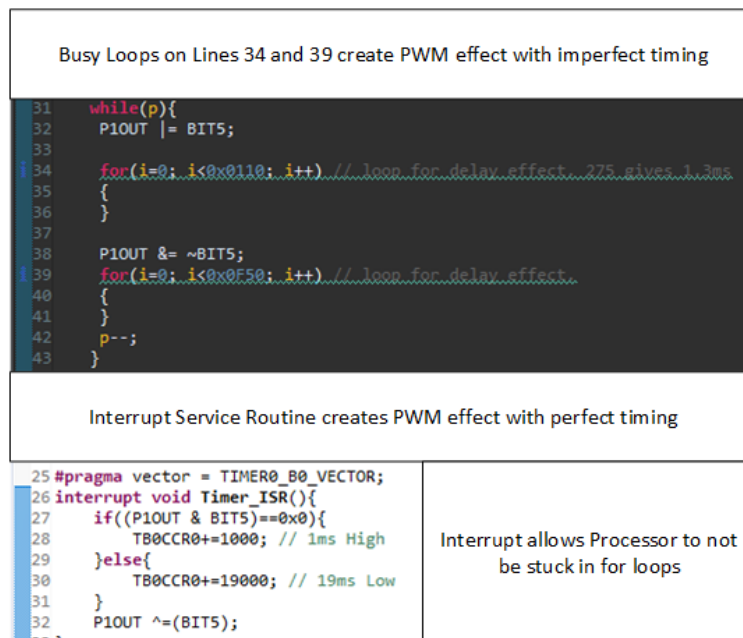


Figure 6.6: Code Comparison

memory. Using a combination of simulation tools, external logic analyzers, and a hardware

setup with servo motors, the timers and interrupts were confirmed as functioning. Figure 6.7 shows the system in simulation and Figure 6.8 shows a measurement on a logic analyzer.

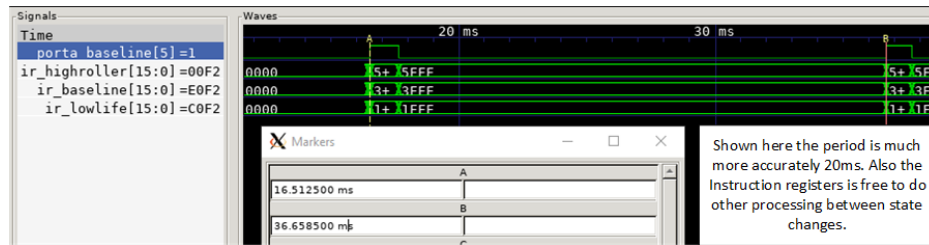


Figure 6.7: Timer PWM Simulation

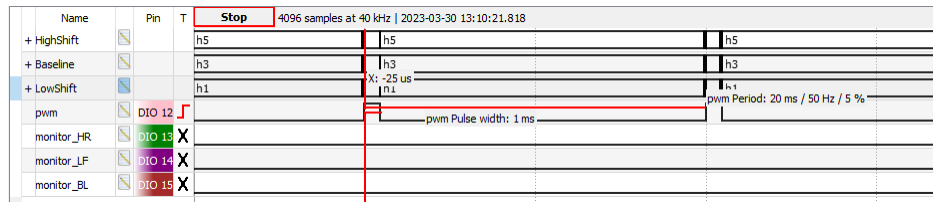


Figure 6.8: Timer PWM Measurement

Serial Link Verification

Processors use serial communication links to interface with other advanced devices. Through the use of serial links large amounts of data can be transferred using minimal amounts of hardware. The serial protocol implemented for this system was a UART module. The UART protocol uses no clock; Instead each member of the bus agrees upon a baud (data) rate when the bus is designed. Using the predetermined baud rate, devices either transmit or receive data on either a TX or RX line respectively. Testing of the UART module was accomplished through simulation and using an “echo” program. The “echo” program was a simple program where a character is received through UART, and the system sends the same character back across UART. This test verifies both transmit and receive capabilities.

The baud rate used for this test was 115.2 Kbps. Figure 6.9 shows the value 0xAA being received, loaded into the transmit buffer, and being transmitted back in simulation. Figure 6.10 shows the same test occurring on hardware monitored by an external logic analyzer. The logic analyzer transmitted the information to the system and observed the response.



Figure 6.9: UART Echo Simulation

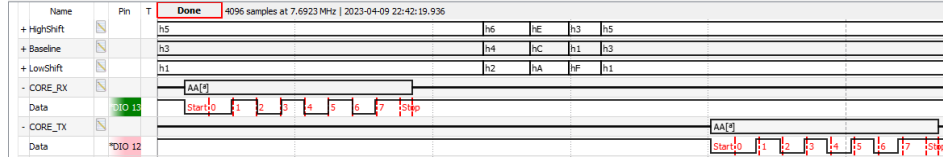


Figure 6.10: UART Echo Measurement

Recovery Verification

The recovery subsystem allows the whole system to function despite malware attacks. In order to test the recovery system, a malware attack was imitated. This was accomplished by manually changing a single program instruction across all three tiles. When the tiles loaded the faulty opcode, the recovery system detected the matching opcodes as an attack and began the recovery sequence. This can be seen in Figure 6.11 showing the recovery in simulation. The same test in hardware was monitored with an external logic analyzer. The fault detection flag was seen to properly show when the faulty opcode was loaded, and the memory refresh procedure executed properly, as seen in Figure 6.12.

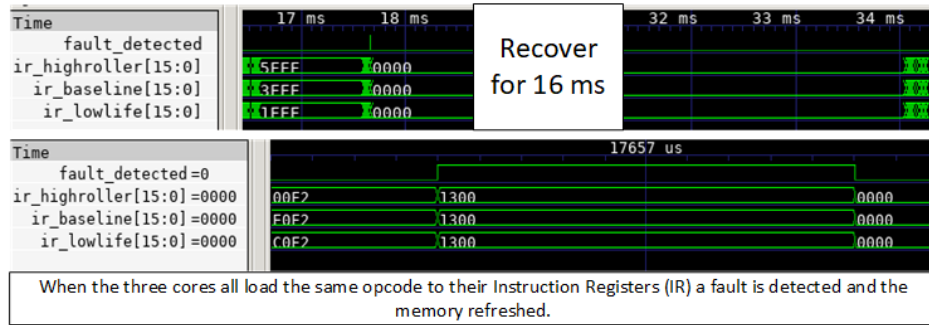


Figure 6.11: Simulation of Recovery Procedure

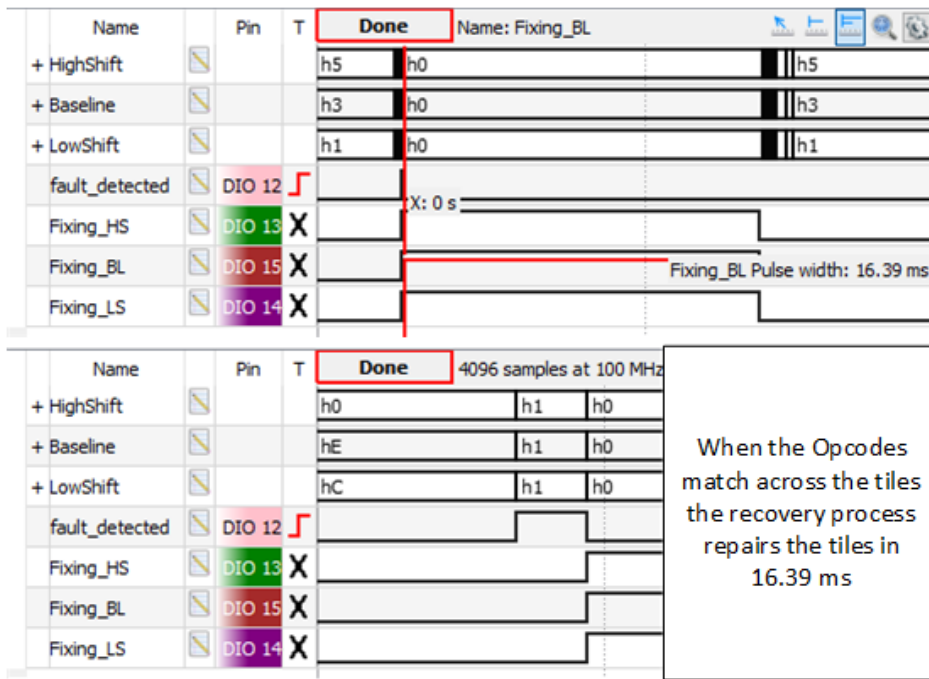


Figure 6.12: Recovery Procedure on FPGA

Full System Verification

A full system verification test was performed in order to confirm the obfuscation techniques implemented could defeat a malware attack. The test demonstrated that the processor was capable of performing a typical task for a microprocessor while also being resilient to a malware injection attack.

System Application

Nearly any program could be designed with a malware vulnerability to be attacked, but in order to properly demonstrate the system, a non-trivial program was designed. The program created needed to utilize the various peripherals and have an attack vector to be exploited. A closed loop feedback system utilized both the input and output peripherals. Using a motor utilized the timer module to create a Frequency Modulated signal. Finally changing an input parameter in real time allowed for use of the serial link. A physical setup was created using a 3d printed model rocket, a rotary encoder, a stepper motor, and a UART port. The rotary encoder would serve to input the angle at which the rocket was suspended. The UART port would allow different angular set points to be selected. Using the selected set point and the current angle of the rocket, the processor would calculate the speed and direction in which the stepper motor should be driven. Using a timer to create a frequency modulated signal, the motor would be driven at the calculated speed closing the loop back to the rotary encoder. A block diagram of the system is shown in Figure 6.13

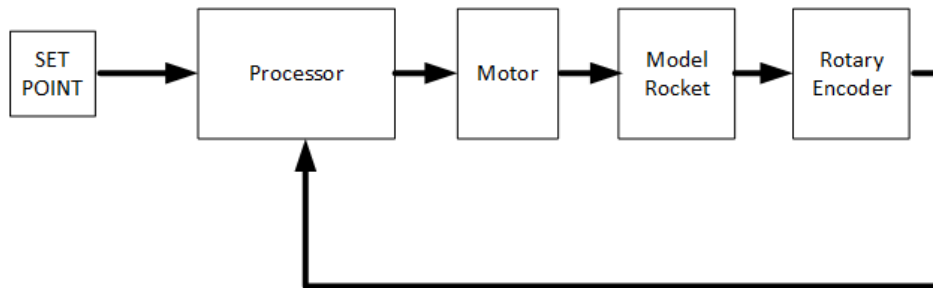


Figure 6.13: Demo Block Diagram

Attack Vector and Method

The program described above was intentionally left with a vulnerable attack vector. The serial link received data from an external computer. This data was received one byte at a time and the byte is transferred to a buffer in data memory. The program determined

the angular set point based on the first byte stored in the data memory buffer. The primary vulnerability stemmed from improper indexing into the buffer. The buffer index incremented anytime a character is received. The index returns to zero inside the main subroutine. This worked well if the user interacts with the system in the intended way by sending a single character. A malicious user could stream many bytes and overflow the buffer. If enough malicious bytes were transmitted, the overflowed buffer would overwrite values on the stack. The stack stored important values such as the return address of the last function called. An attack on the stack could inject arbitrary malicious code and change the return address to execute the injected code. The attack was delivered by a python script sending a malicious payload over the serial link to the system. This attack vector and method were developed using the commercial MSP430. Once the attack had been refined enough to properly disable the commercial processor, the program was implemented on the soft-core system.

Full System Results

Identical physical setups were used to test the vulnerable commercial microprocessor and the obfuscated soft-core system. The setups are shown in Figure 6.14. The processors performed as expected under normal operating circumstances. Both processors were able to read the angle from a rotary encoder and create a frequency modulated signal to drive stepper motors. The motors' speed and direction adjusted the model rocket to hold an angle specified over the serial link. When attacked, the commercial processor executed the maliciously injected payload. The payload disabled the rocket feedback and turned on an LED. When the obfuscated system was attacked, the recovery system was triggered, and the processor recovered before continuing normal operation. A slight wobble by the rocket and a return to the default set point were the only external signs that an attack had occurred. Figure 6.15 shows the instruction registers loading a malicious opcode after a long stream of serial character transmissions. Once the malicious opcode was loaded, the recovery process

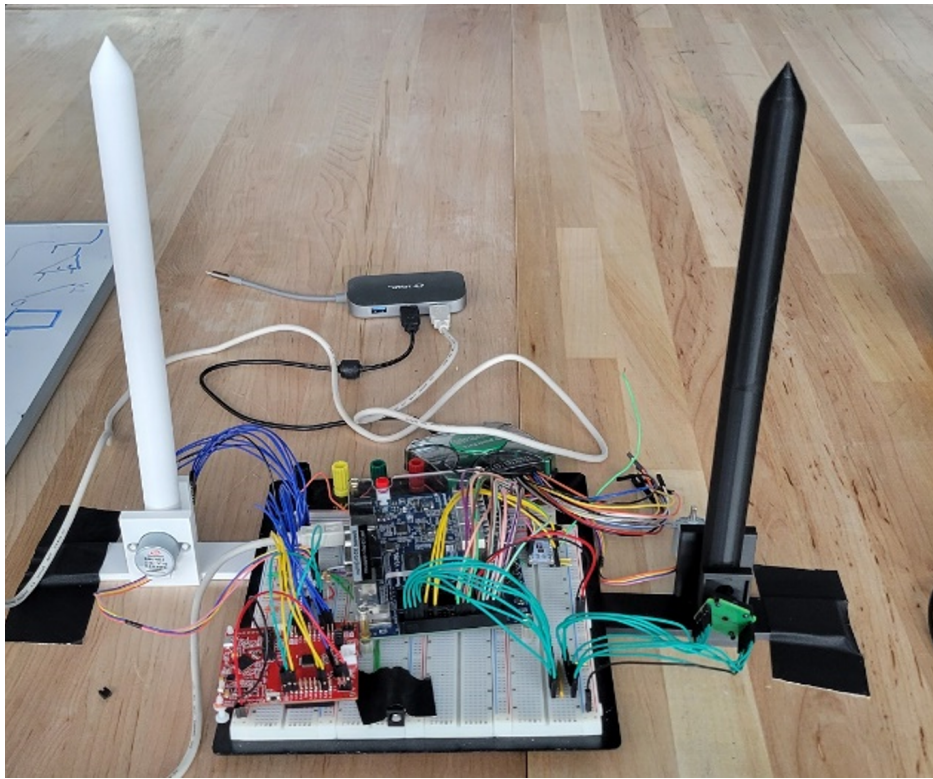


Figure 6.14: Demo Physical Setups

began and the system repaired itself before continuing normal operation.

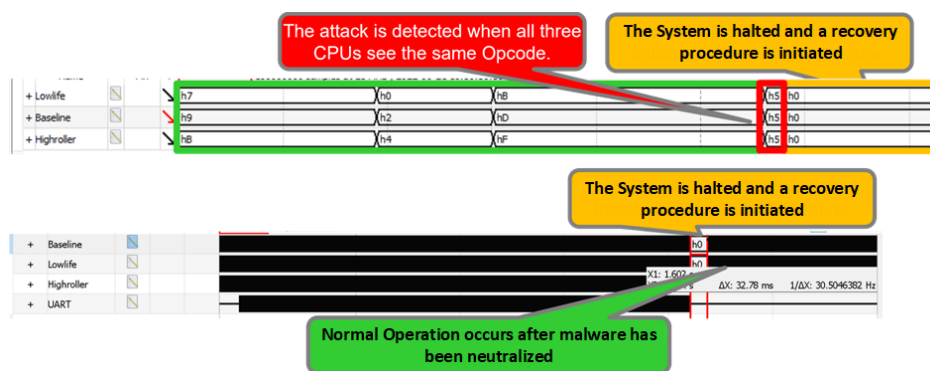


Figure 6.15: Attack on Hardware

Performance Evaluation

The system gains security via redundancy and obfuscation. There are some sacrifices made to gain these benefits. Triplicating the cores will inherently increase the usage of FPGA resources. Figure 6.16 shows the usage of FPGA resources with a single processing core.

Logic utilization (in ALMs)	3,850 / 18,480 (21 %)
Total registers	5246
Total pins	62 / 224 (28 %)
Total virtual pins	0
Total block memory bits	720,896 / 3,153,920 (23 %)

Figure 6.16: Single Core Resource Usage

Resource usage of the redundant system is shown in Figure 6.17. The LUT usage increased by 193% from 3,850 elements to 11,293 elements. The BRAM usage increased by 190% from 90 kilobytes to 262 kilobytes. There is no speed overhead for using redundant cores. The cores differ at a hardware level in order to handle opcode obfuscation. This technique allows the opcodes to be executed without any form of decryption needed, negating any slowdown from decryption.

Logic utilization (in ALMs)	11,293 / 18,480 (61 %)
Total registers	12775
Total pins	62 / 224 (28 %)
Total virtual pins	0
Total block memory bits	2,097,152 / 3,153,920 (66 %)

Figure 6.17: Redundant Cores Resource Usage

FUTURE WORK

This malware resistant architecture has many applications in the current state. Future work could allow the architecture to be more applicable. Increasing the processing power through larger registers, faster clocks, and more memory would be extremely beneficial. Implementing attack prevention techniques would also benefit the security of the system.

Partial Reconfiguration

Operating on an FPGA has many benefits. One of the benefits of newer FPGA is a feature called Partial Reconfiguration. This feature is the ability to change the configuration of a segment inside the FPGA while the rest of the segments remain operating. This feature has many applications, and for this specific project provides a better recovery method. The recovery method implemented here stored a read only copy of the original program in FPGA resources. When an attack was detected the active program was refreshed with the read only version. In a future implementation the backup program memory should be stored as a partial reconfiguration block outside of the FPGA. When an attack is detected the active program memories would be refreshed by partial reconfiguration. Using this technique the FPGA resource utilization and the recovery time would both be reduced.

Processor Shift

The current processor architecture is based on Texas Instruments' MSP430 microprocessor architecture. This architecture uses 16 bit registers and is sophisticated enough to be applied to many embedded system design problems. A more powerful architecture used for embedded system development is the 32 bit architecture used to create ARM processors. The concepts of opcode obfuscation and recovery through attack could be expanded to a 32

bit architecture in the future. Expansion to a larger and more standard architecture would increase the usability and applicability of a heterogeneous system.

Attack Prevention Techniques

The main focus of this paper was to create a system that could detect and recover from injected malware attacks. The security of this system operates on the assumption that once it is attacked the attack is nullified. One method to further increase the system security would be to implement attack prevention techniques often found in more advanced processors.

ASLR

A commonly implemented protection measure is a technique called Address Space Layout Randomization(ASLR). ASLR places important data storage structures at random locations in memory. This makes attacking the system more difficult because an injected return address is probably not the same as the randomly assigned location of the stack. When implemented on more advanced processors, this technique is quite effective due to a couple factors. One such factor is the modular nature of modern processors and operating systems. Programs are stored in external memories. When a program is needed, the executable is loaded into fast access RAM and assigned a portion of that RAM. ASLR takes advantage of this loading and assigns the program to a random location in the RAM. This run time loading style is not used on embedded systems; the programs are loaded on internal chip memories at the time of processor deployment. This would present some issues with ASLR. ASLR also takes advantage of the massive amounts of extra memory to reduce the probability that the injected return address does not match the randomly assigned stack location. Embedded systems inherently have less memory. Consequently, the probability of an attack still being successful despite ALSR increases. A future system may determine the benefits of ASLR

outweigh the cost to develop the system despite the shortcomings created by operating on an embedded system.

Stackguard

Desktop processors almost always implement an injection prevention measure called stackguard. Stack guard places an additional field on the stack. This field exists at the edge of the stack and acts as a stack overwrite protection measure. When a write attempt is made on the stackguard field, an error is thrown. The error can then be handled in a separate thread of execution. This is rarely implemented on embedded systems due to their single threaded nature. A single compromised thread is unable to handle an error caused by the thread itself. Stackguard could be implemented on a future system where error detection triggers the recovery system.

Non-executable Stack

Another technique to prevent malware attacks like the one used in this paper is a non-executable stack. Non-executable stacks throw an error anytime the program counter enters memory space dedicated to the stack. This technique could also be expanded to throw an error anytime the program counter exits memory dedicated to program memory. While this technique is effective at preventing attacks, it limits some forms of development and can be circumnavigated by more sophisticated attacks.

Recovery Subroutines

The recovery method used in this paper was to revert to initial conditions and reset the processor. While this method works for many systems, there are systems that cannot simply be reset in response to an attack. This is where recovery subroutines could be utilized. Recovery subroutines would be a function stored in a section of read only memory. These

routines would be written by developers and would be triggered whenever an attack was detected. Using these routines, developers could customize their system's response to an attack. Using this recovery method, attacks could be detected, handled, and defeated in a more robust manner.

CONCLUSION

Embedded systems need protection from cyber attacks. The use of embedded systems is constantly increasing and so are the available attack vectors. The homogeneous nature of microprocessors makes writing malware for an embedded systems relatively simple. The system outlined in this paper gives the benefits of a secure system while maintaining the usability of standard microprocessors. This system does require more resources than an insecure system, but any security has some form of design trade off. This system is more resource efficient than Polyglot, 193% resource increase instead of 256%, while circumnavigating the speed loss from decrypting opcodes during operation. Opcode obfuscation and hardware diversification are extremely powerful techniques to increase the security of embedded systems.

REFERENCES CITED

- [1] Michael Barr. *Programming embedded systems : with C and GNU development tools*. O'Reilly, Beijing, 2nd ed. edition, 2006.
- [2] Elana Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003.
- [3] Trevor Gahl. Triplicated instruction set randomization in parallel heterogeneous soft-core processors. Master's thesis, Montana State University, April 2019.
- [4] Danielle Hoverman. 6 common types of cyber attacks, 2017.
- [5] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003.
- [6] D. Kleidermacher and M. Kleidermacher. *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Elsevier Science, 2012.
- [7] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. Asist: Architectural support for instruction set randomization. In *"Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security"*, CCS '13, page 981–992, New York, NY, USA, 2013. Association for Computing Machinery.
- [8] Jennifer Read. *The Five Biggest MCU Suppliers Accounted for 82% of 2021 Sales*. EMS Now, June 2022.
- [9] Linton Salmon. System security integrated through hardware and firmware (ssith) proposers day overview, 2017.
- [10] Brian Santo. *25 Microchips That Shook the World*. IEEE Spectrum, 2009.
- [11] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28, 2017.
- [12] Murugiah Souppaya and Karen Scarfone. *Guide to Malware Incident Prevention and Handling for Desktops and Laptops*. NIST, July 2013. Rev. 1.
- [13] Texas Instruments. *MSP430FR235x, MSP430FR215x Mixed-Signal Microcontrollers*, December 2019. Rev. D.
- [14] Texas Instruments. *MSP430FR4xx and MSP430FR2xx family User's Guide*, March 2019. Rev. I.