12-2019

Built-In Return-Oriented Programs in Embedded Systems and Deep Learning for Hardware Trojan Detection

Nathanael R. Weidler
Utah State University

BUILT-IN RETURN-ORIENTED PROGRAMS IN EMBEDDED SYSTEMS AND DEEP LEARNING FOR HARDWARE TROJAN DETECTION

by

Nathanael R. Weidler

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

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Ryan Gerdes, Ph.D. Major Professor	Koushik Chakraborty, Ph.D. Committee Member
Sanghamitra Roy, Ph.D. Committee Member	Rose Hu, Ph.D. Committee Member
Curtis Dyreson, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

Topics on Security : Return-Oriented Programming and Hardware Trojans

by

Nathanael R. Weidler, Doctor of Philosophy

Utah State University, 2019

Major Professor: Ryan Gerdes, Ph.D.
Department: Electrical and Computer Engineering

The massive proliferation of integrated circuits brings with it an increased number of bad actors seeking to exploit those circuits for personal gain or other nefarious reasons. This is not surprising as integrated circuits are integrated into every aspect of human society. Humans have their personal lives, money, business secrets, and national security issues all controlled by integrated circuits. This dissertation explores several aspects of hardware security including the identification and prevention of integrated circuit vulnerabilities. The contents of this dissertation include three papers dealing with this topic. The first paper discuss the inclusion of a read-only memory on a microcontroller that comes factory loaded with peripheral drivers to aid in rapid development and to safe program memory space. The contents of this read-only memory include a Turing-complete gadget set that can be used by an attacker to erase and reprogram the Flash memory of the device. This is where the program memory resides. The final two papers branch into the field of hardware trojan detection and prevention. The first of the papers on hardware trojans discusses the premier method of preventing hardware trojan insertion through split manufacturing. It is shown that this method does not work on highly redundant circuits such as cryptographic ciphers. The second of the papers on hardware trojans includes a novel method to detect them using current state of the art deep learning models. Procedures that add generally to the field

of hardware trojan research are also included. These procedures detail a process by which hardware trojan data can be created at a scale previously unseen. A dataset that is four orders of magnitude larger than a current data set is provided to the public.

(137 pages)

PUBLIC ABSTRACT

Topics on Security : Return-Oriented Programming and Hardware Trojans

Nathanael R. Weidler

Microcontrollers and integrated circuits in general have become ubiquitous in the world today. All aspects of our lives depend on them from driving to work, to calling our friends, to checking our bank account balance. People who would do harm to individuals, corporations and nation states are aware of this and for that reason they seek to find or create and exploit vulnerabilities in integrated circuits. This dissertation contains three papers dealing with these types of vulnerabilities. The first paper talks about a vulnerability that was found on a microcontroller, which is a type of integrated circuit. The final two papers deal with hardware trojans. Hardware trojans are purposely added to the design of an integrated circuit in secret so that the manufacturer doesn't know about it. They are used to damage the integrated circuit, leak confidential information, or in other ways alter the circuit. Hardware trojans are a major concern for anyone using integrated circuits because an attacker can alter a circuit in almost any way if they are successful in inserting one. A known method to prevent hardware trojan insertion is discussed and a type of circuit for which this method does not work is revealed. The discussion of hardware trojans is concluded with a new way to detect them before the integrated circuit is manufactured. Modern deep learning models are used to detect the portions of the hardware trojan called triggers that activate them.

To my wife
Jenni
Hopefully you will see dividends for years to come.

ACKNOWLEDGMENTS

I would like to thank Dr. Gerdes for all of his help and support over the years. Even when he took a new position at Virginia Tech he agreed to remain my major professor and to work with me through this laborious process. He has taught me so much about hardware security and I am grateful for that. He has been patient in teaching me how to be a good researcher and how to submit papers for publication. We have spent countless hours together discussing research topics on various video chat platforms. Thank you for the time and guidance you have given me over the years. I truly appreciate your insights, friendship, and help.

I would like to thank my employer, the Space Dynamics Laboratory and my supervisors there. They have allowed me to work at three-quarters time during the five years it has taken for me to earn this degree and write this dissertation. This was not always easy, but they were always understanding and allowed me the space I needed to complete my school work.

Thank you to each member of my committee who have fit me into their busy schedules and helped me succeed. Thank you Dr. Chakrabory, Dr. Roy, Dr. Hu, and Dr. Dyreson.

Lastly, I would like to thank my wife and children who put up with seeing less of me over the past several years. The burdens of running our household and helping our children often fell to my wife. Thank you for your patience and understanding. My children often ask me when we wake up in the morning if I am going to work or to work on my Ph.D. when we wake up in the morning. I will be excited to tell them that I have completed my Ph.D. and that we will have a lot more time to spend together.

Nathanael R. Weidler

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ACRONYMS

AES	advanced encryption standard
ARM	advanced RISC machine
ASIC	application-specific integrated circuit
DES	data encryption standard
GAN	generative adversarial network
GBC	gradient boosting classifier
GCN	graph convolutional network
GPIO	general purpose input output
GRU	gated recurrent unit
HDL	hardware descriptive language
IC	integrated circuit
IoT	internet of things
LR	link register
LSB	least significant bit
MMU	memory management unit
MPU	memory protection unit
PC	program counter
RISC	reduced instruction set computing
RNN	recurrent neural network
ROM	read-only memory
ROP	return-oriented programming
ReLU	rectified linear unit
SPN	substitution-permutation network
TEE	trusted execution environment
UART	universal asynchronous receiver/transmitter
VHDL	very high speed integrated circuit hardware description language

CHAPTER 1

INTRODUCTION

This dissertation covers two research thrusts in the field of hardware security and consists of three papers. Both research areas deal with securing an integrated circuit (IC) from a would-be attacker. In the first part of the dissertation (the first paper) a vulnerability on a Tiva TM4C123GH6PM (Tiva C) utilizing a Cortex-M4F microprocessor is discovered and exploited in order to make it known to the community [1]. This vulnerability is a factory loaded code base in a read only memory (ROM) that contains a Turing-complete gadget set. It is important for exploits to be reported to the community so that the vulnerabilities can be prevented in the future or fixed. However, if they cannon be fixed, but are known then they may be able to be detected by an end user before critical systems or data are exploited.

The second half of the dissertation (the final two papers) addresses issues with hardware trojans. A prominent paper is explored that attempts to frustrate hardware trojan implantation [2]. A class of circuits is identified for which the methods presented in that paper would not work. Then in the last paper, a novel method to detect hardware trojans is presented.

The first paper, found in Chapter 2 was published in the Journal of Sustainable Computing: Informatics and Systems [3]. In this paper the Tiva C and a particular vulnerability it has is discussed. Specifically, it is revealed that a seemingly helpful read-only memory is in fact a security vulnerability. This ROM contains several libraries factory-loaded to ease the development process and to save space in program memory. Examples of libraries loaded on the ROM are peripheral libraries such as a universal asynchronous receiver/transmitter (UART), general purpose input/output (GPIO), and an Ethernet controller. However, it is shown that gadgets, or small fragments of executable code, can be found in these libraries and used for nefarious purposes. This paper was an expansion to another paper written by the author of this dissertations and others that was first published in the proceedings of

the 14th IEEE International Conference on Embedded Software and Systems (IEEE ICESS 2017) [4]. This paper concentrates on the fact that often energy-efficient devices lack sufficient security assurances for mission-critial applications. Too often security is sacrificed for energy-efficiency and ease of use. This is a bigger problem as the world trends towards tiny devices and the internet of things (IoT).

The process is demonstrated by which a stack overflow attack can be performed on a Tiva C making use of gadgets found in the ROM to carry out a return oriented programming (ROP) exploit. The paper in Chapter 2 shows the procedure to perform an ROP exploit in order to erase and then write to the Flash memory, where the program data is stored. This means that the original program can be deleted and replaced by a new one. Alternatively, portions of a program can be erased and re-written, if the attacker should choose to do so.

This brings to light a security flaw in the production of the Tiva C. One that is open for attackers to make use of, to either change or completely reprogram the device. Engineers must be vigilant to not create security vulnerabilities in the name of ease of use. The intention of the factory loaded ROM was to aid in quick development, but it has other security consequences.

This first paper of this dissertation found in Chapter 2 also finds a Turing-complete gadget set in the ROM. The measure for Turing-completeness was taken from [5]. In that paper, a Turing-complete gadget set is defined as one containing the following capabilities:

- The ability to exchange register values.
- The ability to load a value from the stack into a register.
- The ability to implement conditional branches.
- The ability to increment or decrement the value contained in a register.
- The ability to clear flags before comparisons.
- The ability to copy the stack pointer into another register.
- The ability to load a value from an address into a register.

- The ability to store a value from a register into memory.
- The ability to add or subtract two values.
- The ability to perform logical operations on two values. A full list of logical operations would include `NOT`, `AND`, `OR`, and `XOR`, however only a total of two are needed to be considered Turing-complete. Either `AND` or `OR` in addition to either `NOT` or `XOR`. If one logical operation is present from the following two all others may be derived.
- The ability to perform a conditional branch.

Although all of this functionality is present in Chapter 2, there are not necessarily individual gadgets for each of the above list. For example, this chapter does not contain a separate gadget to clear the flags before comparisons; instead wherever the flags must be cleared or set, that ability is built into the gadget itself. As the gadget set in Chapter 2 adheres to the requisites described by [5] it can be said that it is Turing-complete.

In addition to describing a Turing-complete gadget set, Chapter 2 includes an experimental results appendix in which the results of executing the gadgets described are displayed. This is a contribution because it proves beyond the theoretical that all of these gadgets are executable and reachable by an ROP exploit.

The second paper can be found in Chapter 3. This is the first paper touching on the topic of hardware trojans. This paper has been submitted to the Journal of Hardware and Systems Security.

Hardware trojans are modifications made to a design at one of four stages during the development cycle of an IC: the specification phase, the design phase, the fabrication phase and the assembly phase [6]. Hardware trojans can be used to disable circuits, or leak sensitive information from the circuit. Not only are governments concerned about the existence of these malicious circuits, private businesses, and individuals could be at risk as well. One major issue facing the industry of IC manufacturing is balancing trust with the cost to create a foundry with the ability to create the latest circuit technology.

Nations have more trust in a foundry located within it's borders than one outside of it's borders [7]. If a nation is worried about a foreign adversary, they surely might not trust the foundries located within the borders of their adversary. Within it's own borders, a nation will have greater ability to perform background checks and keep tabs on the people working in the foundries. This presents a problem for many highly advanced nations which do not have foundries with the latest technology domestically located [7]. It is much less expensive to use a foundry as needed then to build, maintain, and upgrade a foundry with the latest technology. The cost of building a new foundry with the latest technology is in the billions of dollars [8].

Chapter 3 examines a very useful method which prevents the insertion of hardware trojans by utilizing split manufacturing. This process is laid out in [2]. The process is to divide a circuit into two layers, a trusted and an untrusted layer: the untrusted layer contains all the gates and some interconnects, and the trusted layer only contains interconnects, or metal wires. The untrusted layer can be manufactured at an untrusted foundry without any worry because without access to the trusted layer, the manufacturer would not be able to deduce the layout of the entire circuit, and therefore it would be much more difficult to insert a hardware trojan. The trusted layer can be manufactured at a trusted foundry, but it does not need to have the capability to manufacture the latest technology as it is only interconnects. When manufactured in this way hardware trojans are very difficult to be inserted successfully into the design, unless the circuit happens to be a highly redundant circuit as described in Chapter 3. In this case the methods described in [2] are not as secure as they purport to be. This is demonstrated on the two main types of cryptographic ciphers: the data encryption standard (DES) which uses a Fiestal structure, and the advanced encryption standard (AES) which utilized a substitution-permutation network (SPN).

The third and final paper contained in this dissertation can be found in Chapter 4. Chapter 4 contains the work as submitted to the workshop on Attacks and Solutions in Hardware Security (ASHES). This paper explores a novel method to detect hardware trojans. This method will be useful in detecting them in third party intellectual property (IP). The

novel method is to use deep learning to identify hardware trojan triggers. If a hardware trojan is trigger-based and the trigger can be found, so can the rest of the trojan. The two parts of a trigger-based hardware trojan are the trigger and the payload [6]. The payload lays dormant until it is activated by the trigger, when a pre-determined combination of values appear on the trigger inputs. Basic triggers consist of `AND` and `NOT` gates placed in a way that the output will be `1` only on the rare occasion that the pre-determined value is present on the trigger inputs.

Deep learning algorithms have shown promise in several fields, including image recognition [9–12]. ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual image recognition competition running since 2010. The ISCVRC has brought awareness to the need for image recognition and has therefore encouraged developments in the field. It includes a public dataset and an annual competition. There are varying categories and levels of image recognition by computer algorithms [13]. A team of researchers working on the ILSVRC challenge and using deep learning algorithms was able to surpass human-level image recognition on the 1000-class ImageNet dataset [14].

As deep learning has shown such promise in other fields, it is worthwhile to experiment and expand the applications of deep learning. Capter 4 is is a paper which plows new ground in the application of state of the art methods to a new problem. One of the difficulties associated with this was creating the vast amounts of training and verification data required for the various trials. One major contribution of this work was a new methodology to create data for the field of hardware trojan research. A large dataset was also made publicly available. This data is not only useful for deep learning research, but can be used in other hardware trojan related research as well.

1.1 Authorship

Each of the papers contained in this dissertation have the author of this dissertation as the first author, however there were others who contributed as well, as can be seen by the citation given or explicitly mentioned here. In an effort to be transparent the contributions of others will be noted here. All other contributions can be assumed to be the original work

of the author of this dissertation.

In Chapter 2, the second gadget set referenced in Section 2.3.4 and seen in Listing 2.6 was identified by other authors. The gadgets in section Section 2.4, containing the Turing-complete gadget set were also identified by other authors. The majority of the stack diagrams were created by others. Much of the discussion on the Turing-complete gadget set was written by another author. The remainder of the work, including Appendix A 2.6, the procedure to erase and reprogram the Flash memory, and the discussion on sustainability was the work of the author of this dissertation.

Chapter 3 was the original work of the author of this dissertation with Dr. Ryan Gerdes and Dr. Thidapat Chantem mentioned on the author list. They were both readers and acted in a large part as consultants during the research process.

For Chapter 4 Nikhil Muralidhar and Dr. Ryan Gerdes were on the author list. Nikhil wrote the abstract, Subsections 4.2.2 and 4.5.3. He also wrote Section 4.6 compiling the results we created together. These contributions by Nikhil include the background on deep learning architectures and applications, and the additional graphical and recurrent models examined. The genesis of Chapter 4 was the original work of the author of this dissertation who brought expertise in the field of hardware trojans and worked with the feed-forward neural network. The author of this dissertation is responsible for creating the process to create feature vectors for deep learning models, as well as identifying the various forms of data described such as the circuit adjacency matrix and the inverse node fanin forms. The author of this dissertation created the dataset of trigger-inserted circuit adjacency matrices made publicly available. Nikhil was the expert on deep learning and aided in expanding the findings to multiple deep learning models. Dr. Gerdes acted as a consultant and advisor throughout the research process.

REFERENCES

[1] *Tiva TM4C123GH6PM Microcontroller*, Texas Instruments Incorporated, 2014, rev. E.

[2] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara, "Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 495–510.

[3] N. R. Weidler, D. Brown, S. A. Mitchell, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-oriented programming on a resource constrained device," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 244–256, 2019.

[4] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-oriented programming on a cortex-m processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 823–832.

[5] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 2012, pp. 7–7.

[6] M. T. C. Wang, *Introduction to Hardware Security and Trust*. 233 Spring Street, New York, NY 10013: Springer, 2012.

[7] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable asics," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 759–778.

[8] C. M. Christensen, S. King, M. Verlinden, and W. Yang, "The new economics of semiconductor manufacturing," *iEEE SpEctrum*, vol. 45, no. 5, pp. 24–29, 2008.

[9] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*. IEEE, 2012, pp. 3642–3649.

[10] Y. Sun, Y. Chen, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, 2014, pp. 1988–1996.

[11] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701–1708.

[12] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, 2013, pp. 1058–1066.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

CHAPTER 2

Return-Oriented Programming on a Resource Constrained Device

ABSTRACT Microcontrollers are found in many everyday devices and will only become more prevalent as the Internet of Things (IoT) and other low power devices gain momentum. As such, it is increasingly important that they are reasonably resilient to known exploitation techniques. Modern enterprise-grade systems with virtually unlimited resources have many options when it comes to implementing state of the art intrusion prevention and detection solutions. These solutions are costly in terms of energy, execution time, circuit board area, and — of course — money. Sustainable IoT devices and power-constrained embedded systems cannot afford such costs and are forced to make suboptimal security trade-offs. One such trade-off is the design of architectures which prevent execution of injected shell code, yet have allowed Return Oriented Programming (ROP) to emerge as a more reliable way to execute malicious code following attacks. ROP is a method used to take over the execution of a program by causing the return address of a function to be modified through an exploit vector, then returning to small segments of otherwise innocuous code located in executable memory one after the other to carry out the attacker's aims. It will be shown that the Tiva TM4C123GH6PM microcontroller, which utilizes a Cortex-M4F processor, can be fully controlled with this technique. Sufficient code is pre-loaded into a ROM on Tiva microcontrollers to erase and rewrite the flash memory where the program resides. Then, that same ROM is searched for a Turing-complete gadget set which would allow for arbitrary execution. This allows an attacker to re-purpose the microcontroller, altering the original functionality to their own malicious ends. Our results show that advanced exploitation techniques are still effective against embedded systems which prioritize energy-efficiency and that more research needs to be focused on finding the right balance of security for devices with a small energy footprint.

2.1 Introduction

Energy-efficient microcontrollers are becoming increasingly important as the Internet of Things and the Internet of Everything become a real part of life. Everything from wearables to remote sensors become more feasible when they consume less power making them less expensive to operate. In the case of battery-operated devices they will last longer to allow them to fulfill their mission for longer periods of time. The Tiva TM4C123GH6PM (Tiva C) is advertised as a microntroller capable of supporting applications such as low power and hand-held smart devices [1].

The Tiva C makes use of an Advanced Reduced Instruction Set Computing (RISC) Machine (ARM) Cortex-M4F microprocessor. ARM advertises this as a low-power, low cost solution. The Cortex-M4 processor has been developed for a broad range of embedded markets including automotive control systems, building automation, connective clothing, the energy grid, wearables, medical instrumentation, household appliances, and space applications just to name a few [2]. ARM claims that tens of billions of ARM Cortex-M processors have already been shipped [2]. With these devices widely deployed in safety-critical products, their security is vital.

The Tiva C makes use of the Cortex-M4F microprocessor [1], adding memory and the ability to interact with peripherals such as a Universal Asynchronous Receiver/Transmitter (UART), General-Purpose Input/Output (GPIO), and Ethernet controller [3]. It is important for the manufacturers of microprocessors and microcontrollers to have security in mind when they design these devices. Attackers would be able to wreak havoc on individuals and industries if they were able to exploit vulnerabilities in these systems for their own pernicious purposes. We have discovered that a Cortex-M4F microprocessor on a Tiva TM4C123GH6PM microcontroller can be forced to execute arbitrary code by means of Return-Oriented Programming (ROP).

2.1.1 Return-Oriented Programming

The concept of return-oriented programming (ROP) was first introduced for x86 processors in 2007 by Hovav Shacham [4]. The motivation for this work was the memory policy

of "write xor execute." This policy specified that system memory regions could either be written to or executable, but not both. Typically, the region of memory which contains the stack and the heap would be set to be writable but not executable [5] and, during process execution, regions of memory containing code would be executable but no longer writable. This prevents code execution following basic buffer overflow attacks such as those outlined in the classic work, "Smashing the Stack for Fun and Profit" [6]. With ROP, the same technique is used to overflow a buffer, but instead of actually inserting instructions onto the stack, addresses and constants are placed on it. These addresses point to special segments of code and the constants are used by that code. These special segments of code are called gadgets. The main difference between code injection and ROP is that gadgets used in ROP are already somewhere in memory, there is no new code being introduced into the system. The attacker only needs to know the address of the gadget in order to force the processor to execute the instructions of the gadget at a time when it wasn't meant to be executed. The constant values on the stack are values that the gadgets use, for example a gadget might load a value off the stack into a specific register.

Gadgets are short sequences of instructions that in the x86 world always end with a return instruction. The short sequence of code is meant to do a small amount of work towards the attacker's goal and then the return instruction transfers program control to the next address on the stack, where the attacker has placed a subsequent gadget. In this way the attacker can carry out their nefarious purpose by piecing together snippets of executable code that already exist in the program's memory. The attacker uses the program's own code to carry out the attack. The beauty of this method is that it circumvents the "write xor execute" memory policy. This memory policy has been defeated elsewhere as well [7].

The original ROP work [4] has been extended to many platforms. It was extended to SPARC, a fixed instruction length RISC architecture by Buchanan et al [8]. Checkoway extended it to not require return instructions [9].

2.1.2 Security versus Sustainability

The ability to secure systems has made impressive strides in recent years. End users

who purchase top-of-the-line systems, configure them properly, and patch regularly can have reasonably high levels of assurance that their devices are protected from all but the most persistent and well-resourced attackers. Methods to detect and prevent buffer overflow attacks are commonplace cowan1998stackguard,cowan2000buffer,alouneh2016software and similar defenses against ROP are beginning to be implemented in production systems fratric2012ropguard,cheng2014ropecker,pappas2012kbouncer. Unfortunately, these protections are not extended to low-power embedded devices as the overhead to implement them is generally incompatible with green computing goals [10]. As a result, security for green embedded and IoT devices is frequently off-loaded and handled external to the device, if at all. Specifically, external systems look for evidence of tampering either in network communications [11] or via side channel analysis [12]. Neither of these methods is effective at preventing local compromise or corruption in a device.

This research studies the case of the ARM Cortex-M4F in the Tiva C microcontroller. ARM specifies three families of processors: Cortex-A, Cortex-R, and Cortex-M. The Cortex-M family is specifically tailored for embedded system development [2], so any entity implementing a green device with the ARM architecture will do so using the Cortex-M family. Even though the Cortex-M4F is ideal for green applications, it is not ideal for security. The only applicable security feature is its Memory Protection Unit (MPU) [13]. The MPU allows a developer to specify granular constraints for reading, writing, and executing various memory regions which may be useful in preventing execution of malicious code injected into a data area, but is not effective at stopping ROP which adds addresses - not code - to writable areas and only executes existing code from executable areas.

ARM does have stronger protections in place in its Cortex-A family. Cortex-A processors are intended to deliver good performance for general purpose applications, but this prioritization of performance comes at the expense of energy-efficiency. While the Cortex-A is not ideal for green applications, it is the best ARM family for secure computing as it includes a full-featured Memory Management Unit (MMU) and the ARM Trust Zone [14]. The MMU handles virtual to physical address translation, extended permissions checking capa-

bility, execute never specification, and a non-secure bit. ARM Trust Zone creates a Trusted Execution Environment (TEE) in hardware which causes access to trusted applications and resources to cross a trust boundary with additional scrutiny and verification.

While clearly desirable, the security features of the Cortex-A family have not been feasible to implement on energy-efficient processors like the Cortex-M4F. ARM claims that the new Cortex-M23 specification will be the best of both worlds. Announced in late 2016, the Cortex-M23 and Cortex-M33 will be the first cores in the Cortex-M family with Trust Zone hardware protection built-in [15]. Of the two, the Cortex-M23 will be specially geared towards energy-efficiency while still providing Trust Zone security assurances. However, as of this writing, no chips or development boards have yet been available with a Cortex-M23 processor and the only public implementation is an IoT FPGA image for the Cortex-M Prototyping System, MPS2+.

2.1.3 Contributions

The four main goals of this work are to demonstrate:

1. The ability to create a gadget set capable of erasing flash memory. This is the first step in taking control of a microcontroller and could also result in a denial of service.

2. The ability to create a gadget set capable of programming the region of flash memory that was previously erased. This is the second step in taking control of a microcontroller.

3. The ability to create a Turing-complete gadget set from the TivaWare ROM. This allows for arbitrary code execution with ROP.

4. That modern energy-efficient embedded devices lack sufficient security assurances for mission-critical applications.

The contributions of this work are as follows: It is shown hereafter that an ARM Cortex-M4F processor using the Thumb-2 instruction set can be forced to execute arbitrary code. Specifically, the processor of the Texas Instruments Tiva TM4C123GH6PM microcontroller

is attacked and the results shared. Even a simple program loaded into main memory is sufficient to locate enough gadgets to carry out an attack. A small code base is vulnerable to many exploits.

We further show that the Tiva microcontrollers are particularly vulnerable because large portions of code, even code that may never be executed, is made available in the Read Only Memory (ROM) of the microcontroller. Included on this ROM are libraries which make interacting with peripherals such as a UART, GPIO and Ethernet controller easy. This code base is a treasure trove of potential gadgets to the attacker who has learned the technique of ROP. It is believed that the practice of loading this ROM with unnecessary code weakens the security of the microcontroller. The ROM is not, however, necessary to locate enough gadgets to carry out an attack.

Two sets of gadgets are shown, one taken from an example program in main memory and a second taken from the ROM. Both gadget sets can accomplish goals one and two (defined below) and make use of otherwise benign instructions that already exist either in system memory or on the ROM. Buffer overflow techniques combined with ROP makes this microcontroller an easy target for malicious attacks.

In addition to the above contributions, a novel gadget that loads the link register with the address of a `pop` of the program counter is discussed. A technique using this gadget which is similar to the update-load-branch [9] model is introduced.

This solution is simpler than the update-load-branch technique. A Turing-complete gadget set is also included. The search space for the gadget sets described below is restricted to the peripheral drivers loaded on the ROM of the Tiva C. Any program from main memory may be used to create a Turing-complete gadget set, if the appropriate gadgets can be located. The ROM was used here as an example as it ships pre-loaded on every Tiva C.

A novel gadget chaining mechanism is set forth as well. We also demonstrate the ability to manipulate the stack pointer which allows for efficient loops.

2.1.4 Related Work

A topic of research that is related to this work but not included is protecting a system

from buffer overflow. This has been extensively researched by others [16–19]. To counter that effort, several works have been dedicated to bypassing stack protections. Buffer overflows and bypassing stack protections will not be discussed here as they have been extensively discussed elsewhere [20, 21].

Francillon and Castelluccia published their research about an ROP procedure on an Atmel AVR atmega 128 8-bit microcontroller [22, 23]. In their paper they were able to successfully demonstrate a buffer overflow and permanent code injection attack using ROP against a sensor node using this microcontroller. In order to do so they performed several buffer overflows to build a fake stack one byte at a time. This fake stack was eventually used to perform the re-write to flash memory.

Our work differs from Francillion's. We are targeting the Cortex-M4 processor. The Cortex-M4 processor uses the Thumb-2 instruction set, whereas the AVR atmega 128 utilizes the AVR instruction set [24]. We also employ different return strategies which are discussed in Section 2.2.

2.1.5 Thumb Instruction Set

Cortex-M processors utilize the Thumb and Thumb-2 instruction sets. The Thumb-2 instruction set augments the original Thumb instruction set with several 32-bit instructions. The 16-bit instructions of Thumb map directly to an equivalent 32-bit ARM instruction [25], although not all instructions are accounted for. The advantage is that the instructions take up less space in memory which is desirable for a microcontroller as memory space is typically a premium [1]. For example, on a 16-bit memory system when Thumb is utilized the code size will typically be 65% of what it would have been if the ARM instruction set was used, and it will provide 160% of the performance [25]. The Thumb-2 instruction set adds 32-bit instructions on to the Thumb instruction set in order to allow for operations that were not previously accounted for [26].

This variable size instruction set does not introduce any insurmountable hurdles into the execution of an ROP attack on ARM devices. Some care must be taken to ensure that the processor is in the appropriate execution mode if it is capable of switching between

Thumb and ARM instruction sets. Similarly, jumping into a mis-aligned instruction, while theoretically possible, introduces many potential complications and should be avoided.

2.1.6 Threat Model

The threat model for this work is defined here. As previously stated we are attacking the ARM Cortex-M4F processor using the Thumb-2 instruction on a Texas Instruments Tiva TM4C123GH6PM microcontroller.

1. We assume that there exists a vulnerability in the code executing on the microprocessor to allow a buffer overflow to occur. Buffer overflows on the ARM architecture has been sufficiently shown elsewhere [9].

2. No execution of code that lies on the stack will be allowed.

3. The attacker has access to the contents of the ROM on the target device in advance of the attack.

2.1.7 Organization of the Paper

The remainder of the paper is organized as follows: Section 2.2 describes in more detail ROP on ARM devices and some differences between ROP on ARM vs. an x86 architecture. It also describes the various return-like sequences used to maintain control of a compromised device. Section 2.3 describes the process to erase and reprogram the flash memory on the Tiva microprocessor. Also included are the gadgets used for two reprogramming attempts. A Turing-complete gadget set is discussed in Section 2.4 with stack diagrams of the gadgets shown. Conclusions are drawn in Section 2.5.

2.2 Return-Oriented Programming on ARM Architectures

The Cortex-M4 does not explicitly follow the "write xor execute" memory policy. It is however, a modified Harvard-Architecture so the stack is innately non-executable [22, 27]. The only known way to modify the control flow of a program on such a device is to use ROP techniques. ROP has been proven to be effective at bypassing execution protections

on ARM-based devices [28]. Tim Kornau created an extensive work outlining ROP against the ARM architecture [29] in which the author specifically attacks a mobile phone running Windows Mobile 6.x.

There exist several differences between ROP on ARM architecture and x86 architecture [30]. A major difference lies in the structure of the gadget needed; ARM lacks the straightforward return instruction that x86 provides. Routines instead use other specialized instructions to change control flow between different sections of code. This is very important for ROP on a resource-constrained device. Because the code base to search for gadgets is limited the attacker must be creative in their ability to find return-like instructions that will allow them to maintain control of the code execution after each gadget. We identify four control-flow mechanisms used in ARM that can accomplish this purpose which enlarges the set of gadgets available to us.

The first of these control flow mechanisms is the `push` and `pop` set of instructions. A program utilizes these by storing register values on the stack by means of the `push` instruction. Then the program will execute the new routine. When that routine is completed, the state of the register is restored by pulling the previously stored value back off the stack by making use of the `pop` instruction. In this style of for routine calls, the program counter is one of the registers that is stored on the stack in this style of routine calls, meaning that if a `pop` instruction is found that contains the program counter register (and ideally no others) it can be treated in the same manner as a return instruction in x86 architectures. Figure 2.1 is a stack diagram which depicts an example of this type of return-like mechanism.

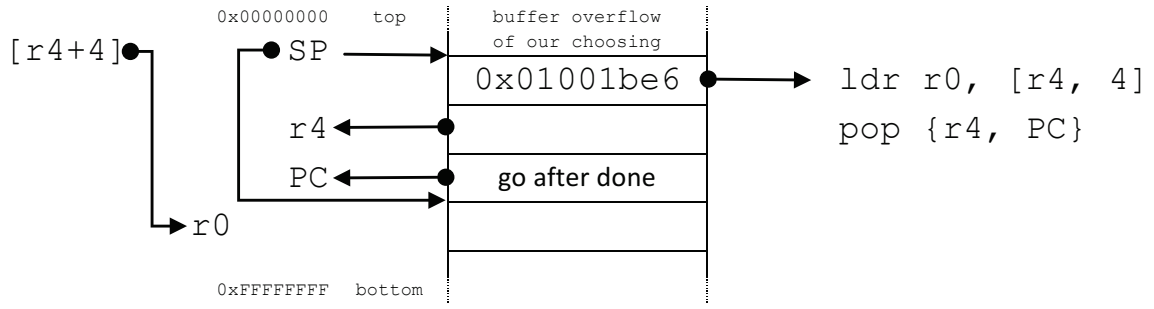


Fig. 2.1: Example of using a `pop pc` as a return.

The second control flow mechanism is a branch instruction that uses another special purpose register called the link register. Routines utilizing this style of return simply call another routine with the specialized `bl` or `blx` instruction, which loads the link register with the appropriate return address before branching to the new location. At the end, the called routine loads the link register to the program counter, and execution returns to the original point.

This second method has some subtleties that require more attention than the simple `pop` instruction. First, this style does not allow nested routine calls, as the inner call would overwrite the original value in the link register without being able to restore it. Therefore, the link register must be stored onto the stack using a `push` before the call, and restored after the call with a `pop`. Secondly, because the link register is used as the return address, it must be loaded with the address of the next gadget before being used as a gadget return.

These sequences can be combined to allow branches to the link register to be used as an equivalent of return for ROP gadgets. Code which uses a `pop` to restore the link register from the stack after a call using the `bl` instruction can be used as a gadget to load the link register with the address of a `pop` of the program counter. Once this is accomplished, all branches to the link register will jump to the `pop` of the program counter, which will then pull the next address from the stack as in a traditional ROP attack. If a pop instruction containing both the link register and program counter is found, the link register can be conveniently loaded in a single gadget.

This bears some similarity to the update-load-branch technique described in [9], which searches for gadgets characterized by indirect branches to the address held in a register which is loaded in a directly preceding instruction. The use of the `bx lr` instruction allows for a less complicated gadget sequence, where the address of a `pop pc` instruction is placed in the link register to provide what [9] refers to as a trampoline. This means that our approach does not need to use additional registers to maintain control flow and also does not need to provide an explicit ability to for advancing the stack pointer, two limitations of the work in [9]. Instead the sequence of gadgets using our method is to simply first load the

link register with the address of a `pop pc` instruction, and then call any number of gadgets ending with a `bx lr` instruction.

The mechanics of this gadget return style are demonstrated in Figure 2.2, which gives an example of a simple store gadget. First a load immediate gadget puts the address of the gadget using the `bx lr` return onto the stack. Next a `pop` gadget fills the link register with the address of a `pop pc` instruction (outlined in red). Finally, the unconditional branch jumps to the location of the `bx lr` gadget, which is in this case a simple store instruction. Note that any subsequent gadgets that use the `bx lr` return do not need to load the link register again, unless it is overwritten as some gadget's side effect. Any time the `bx lr` instruction is executed it will immediately jump to the `pop pc` instruction, which will in turn load the next gadget address from the stack into the program counter.

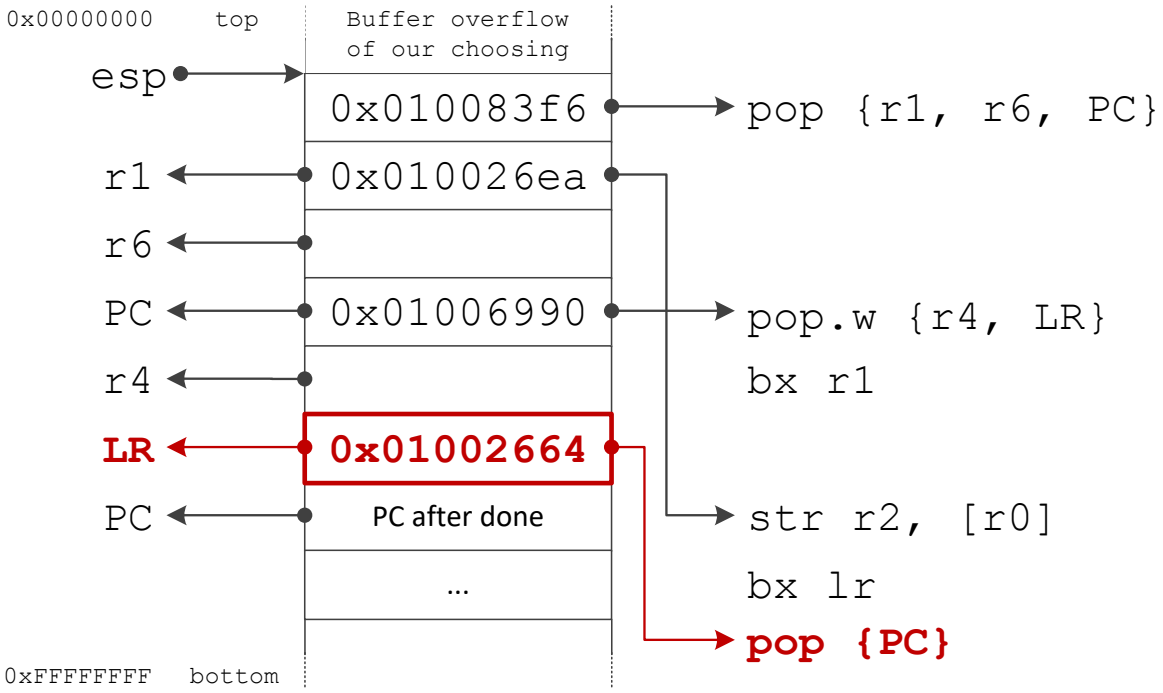


Fig. 2.2: Stack Diagram - `bx lr` Return.

A third control flow mechanism is a direct branch to an address held in a register. This can also be utilized by first loading the register with the address of the next gadget. These branches do not occur as often in code but if found they do provide an opportunity wherever

they are present. An example of this style of return can be seen in Figure 2.2, where the `pop` instruction responsible for loading the link register is followed by an unconditional branch to the address held in `r1`.

The fourth control flow mechanism is a branch with link to a general-purpose register. This appears as a `blx reg` where `reg` may be any general-purpose register such as `r1`, `r2`, `r3` and so forth. This is very similar to the third control flow mechanism of `bx r1`. However, the `blx` instruction not only updates the `pc` with the address specified by the value stored in the register, it also stores the address of the next instruction after the `blx` instruction in the `lr` register [31]. This is the equivalent of a function call and storing the return address in x86, and is used in [9]. An example of this can be seen in Figure 2.3 found in Section 2.4.1.

2.3 Erasing and Programming Flash Memory

This section describes two sets of gadgets for an ROP carried out on the Texas Instruments Tiva TM4C123GH6PM containing an ARM Cortex-M4F processor [1] revision 1. The Thumb-2 instruction set is utilized. The Cortex-M4F is designed to be integrated with a ROM which contains peripheral drivers [13]. Addresses `0x01000000-0x1FFFFFFF` are reserved for the ROM containing the TivaWare for C Series software. The first set of gadgets is shown to demonstrate that even if the ROM eliminated, this would not make ROP impossible. The first attack uses gadgets taken from an example program meant to demonstrate the Sensor Hub BoosterPack, BOOSTXL-SENSHUB [32], an available daughter card for the Tiva C. The search space gadgets for the first ROP example includes addresses `0x00000000` to `0x00005AA8`. The second set of gadgets can be found on the ROM.

Either set of instructions could be used with a buffer overflow to reprogram the flash memory of the microcontroller. This work is not focused on the method of the buffer overflow attack itself. However, it is assumed that a buffer overflow must occur to initiate the ROP procedure. Proof of successful buffer overflow has been sufficiently demonstrated on the ARM architecture [9].

A denial of service attack could be accomplished by simply erasing the region of flash memory which starts the default program's execution and then restarting the microcon-

troller. Once that region of memory is blank the microprocessor would not be able to boot. A second attack could be carried out by reprogramming that portion of flash memory with an arbitrary sequence of instructions after it had been erased. This other exploit is illustrated in both examples.

2.3.1 Finding Gadgets

To find the first set of gadgets, we dissembled the binary of the example program and used that to search for gadgets. For the second, we extracted the ROM binary file from the microcontroller and used its contents to locate gadgets. The open-source Radare2 [33] disassembler program revealed the assembly-code contents of the peripheral drivers included by Texas Instruments on the ROM.

After that, we ran simple `grep` commands against the resulting files as the starting point in the search for gadgets. However, more sophisticated methods than `grep` exist to find gadgets. As a result of the ARM instruction set being published and Kornau demonstrating automated searches for gadgets in ARM code, open-source tools exist to assist in the search for gadgets [29, 34]. `ROPgadget.py`, an open-source python script, significantly sped up the search for gadgets [35]. Several sources on the ARM architecture also proved to be invaluable in the search for gadgets [36–38].

`ROPgadget.py` allows users to extract all candidate ROP gadgets from a binary code segment. The ROM file yielded 6236 potential gadgets.

2.3.2 Reprogramming Method

In the following attack, the aims of goals of erasing and programming the flash are achieved. The procedures to perform these flash memory operations are found in Section 2.3.2. Gadgets to perform the first exploit are described in Section 2.3.3 and for the second they are described in Section 2.3.4. The character sequence placed on the stack is demonstrated in Section 2.3.3.

Flash Memory Write Sequence

In order to accomplish goals one and two the procedure to erase and write to the flash of the Tiva C must be understood. Programming the flash can only change a bit that is already a `1` to a `0` or just leave the bit as its current value. Programming cannot transition a `0` to a `1`. Based on this limitation, it is unlikely for any attack to be successful without first erasing portions of flash.

First flash must be erased and then programmed. This memory is erased by setting all bits to a 1. The flash on the Tiva C can be erased completely or in 1 kB blocks. The attack utilizes a 1 kB erasure. The erase procedure is as follows: first, identify the start address of the 1 kB-aligned Central Processing Unit (CPU) byte address which specifies which block of flash is the target for the erasure. Next, place that address into the Flash Memory Address (FMA) register `0x200FD000`. Finally, the write key must be loaded into the Flash Memory Control (FMC) register `0x400FD008` to bits 16 to 31 and the erase bit (bit 1) must be set. The write key is determined by the state of the key bit (bit 4) of the Boot Configuration (BOOTCFG) register `0x400FE1D0`. The possible values of the write key are `0x71D5` or `0xA442` for `0` or `1`, respectively. In this case the key bit is set to `1` indicating the value of the write key to be `0xA442`. Thus the 32-bit value that must be entered into the FMC register erase the 1 kB block is `0xA4420002`. The erase sequence can be found in Listing 2.1. In the attack demonstrated in Section 2.3 the start address of main happens to be located at `0x4BA0` and the minimum block size of flash memory that can be erased is 1 kB. Therefore, the start address for the erase is `0x00004800`. When this procedure is carried out all bits will be set to `1` between `0x00004800` and `0x00004BFF` in the flash which includes the start address of main. If the desire was to erase the entire flash instead of just a 1 kB block, simply write the key to the upper 15 bits of the FMC register and set the Mass Erase (MERASE) bit (bit 2). This can be seen in Listing 2.2.

Listing 2.1: Flash erasing sequence for the Tiva C.

```
// address of FMA register
uint32_t * FLASH = (uint32_t *) 0x400FD000;
FLASH[0x0] = 0x4800; // address to erase
// clear the area 0x4800-0x4BFF
```

```
// perform erase command by writing to
// FMC register
FLASH[0x2] = 0xA4420002;
```

Listing 2.2: Mass erasing sequence.

```
// address of FMA register
uint32_t * FLASH = (uint32_t *) 0x400FD000;
// erase entire flash by writing the key and
// setting the MERASE bit of FMC register
FLASH[0x2] = 0xA4420004;
```

Once all bits are erased (set to `1`) in the region that is to be re-programed, the flash is prepared to be written. The flash write procedure begins with writing the address to be programed to the same FMA register described in the flash erase procedure. Then the 32-bit word to be written is placed in the Flash Memory Data (FMD) register `0x400F004`. Finally, the same write key is written to the upper 16 bits of the FMC register as described above, however during a program command, bit `1` the write bit, is set. The programming procedure for writing the exploit program can be seen in Listing 2.3.

An alternate programming procedure includes writing the address to be programed into the FMA as seen earlier. Then the desired words are written to the appropriate Flash Write Buffer n (FWBn) registers: `0x400FD100` to `0x400FD17C`. This technique to write to the flash allows up to 32 32-bit words to be written at once. Then the Flash Write Buffer Valid (FWBVAL) register `0x400FD030` must be set to a mask indicating which FWBn registers are to be written. In the case of this example, all 32 bits are set. Finally, the Flash Memory Control 2 (FMC2) register is written to. This register is very similar to the FMC register. The write key is entered into the upper 16 bits and bit 1 (the write buffer bit) is set. The address of the FMC2 register is `0x400fd020`. Listing 2.4 illustrates this second method to program the flash. The first programming procedure writes one word at a time and is therefore easier to follow. The second procedure allows for up to 32 words to be written at the same time. As can be seen, Listings 2.3 and Listing 2.4 both write the same words to the same addresses, but Listing 2.4 requires one less instruction because it only needs to

write to the FMA register and the FMC register once. This simplification reduces the stack space needed for this exploit which is helpful when attacking a resource constrained device.

Listing 2.3: Tiva C flash programming sequence.

```
// place address to program (main) into FMA
FLASH[0x0] = 0x4BA0 ;
// assembly add instruction placed in FMD
FLASH[0x1] = 0xF1000001 ;
// write command - write key and write bit to FMC
FLASH[0x2] = 0xA4420001 ;
// second address to program placed into FMA
FLASH[0x0] = 0x4BA4 ;
// assembly branch instruction placed in FMD
FLASH[0x1] = 0xE7FC0000 ;
// write command - write key and write bit to FMC
FLASH[0x2] = 0xA4420001 ;
```

Listing 2.4: Alternate Tiva C flash programming sequence.

```
// base address to program 0x4B80 is the closest
// 32-word alligned address to main at 0x4BA0
FLASH[0x0] = 0x4B80 ;
// load add instr into FWBn - offset of 0x20
FLASH[0x48] = 0xF1000001 ;
// load branch instr into FWBn - offset of 0x24
FLASH[0x49] = 0xE7FC0000 ;
// set every bit in FWBVAL register
FLASH[0xC] = 0xFFFFFFFF ;
// write key and sett WRBUF bit of FMC2 register
FLASH[0x8] = 0xA4420001 ;
```

2.3.3 Demonstration of Writing a Simple Program to Flash

Here we show a demonstration in which the flash is reprogrammed with a sequence of instructions that will no longer execute the original program at all, but will instead simply enter into an infinite loop. This loop will begin immediately after the ROP is performed.

In addition, it will start again if the system reset push-button is ever pressed or if the Tiva C is powered off and then back on as this will reside in the flash memory at the location of `main`. This example will prove that an attacker is able to erase the flash memory and reprogram the microcontroller.

Gadgets

This example ROP procedure will demonstrate that even a small amount of existing code can be leveraged by a creative attacker. The search space for gadgets for this attack was limited to the example Sensor Hub Booster Pack program.The flash rewrite sequence contained in Listings 2.3 and 2.4 requires two operations: load and store. The search for gadgets resulted in two gadgets, two lines each which can accomplish these tasks. They are shown in Listing 2.5.

Listing 2.5: Gadgets that provide the load and store operations.

```
; Gadget A at 0x3673
str r0, [r4, #0x0]
pop {r4, pc}
; Gadget a0 at 0x3675
pop {r4, pc}
; Gadget B at 0x42A7
mov r0, r4
pop {r4, pc}
```

Gadget A provides the ability to store data from `r0` into the address specified at `r4`. It also causes the program to jump to the next instruction, while filling `r4` with more data from the stack. This gadget is effective because `r4` is constantly updated, and can thus be used to load immediate values off the stack. Note that Gadget A0 is the second line of Gadget A and could be useful if the `str` operation on the first line was not needed. This gadget is only used as the first gadget as there was no need for a store before the a value was popped into `r4` for the first time.

Gadget B transfers the data from `r4` into `r0`. There were no gadgets that would load `r0`

directly, so this method was a sufficient substitute. The data from the stack is transferred from the stack to `r4` by using Gadget A0, followed by Gadget B where the data is shuttled to `r0` while `r4` is repopulated. Finally, the data is stored into the desired location via Gadget A.

ROP Procedure

The design of the ROP was taken directly from the code in Listings 2.1 and 2.4. The flash programming method shown in Listing 2.4 was chosen because it needed only 5 total writes to flash registers while the sequence in Listing 2.3 required 6. The gadgets from Listing 2.5 were combined in a pipelined fashion in order to minimize operations. The implementation was still rather bulky at 23 required returns. Table 2.1 describes the order that the gadgets should be executed in order to rewrite the flash memory of the microcontroller.

The ROP attack was first approached by determining the size and boundaries of the stack. Once the boundaries were determined, the location on the stack where the program counter (`pc`) was stored was overwritten with the address of Gadget A. Each successive call (shown in Table 2.1) was determined by overwriting the values to be placed into the `r4` and `pc` registers.

This attack was successful and resulted in the flash being permanently reprogrammed. Even after the reset button of the Tiva C was pressed, an infinite loop was entered and nothing else was ever executed. This was verified by stepping through execution on the Tiva C using a debugger.

2.3.4 Second Gadget Set

The second set of gadgets can be seen in Listing 2.6. This gadget set was derived entirely from the ROM. We will not illustrate the second ROP procedure here, as it uses the flash erase and re-write procedures found in Section 2.3.2, and the procedure is very similar to Section 2.3.3. Using the four gadgets in Listing 2.6 we were able to successfully replicate a similar ROP sequence as has been already illustrated. The purpose of this exercise is to

Table 2.1: The ROP design.

Gadget	Pop into `r4`	Pop into PC	Description
	0x30303030	0x30303030	Don't care
	0x30303030	0x30303030	Don't care
	0x00000000	0x00000000	Don't care
	0x00000000	0x00000000	Don't care
	0x00000000	0x00000000	Pop {`r4-r5`}
pop {pc}	0x75360000		Return to A0
A0	0x00480000	0xa7420000	Erase address
B	0x00d00f40	0x73360000	Write erase address
A	0x020042a4	0xa7420000	Erase command
B	0x08d00f40	0x73360000	Write erase command
A	0x804b0000	0xa7420000	main address
B	0x00d00f40	0x73360000	Write main address
A	0x00f10100	0xa7420000	add `r0`,#0x1
B	0x20d10f40	0x73360000	Write add
A	0xfce75555	0xa7420000	b main
B	0x24d10f40	0x73360000	Write b
A	0xffffffff	0xa7420000	Clear write buffer
B	0x30d00f40	0x73360000	Write clear
A	0x010042a4	0xa7420000	Flash key
B	0x20d00f40	0x73360000	Write flash key
A	0x584b0000	0xa7420000	Scatter addr
B	0x00d00f40	0x73360000	Write scatter addr
A	0x42e00000	0xa7420000	b main
B	0x18d10f40	0x73360000	Write b main
A	0xffffffff	0xa7420000	Clear write buffer
B	0x30d00f40	0x73360000	Write clear
A	0x010042a4	0xa7420000	Flash key
B	0x20d00f40	0x73360000	Write flash key
A	0x00000000	0xa14b0000	Return to main
	0x1b		

demonstrate the ability to find a gadget set in the ROM similar to the gadget set already found in the code of a basic program.

Listing 2.6: Gadgets 1-4, which provide the functionality to erase and reprogram memory.

```
; Gadget 1 at 0x01007550
pop {r0,r1,r3,r6,pc}
; Gadget 2 at 0x010067aa
str r0, [r1, #0]
pop {r4,pc}
; Gadget 3 at 0x01006990
ldmia.w sp!, {r4, lr}
bx r1
; Gadget 4 at 0x101001024
subs r0, #1
bne.n 0x1001024
bx lr
```

The first gadget is a simple command that pops five values off of the stack. The first four values are stored in registers and the last value is stored in the program counter (`pc`). This is a very useful gadget because a command that pops the `pc` off of the stack can be used as a branch command. The address in the `pc` will be the next command executed by the program.

The second gadget is used to store a value in memory. The value in `r0` is stored to the address in `r1`. This gadget is particularly useful because it stores a 32-bit value to an address without an offset. Many of the possible gadgets in the provided code space will only store bytes or halfwords and require an offset. More analysis and longer gadgets would be required for these to be used. Another important feature is that it ends with a pop command that includes the Program Counter (`pc`), so it can be used as a branch to the next gadget.

The textttldmia.w command is used as a load immediate. The registers in the curly brackets are loaded with the values located at the address of first argument. If there is more than one register in the curly bracket the word that is in the next address (following the first argument), is loaded into each following register. In this case, the value at the

Stack Pointer (`sp`) is loaded into `r4`, and the next value in the stack is loaded into the Link Register (`lr`). Then the program branches to the address in `r1`. This gadget is very useful because it manipulates the link register.

The last gadget is a subtraction command to be used as a delay for the erase and write commands. Whenever a word was written or erased from flash memory a delay of between 50 µs and 300 µs is required. This gadget allowed for a delay long enough for the write or erase command to complete. Register `r0` was preloaded with the wait value. If the result of the substitution was 0, a flag would be set and a command can be used as if it were a comparison. In this case, if the result is not equal to zero the program branched to the beginning of the gadget `0x1001024`. The program looped through the subs command until a result of 0 at which time the program branches to the `lr`.

2.4 Turing-complete Gadget Set

The ultimate goal of an attacker in crafting a ROP library of gadgets is to establish a Turing-complete gadget set. This allows the attacker to accomplish an arbitrary computation using a single predefined gadget set, and even to go as far as creating a specialized compiler capable of generating an attack payload directly from C code [39].

The successful generation of such a gadget set depends heavily on the size and nature of the code base that is available to the attacker. An ideal environment for the attacker includes standard libraries that the attacker can easily depend on for widespread functionality. However, in embedded systems this is often not the case as memory is at a premium and only specialized libraries are available. For example, a library designed specifically for the purpose of configuring device peripherals will likely lack explicit functionality needed to perform complex arithmetic operations. Similarly, a library that provides such mathematical operations may be lacking in load and store instructions needed to work with the device memory. It is also possible that while a particular functionality exists within the library, there are instructions between the desired code and the return instruction which lead to undesirable side effects of the gadget.

The attacker may overcome such limitations with a careful and methodical approach.

After building a gadget set containing as many pure gadgets (that is, gadgets that are free from side effects) as could be found, these may then be combined to perform more complex operations. An `xor` instruction may be used to create gadgets for register clearing, register value swapping, and other operations such as negate. Development of gadgets for saving and restoring states and register values are also very useful, as these allow the use of gadgets with side effects between a save/restore pair of gadgets to avoid the unwanted consequences. By starting with the simplest gadgets and acquiring this functionality as early as possible, more complex functionality can be implemented without finding specific gadgets for each and every operation.

The basic starting gadgets that provide this base to build are easy to identify. Immediate loads of registers are almost trivial: all that is needed is a `pop` instruction that includes both the program counter and the register in question. Similarly, loads and stores can simply be used directly as long as a gadget return follows close behind. In contrast, a gadget for the equivalent of a branch instruction poses significant difficulty. Conditional execution of one of two gadgets (the equivalent of the if-then-else, or `ite` instruction) can be accomplished by utilizing the same instruction in the code, which is detailed in the conditional execution gadget below.

A fully-featured conditional branch gadget is much more challenging. To simulate the conditional branches of the instruction set the gadget must be capable of executing gadgets ahead in the stack, jumping over others if they are unneeded. The branch must also be capable of returning to an earlier point in the gadget sequence, looping back on itself and allowing gadgets to repeat. This backwards motion is an essential programming paradigm and is necessary for a Turing-complete set of gadgets, as forward and backwards motion must be present in a Turing machine [40].

Executing gadgets that are further ahead in the sequence can be accomplished without much trouble. A `pop` instruction removing several values from the stack can be conditionally executed, several times if necessary, to skip the gadgets that should not be executed. A gadget that increments the stack pointer can also be used to more efficiently adjust the next

gadget to execute.

Gadgets that decrement the stack pointer are much more difficult to identify. While there is code in the library in question that decrements the stack pointer, it is followed by an increment operation before any return statements such that a gadget cannot be built from it. An attacker would need to identify a way to either decrement the stack pointer to jump to previous gadgets, or copy the previous addresses in the stack to the next portions so that they are executed once again. We introduce a gadget that is capable of this backward looping in this work.

2.4.1 A Turing-complete Gadget Set

In order to show that our collection of gadgets are Turing-complete we use a paper written by Homescu et al. [41]. Here the authors attempt to create a set of Turing-complete gadgets with each gadget taking up the least amount of bytes possible. The functionality that was needed included twelve gadgets. We capture the same functionality in our gadget set but not in the same way. For example, the authors of [41] included two gadgets whose only functionality was to operate on system flags, while this functionality is built into our gadgets where needed. We were not attempting to identify the smallest possible gadgets, so this method worked for our needs. Besides the flag operations the set of gadgets identified by [41] included functionality to move or exchange register values, pop a value from the stack into a register, control the stack pointer, increment or decrement a value in a register, load a value from an address to a register, store a value to memory, add two values, and subtract two values. In addition to the above functionality the logical operations of `and`, `or`, `xor` and `not` are needed. The authors of [41] remind us that because of DeMorgan's laws only two gadgets are really needed to create the behavior of all these logical operations. The gadgets needed are either `and` or `or` and `xor`, `not` or `neg`. We chose to identify all of these gadgets, although the `xor` gadget is perhaps impractically large. The final functionality needed to make a gadget set Turing-complete is the ability to compare two values and branch based on their result. This section describes the gadgets we have found which cover all of this functionality.

The full gadget set developed from the included ROM implements the following functions, explained with stack diagrams. These gadgets illustrate the mechanics of a ROP attack on an embedded ARM device without any loss of applicability or effectiveness due to the lack of an explicit return instruction.

move: It is critical to have the ability to move values between registers. As a common operation, there are plenty of gadgets which are suitable for this. The following gadget shown in Figure 2.3 moves the value from register `r4` into register `r0` then transfers control to the location popped into `r1`.

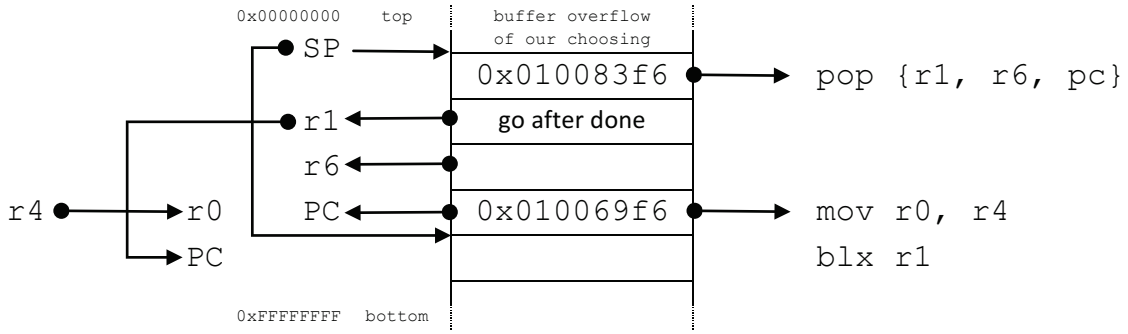


Fig. 2.3: Stack Diagram - `move`.

Note that this gadget has two side effects. First, register `r6` is overwritten by the first `pop` instruction. The register may be filled with either a "don't care" value or a value used by a future gadget. However, if the value in `r6` must be preserved then it should be saved before this gadget and restored afterwards.

The second side effect is the overwriting of the address in the link register that is used as a return address in the link register style of returns. Therefore, if any gadgets using this style of return follow this gadget then the link register must be reloaded with the appropriate address.

These two side effects demonstrate that for any gadget the potential impacts must be understood and accounted for. The attacker can compensate for them with planning, but obviously gadgets without these caveats are preferred. Similar side effects will occur with

many of the gadgets presented here due to the small size and specialized nature of the code base in question.

load: The load gadget loads the value from memory location `r4+4` into `r0`. Note that it assumes that `r4` is pre-loaded, but it happens to also `pop` a value to `r4` before returning. Therefore, if necessary, it could be called twice, the first time putting the desired `r4` address on the stack. The load gadget is illustrated in Figure 2.4.

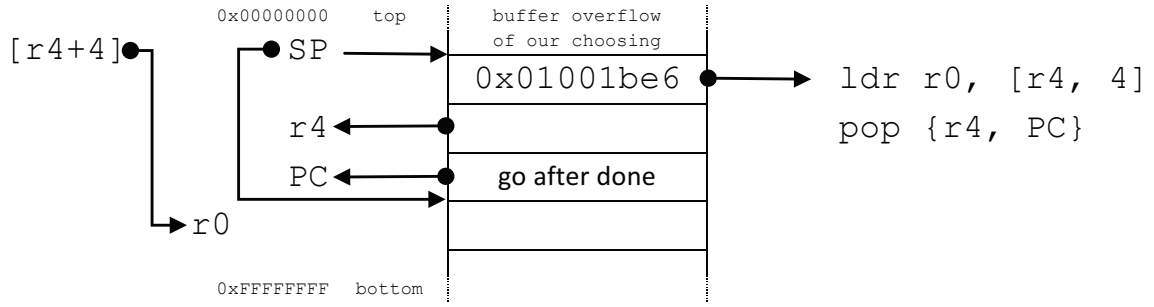


Fig. 2.4: Stack Diagram - `load`.

load immediate: The load immediate gadget shown in Figure 2.5 will take an arbitrary value and load it into a given register. In this case, the attacker can place the arbitrary value on the stack and `pop` it into `r4`.

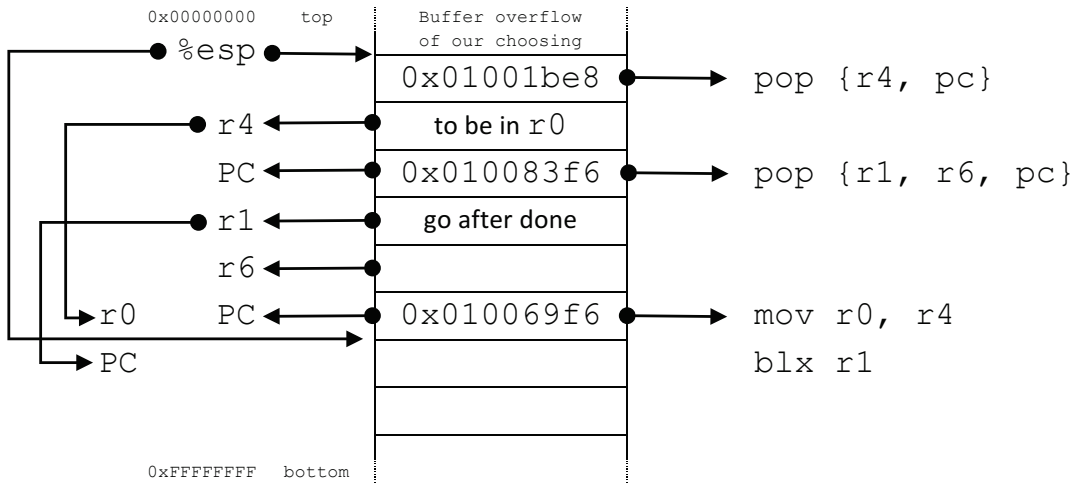


Fig. 2.5: Stack Diagram - `load immediate`.

store: The store gadget will store the value in register `r0` to a memory location at an offset of 12 from the address the attacker places in `r3`. Figure 2.6 shows a stack diagram of the store gadget.

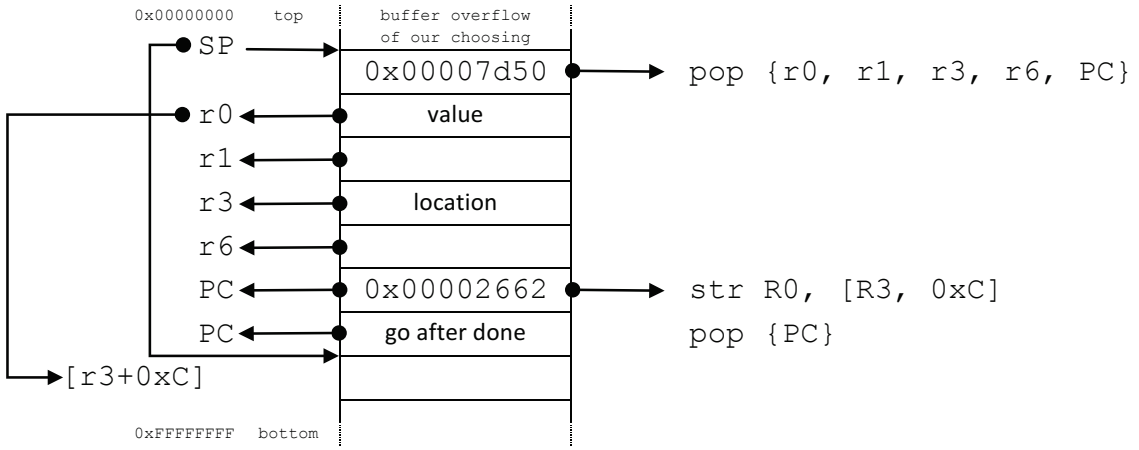


Fig. 2.6: Stack Diagram - `store`.

The operations below can be carried out strictly by register operations, e.g., AND, or through register operations with an immediate operand, e.g., AND immediate. Above we showed that immediate values can be loaded into registers, so the operations below focus strictly on register operations assuming a value has been pre-loaded into a register, if necessary.

add: The add gadget sums the values in `r1` and `r3`, then stores that sum in `r1`. This gadget complicates matters by ending in a `bx lr` command, which is the most common way to return from a legitimate subroutine. The issue is that control will be passed back to the address contained in the link register (`lr`), so we must proactively use the first two jumps to set `lr` to a value we control from the stack.

sub: As seen in Figure 2.8, `r0` is subtracted from `r1` and the difference is stored in `r0`.

negate: The gadget for negate is no different from sub. The attacker just needs to ensure zero is loaded into `r1` from the stack in the first command. Thus, negation is performed via subtraction from zero.

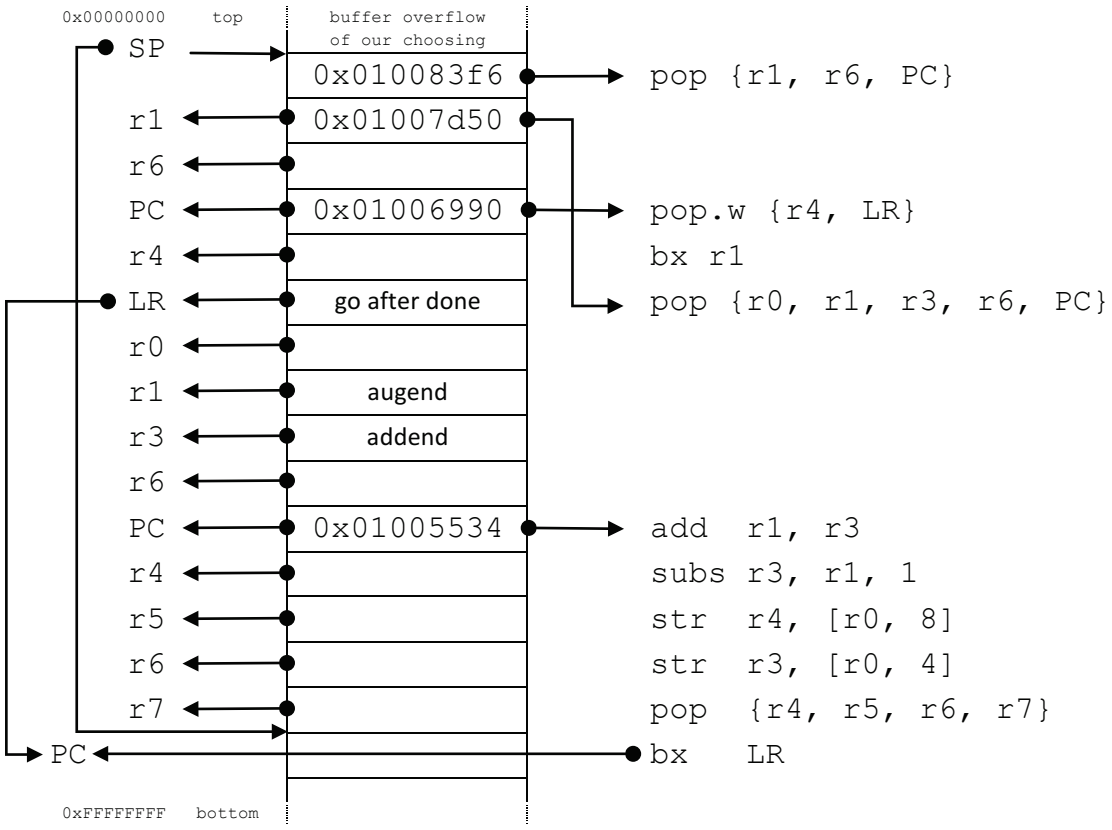


Fig. 2.7: Stack Diagram - add.

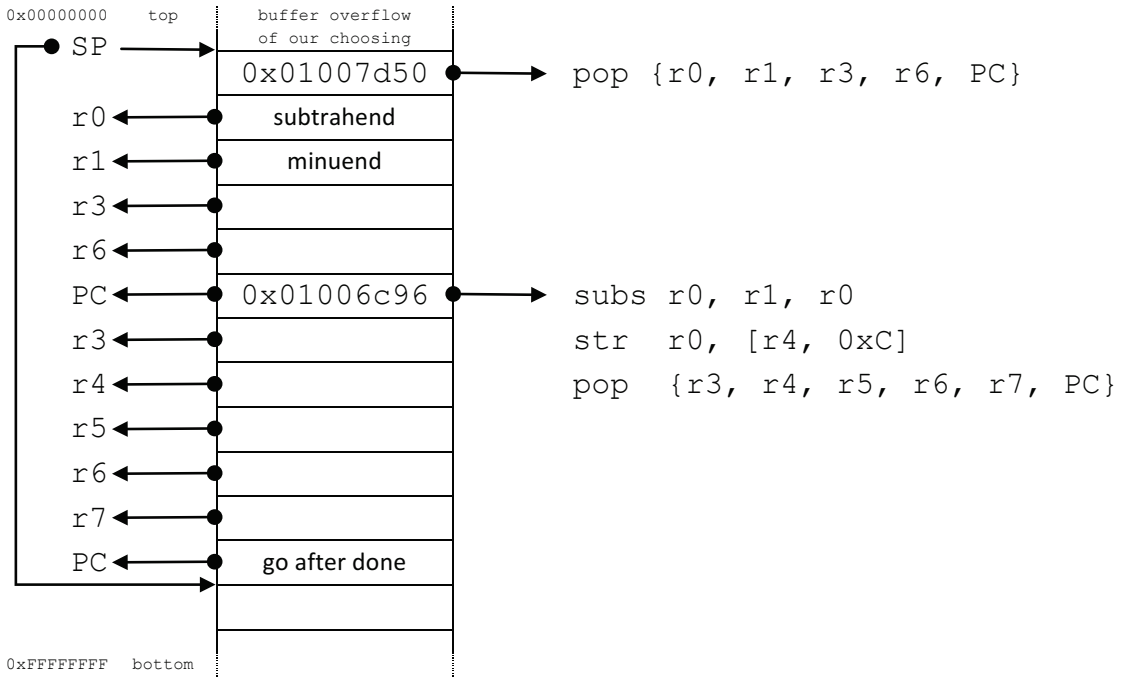


Fig. 2.8: Stack Diagram - `subtract`.

not: To perform a not, an attacker could simply load -1 into `r1`, then use the sub gadget shown in Figure 2.8. As this value is off by 1 from a true not, subtract 1 from `r0`. After that, Listing 2.7 would come right after to finish the operation.

Listing 2.7: Final part of the not gadget.

```
@ 0x010083f6
  pop   {r1, r6, pc}
@ 0x01006990
  pop.w   {r4, lr}
  bx  R1
@ 0x010058f2
  subs  r0, r0, 1
  bx  LR
```

and: The and gadget as shown in Figure 2.9 performs a bitwise and between `r2` and `r3`, storing the result in `r2`. Again, the link register needed to be set up to maintain continued control over the instruction sequence.

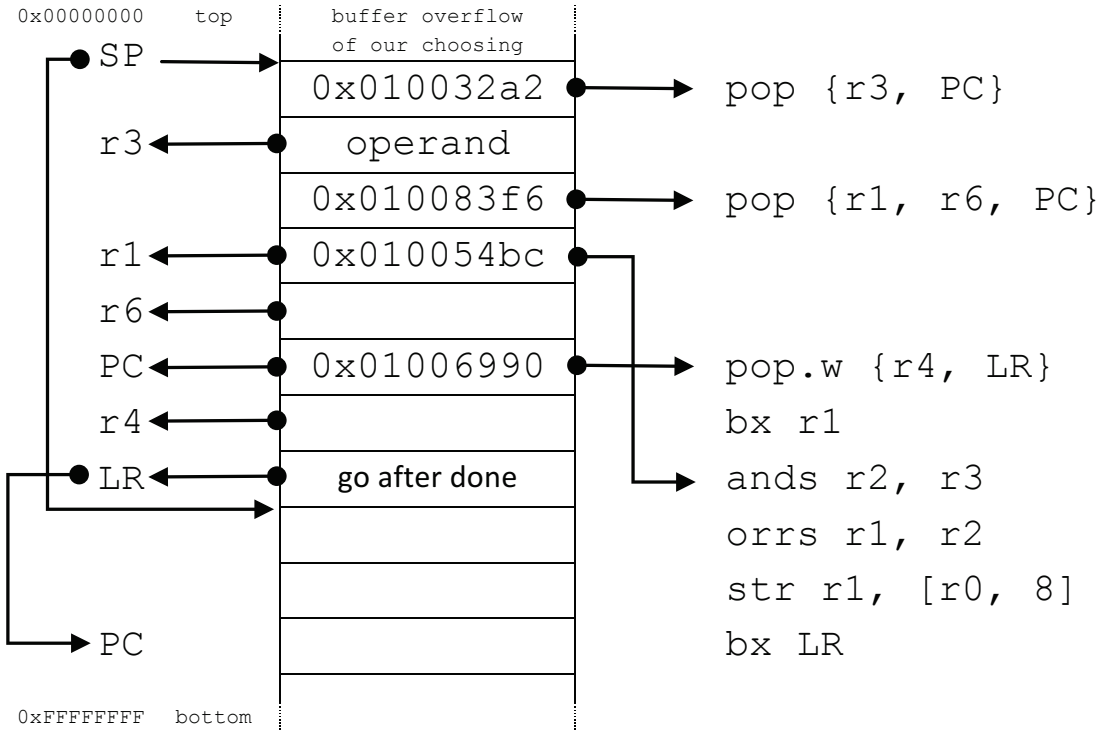


Fig. 2.9: Stack Diagram - `and`.

or: Figure 2.10 shows the OR gadget. The values over which a logical OR must be performed are placed in the `r0` and `r2` registers. Register `r0` gets the result and can then store it into a memory location offset from the address in `r1`.

branch: A simple pop into the program counter simulates an unconditional jump to the address on the stack.

Conditional Branch: This conditional execution examines the value of `r2` which is supplied from the stack by the attacker. If `r2` equals 1, the system will branch to the address of condition A, otherwise the branch to condition B will be followed. These branch addresses will come from the address relative to `r0`, which will be slightly offset from the stack pointer. This gadget can be seen by examining Figure 2.11.

set less than: The set less than gadget as seen in Figure 2.12 adds the ability to conditionally set a value that will compare `r0` to `r1`. If `r0` is less than `r1`, it will set `r0` to 1, otherwise `r0` will be cleared to 0.

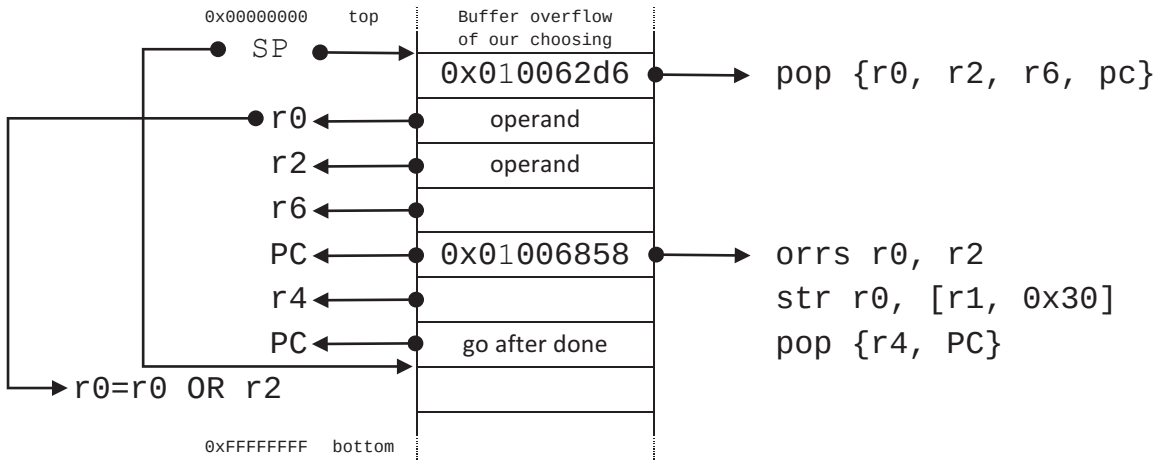


Fig. 2.10: Stack Diagram - `0r`.

Control Stack Pointer: The final step in making this set of instructions Turing-complete, is to have fine control over the stack pointer. This allows these ROP routines to efficiently be used multiple times without exhausting the memory resources allocated to the stack. As ROP gadgets are called and executed, the stack pointer (`sp`) moves down continuously. A separate gadget must then be implemented to allow the `sp` to return back upwards to a specific previous location. When implemented, an attacker is then able to perform repetitious operations, such as for loops, while loops, and recursion. Without this looping ability, an attacker would have to copy repetitive segments of code to the stack the exact number of times that code would need to be executed. Inevitably, they would quickly run out of stack space if even a simple routine needed to be executed a significant number of times.

The included code for controlling the `sp` does just this. It allows the saving of the stack pointer at the beginning of a repetitious construct into the `r0` register. Later, there will need to be a decision made as to whether that construct should repeat, or whether program flow should continue sequentially. The conditional branch gadget will be used to make this decision and, if the stack pointer needs to be reinstated to the beginning of a loop, the final command in this gadget can take the desired `sp` address loaded from memory with a load gadget into `r0` and then set the `sp` to that address.

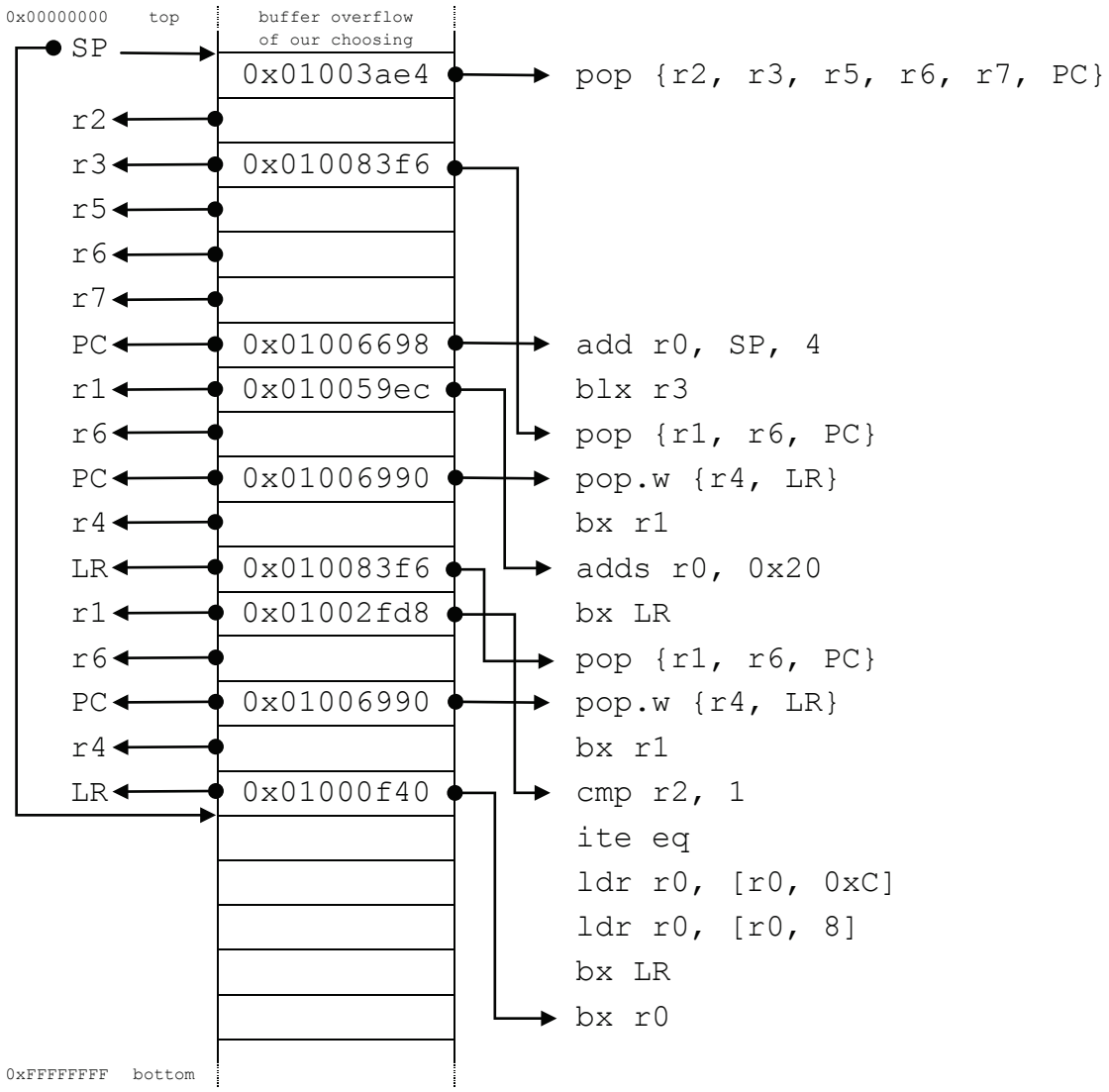


Fig. 2.11: Stack Diagram - `conditional branch`.

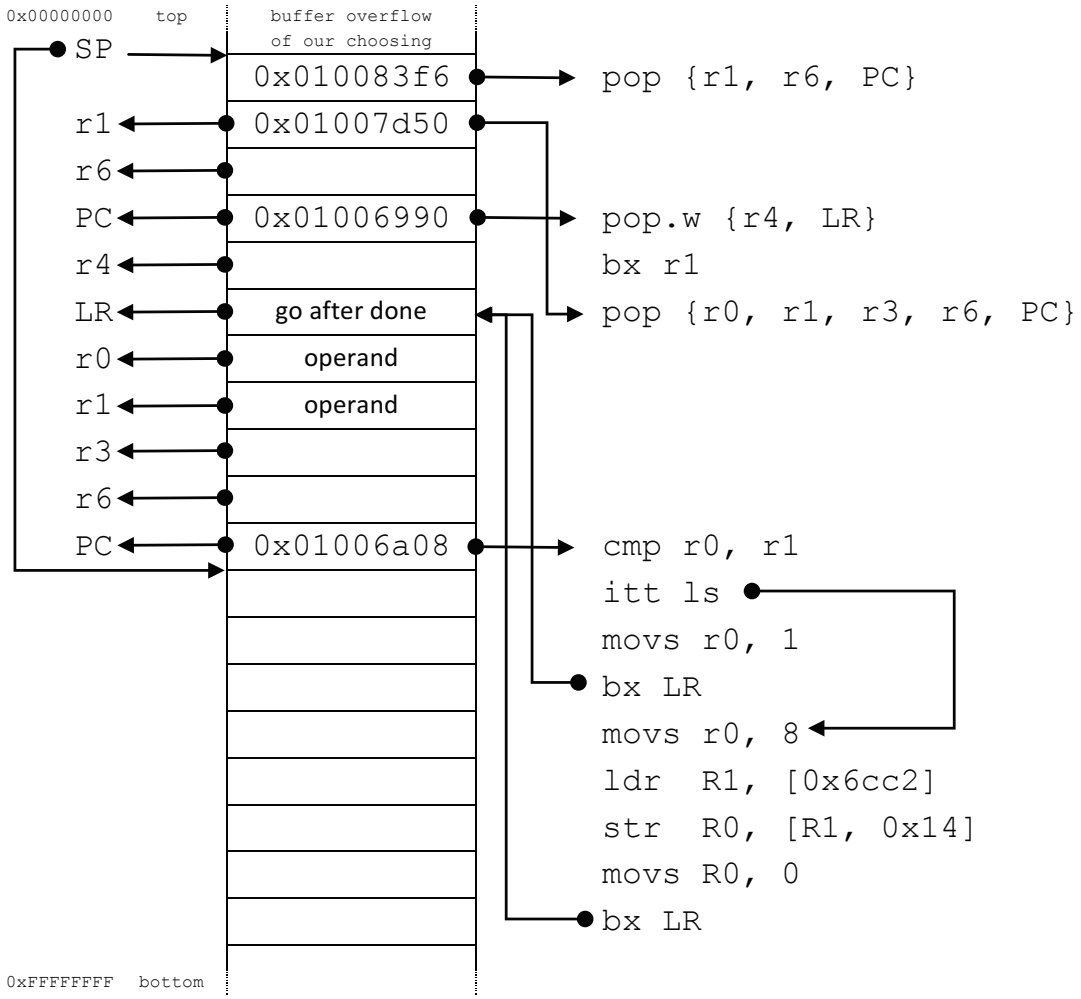


Fig. 2.12: Stack Diagram - `set less than`.

Listing 2.8: Control Stack Pointer.

```
Store initial SP value to be restored later
@ 0x01006620
mov r0, sp ; blx r2
\#Then use store gadget to place r0 in RAM

After a conditional branch,
one target could be to restore the sp
@ 0x01000e74
ldr.w sp, [r0] ; bx r1
```

Delay Loop: We have implemented a delay loop using ROP techniques. This delay loop has the mechanics of a simple loop that merely increments its counter and stops once that counter has reached a pre-determined value. This would be useful for reprogramming the flash memory.

To begin, the loop counter variable is set in `r5`. Figure 2.13 implements a 9 iteration loop and, in each iteration, the repeat condition is determined by a comparison the value from `r3`. So in this example we set `r3` to be 9 and `r1` to start at 0. On each iteration of the loop `r5` is incremented by 1. When `r5` equals `r3` the branch back up is not taken and instead a branch to the `lr` is taken instead.

xor: The real utility of these gadgets is that they can be cleverly tied together to perform more complex or higher level functions. For example, no suitable single xor gadget was found in this code base. However, since all logic gates can be composed of AND and NOT gates, an XOR gadget can be realized through a composition of several primitive building blocks. The truth table for XOR shows that `r0` XOR `r1` can be implemented with an OR between two operands. The first operand is `r0` AND NOT `r1` and the second operand is NOT `r0` AND `r1`. The stack diagram for this instruction is located in Figure 2.14 which refers the XOR arguments as A and B.

2.5 Conclusion

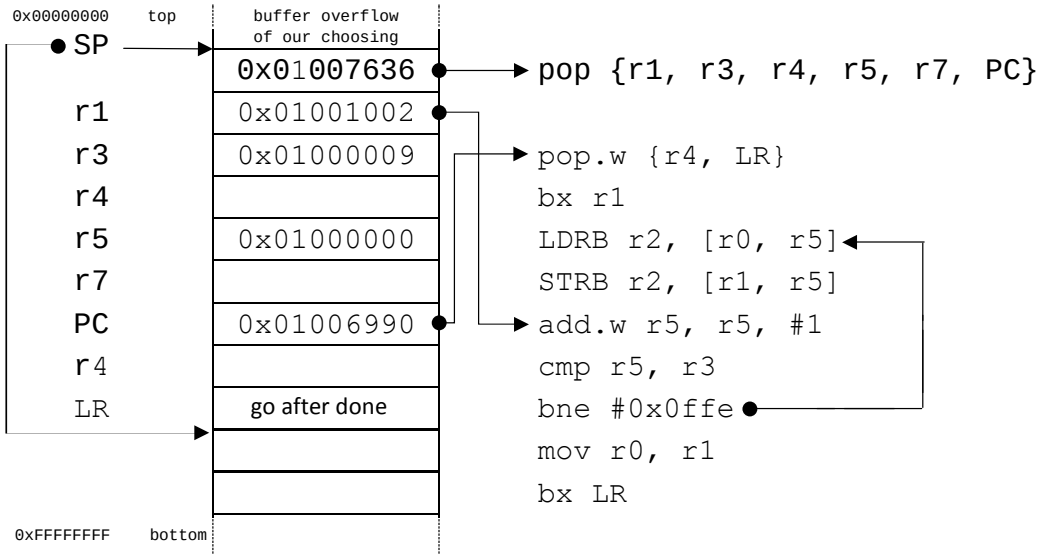


Fig. 2.13: Delay Loop.

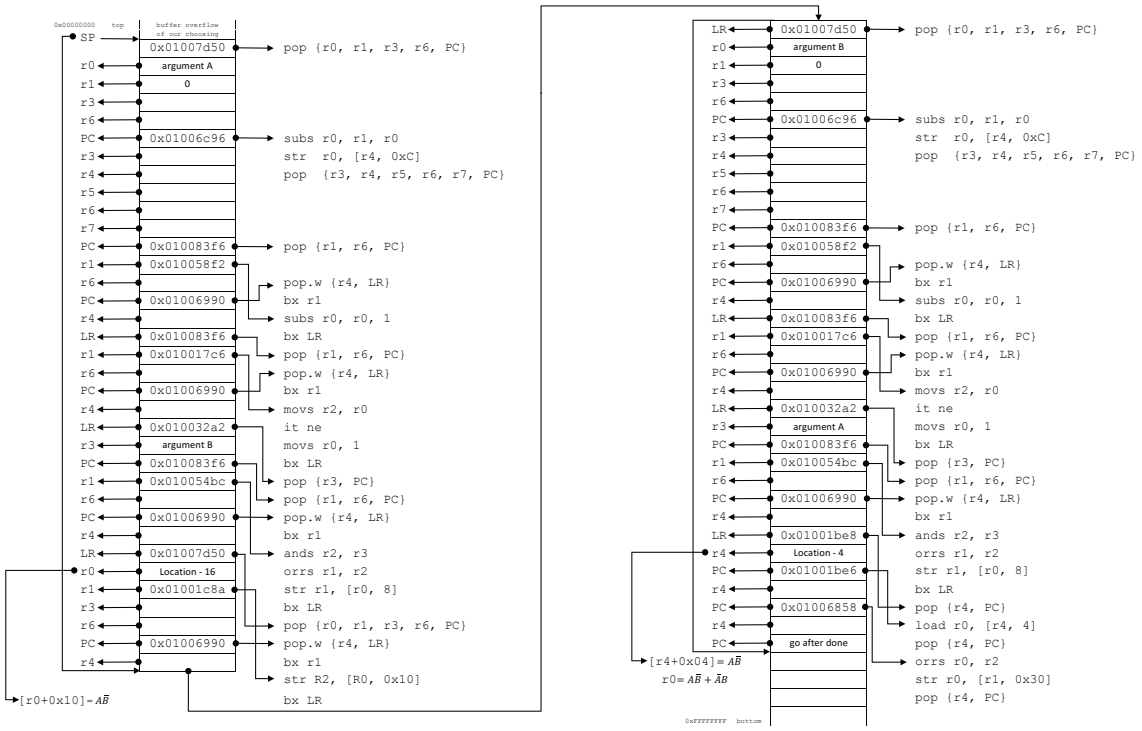


Fig. 2.14: Stack Diagram - `xor`.

This work has brought to light some security concerns on energy-efficient architectures like the Cortex-M4F using the example of a Tiva TM4C123GH6PM. It has been shown that even small areas of program code are enough to reprogram the flash memory of a resource constrained device. The practice of loading the on-chip ROM with peripheral libraries and other potentially unused code makes it a prime target for ROP attacks. The gadget sets needed to erase and reprogram the flash memory were quite small, and therefore could likely be found in many simple programs. A portion of flash memory can be erased using only 4 gadgets. It has also been shown that a Turing-complete gadget set can be identified in the peripheral driver libraries which would allow for arbitrary execution if a device was compromised. We discussed various return-like instructions that can be used for ROP on this device and have introduced a novel technique. We have also located gadgets which allow us to control the stack pointer making efficient loops possible.

Defense against ROP attacks has been explored with popular approaches to include ROPGaurd [42], ROPecker [43], and kBouncer [44]. Unfortunately, these techniques add significant overhead to the system and have all been bypassed in a more rigorous analysis [45]. Therefore, they are not currently practical to implement in a device geared towards sustainability. We have shown that omission of modern security controls leaves a plethora of energy-efficient products wide open to attack. Successful defense against ROP that is practical for resource constrained, low power embedded systems remains a topic that requires more attention in future research.

2.6 Appendix A. Experimental Results

This appendix shows screen shots taken from the Keil μVision [46] debugger [47] of the program stack containing the ROP chain about to be executed. It also shows the register values both before and after each of the gadgets was executed on an example program. This is to demonstrate to the reader that these gadgets can be used to exploit the target hardware. In each of these cases, a simple buffer overflow was used to take control of the microprocessor and the exploited program as well as the overflow strings are included with the submission of this article.

For each figure below the left most screen shot of the registers shows their state just before the ROP chain begins executing and the registers on the right show their state after. The memory that contains the stack is shown in between them.

In the case of the store gadget, Figure 2.18, an additional screen shot of the memory containing the stack and just below it is shown so that the location where the value was stored can be seen. The move gadget set can be seen in Figure 2.15. The load gadget is shown in Figure 2.16 and the load immediate in Figure 2.17. Figure 2.19 and Figure 2.20 show the add and the subtract gadget. The and gadget is seen in Figure 2.21 with the or gadget in Figure 2.22. Figure 2.23 shows the conditional branch gadget. The set less than gadget is shown in Figure 2.24. The delay loop is shown in Figure 2.25.

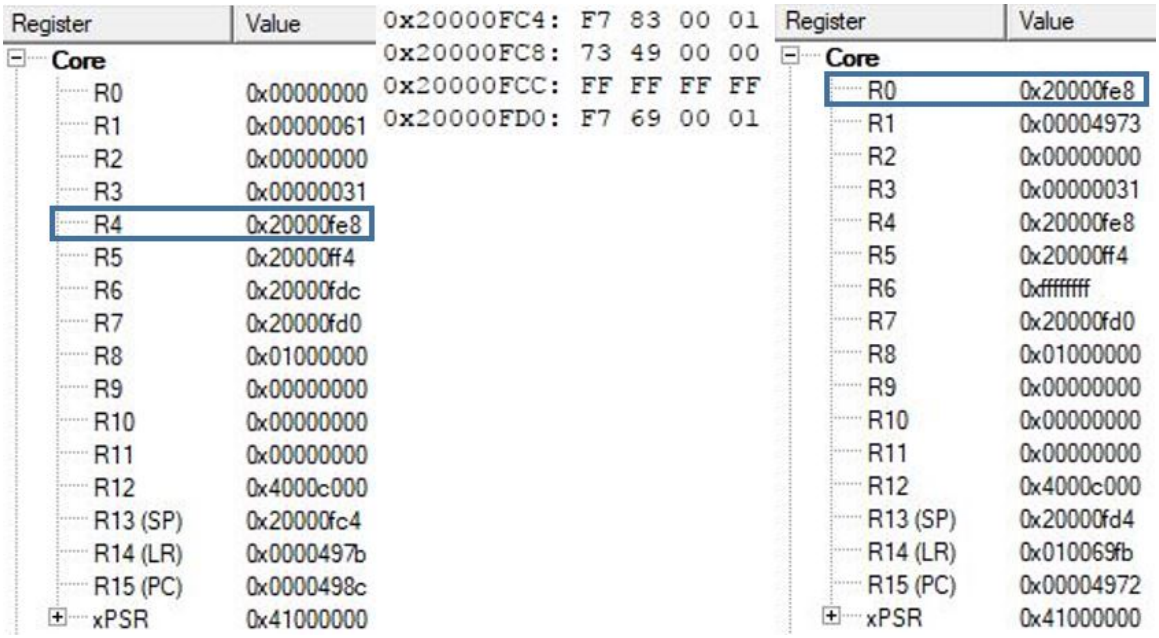


Fig. 2.15: Move. The value contained in `r4` moved to `r0`.

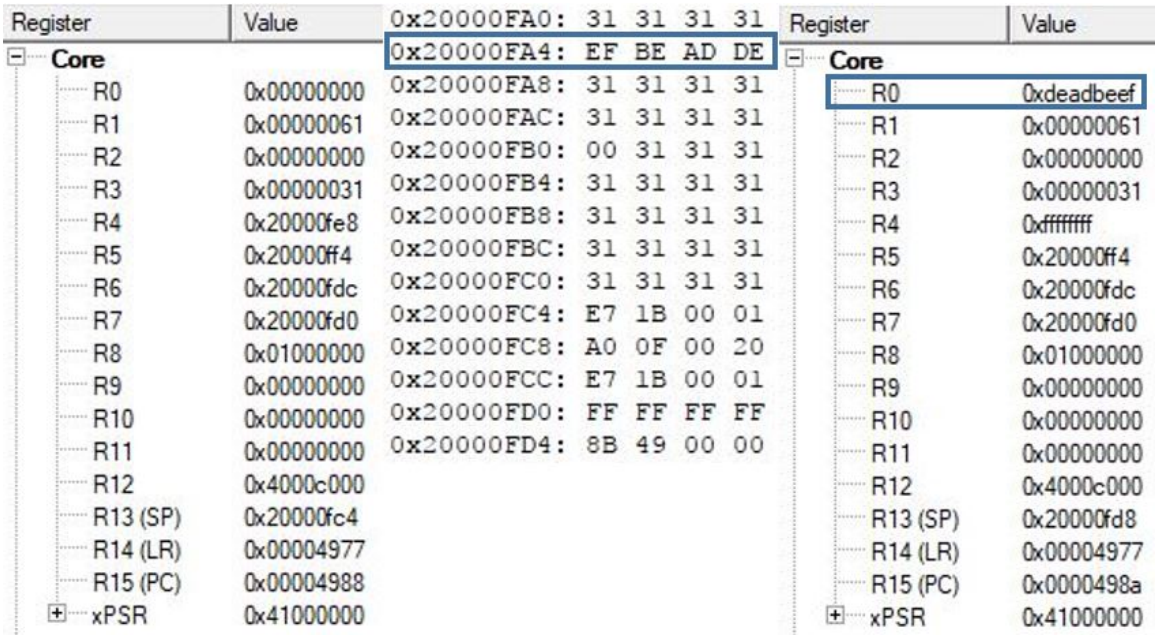


Fig. 2.16: Load. The value `0xDEADBEEF` from memory location `0x020000FA4` to `r0`. Note that the stack pointer shows the bottom of the stack as `0x20000FC4` before execution begins.

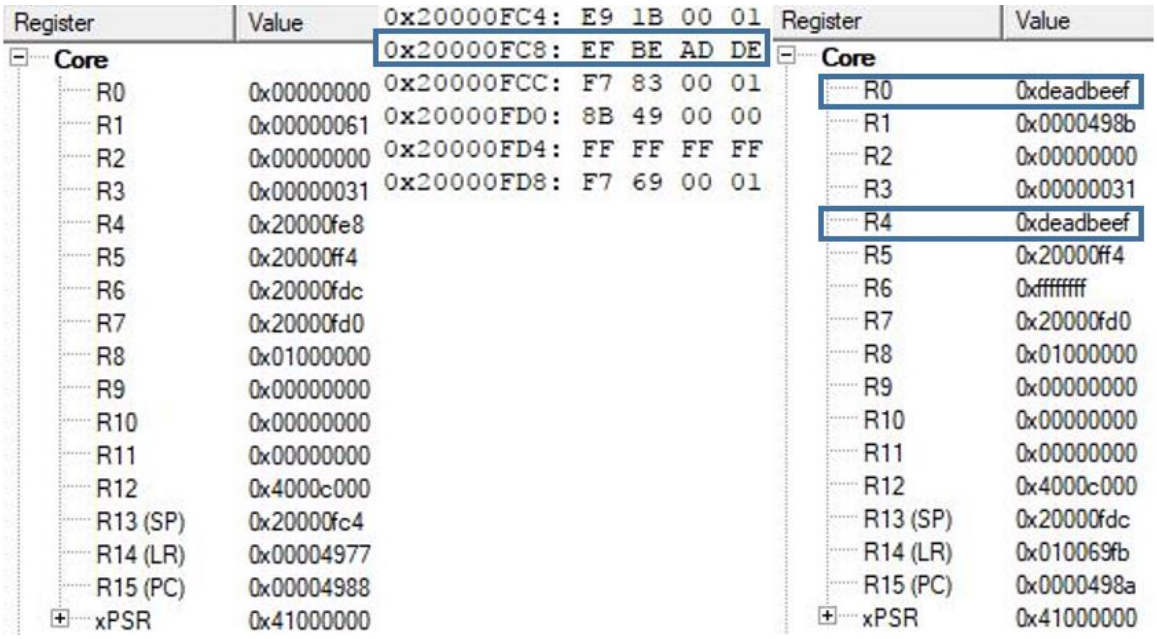


Fig. 2.17: Load Immediate. The value `0xDEADBEEF` is popped from the stack into `r4` and then moved to `r0`.

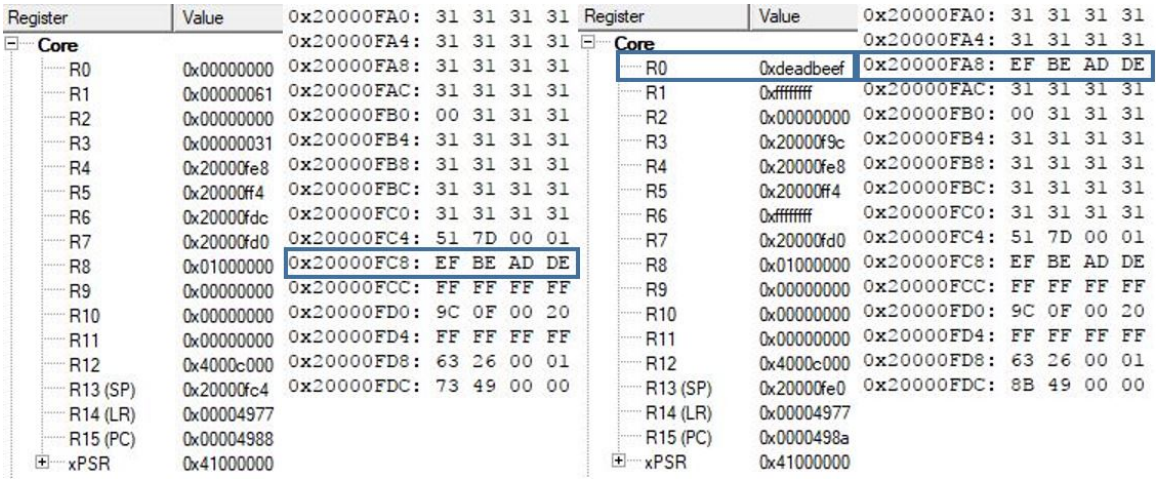


Fig. 2.18: Store. The value `0xDEADBEEF` is taken from the stack, placed in `r0` and then written to memory location `0x20000FA8` which is 12 plus `0x20000F9C`.

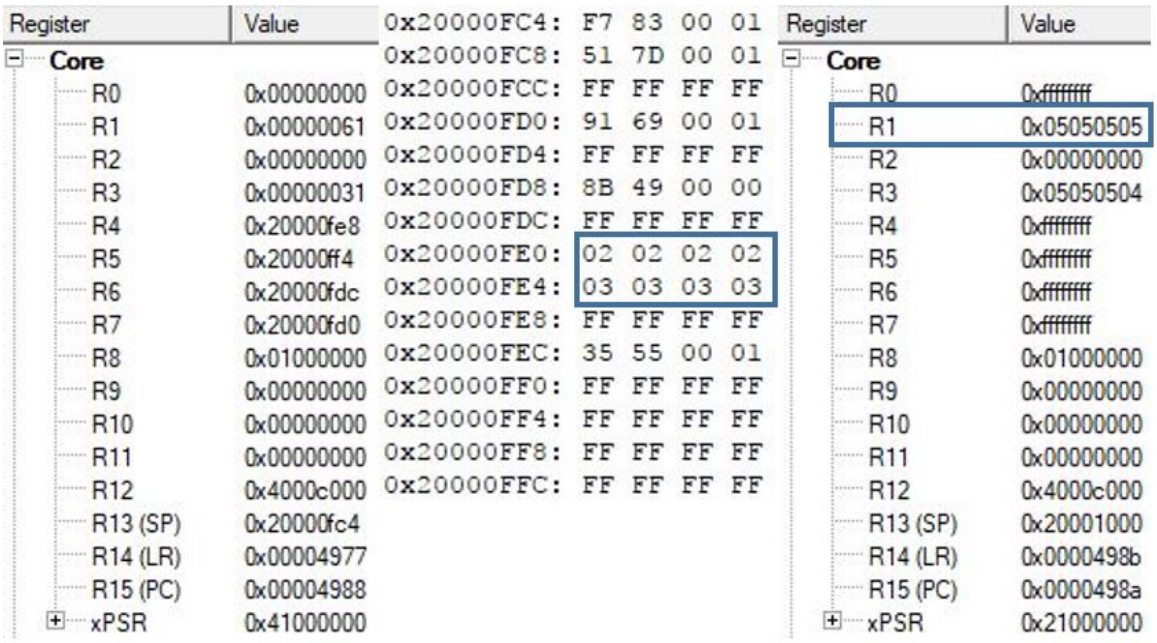


Fig. 2.19: Add. The values `0x02020202` and `0x03030303` from the stack are added together and the result, `0x05050505`, is placed in `r1`.

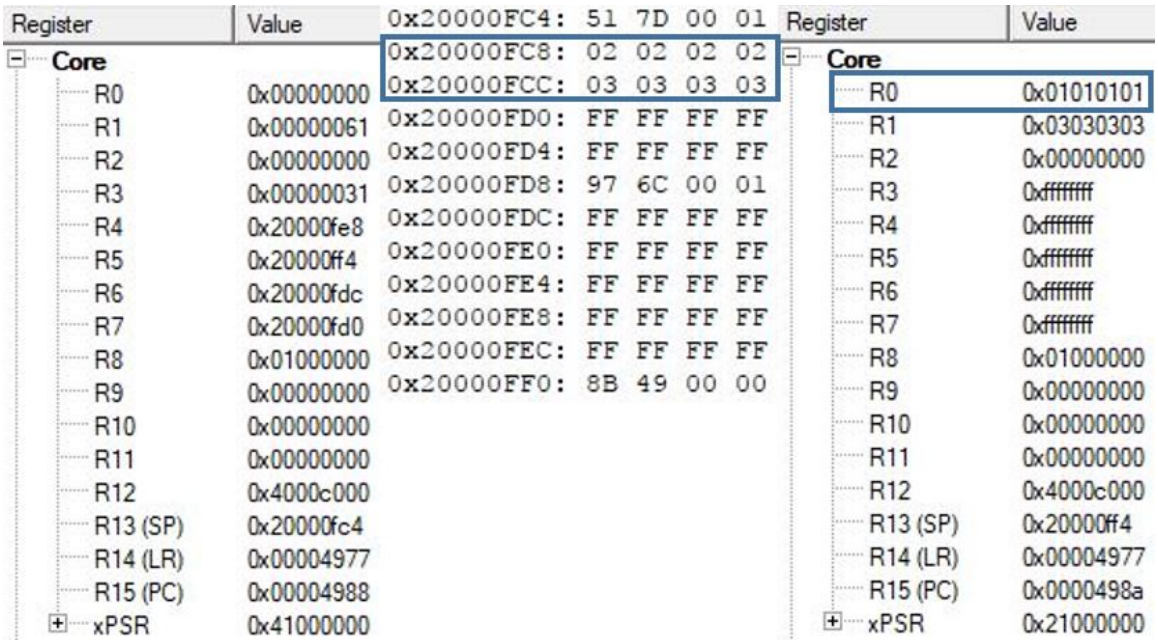


Fig. 2.20: Sub. The value `0x02020202` is subtracted from `0x03030303` (both values are found on the stack) and the result, `0x01010101`, is placed in `r0`.

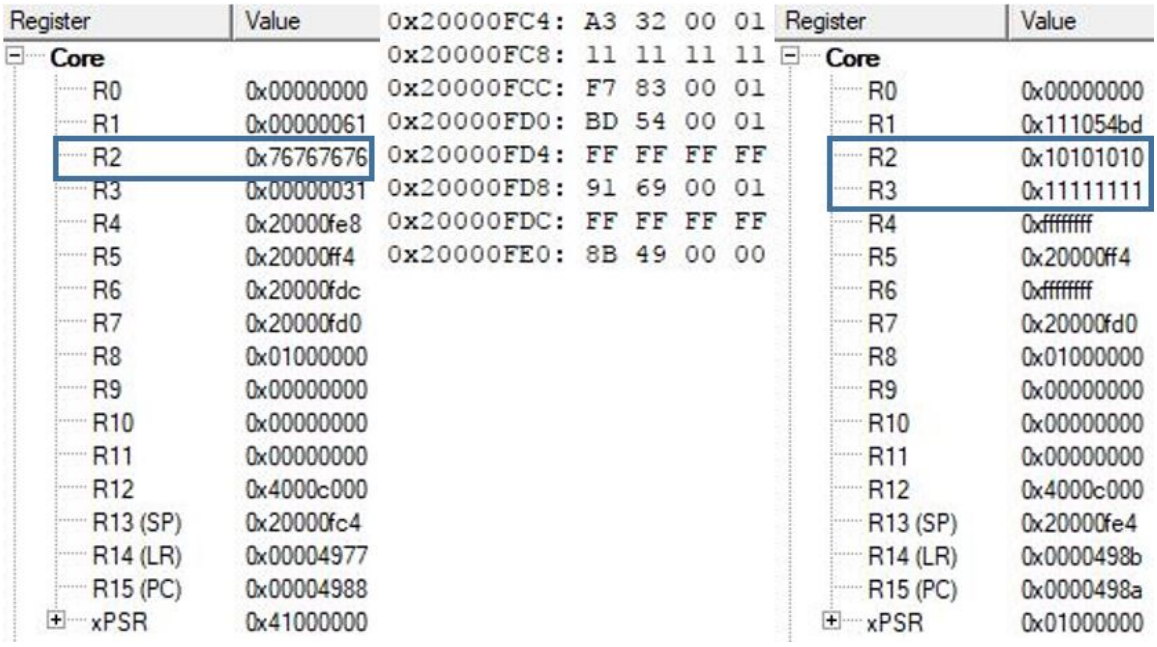


Fig. 2.21: And. The value `0x11111111` is placed in `r3` and anded with the value, `0x76767676` already in `r2`. The result, `0x10101010` finishes in `r2`.

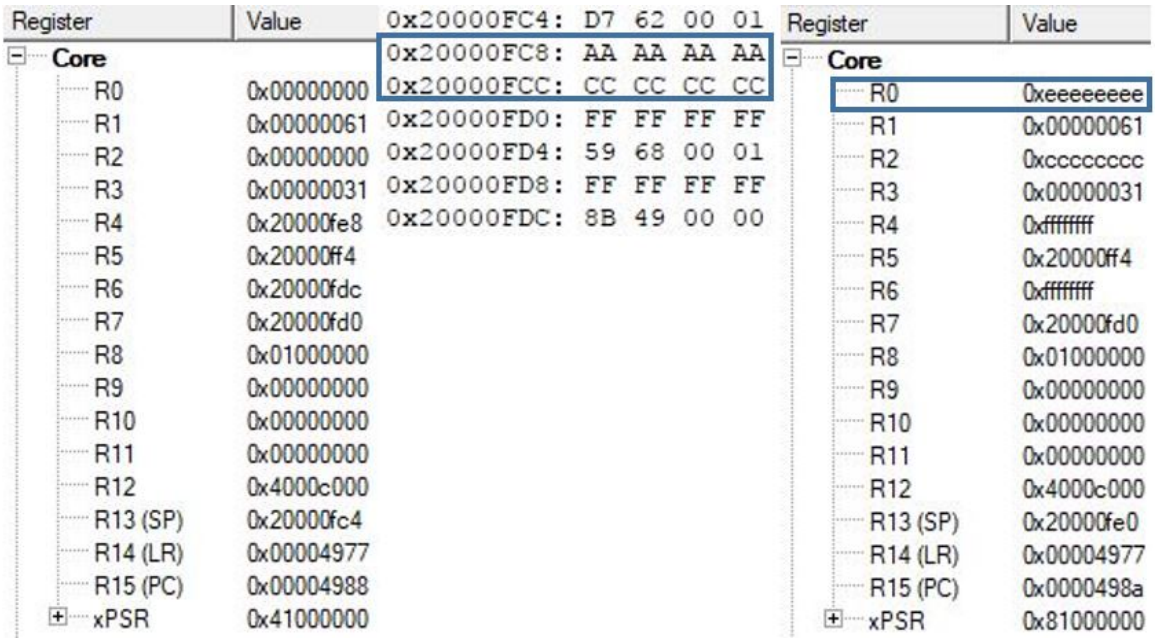


Fig. 2.22: Or. The two values `0xAAAAAAAA` and `0xCCCCCCCC` are taken from the stack and ored with each other. `0xEEEEEEEE` is the result of that operation and it is placed in `r0`.

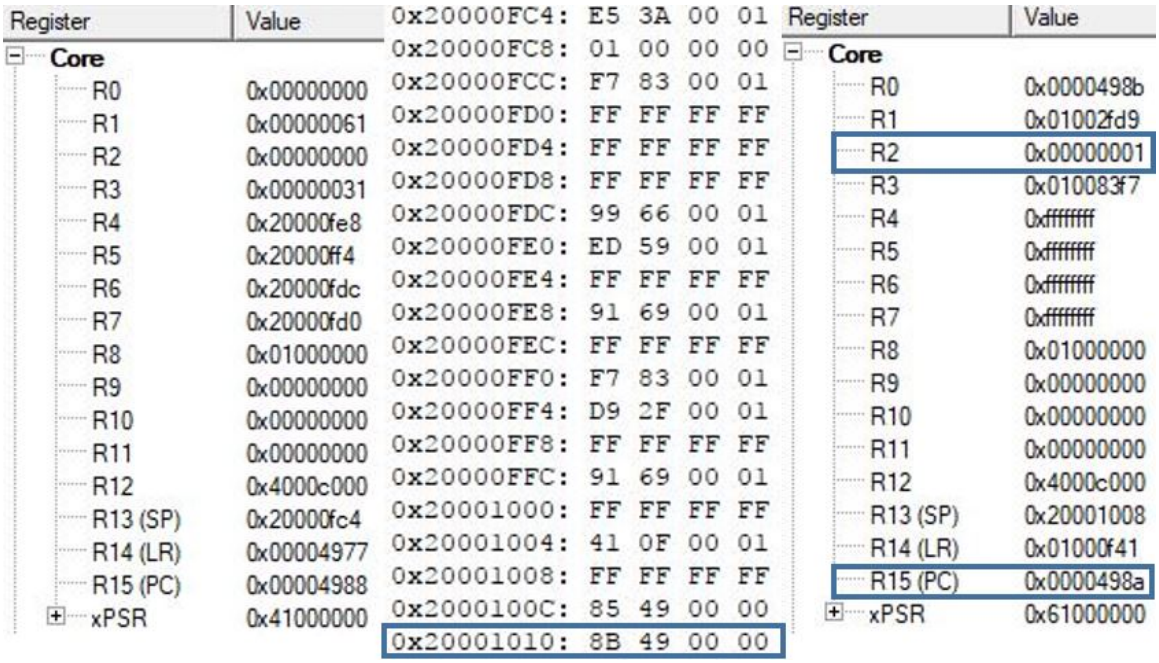


Fig. 2.23: Conditional Branch. The value `0x00000001` is placed in `r2` which indicates to branch to the location found at memory location `0x20001010`. If any other value besides `0x00000001` was placed in `r2`, the branch to the address located at `0x2000100C` would have been followed.

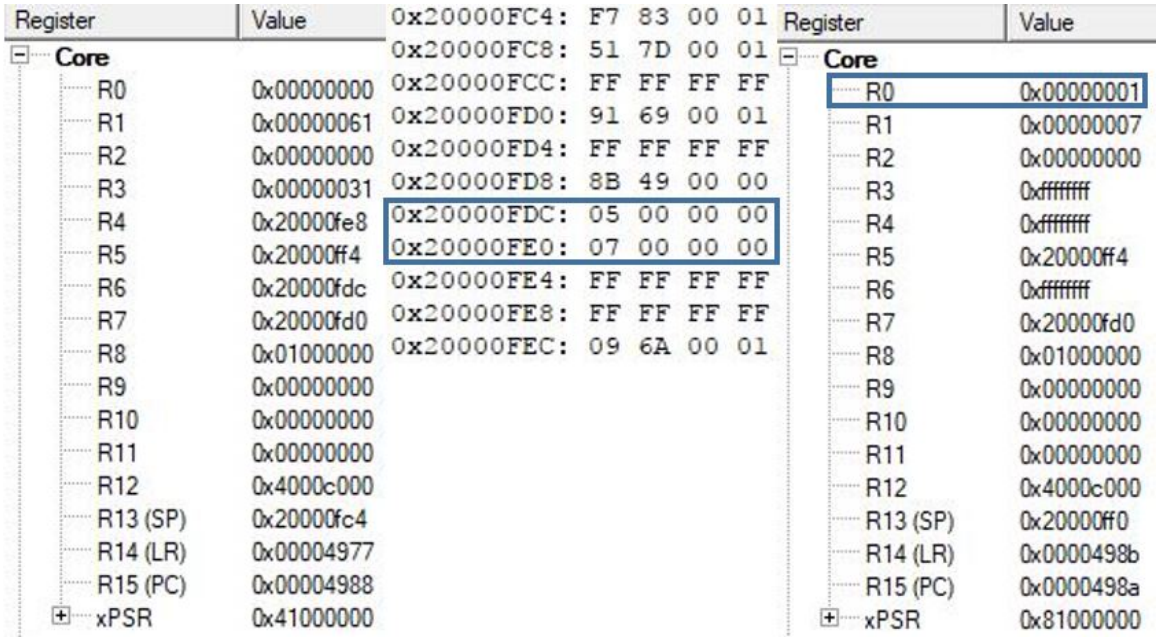


Fig. 2.24: Set Less Than. The values `0x00000005` and `0x00000007` are tested. As the first value is less than the second a `0x00000001` is placed in `r0`. If the first value was not less than the second value, a `0x00000000` would have been placed in `r0`.

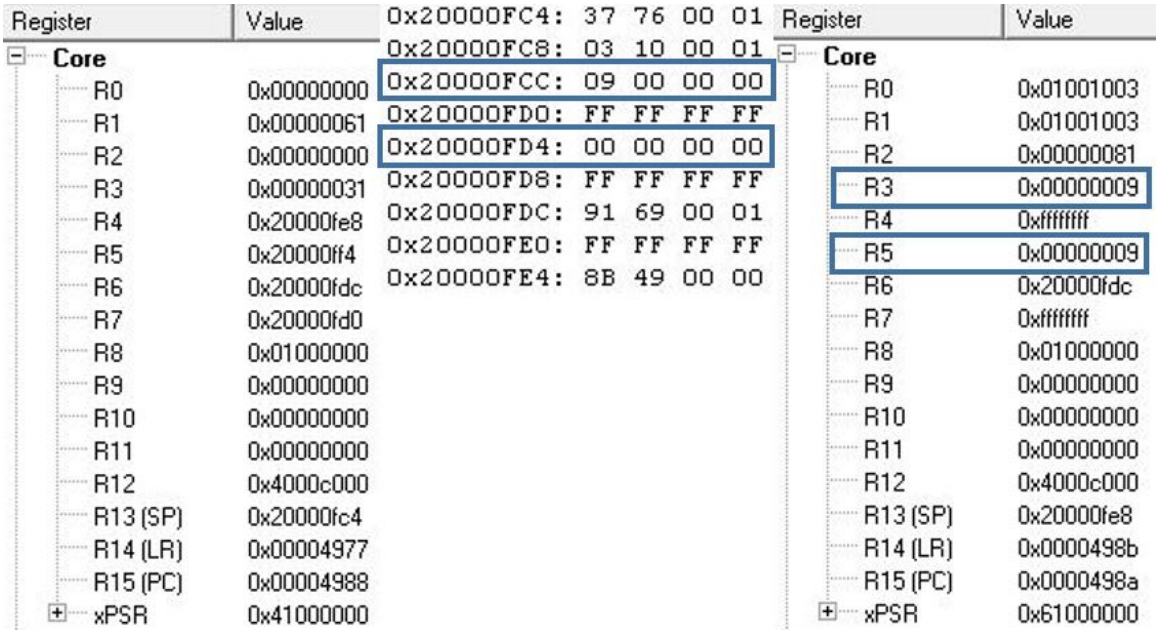


Fig. 2.25: Delay Loop. `0x00000000` is initially loaded into `r5` and `0x00000009` is loaded into `r3`. `r5` is incremented until it equals `r3` and then the loop finishes. The final value of `0x00000009` can be seen in both `r5` and `r3`.

REFERENCES

[1] *Tiva TM4C123GH6PM Microcontroller*, Texas Instruments Incorporated, 2014, rev. E.

[2] "Cortex-M Series Family description," http://www.arm.com/products/processors/cortex-m, accessed: 2017-04-25.

[3] *TivaWare^{TM} Peripheral Driver Library*, Texas Instruments Incorporated, 2016, version 2.1.3.156.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[5] "Thumb-2 technology," https://pax.grsecurity.net/docs/noexec.txt, PaX Team, accessed: 2017-04-25.

[6] A. One, "Smashing the stack for fun and profit (1996)," *Phrack*, vol. 7, p. 49, 2007.

[7] J. McDonald, "Defeating solaris/sparc non-executable stack protection," *Bugtraq, Mar*, 1999.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 27–38.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[10] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[11] A. P. Lauf, R. A. Peters, and W. H. Robinson, "A distributed intrusion detection system for resource-constrained devices in ad-hoc networks," *Ad Hoc Networks*, vol. 8, no. 3, pp. 253–266, 2010.

[12] X. Wang, H. Salmani, M. Tehranipoor, and J. Plusquellic, "Hardware trojan detection and isolation using current integration and localized current analysis," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 2008, pp. 87–95.

[13] *Cortex-M4 Technical Reference Manual*, ARM, 2010, revision r0p0.

[14] ""fundamentals of armv8-a"," https://static.docs.arm.com/100878/0100/fundamentals_of_armv8_a_100878_0100_en.pdf, accessed: 2018-04-07.

[15] ""arm trustzone developer guide"," https://developer.arm.com/technologies/trustzone, accessed: 2018-04-07.

[16] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with stackguard," in *Linux Expo*, 1999.

[17] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[18] S. Alouneh, M. Kharbutli, and R. AlQurem, "A software approach for stack memory protection based on duplication and randomisation," *International Journal of Internet Technology and Secured Transactions*, vol. 6, no. 4, pp. 324–348, 2016.

[19] M. Lackner, R. Berlach, R. Weiss, and C. Steger, "Countering type confusion and buffer overflow attacks on java smart cards by data type sensitive obfuscation," in *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*. ACM, 2014, pp. 19–24.

[20] K. Bulba, "Bypassing stackguard and stackshield," 2000.

[21] P. M. Dovgalyuk and V. A. Makarov, "When stack protection does not protect the stack?" *Trudy instituta sistemnogo programmirovaniya RAN*, vol. 28, no. 5, pp. 55–72, 2016.

[22] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 15–26.

[23] *8-bit Atmel Mirocontroller with 128KBytes In-System Programmable Flash*, ATMEL, 2011, rev. 2467X-AVR-06/11.

[24] *AVR Instruction Set Manual*, ATMEL, 2016.

[25] "The thumb instruction set," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html, ARM, accessed: 2017-04-25.

[26] "Thumb-2 technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471k/pge1358786963523.html, ARM, accessed: 2017-04-25.

[27] "Cortex-m4 devices generic user guide," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CHDCHEAG.html, ARM, accessed: 2017-04-25.

[28] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage." *EVT/WOTE*, vol. 2009, 2009.

[29] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Master's thesis, Ruhr-Universität Bochum, 2010.

[30] L. Le, "Arm exploitation ropmap," in *BlackHat 2011 Briefings and Training*. BlackHat, 2011.

[31] A. N. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide*. Elsevier, 2004.

[32] "Boostxl-senshub sensor hub boosterpack," http://www.ti.com/lit/ug/spmu290/spmu290.pdf, Texas Instruments, accessed: 2018-04-14.

[33] ""radare2 download page"," https://radare.org/r/down.html, accessed: 2017-04-25.

[34] ""arm and thumb-2 instruction set quick reference card"," http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf., accessed: 2017-04-25.

[35] J. Salwan, ""ropgadget"," https://github.com/JonathanSalwan/ROPgadget, accessed: 2017-04-25.

[36] ""arm infocenter"," http://infocenter.arm.com/help/index.jsp, accessed: 2017-04-25.

[37] W. H. Christopher Hinds, *Arm Assembly Language: Fundamentals And Techniques*, 2nd ed., 2014.

[38] J. Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, 3rd ed. Newnes, 2014.

[39] S. J. and D. V., "Rop compiler," http://www.keil.com/support/man/docs/uv4/uv4_debugging.htm, accessed: 2018-04-14.

[40] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. International Thomson Publishing, 2006.

[41] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 2012, pp. 7–7.

[42] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," 2012.

[43] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[44] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," *tech. rep. Citeseer*, 2012.

[45] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 88–108.

[46] "Microcontroller development kit," https://www.keil.com/demo/eval/armv4.htm, Keil, accessed: 2018-04-14.

[47] "μvision user's guide," http://www.keil.com/support/man/docs/uv4/uv4_debugging.htm, Keil, accessed: 2018-04-14.

CHAPTER 3

On the Limitations of Obfuscating Redundant Circuits in Frustrating Hardware Trojan Implantation

ABSTRACT Split manufacturing is a method to secure circuits by creating layers of a circuit separately— one layer would be manufactured at a trusted foundry and the other at an untrusted foundry. The complete design would not be known without both pieces. A prominent example of this approach is a paper entitled “Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation" [1]. This paper is very important in the field because it gives strong theoretical proofs as apposed to more recent publications which only provide empirical results [2–7]. The authors claim that even if an attacker knew the exact layout of a circuit before it was divided, the technique set forth would prevent the attacker from inserting an effective hardware Trojan into the circuit as long as they did not have knowledge of the trusted layer.

We claim that this method of split manufacturing for security from [1] is not effective for protecting redundant circuits such as cryptographic ciphers from the implantation of hardware Trojans. Specifically, we were able to insert a hardware Trojan with a much higher success rate and smaller footprint than the example discussed in the original work. Our analysis was carried out on the data encryption standard (DES), the cryptographic circuit based on the Feistel structure, which was used as an example in the original work.

In order to demonstrate more broadly that cryptographic ciphers are not protected by the method of split manufacturing, we also inserted a hardware Trojan into the advanced encryption standard (AES) after this method of split manufacturing had been applied. AES was chosen because it utilizes a substitution-permutation network as its structure, instead of the Feistel structure used by DES. Our analysis and results from simulations show that the methods described in [1] to obfuscate cryptographic ciphers as a class of redundant circuits do not create the intended amount of security. We do not discuss the detectability of the

hardware Trojans used as this was not discussed in the original work. We only show vast improvement over the metrics used in [1].

3.1 Introduction

Hardware Trojans have been a topic widely discussed and analyzed recently [8–13]. They are malicious modifications made to a circuit which can leak confidential information such as secret keys, or cause a circuit to malfunction [10]. A hardware Trojan must be inserted at some point during the development of an integrated circuit (IC). There are four possible points of insertion during this cycle: the specification phase, the design phase, the fabrication phase and the assembly phase [14]. Hardware Trojans present a potential security concern to anything that utilizes integrated circuits. The U.S. Department of Commerce has estimated that counterfeit electronic components have appeared in 39% of the Department of Defense supply chain [15]. Such counterfeit parts have been discovered in Navy helicopters and Air Force planes [16]. Thus, there exists a very real possibility that ICs that include malicious logic designed to leak critical information or make systems fail could make their way into either military or civilian systems.

In an effort to thwart malicious actors, researchers have discovered new forms of Trojans and ways to defeat them [8]. Others have concentrated on finding a way to detect hardware Trojans in a compromised circuit [17,18]. Another approach taken to mitigate these threats is to harden circuits against hardware Trojans. This approach designs the circuit in a way which increases the difficulty of successfully inserting a Trojan. An example of this approach was proposed by Imeson et al. [1] whereby a wire lifting procedure, coupled with 3D fabrication, was used to obscure the target circuit from the attacker. This method concentrated on obfuscating the circuit— if an attacker could not distinguish different parts of the circuit from one another, they would not be able to insert an effective hardware Trojan. This procedure provided a novel method to secure a circuit against hardware Trojan implantation at the foundry. The remainder of this work will focus on the paper by Imeson et al. [1].

The paper in question by Imeson et al. had promising results and presented a novel

method of securing circuitry from hardware Trojan implantation [1]. This is an important paper in the field of 3D circuit obfuscation because it provides strong theoretical proofs [2], unlike most of the literature on the topic, including more resent papers, which merely provide empirical results [2–7]. For this reason it is important to point out flaws in the methods as it is a foundational work in the field.

The paper made contributions that have applicability to many types of circuits. However, the claims of increased security made were too broad in the applicability of the wire lifting procedure. This was made obvious by the provided example of a DES circuit. We do not dispute the claims of increased security for all circuits generally, but we do dispute them in that they do not apply to redundant circuits (defined in Section 3.3.3). We hypothesize that there exists a class of circuits for which the methods proposed by [1] do not provide the claimed security. This class of circuits is redundant circuits, with cryptographic ciphers being a subclass of redundant circuits.

We present methods to bypass the defense of [1] that we believe to be applicable not only to DES and AES but also other redundant, pipelined circuits, which are widely used in cryptography; e.g., the Feistel structure and substitution-permutation networks (SPN) [19–23]. Such ciphers and cryptographic devices are known to be vulnerable to fault injection attacks [24, 25], to which our attack is a Trojan-based variant.

This paper is outlined as follows. Our specific contributions comprise Section 3.2. Section 3.3 provides some background on 3D circuit obfuscation, the fault-style attack under consideration and defines the term redundant circuits, as used by us. The motivation for our work is set forth in Section 3.4. Section 3.5 discusses the threat model and sets forth several weaknesses in the 3D obfuscation model, specifically pertaining to DES (Feistel structures). Section 3.6 demonstrates how these weaknesses in 3D obfuscation extend to AES (SPN). Finally conclusions and future work are put forth in Section 3.7.

3.2 Contributions

We allocated a significant amount of time to reproducing the results from the wire lifting procedure found in [1], running the author's publicly available code and their examples.

This is itself a contribution because we were unable to reproduce their results, and this is something that the community should know about. We were able to reproduce their results for small circuits, however for the larger example they provide of the DES circuit, we could not. The code ran for months and never completed. This was the case even after being in contact with the authors. This is discussed further in Section 3.4. With this limitation in mind, we make the most pessimistic assumptions in our work, i.e., we consider the most secure possible outcome for the result of the wire lifting procedure. The following is a list of additional contributions found in this work:

1. A discussion demonstrating how redundant circuitry weakens the defenses proposed in [1].

2. A demonstration showing that the wire lifting procedure from [1] does not provide the intended amount of security to highly redundant circuits, particularly cryptographic ciphers. This is shown on both DES and AES circuits. These circuits were chosen to show that the wire lifting procedure does not frustrate the implantation of a hardware Trojan for both a Feistel structure or an SPN.

3. An approach which allows a hardware Trojan to be inserted in a DES circuit that has undergone the wire lifting procedure is explained. This is a modification to the Trojan described in [1]. This new approach entails attacking all portions of the circuit that are indistinguishable from one another at the same time, instead of choosing one portion and only attacking it, or attacking each portion one at a time.

4. A second approach is introduced which allows a hardware Trojan to be inserted in an AES circuit that has undergone the wire lifting procedure. This method is similar to the approach outlined for inserting a Trojan in a DES circuit, but does not attack every indistinguishable portion of the circuit at the same time. It attacks enough of the indistinguishable portions of the circuit to be able to recover the key through an exhaustive search. This allows the size of the Trojan to be less than it would have to be if every portion of the circuit indistinguishable from another were to be attacked.

This method can also be applied to a DES circuit, increasing the probability of success of against a DES circuit to 100%.

3.3 Background

In this section, a brief background on 3D obfuscation is presented and followed by a short explanation of fault injection analysis. Then the term redundant circuits is defined as used in this work.

3.3.1 3D Obfuscation

Imeson et al. introduced a wire lifting procedure to select wires to lift to the trusted tier for split manufacturing. The wire lifting procedure is a greedy heuristic to make individual gates or groupings of gates indistinguishable from one another [1]. The gates left on the untrusted layer can even be scrambled in space to remove any hints of what they may be used for. Hence, unlike in a traditional single-layered layout, the locality of the gates has no correlation to their connections. If the outputs of two `AND` gates were connected to the inputs of another `AND` gate these gates would very likely have close proximity to one another. However implementing this method, the three gates may occupy three different corners of the chip. Without the traces between them to identify their functionality an attacker would not be able to tell how the gates are connected and therefore could not implement a Trojan in the right place to create the intended adverse result.

A gate is *k-secure* when there are $k-1$ other gates in the circuit that are indistinguishable from that gate. Imeson et al. define the *k-security* of the circuit as each gate in the circuit being at least *k-secure*. As an example, after the wire-lifting procedure, if there was a set of gates that was 3-secure and the remaining gates were 5-secure, the *k-security* of the circuit would be 3-secure, as each gate is at least 3-secure. Depending on the target *k-security* of a circuit, more or fewer wires may need to be lifted. An illustration follows:

Figure 3.1 shows an illustration of lifting wires to create *k-security*. Graph 1 represents a circuit in the original state with inputs `A`, `B`, and `C` and outputs `D`, `E`, and `F`. Graph 2

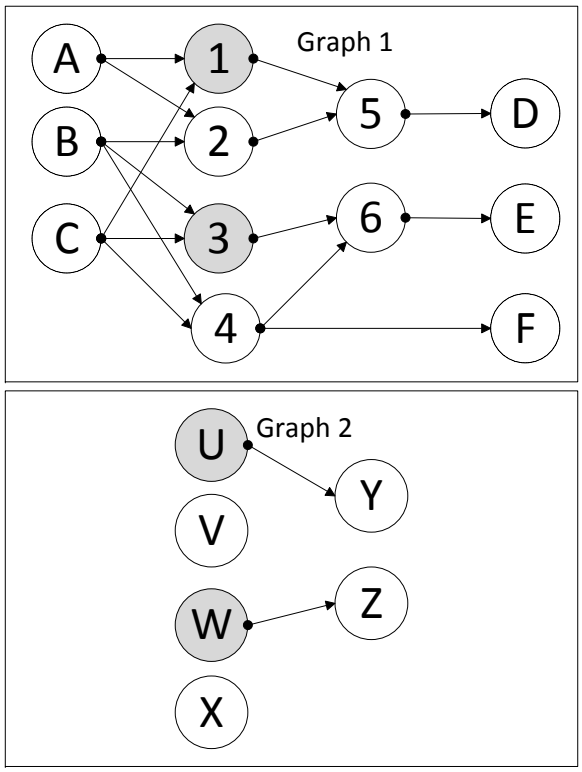


Fig. 3.1: Wire lifting example. Graph 2 has a *k-security* of 2, as each subgraph has at least one other that is identical to itself.

represents how the circuit would look after a wire lifting procedure to make the circuit 2-secure. Notice that the inputs and outputs are removed in Graph 2, as those wires have been entirely removed. In this example each gate or subgraph is indistinguishable from *2-1* or one other gate or subgraph making the circuit 2-secure. It is unknown if node `V` in Graph 2 represents node `2` or node `4` from Graph 1. Also, it is unknown if the subgraph of node `U` to node `Y` is the same as the subgraph of node `1` to node `5`, or node `3` to node `6`.

3.3.2 Fault Injection Analysis

The method of circuit obfuscation by wire-lifting set forth in [1] is said to protect a DES circuit against a hardware Trojan which would create a fault attack on the LSB of the 14th round. This technique of discovering the secret key by fault injection on the output of the 14th round or input of the 15th round is set forth in [26]. The technique is as follows: a known fault on the output of the 14th round propagates through the 15th and 16th rounds.

The attacker deduces the DES S-boxes that were affected by the fault by comparing the corrupted cipher text with a previously captured uncorrupted cipher text. Then through a series of trial and error the attacker guesses the values of the S-boxes and uses these guesses to create a possible round key for the 16th round. This round key can then be reversed through the DES key schedule to attain a DES secret key with 8 missing bits. These final bits are then searched in a brute force manner by running the DES algorithm with possible full keys and the plaintext that should give the uncorrupted cipher text. This process is continued from the S-box guesses until a possible full key yields the expected cipher text from a known plaintext, at which point the secret key is known.

3.3.3 Redundant Circuits

We define a redundant circuit to be a circuit in which logic is duplicated multiple times, separated by pipeline stages. The methods set forth in [1] are very good at creating portions of circuits that are indistinguishable from one another, but perhaps the consequences of these methods were not considered in the case of them being applied to circuits which are redundant in nature. Redundant circuits run through the wire lifting procedure would be subject to one of the following two outcomes, assuming the same number of wires are lifted from each circuit:

1. Each redundant portion of the circuit will be identical to one another after the wire lifting procedure.

2. The *k-security* of the circuit will be greatly reduced.

Figure 3.2 shows the graph representation of a redundant circuit. This is Graph 1 taken from Figure 3.1 duplicated through a pipeline stage. It has two redundant portions represented by nodes `1-6` and nodes `7-12`. If we remove the inputs (`A`, `B` and `C`) and the outputs (`D`, `E` and `F`) as well as the pipeline stage and allow the circuit to undergo the wire lifting procedure Figure 3.3 and Figure 3.4 are two possible results. Figure 3.3 aligns with item 1 from above. The two redundant portions of the design have had identical wires lifted. If a single redundant portion was examined by itself, it would look identical to Graph 2 taken

from Figure 3.1 and would have a *k-security* of 2. However, the circuit as a whole has a *k-security* of 4, each subgraph is indistinguishable from 3 other subgraphs.

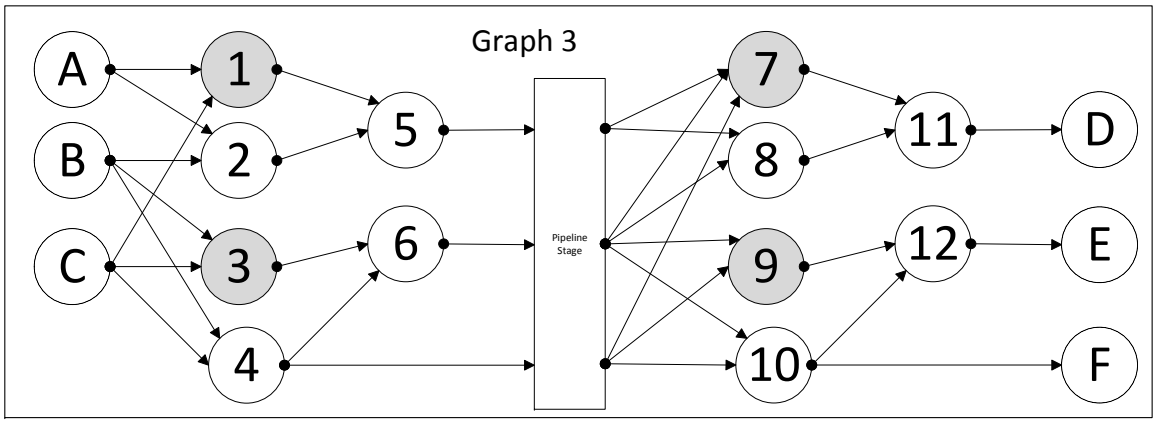


Fig. 3.2: An example of a redundant circuit with two redundant portions and a pipeline stage separating the two.

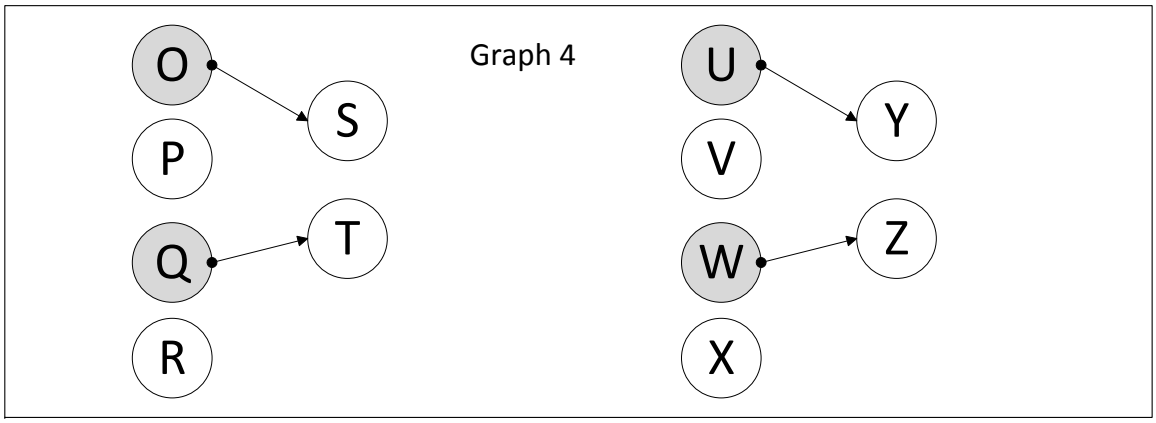


Fig. 3.3: A redundant circuit which has undergone the wire lifting procedure and each redundant portion is identical to each other. The *k-security* of the circuit is 4.

Figure 3.4 aligns with item 2 from above. In this case, after the wire lifting procedure the two redundant portions of the circuit are not identical to one another, but it has the same number of wires lifted as Figure 3.3. Each redundant portion of the circuit when examined independently has a *k-security* of 2 and the circuit as a whole also has a *k-security* of 2. The only way to increase the *k-security* of Figure 3.4 to 4 would be to remove the

remaining wires. Minimizing the number of wires lifted is important because as the number of lifted wires increases, so increases the power consumption, delay, and area of a circuit [1].

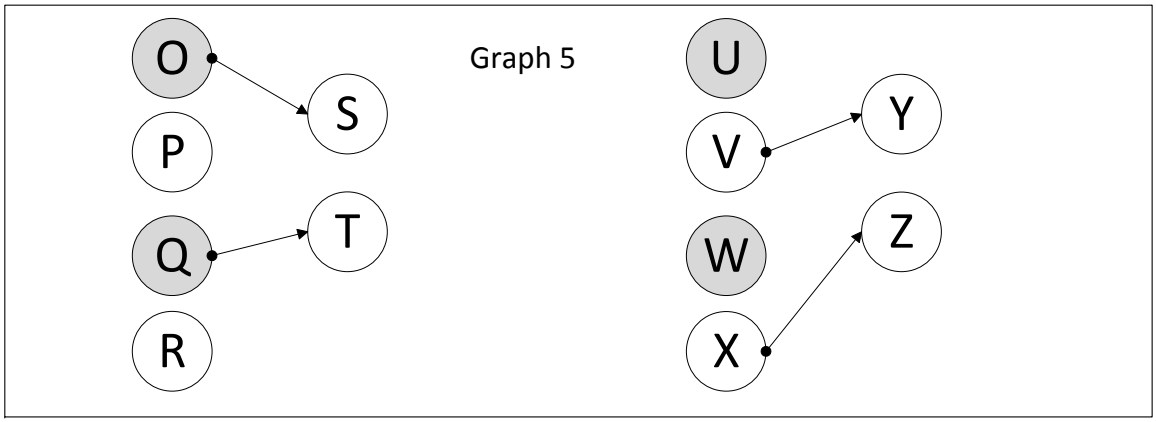


Fig. 3.4: A redundant circuit which has undergone the wire lifting procedure and the two redundant portions are not identical to each other. The *k-security* of this circuit is 2.

This example illustrates the security concern that redundant circuits introduce. If an attacker wanted to modify the circuit represented by Graph 3 in Figure 3.2 so that the output D was always held to `0`, they would need to attack node 11. The DES circuit example in [1] suggested that a successful Trojan would need 5X the number of gates as the *k-security* of the node to be attacked. But in fact, this could be done with exactly the number of gates equal to the *k-security* of the node to be attacked. In Figure 3.3 an `AND` gate with one input held to `0` once triggered, would be attached to nodes `S`, `T`, `Y`, and `Z`. This would guarantee that output `D` was held to zero, and would require exactly four `AND` gates.

3.4 Motivation

With a work as highly regarded in the community as [1], it is important to point out shortcomings. Although we are not attempting to discredit this work as a whole, it is important to point out the limitations that it has so that the methods described are not erroneously applied to a circuit for which the promise of increased security cannot be fulfilled.

In the redundant circuit example given in the paper, DES, the claim is made that after the methods from [1] were employed, an attacker could either place a small Trojan in the

circuit with a 1/256 chance of success, or place a large, 1280 gate Trojan in the circuit with a 100% chance of success, but the number of plaintexts would increase by 255×. We present another possibility: a hardware Trojan containing only 256 gates. Simulations show a 75% chance of recovering the secret key with only two plaintexts presented using this Trojan. This is a 191× improvement over the success rate of the small Trojan presented, and at the same time, the Trojan comprises 5× fewer gates and requires exactly as many plaintexts as the small Trojan, not the 255X number required by the large Trojan.

We believe that the example of a cryptographic cipher was a poor choice for the wire lifting procedure not only because its size makes it time prohibitive to apply the method to (as discussed in Section 3.5), but also because these circuits are highly redundant in nature, which allows for effective fault-style attacks. DES has 16 identical rounds making it fit into the class of highly redundant circuits. It has been broken using differential fault analysis on early, middle and late rounds [26]. That is, even if a specific bit cannot be targeted because of the protections the wire lifting procedure provides, the circuit could still be compromised by a hardware Trojan. So long as faults can be induced in the DES circuit, even if the location of the Trojan or the round in which it was implanted in is not certain, the secret key can still be discovered (as shown in Sections 3.5 and 3.6).

Furthermore, we attempted to generalize the attack by investigating another highly redundant cryptographic cipher, the Advanced Encryption Standard (AES), which uses a different cryptographic structure (SPN) from DES. We show that for AES the wire lifting procedure again does not provide the claimed security as the the secret key can be recovered using at most six plaintexts (Section 3.6). Specifically, We show that with a Trojan of 640 gates the key can be recovered 53% of the time, 700 gates 89% of the time and an 800 gate Trojan would be able to recover the key 99.9% of the time. These Trojans are still 50%, 45% and 37.5% smaller than the style of Trojan suggested for DES (and adapted to AES), and they use 97.6% fewer plaintexts in the worst case.

We also feel it important to point out the difficulties we experienced attempting to reproduce the results of the DES example given in [1]. We were unable to perform the

wire lifting procedure as outlined in the original work. We used the code base and examples from [1] that were made available to the public [27]. We have had a varying degree of success using this code. Over the course of several years, we have been in contact with the authors of the original work, who were at first very helpful in aiding our efforts to reproduce their work. They fixed errors in the code base and added files that were needed to build the code but were not originally included. The small example from the `README` provided does work, for instance. However, for the more complex circuits that are included with the code base, e.g., the DES circuit, the wire-lifting procedure never successfully terminated.

One of the points in the original paper was that the wire lifting procedure is scalable, e.g., can be used on a DES implementation. We believe that our failure to produce a *k-secure* DES circuit, using the code provided by the authors of [1], seems to indicate a significant weakness of the original work. That is, although the wire lifting procedure, as implemented in the authors' code, works on smaller, simpler circuits, we have not been able to have a partitioned portion of the DES circuit complete the procedure. Even after gaining access to the cluster resources at the University of Utah and running the code for three weeks, the wire lifting procedure made little progress. We estimated that given the progress it would take additional years to complete. On another virtual machine with the ability to run 24 threads and 128 GB of RAM, the example DES wire lifting procedure ran for over 135 days without finishing or showing significant progress. According to the authors, when queried, such computational resources should have been sufficient but they were unable to provide a time frame for completion.

3.5 Weaknesses of 3D Obfuscation on Redundant Circuitry

The following section describes the threat model, the specific DES circuit that was attacked, and the attacks which demonstrate that the wire lifting procedure set forth in [1] does not provide the the level of security that it attempts to because of the redundant nature of the DES circuit.

3.5.1 Threat Model

As in the existing work [1], we assume that the Trojan is inserted during the fabrication phase. In addition, the attacker has full knowledge of the original circuit. Also, a trusted party has performed the wire lifting procedure as described earlier, and manufactured the trusted tier. However, the attacker does not have a knowledge of the results of wire lifting procedure. Hence, if the attacker intends to change the behavior of the circuit, they can only do so with a one in k chance of success, where k is the *k-security* of the circuit. This is because the attacker will not be able to differentiate any gate between itself and $k-1$ other gates.

We assume a pipelined DES circuit on an application-specific integrated circuit (ASIC) as explained in Section 3.5.2. The attacker is able to probe the device with plaintext challenges that the attacker chooses and observe the cipher text outputs. The attacker is also able to trigger the Trojan. This will allow the attacker to learn the secret key.

3.5.2 The DES Circuit

We begin by discussing the DES circuit.. DES has 16 rounds of logic that are identical to one another [26], hence it is a redundant circuit. DES was used in [1] as a demonstration of the difficulty an attacker would have implanting a successful hardware Trojan into a design that had undergone the wire lifting procedure. Although the specific details of the implementation of DES used were not given, the circuit is described as having approximately 35,000 logic gates. This matches well with a pipelined implementation of DES. Since a *k-security* of 16 is achieved by simply removing the interconnects between the rounds, we assume that each round is indistinguishable from any other round in the circuit. After the wire lifting procedure the final circuit is 64-secure, and the specific bit given as an example of an attack, the LSB of the 14th round, is in fact 256 secure.

The following assumptions are also made in this work because the exact implementation used in [1] was not obvious: a full encryption or decryption will take place with every clock cycle regardless of the delay through the complete circuit. Also the minimum number of pipeline stages would be 17, one between each round and one for the inputs and another for the outputs.

3.5.3 Attack Outline

Imeson et al. gave only two options for a Trojan to attack the LSB of the 14th round [1]. Either the attacker could choose one of the 256 possibilities and have a 1/256 chance of success, or attack each of the options one at a time in a multiplexed attack which would yield a large Trojan. However, we suggest that an attacker might attack each of the 256 bits all at once, holding them at zero simultaneously for a single clock cycle. This new attack would have two effects. The first effect is that it would hold the LSB of round 14 to zero for a particular plaintext that is in flight through the circuit. It would also hold 15 other bits to zero for the 14th round. The LSB output of any round is 16-secure within each of its rounds viewed independently. The other effect is that each of the other rounds will have bits that are also held to zero. However, as this is a pipelined design the bits held to zero in the other rounds will have no effect on the ciphertext output in question. This new design for the Trojan is compared to the original in Figure 3.5. The removal of the multiplexer decreases the size of the Trojan dramatically, from 1280 gates required down to 256.

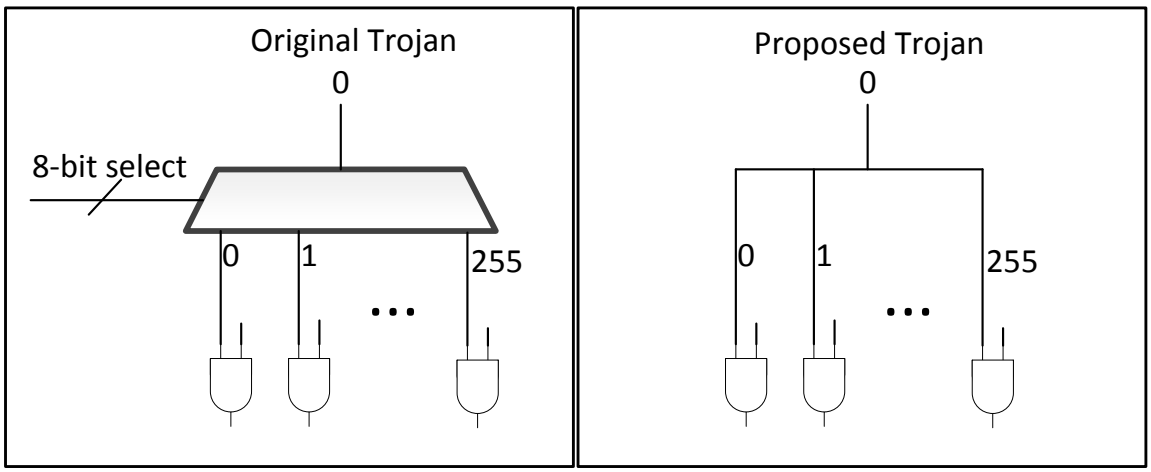


Fig. 3.5: The figure on the left shows the original Trojan which requires 1280 gates. The figure on the right shows a smarter Trojan which requires 256 gates, numbered 0 to 255.

Note that even though each plaintext that is being processed by the crypto device will be corrupted by the hardware Trojan forcing its bits to zero, the plaintext that is corrupted at the end of the 14th round will not be corrupted in previous or later rounds. As this cypher

text is identified at the output of the crypto device it will contain information needed to determine the secret key of the system.

A plaintext is presented at the inputs and cipher text exits the circuit at the outputs on every clock cycle, so with 16 rounds, there can be 16 unique plaintexts being encrypted at once, each in a different round. Figure 3.6 demonstrates corruption occurring in 3 separate rounds of a pipelined implementation of DES. Assume the corruption in rounds 13, 14 and 15 occur on the same clock cycle, t. Then at t+1 an uncorrupted cipher text will emerge from round 16. At time t+2, a corrupted cipher text will emerge with corruption15, at t+3 a corrupted cipher text will emerge with corruption14 and at t+4 a corrupted cipher text will emerge with corruption13.

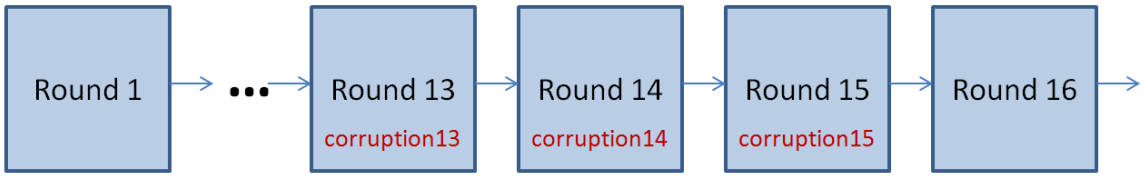


Fig. 3.6: DES corruption example. In this pipelined implementation of DES there are 16 unique encryptions occurring at the same time. If a there is corruption in rounds 13, 14 and 15 all at the same time then there will be three corrupted cipher texts. One cipher text will have been corrupted by corruption15, one by corruption14, and the third by corruption13.

3.5.4 Attack Implementation

We will discuss two attacks. A pipelined implementation of DES was obtained from opencores.org [28] in order to be unbiased, and used for these attacks. The design obtained was unmodified with the exception of inserting the hardware Trojan into the design.

In our simulations we added the ability for every bit to become corrupted, or held to zero if the Trojan was activated. Using the wire lifting procedure, the number of places in the circuit that would have been indistinguishable from the LSB of the 14th round would be known. To be conservative, we were generous in the number of locations in the DES circuit that may be indistinguishable from the LSB of the 14th round. Namely we allowed

every location a possibility to be included in the Trojan, resulting in the ability to corrupt locations which could make it more difficult to recover the key that would not have been included if the wire lifting procedure was completed. With 10,000 runs we randomly selected bits to hold to zero along with the LSB of the 14th round.This was done in lieu of the wire lifting procedure.

Attack One: Limit Scope to the Round 14

The first attack entails attacking a single round in the unbiased DES circuit. We identified the 14th round and attached an `AND` gate to the LSB output of that round. When triggered, this `AND` gate will hold that output bit to zero. If we assume that the entire circuit is 256-secure and there are 16 identical rounds, then if the LSB has 255 other gates that look exactly like it after the wire-lifting procedure then the 14th round would have 15 other gates that are indistinguishable from it. This amounts to 16 bits per round. (This assumption is based on the statement from page 12 of [1] which states, "We note that a security level of 16 is obtained in the first few rounds of partitioning by removing 13% of the wires, i.e., all wires that lie between successive DES rounds." [1]) Therefore, in addition to the `AND` gate placed on the LSB of the 14th round to hold it to zero we also randomly selected 15 other locations where a value would be held to zero by an `AND` gate triggered at the same instant as the gate tying the LSB to zero.

The DES circuit was modified in the following manner. In the original circuit there was a single description of a DES round written in several Verilog files. These files were duplicated in order to differentiate the 14th round from the other rounds. The duplicated 14th round was modified to include `AND` gates associated with every bit of the computation. These `AND` gates are associated with each bit of each input and output as well as the inter-round logic and each s-box lookup. In total, 2240 additional `AND` gates were added to the 14th round and attached to an individual trigger which resulted in a 2240 bit bus at the top level of the circuit hierarchy. The default value for that bus was set to `1` for each bit in the test-bench used to simulate the circuit. A value of `1` on the second input of an `AND` gate would not cause any change in the output. When the simulation is run, the test-bench

contains code to randomly select 15 of those 2240 bits to tie to zero as well as the LSB output of the 14th round.

We selected 15 random locations to provide an unbiased simulation of what gates an attacker might be faced with in a 16-secure round. Only 15 random locations were needed in the 14th round because the other 240 other locations that would be indistinguishable from the LSB of the 14th round are corrupted as well, however, they are in different rounds which cause corruption to different plaintexts being encrypted. Sixteen total corrupted plaintexts would result, but we are only interested in the corrupted plaintext resulting from the corruption in the 14th round.

Attack Two: Allow any bit to be corrupted

In an effort to expand our findings, we designed a second attack. Although we felt that our assumptions used in the first attack (that if the LSB of the 14th round was 256-secure, that it would be 16-secure inside of each round) were sound based on the wording in [1], we were not able to verify by successfully running the pipelined DES circuit through the wire-lifting code. Thus a second, more restrictive, attack would confirm the weakness of the wire lifting procedure.

This second attack extended the first attack to each of the 16 rounds to be more broad. As the LSB of the 14th round is said to be 256-secure, we randomly selected 255 bits from anywhere in the circuit and held those bits to zero in the same clock cycle that we held the LSB of the 14th round to zero. Note, there was other logic in a pipelined DES circuit that was outside of the round logic, such as the key scheduler. The setup for this attack included all logic in the design, not just the logic found within the 16 rounds. As was done to the 14th round, each bit of the other rounds as well as all additional logic in the design was associated with an `AND` gate. One of the inputs to this `AND` gate was the original logic from the circuit. The other input was fed from the hierarchy of the Verilog design from a bus at the top level. This bus was used to control each of the `AND` gates. When this bus drove a `1` on each of its bits the DES circuit functioned normally. When a `0` was driven on any of the bits of that bus, the associated `AND` gate would have an output of `0`. The output of the

logic in the affected area of the circuit may have already been a `0`, in which case no effect would be seen. However, if the output should have been a `1` then this may cause an error in the DES computation.

For this second attack the bus at the top level which controlled each of the added `AND` gates was 36,823 bits wide. We take this to be a reasonable comparison to the DES circuit referenced in [1] which contained approximately 35,000 gates. The attack procedure is very similar to the first attack. The 36,823-bit bus is initialized to `1` to allow for normal operation of the circuit. The DES circuit is allowed to run for a set amount of time and then 255 bits of the 36,823 bit bus are held to `0` along with the LSB output of the 14th round for a single clock cycle.

Each simulation represents a single implementation of Trojan logic. The actual Trojan circuit would not contain 36,823 bits but instead only the 255 bits in addition to the LSB output of the 14th round. These simulations will help us determine the likelihood of success of an attack on a DES circuit that has undergone the wire lifting procedure.

3.5.5 Attack Results

For each of the two attacks, the plaintext input always remained constant to reduce the number of variables. Both the corrupted and uncorrupted cipher text outputs were collected from the Verilog simulations. These outputs were analyzed using the fault injection analysis methods described in [29] and expanded in [26]. A C program was written to automatically implement the methods identified in [26] and determine whether or not the secret key could be identified.

These methods created several guesses for the round key of the 15th round. Each guess contained 48 of the needed 56 bits of the secret key. Then the remaining 256 possibilities for the remaining 8 bits were exhaustively searched through a software DES implementation. If the cipher text resulting from the key guess matched the uncorrupted cipher text output then the key that was used to create it was indeed the original key.

For the first attack, Figure 3.7 displays the results. Out of the 10,000 simulations run, there were 7,554 that resulted in a key that was easily recoverable. The results of

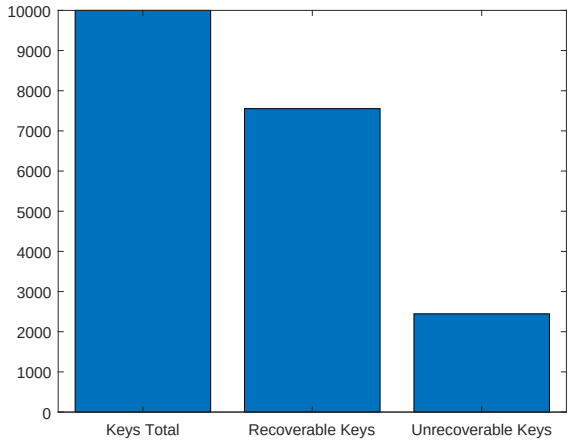


Fig. 3.7: Attack 1 Results.

the second attack can be seen in Figure 3.8. Out of the 10,000 simulations run there were 4,437 recoverable keys. Truly, the completely random nature to which we selected the 255 points in the second attack may have actually biased us towards failure. It is likely that in the actual obfuscated circuit the LSB of the 14th round would not have 36,823 other possibilities to be confused with. Still, a success rate of approximately 75% (191/256) and 44% (112/256) is far better than the 1/256 chance, or 0.4% success rate that was discussed in [1].

The other alternative that [1] offered was a guaranteed success rate, but a Trojan that was 1280 gates large not including any trigger logic. While the existing work [1] did not discuss an acceptable Trojan size, the Trojan proposed here would be 256 gates not including any trigger logic. This Trojan is 5x smaller in size, which is again a substantial improvement.

3.5.6 Discussion

The results above show a significant improvement using the metrics that the authors of [1] used. However, these results would have been even better if we allowed ourselves to make further assumptions. We did not allow this in our experiments because we were not able to verify using the wire lifting procedure. However, when the actual architecture of the circuit is considered, the bits that will be indistinguishable from the LSB of the 14th round are output bits 1-15 of that same round. The 240 other bits in the other rounds that would

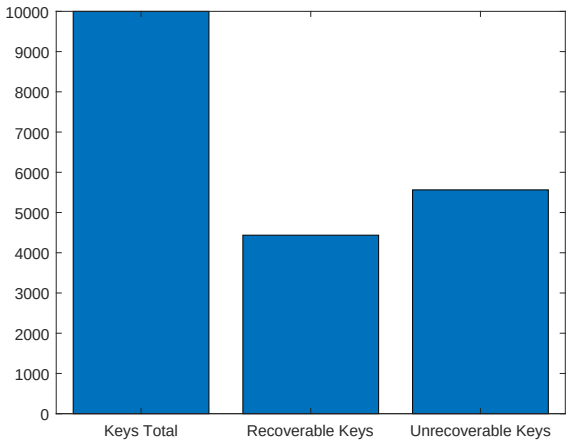


Fig. 3.8: Attack 2 Results.

be indistinguishable from the LSB of the 14th round are output bits 0-15 of each of the other 15 rounds. If this assumption was allowed to be made, the success rate of the Trojan would be 100%. Fault injection analysis teaches that the greater the number of known bits that have a fault induced on them, the easier it is to recover the key [26].

3.6 Weaknesses extended to AES

Hardware Trojan attacks can also be created to make AES circuits vulnerable, even after the wire lifting procedure has been performed. The example of AES being vulnerable is important because it extends the previous attack to another class of cryptographic schemes (those based on SPN). For our purposes, 128-bit AES will be considered, but the attack can be extended to 192 and 256-bit AES, as well. It must be reiterated that because the publicly available wire lifting procedure could not be reproduced, several pessimistic assumptions were made.

3.6.1 AES Attack Background

The original proposal for AES was submitted to the National Institute of Standards and Technology in 1999 which describes the Rijndael algorithm, a symmetric block cipher [30]. It was adopted as a standard in 2001 [31]. The standard allows for the processing of 128 bit data blocks with an option of key sizes of 128, 192 or 256 bits. The 128-bit data is organized

into a 2D array of 4x4 bytes called the State. For a 128-bit key operating on 128-bit data, AES has 10 rounds. Rounds 1-9 are all identical with their State manipulation steps as follows [31]:

1. Perform byte substitutions.
2. Shift the rows of the matrix.
3. Perform the mix column transformation.
4. Add the round key to the result.

The final round, round 10 performs steps 1, 2 and 4, but leaves out step 3, the mix column transformation. This is an important distinction from DES which has 16 identical rounds [26]. This difference is important because in [1] the idea of manually partitioning the circuit and removing the wires between the rounds is the first step of the procedure, which creates a *k-security* of 16. With AES, if the connections between the rounds are broken, no such *k-security* is achieved because the final round is not identical to the others. With this in mind, we would suggest partitioning the AES circuit in a way that would allow for the greatest *k-security* as was done in the example of DES. We submit that the best case is for removing the wires between all rounds, as well as between steps 2 and 3 and between steps 3 and 4 before performing the wire lifting procedure, as seen in Figure 3.9. This is important to do, otherwise the final round would be easily attacked as it would have a different footprint than all the other rounds.

After this partitioning procedure is completed the final round circuit of SubBytes to ShiftRows, as well as the AddRoundKey circuit, is 10-secure, as there would be nine other rounds in which those identical circuits existed. We would now pass the partitions through the wire lifting procedure with a goal of having each circuit be x-secure. Note that in order to add anonymity to the final round, we have isolated the AddRoundKey circuit which is by default 128-secure in each round (this is because that circuit is made up entirely of 128 `XOR` gates). The whole circuit, then, is in fact 1280-secure. There are 1279 other gates in the

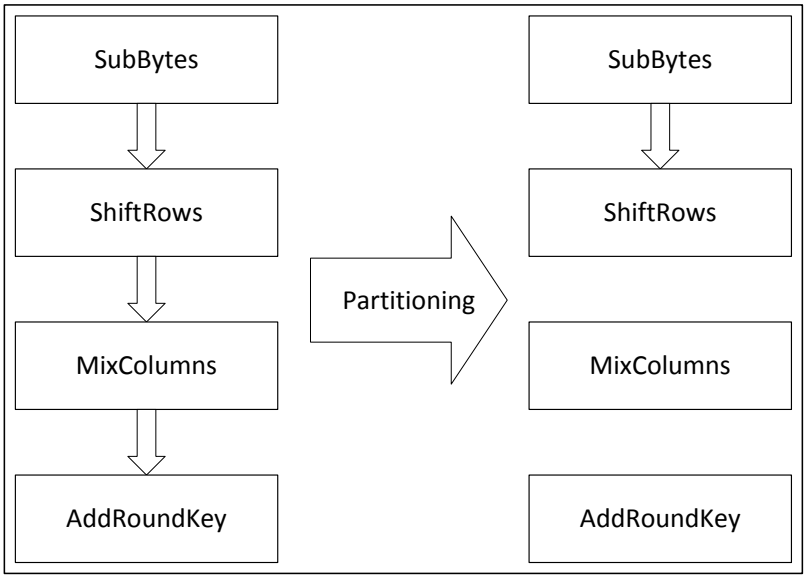


Fig. 3.9: After partitioning the rounds to remove wires between their components, there would be nine identical MixColumns circuits, ten identical circuits of SubBytes followed by ShiftRows, and ten AddRoundKey circuits.

circuit which cannot be distinguished from a particular `XOR` gate in the final AddRoundKey circuit. 127 of these 1279 gates also exist in the final AddRoundKey circuit.

3.6.2 AES Attack Outline

In order to recover the 128-bit key of AES, we propose to attack the circuit in the final round during the AddRoundKey step (when the round key is added in). That is, if we can implant a hardware Trojan that can cause a fault that holds a bit to `0` in at least 64 of the 128 bits that are `XOR'd` with the round key, those bits of the round key can be revealed. The final bits of the key (up to 64 bits) can be brute forced by an exhaustive search until the full key is recovered. After the attack the remaining bits of the round key would be guessed, and the round key would be propagated through the AES key schedule in reverse to reveal the key. That key would be used to encrypt a known plaintext-ciphertext pair. If the ciphertext encrypted under the guessed key matched the original ciphertext then the guessed key is correct. If not, then the process starts over by guessing the remaining bits in the final round key, again.

This attack uses the properties of the XOR operation: A XOR B = B where A = 0. Any bit of the final round key will be revealed if it is XOR'd with 0. If our hardware Trojan affects random bits that are XOR'd with the final round key, we do not need to attack specific bits. In fact, we do not even need to know how many bits were attacked, we only need enough plaintexts to recover all the bits. The following is an illustrative example.

Let us consider the least significant byte of the final step, adding the round key, of the final round of an AES encryption. Assume that two bits of this byte are held low when a hardware Trojan is active but it is not known which bits are affected by the Trojan. Given that enough ciphertext pairs (C,C'), where C is a correctly encrypted ciphertext and C' is a cipher text encrypted while the hardware Trojan was active, and they are both the encryption of the same plaintext, we can determine which bits are associated with the Trojan and the value of those bits of the round key. For example, assume that the least significant byte of the round key (from here on referred to as "the key") is 10101010 and the least significant byte of the State (from here on referred to as "the State") being XOR'd with the key is 11110000. In this case, C = 01011010 and C' = 01001010. The errored byte *E* is calculated by C XOR C', which in this case yields 00010000, where a 1 indicates a bit that was affected by the Trojan.

Now, using *E* as a mask over C' the result is: 0100101. This indicates that bit 4 (starting with 0 on the right) of the round key is 0 and we have recovered one of the two bits. Continuing this example, after several more ciphertext pairs, in which *E* = 00000000, we find the pair (00101101, 00101111) and *E* = 00000010. Again using *E* as a mask over C' to select the revealed portion of the key, this indicates that bit 1 of the key is 1. At this point we have recovered both bits of the round key that this Trojan allows; i.e., we discovered that the round key looks like --0-1-. More bits affected by the Trojan would have revealed more key bits.

The attack above is similar to the attack on the DES circuit in that the Trojan consists of AND gates designed to hold the bits they affect to 0 when activated. Like the attack on the DES circuit, a pipelined AES implementation will be used. The entire Trojan will be

activated at the same time, for a single clock cycle, so if gates of the Trojan affect bits in rounds besides the final round, they will not change the ciphertext output that we are concerned with. Only the gates of the Trojan that lie within the final round will cause any change to that ciphertext.

3.6.3 AES Attack Implementation

A pipelined implementation of AES was obtained from opencores.org [32], in order to start with an unbiased implementation. This design also came with a test bench that was utilized for simulation. The circuit and test bench were modified only to add the hardware Trojan; in no other way was the circuit tampered with.

The *k-security* of any of the `XOR` gates in the final round during the AddRoundKey step is 128 with respect to that round and 1280 with respect to the entire circuit. If our intention was to attack a specific `XOR` gate we would need a hardware Trojan to contain 1280 `AND` gates to be sure the crucial gate was attacked. Instead, we only want enough gates to be attacked so that we can recover the key. We claim that at least 64 of these `XOR` gates in the final round must be attacked, leaving up to 64 bits of the key to be exhaustively searched. In order to decide how large to make the Trojan, we simulated different Trojan sizes 10,000 times. For each simulation we kept the number of Trojan bits constant, but randomly selected their locations. This is akin to seeing the netlist that the attackers have access to but not knowing which of the 1280 `XOR` gates fall within the final round. Instead of attacking each gate the attacker would select a number of them at random to attack. These simulations are used to calculate the probability of success with varying Trojan sizes. The sizes of Trojans simulated were 640 bits, 700 bits and 800 bits.

A Python program was written to generate the 10,000 locations of each bit of the Trojan for each of the three Trojan sizes. The Verilog code was modified to incorporate these locations and the resulting circuits were simulated with the `(C,C')` pairs being written to a file. Those `(C,C')` pairs were then analyzed by another Python program to determine the number of bits of the round key that were discovered as well as how many pairs were needed to find those bits.

3.6.4 AES Attack Results

640-bit Trojan

The 640-bit hardware Trojan was selected as a starting point as it was half as large as the 1280 which would affect all gates. The results show that in 5340 of the 10,000 simulations at least 64-bits of the key were recovered. The maximum number of round key bits recovered was 85 and the minimum number recovered was 43. The probability of success for this Trojan size was 53.4%. All bits of the round key were recovered using at most six `(C,C')` pairs. These results can be seen in Figure 3.10.

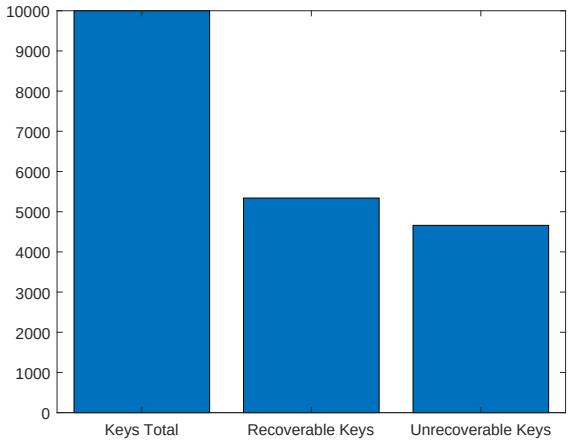


Fig. 3.10: The numbers of recoverable vs. Unrecoverable keys are displayed for the 640 gate Trojan.

700-bit Trojan

A small increase in Trojan size revealed an increased probability of success. 8938 simulations resulted in 64 bits or more of the round key being revealed. This is an 89.38% chance of success with this Trojan size. The maximum number of round key bits recovered was 92 and the minimum was 52. These bits were again found using at most six `(C,C')` pairs. These results can be seen in Figure 3.11.

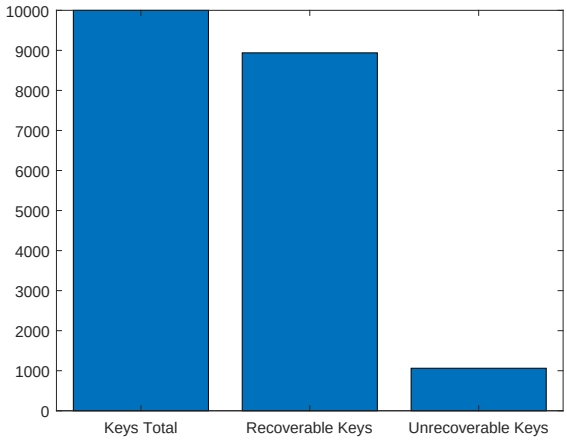


Fig. 3.11: The numbers of recoverable vs. Unrecoverable keys are displayed for the 700 gate Trojan.

800-bit Trojan

This final simulation revealed that at least 64 bits of the round key were recovered in 9992 of the simulations which is a 99.92% chance of successfully recovering the key. The minimum number of bits recovered in this simulation was 61 and the maximum number of bits was 101. The maximum number of bits recovered would only leave 27 bits of search space, or 134,217,728 combinations, required to discover the full key. This is a problem a typical modern-day personal computer could easily solve. The maximum number of `(C,C')` pairs required to recover each bit was again six. These results can be seen in Figure 3.12.

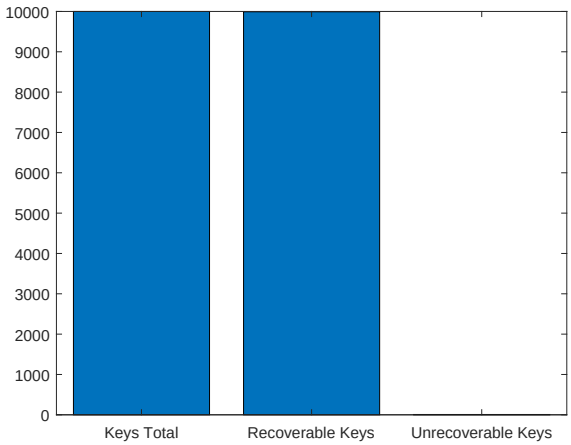


Fig. 3.12: The numbers of recoverable vs. Unrecoverable keys are displayed for the 800 gate Trojan.

3.6.5 Method applied to the DES circuit

A similar Trojan that was used against the AES circuit could be modified to attack the DES circuit previously described. In this scenario, the hardware Trojan would attack the entire left hand output of the 14th round, holding it to zero. This would require 32 `AND` gates per round for a total of 512 `AND` gates. This would allow us to discover the round key for the 16th round with 100% certainty using only a single plaintext.

This works because if the left block output of the 14th round is held to zeros, then the right block input to the 15th round would also be zeros and the left block input to the 16th round would be zeros, as well. Both the right and left block outputs of the 16th round are known because they can be reversed through the final permutation to reveal them. The right block input to the 16th round would also be known because it is equal to the left output. This is illustrated in Figure 3.13. Knowing all inputs and outputs to the round, the only unknown would be the round key which could easily be determined by going through the algorithm in reverse. Upon discovering the round key, the remaining eight bits of the secret key could be discovered by an exhaustive search, as described earlier.

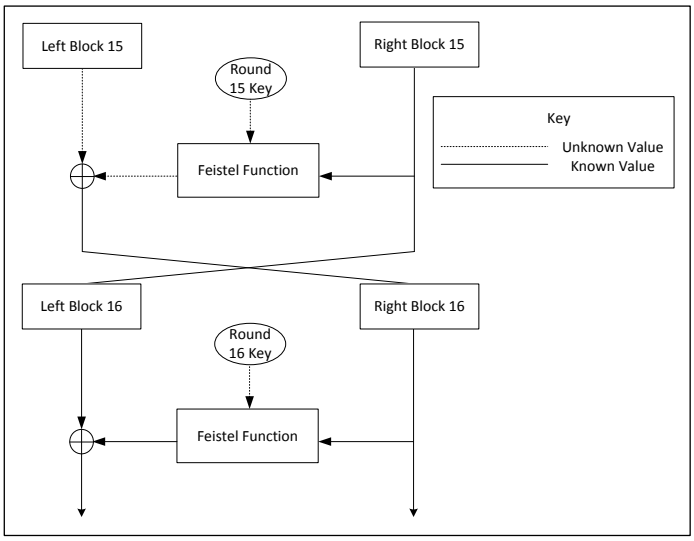


Fig. 3.13: Rounds 15 and 16 of DES are shown. It is illustrated that if a hardware Trojan caused the left block output of round 14 to be zeros then the only unknown for round 16 is the round key.

This Trojan attacking DES would be 512 gates large, which is half the size of the original Trojan suggested by Imeson et al., and it would have a 100% success rate.

3.7 Conclusion and Future Work

We have demonstrated that the obfuscation methods set forth in [1] do not provide the security that they claim for Feistel structured or SPN ciphers. This is because these ciphers contain highly redundant logic. This redundant logic makes these types of circuits vulnerable to hardware Trojan attacks even after the wire lifting procedure set forth in [1] has been carried out. Even if a desired gate cannot be attacked specifically, there is enough redundancy in these types of circuits to allow for an attack to be successful. In these cases, *k-security* gives a false sense of how secure the circuit is because multiple gates in a redundant, pipelined circuit can be attacked simultaneously without the extraneous gates affecting the outcome of the attack, as they are attacked during a different pipeline stage.

This is very important to be brought to light before the community. It is difficult to prove that any method of 3D circuit manufacturing has the same benefits for any arbitrary circuit. It is therefore beneficial when a class of circuits can be identified as being an exception to the rule. In the future of split manufacturing research, redundant circuits must be considered. Once a new method is identified, it must be tested against redundant circuits to see if the claims of security still hold for those types of circuits.

Future work would include investigating other types of redundant circuits in order to show that cryptographic ciphers are not the only types of highly redundant circuits for which the wire lifting procedure does not provided the advertised amount of security. Other classes of circuits may also be identified as not being subject to the claims of increased security provided by 3D manufacturing.

REFERENCES

[1] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara, "Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 495–510.

[2] Y. Wang, P. Chen, J. Hu, G. Li, and J. Rajendran, "The cat and mouse in split manufacturing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, pp. 805–817, 2018.

[3] J. J. Rajendran, O. Sinanoglu, and R. Karri, "Is split manufacturing secure?" in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 1259–1264.

[4] K. Vaidyanathan, B. P. Das, E. Sumbul, R. Liu, and L. Pileggi, "Building trusted ics using split fabrication," in *2014 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE, 2014, pp. 1–6.

[5] K. Vaidyanathan, R. Liu, E. Sumbul, Q. Zhu, F. Franchetti, and L. Pileggi, "Efficient and secure intellectual property (ip) design with split fabrication," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 13–18.

[6] K. Vaidyanathan, B. P. Das, and L. Pileggi, "Detecting reliability attacks during split fabrication using test-only beol stack," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[7] Y. Xie, C. Bao, and A. Srivastava, "Security-aware design flow for 2.5 d ic technology," in *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. ACM, 2015, pp. 31–38.

[8] A. P. Johnson, S. Patranabis, R. S. Chakraborty, and D. Mukhopadhyay, "Remote dynamic clock reconfiguration based attacks on internet of things applications," in *Digital System Design (DSD), 2016 Euromicro Conference on*. IEEE, 2016, pp. 431–438.

[9] K. Hasegawa, M. Oya, M. Yanagisawa, and N. Togawa, "Hardware trojans classification for gate-level netlists based on machine learning," in *On-Line Testing and Robust System Design (IOLTS), 2016 IEEE 22nd International Symposium on*. IEEE, 2016, pp. 203–206.

[10] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, vol. 27, no. 1, 2010.

[11] M. Yoshimura, T. Bouyashiki, and T. Hosokawa, "A hardware trojan circuit detection method using activation sequence generations," in *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE, 2017, pp. 221–222.

[12] H. Salmani, "Hardware trojan attacks and countermeasures," in *Fundamentals of IP and SoC Security*. Springer, 2017, pp. 247–276.

[13] A. Malekpour, R. Ragel, A. Ignjatovic, and S. Parameswaran, "Trojanguard: Simple and effective hardware trojan mitigation techniques for pipelined mpsocs," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 19.

[14] M. T. C. Wang, *Introduction to Hardware Security and Trust*. 233 Spring Street, New York, NY 10013: Springer, 2012.

[15] U.S. department of commerce bureau of industry and security office of technology evaluation. (1999) Defense industrial base assessment: Counterfeit electronics. [Online]. Available: http://www.bis.doc.gov/defenseindustrialbaseprograms/osies/defmarketresearchrpts/final_counterfeit_electronics_report.pdf

[16] Committee on Armed Services, United States Senate. (1999) Inquiry into counterfeit electronic parts in the department of defense supply chain. [Online]. Available: http://www.armed-services.senate.gov/Publications/Counterfeit%20Electronic%20Parts.pdf

[17] H. Salmani, M. Tehranipoor, and J. Plusquellic, "New design strategy for improving hardware trojan detection and reducing trojan activation time," in *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. IEEE, 2009, pp. 66–73.

[18] M. S. Samimi, E. Aerabi, Z. Kazemi, M. Fazeli, and A. Patooghy, "Hardware enlightening: No where to hide your hardware trojans!" in *On-Line Testing and Robust System Design (IOLTS), 2016 IEEE 22nd International Symposium on*. IEEE, 2016, pp. 251–256.

[19] J. Peng, C. H. Tan, Q. Wang, J. Gao, and H. Kan, "More new classes of differentially 4-uniform permutations with good cryptographic properties," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 101, no. 6, pp. 945–952, 2018.

[20] G. Maity, J. Bhaumik, and A. Kundu, "A new spn type architecture to strengthen block cipher against fault attack." *IJ Network Security*, vol. 20, no. 3, pp. 455–462, 2018.

[21] R. Girija and H. Singh, "A new substitution-permutation network cipher using walsh hadamard transform," in *Computing and Communication Technologies for Smart Nation (IC3TSN), 2017 International Conference on*. IEEE, 2017, pp. 168–172.

[22] T. Baigneres and S. Vaudenay, "Proving the security of aes substitution-permutation network," in *International Workshop on Selected Areas in Cryptography*. Springer, 2005, pp. 65–81.

[23] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit blockcipher clefia," in *International Workshop on Fast Software Encryption*. Springer, 2007, pp. 181–195.

[24] C. H. Kim and J.-J. Quisquater, "Faults, injection methods, and fault attacks," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 544–545, 2007.

[25] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[26] M. J. M. Tunstall, *Fault Analysis in Cryptography*. Springer, 2012.

[27] F. Imeson. (2017) circuit_security. [Online]. Available: https://github.com/fcimeson/circuit_security

[28] R. Usselmann. (2009) DES/Triple DES IP Cores. [Online]. Available: http://opencores.org/project/des

[29] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual International Cryptology Conference*. Springer, 1997, pp. 513–525.

[30] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[31] J. Daemen and V. Rijmen, "Specification for the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, 2001.

[32] H. Hsing. (2015) tiny_aes. [Online]. Available: https://opencores.org/project/tiny_aes

CHAPTER 4

Hardware Trojan Detection Without a Golden Model Using Deep Learning

ABSTRACT Integrated circuits (IC) are vulnerable to malicious modifications known as hardware trojans. This is primarily due to the outsourcing of the design and fabrication of ICs to untrusted foundries. Due to the risk posed by such modification of ICs, detection of these malicious entities within hardware circuits is of immense interest.

Hardware trojans are accompanied by triggering mechanisms embedded into the same IC along with the trojan. In this paper we formulate the problem of detecting trojan triggers, in a non-invasive and scalable manner using only the circuit structure and gate types as a supervised machine learning task. We have also created and made available, a dataset including a wide array of trojans of various sizes on a variety of circuit types. We characterize the performance of various machine learning architectures under settings of clean and contaminated training data and also explore the performance of machine learning models when subjected to an adaptive adversary.

4.1 Introduction

Hardware trojans are malicious modifications or additions made to a circuit which can change the intended functionality of a circuit, alter the reliability, leak confidential information such as secret keys, or cause a circuit to malfunction in some other way. They can cause a system to become disabled or compromised, allowing an adversary to gain access to highly protected data [1]. Hardware trojans present a potential security concern to anything that utilizes integrated circuits. The U.S. Department of Commerce has estimated that counterfeit electronic components have appeared in 39% of the Department of Defense supply chain [2]. Such counterfeit parts have been discovered in Navy helicopters and Air Force planes [3]. The possibility exists that counterfeit parts might contain hardware trojans. Not only are governments concerned about the existence of these malicious circuits, private

businesses and individuals could be at risk as well.

Malicious modifications made to hardware are very difficult to implement, but if successful they can be devastating. Recently, Bloomberg reporting alleged that a tiny microchip was discovered on servers assembled by Super Micro Computers Inc. (Supermicro) which was not supposed to be there. These servers were used by many private and governmental agencies including the Department of Defense (DOD), the Central Intelligence Agency (CIA), Apple, Amazon and 28 other companies. Bloomberg asserted that this addition to the design created a backdoor for attackers to access any network connected to an infected server, and that it was added to the design in factories in China [4]. Apple, Amazon and Supermicro have all issued strong denials claiming that these malicious chips were never found on hardware they were using and calling for a retraction [5–8].

Nonetheless, hardware trojans remain a threat to governments and public companies around the world, which is why they are a topic that continues to be researched. Methods to prevent hardware trojans from being detected, as well as ways to detect them once inserted into a design, are constantly being sought after [9–12]. New types of hardware trojans are being invented by researchers and then techniques to mitigate those specific threats are proposed [13, 14]. Hardware trojans can be introduced by modifying the code written in a hardware description language (HDL) used to describe the circuit [15, 16]. Users of third party intellectual property (IP) must be wary that their vendors did not place a trojan into the design.

We present the groundwork for applying deep learning to the task of hardware trojan detection and solve some of the difficulties associated with this application. A good application for this method would be in third party IP detection, as it could be detected using our methods prior to chip manufacture. Typically, a deep learning application requires a large amount of data in a form it can digest. We provide a solution to this problem. Our method involves analyzing the hardware description language (HDL) representation of a circuit or the gate level circuit and detecting hardware trojan triggers. We show promising results applying learning strategies to detect combinatorial hardware trojan triggers comprised of

`AND` and `NOT` gates. One of the highlights of this proposed approach is that it does not rely on a golden model.

4.1.1 Contributions

Our contributions are as follows:

- A methodology to create datasets for hardware trojan research is presented.
- A dataset of trigger-inserted circuit adjacency matricies is provided.
- A methodology to create feature vectors from circuit adjacency matrcies is set forth.
- The groundwork for a new application of deep learning: using state-of-the-art models to identify hardware trojans is presented.

4.2 Related Work

There is constant escalation in the field of hardware trojan detection. A new form of trojan detection breeds a new type of trojan which bypasses that form of detection, only for the detection modification to be altered to detect the new trojan that was discovered [17–19].It will be shown that the methods we present to detect hardware trojan triggers are highly adaptable. If a trigger is created to bypass detection the training data of the deep learning model can easily be expanded to include this new type of trigger so that it will be detected in the future.

Recently, deep learning models have shown promise in various fields like computer vision, natural language processing and time series analysis. In [20], deep learning models were used to develop a system capable of object classification, part segmentation, scene semantic parsing directly from point clouds. Convolutional networks have been especially successful at image classification [21] and object detection [22]. Deep learning models have also shown enormous promise at natural language text classification [23,24], text summarization [25,26] and neural machine translation tasks [27].

4.2.1 Existing Hardware Trojan Detection Methods

Many methods of detecting hardware trojans have been proposed, e.g., [1,28–31]. Integrated circuits (IC) that are produced today are too complex to be exhaustively verified by simulation before production [32]. One method of trojan detection requires a golden version of the IC in which the actual layout of a fabricated IC is compared to the golden in order to determine if any gates were added or removed [33]. Other techniques requiring golden models have been proposed and they typically compare circuit functionality with that of a golden model [34–36].

Reliance on a golden model has been proved to be problematic. For example, it would be possible for an attacker to introduce a hardware trojan into only a small subset of the population of manufactured ICs, which would make these methods of testing samples of the total population less effective [29]. The majority of the existing hardware trojan detection techniques require a golden IC which makes them less useful than the few methods that do not rely on a golden IC [30].

Recently, a new approach has been proposed which does not rely on a golden model: it uses symmetry within a circuit to detect the presence of additional logic, or a hardware trojan [37]. This technique measures path delays within a circuit and detects if a hardware trojan has modified that delay. The technique allows for natural symmetries to be used, or to be designed into the circuit. One advantage to this method is that it completely eliminates inter-die variations as the measurements are taken from the same chip. A major drawback to this method is that the technique becomes very difficult for large circuits when finding the symmetry naturally, and the overhead can get very large when attempting to insert the symmetry artificially.

An order of path delay method was recently described [38]. The authors of this work propose a hardware trojan detection technique which takes pairs of paths and calculates order of path delays for each. During the design phase the path pairs for the entire circuit are analyzed and then during the test phase the actual delays are recorded. The actual delays are compared to the theoretical delays calculated during the design phase, and thus

hardware trojans are detected.

Another method which works without a golden standard is called TeSR, it is a temporal self-referencing method [39]. This approach measures the current profile of a circuit at different times and compares them. The authors claim that this method uses the uncorrelated temporal variations in current caused by sequential hardware trojans to detect them. They also claim that it eliminates process noise. This is a novel form of side-channel analysis and differs from most approaches which rely on golden models.

The advantage that our method has over these other methods that also do not require a golden model is we detect the trojan before the chip is manufactured, saving time and money.

4.2.2 Deep Learning Architectures & Applications

Deep learning architectures have also been successfully applied to problems with sequential and graphical properties. Recently, deep learning models have been used for time series forecasting and anomaly detection [40–42] and for speech recognition tasks [43,44]. Another emerging area of application for deep learning models is unstructured input domains like graphs. Recently, Graph Convolutional Networks (GCN) have proven to be effective in domains with irregularly structured input data which cannot be handled easily by traditional network architectures like feed-forward, convolutional or recurrent neural networks. GCNs have been used for various tasks like labeling nodes in a graph of citation networks, predicting molecular properties given their structure, or automatic hand-written digit classification when the digits are not on a traditional 2D plan but rather on 3D surfaces and many other applications [45–48].

In this paper, we apply variants of all the aforementioned popular deep learning architectures to the problem of hardware trojan trigger detection using the adjacency matrix of a circuit and the corresponding gate types of each node (gate) in the matrix (hardware circuit). To the best of our knowledge, the task of identifying hardware trojan triggers embedded in a circuit by only using the adjacency matrix of the circuit (i.e non-invasively) has not been attempted previously.

4.3 Overview

A trigger-based hardware trojan contains two parts, a trigger and a payload [49]. The trigger activates the payload when a specific combination of values is found on the trigger inputs. The connections made when the trigger is inserted in the circuit are the insertion points. A trigger will have more insertion points as it increases in size. A basic trigger will consist of `AND` and `NOT` gates in order to select the correct trigger inputs to activate the trojan. Fig. 4.1, trigger `A` depicts a hardware trojan trigger consisting of three `AND` gates and two `NOT` gates. The output of this trigger will be a `1` when the inputs, `A,B,C,D` are `1,0,0,1`. Fig. 4.1, trigger `B` shows a variation on the connectivity of the trigger gates from Fig. 4.1, trigger A. The functionality of the trigger is unchanged assuming each input remains attached to the same insertion point. This simple example illustrates why it would be difficult to use an exact match algorithm to detect hardware trojan triggers. An exact match algorithm would be required to contain every iteration of every possible trigger, not only in number of gates and input sequence causing a trigger, but also possible layouts of the trigger creating logically equivalent circuits. The difficulties involved in such an effort lead us to consider using deep learning as a hardware trojan trigger detection method. This will allow for the detection of hardware trojans in third party intellectual property (IP).

We will demonstrate how various deep learning models perform while attempting to identify hardware trojan triggers. We have created a process described more in depth in Section 4.4 which allows us to take a circuit described in HDL and create feature vectors for a deep learning algorithm. This process is illustrated in Fig. 4.2. We take a high-level HDL circuit (which may contain multiple HDL files) and transform it into the gate representation. We then transform that into the circuit adjacency matrix form, which preserves all the data from the gate representation, including the gate types and all connections from input to output of the circuit. A definition for circuit adjacency matrix can be found in Definition 4.4.1. The circuit adjacency matrix form is then converted to the inverse node fanin, defined in Definition 4.4.2. This breaks down the reverse path from gate to input for each gate in the circuit. Finally, the individual inverse node fanins are converted to one-hot

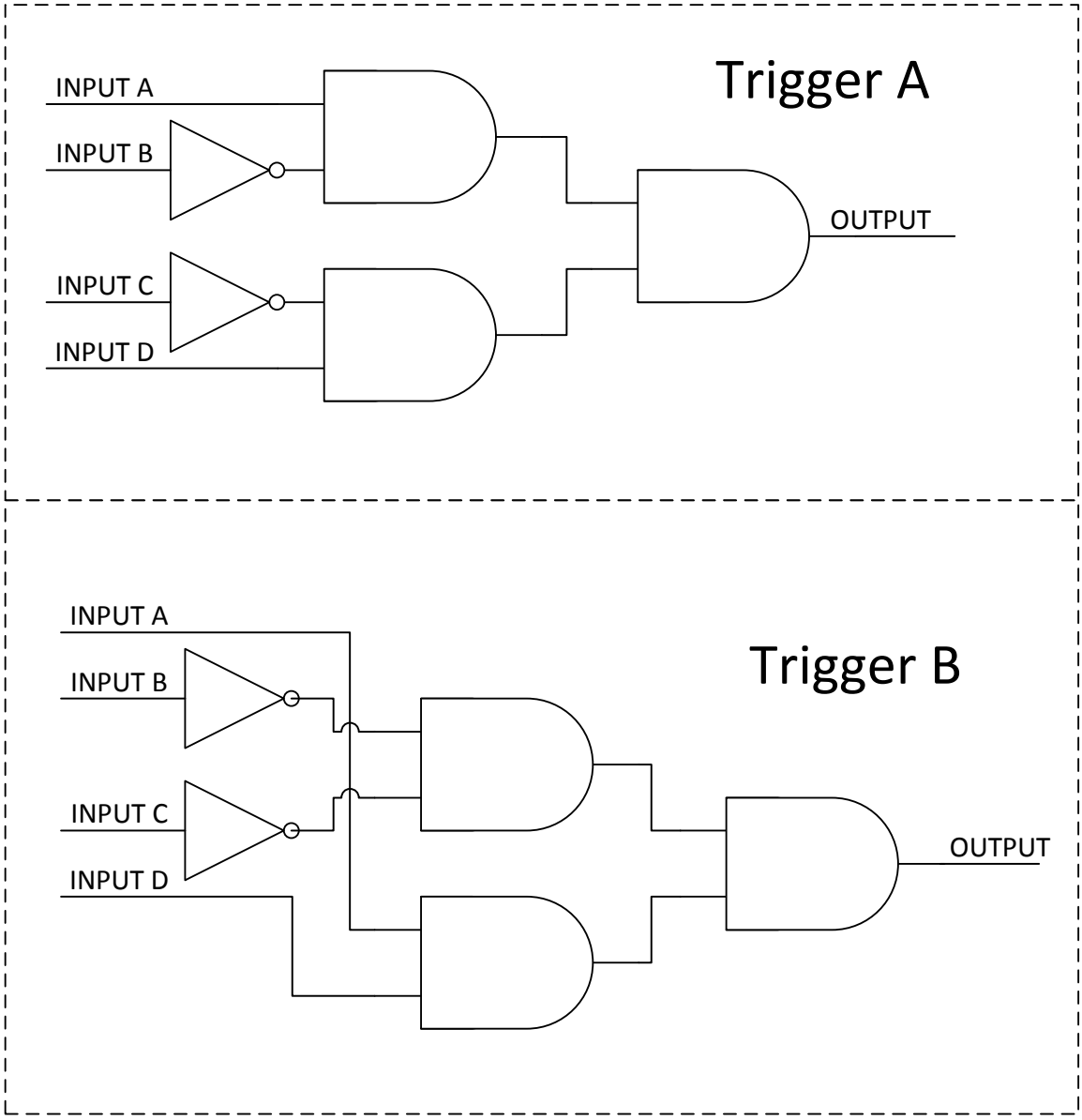


Fig. 4.1: Two equivalent hardware trojan triggers containing the same number of gates but with different connections.

encoded feature vectors, where each gate type is depicted by a six bit one-hot encoding.

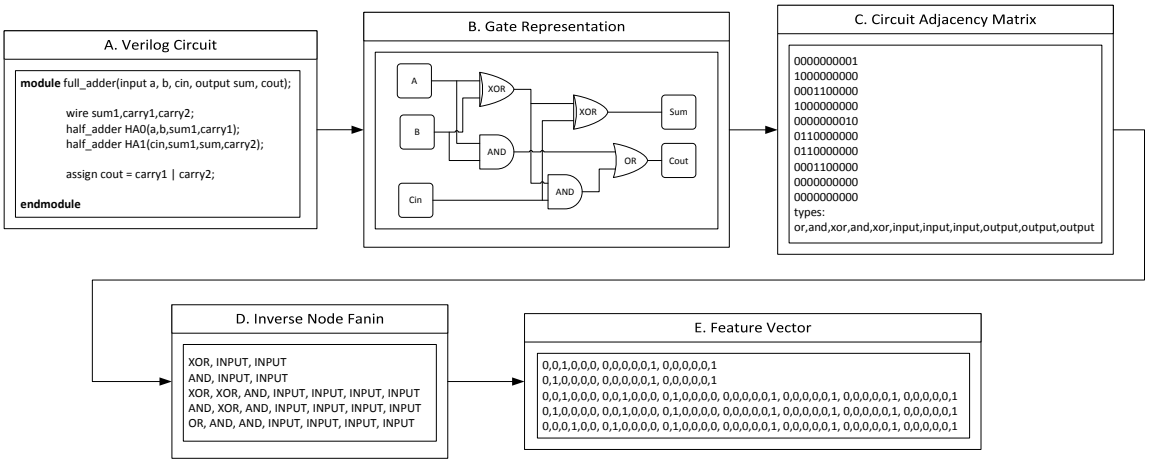


Fig. 4.2: Process required to convert an HDL circuit into feature vectors.

4.3.1 Threat Model

The threat model we assume in this work is that a designer, or team of designers implant a combinatorial trigger-based hardware trojan in a circuit during the design phase. The HDL itself which was altered by these bad actors is available to the manufacturer of the circuit. The source of the hardware trojan may be from a third-party IP vendor, or from an insider threat in the organization creating the circuit. Example triggers can be seen in Figure 4.1.

We are not concerned with the functionality of the hardware trojan once it is triggered, only that it is latent until triggered. The trigger is a logic cone consisting of `AND` gates and `NOT` gates which ends in a single wire which activates the trojan. The trigger may be dispersed throughout the circuit. It may be connected to inputs, or buried deep within the logic of the circuit. The trigger is condition-based and may be activated either internally or externally, when the predetermined sequence of bits appear on the inputs to the trigger [1, 50].

4.4 Procedures

One of the difficulties associated with performing research in the field of hardware

trojans is the lack of publicly available trojan-inserted data. This is particularly difficult when attempting to apply neural network strategies which require large amounts of data for training and verification purposes. The Trust-Hub website [51] contains 88 trojan-inserted benchmark circuits. These vary in the types of trojans and triggers; when looking at particular types of trojans and how they are triggered the researcher is limited to a subset of the 88. For example, only five of the 88 circuits contain a trigger that is activated based on user input. Besides the lack of publicly available trojan-inserted circuits, there is also a lack of benign circuits available in a state that would be acceptable for deep learning consumption.

In order to increase the amount of data available for research purposes, it was necessary to create a method capable of taking circuits described in either very high speed integrated circuit hardware description language (VHDL) or Verilog and forming usable data from those circuits. Knowing that these two HDLs are the most widely used in industry [52] they were chosen as the starting point for the new dataset, but the most important form is the circuit adjacency matrix. It has all the information of a circuit in a compact form which is simple to manipulate and create more data from. We have made datasets of circuit adjacency matrices containing hardware trojan triggers available here: https://github.com/nweidler/circuitAdjacencyMatrix.git The following section will offer a definition for a circuit adjacency matrix and describe the process to create them, as well as feature sets for deep learning use. We are providing detailed instructions so that the community can reproduce our work. We have used this same process to produce feature sets for our deep learning models.

4.4.1 Circuit Adjacency Matrix

Definition 4.4.1 (Circuit Adjacency Matrix) *We say that a Circuit Adjacency Matrix is an nXn matrix used to represent a the nodes of a circuit and their connections. It is the representation of a circuit as a directed graph. A node is any gate, input or output of a circuit. The elements of a circuit adjacency matrix indicate which nodes are connected to each other. Along with an adjacency matrix, a list of nodes is required to preserve the*

functionality of the circuit. This is included in the circuit adjacency matrix with the first node in the list corresponding to the first row and column in the matrix, the second node corresponding to the second row and column, and so forth.

The process to create an adjacency matrix is as follows:

Use Synopsys Design Vision to flatten the heirarchy of a circuit described in either VHDL or Verilog, sythesize the circuit into gates and then regenerate a VHDL file. The VHDL file will be in a standard form which is expected by the following steps. Use the Python program developed by the authors to produce an adjacency matrix for each circuit. The process to create the adjacency matrix is as follows:

1. Read in the VHDL file produced by Synopsys Design Vision.

2. Convert all inputs and outputs in the entity declaration to be of type `std_logic` without losing any bits, this requires inputs and outputs to be created for multi-bit types. Ex. `a : in std_logic_vector(2 downto 0)` becomes `a(0), a(1), a(2) : in std_logic`. Although this is not correct VHDL syntax, we are not compiling these files, this is a shortcut to allow the inputs and the outputs to match to where they are used. The inputs and outputs are only used as single bits because of the way Synopsys creates the VHDL file.

3. Create a new VHDL file with these modifications.

4. Read in the new VHDL file created by the previous step.

5. Using the new VHDL file, create lists of all inputs, outputs and gates used in the design.

6. Create a data structure containing of all the connections leaving each input and each gate in the design. The outputs will have no nodes leaving them. We ignore all incoming connections because for each outgoing connection in the design there is an incoming connection.

7. Create a square zero matrix with the number or columns and rows equal to the total number of gates, inputs and outputs in the design. Each column and each row will represent a gate, an input or an output. This is the frame of a circuit adjacency matrix.

8. Starting with the first gate in the list of gates, and the first row in the square zero matrix, indicate the outgoing connections from that gate by placing a `1` in the column corresponding the the outgoing connection. Ex. The following row indicates that the gate represented by this row has a connection to gate `1` and gate `4` : `0100100000000000000`. The first item in the list of gates will correspond to the first row in the matrix, the second item to the second row, and so forth. After the list of gates is exhausted do the same for the list of inputs. The rows corresponding to the outputs in the matrix will always contain all zeros as outputs do not have any outgoing connections.

9. Output the adjacency matrix to a file, adding at the bottom the type of the gate represented by each row, followed by `input` for each input row and `output` for each output row. The circuit adjacency matrix is now complete.

We created an automated process to run large numbers of HDL files through the above process. The original HDL files we used were taken from Opencores.org [53]. No data is lost when the HDL is represented as a circuit adjacency matrix as the entire circuit could be reproduced from it. An example adjacency matrix can be seen in Fig. 4.3. It depicts a full adder represented as a circuit adjacency matrix, and the gate representation of the circuit is shown beside it for reference.

The translation from the circuit to the circuit adjacency matrix in Fig. 4.3 is as follows:

- Row 0 represents the `OR` gate. The only connection it has is to output `Cout`, indicated by a `1` in column 9 of row 0.

- Row 1 represents `AND 1`, showing the connection from it to the `OR` gate indicated by the `1` in column 0.

- Row 2 represents `XOR 1` showing the connection to `XOR 2` and `AND 2` by the `1`s in columns 3 and 4.
- Row 3 represents `AND 2` and the connection to the `OR` gate is indicated by the `1` in column 0.
- Row 4 represents `XOR 2` and the connection it has to the `Sum` output indicated by the `1` in column 8.
- Row 5 is input `A` and shows the connections it has to `XOR 1` and `AND 1` by the `1`s in columns 1 and 2.
- Row 6 is identical to row 5 as input `B` is also connected to `XOR 1` and `AND 1` like input `A`.
- Row 7 represents input `Cin` and shows the connections to `AND 2` and `XOR 2` indicated by `1` in columns 3 and 4.
- Both outputs `Sum` and `Cout` are shown by rows 8 and 9. Notice they do not show any connections because they have no outgoing connections.

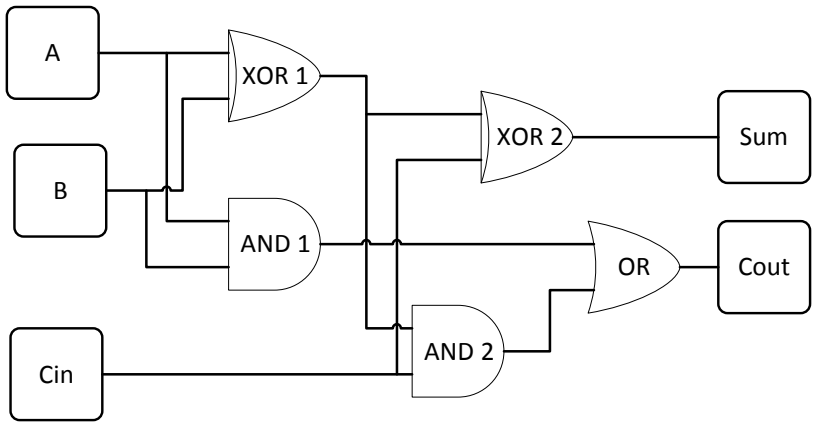
The circuit adjacency matrix is the baseline for the datasets we use in this work. From this form we have discovered other ways to represent the circuits to prepare them for consumption by the neural networks. Other researchers may find additional novel representations for circuits starting from an adjacency matrix as described here. One powerful method we have chosen for the representation of circuit data is called inverse node fanin.

4.4.2 Inverse Node Fanin

Definition 4.4.2 (Inverse Node Fanin) *We say that for each node in a circuit an inverse node fanin exists and is represented beginning with the node itself followed by the the nodes it is connected to, until the inputs of the circuit are reached.*

The entire circuit can be described by specifying the inverse node fanin for each node in the circuit. This is a feature rich representation which contains as much data as the

0000000001
1000000000
0001100000
1000000000
0000000010
0110000000
0110000000
0001100000
0000000000
0000000000
types :



or2i,an2,eo,an2,eo,input,input,input,output,output,

Fig. 4.3: Full adder represented as a circuit adjacency matrix. This is the actual output of the python program. The circuit is shown as gates on the right.

adjacency matrix but can be viewed one node at a time, making it a good candidate for a form of data to supply to the deep learning model.

In order to prepare datasets for deep learning, the inverse node fanin data was converted to a one-hot encoding scheme to represent the type of each node. In order to preserve structure, all zeros in the one-hot scheme represents the lack of a gate. Each gate with only a single input, such as an `INVERTER`, will have a node in front of it, as well as a one-hot encoding representing the lack of a second node. From each adjacency matrix it was straightforward to create this data. The adjacency matrices derived directly from the HDL provide inverse node fanin data that represent nodes that do not contain a hardware trojan trigger. The process is as follows:

- Read in the circuit adjacency matrix to a Python program.

- For each node in the adjacency matrix record which node is in front of it, or in other words, which node is on the other end of the incoming connection. This is accomplished by examining the columns in the adjacency matrix. Each column and each row represent a node in the circuit. When looking at column 2, we are determining which nodes come before node 2. If rows 0 and 1 contain a `1`, that indicates that nodes 0 and 1 are in front of node 2, or nodes 0 and 1 have outgoing edges towards node 2. See Fig. 4.4 as an example. Record the node type and the type of each node in front of it.

- Repeat the previous step until all inputs are reached for each node. For each node, the complete inverse node fanin is known.

- Convert each inverse node fanin to a one-hot encoded scheme. This is the feature set for the deep learning models.

4.5 Representation Learning

In order to test the applicability of using deep learning to solve the problem of hardware trojan trigger detection, several experiments were run. Initially, a proof of concept was required. For this the training data contained only either whole triggers or original circuit data depicted in the inverse node fanin form. No parital triggers were considered.

Although interesting, the proof of concept is not applicable to real world data. When an HDL file is converted to a circuit adjacency matrix and then to inverse node fanin form, each node is considered and therefore partial triggers will be included in the data fed to the deep learning model. We carried out additional experiments in which the training data was labeled in such a way that when 40% of a trigger was included in an inverse node fanin it was labeled as a trigger. This may result in multiple inverse node fanins returning as positive for a trigger from a single circuit, but when examined it will be seen that they are all part of the same trigger.

4.5.1 Feed Forward Neural Network

00**1**0000000000000000
00**1**0000000000000000
00**0**0000000000000001
00**0**0010000000000000
00**0**0010000000000000
00**0**0001000000000000
00**0**0000010000000000
00**0**0000010000000000
00**0**0000000000000010
10**0**0000000000000000
00**0**0000000001000000
00**0**1000000000000000
00**0**0000000010000000
00**0**0000001001000000
00**0**0000001100000000
00**0**0000100000000000
01**0**0100000000000000
00**0**0000000000000000
00**0**0000000000000000
types :

ivi,ivi,an2i,ivi,ivi,an2i,ivi,ivi,an2i,eoi,ivi,ivi,an2i,input,input,input,input,output,output,

Fig. 4.4: The boldface column in this circuit adjacency matrix, representing node 2, has nodes 0 and 1 in front of it in the circuit. We see that node 2 is an `AND` gate with two INVERTER gates in front of it.

We use deep learning to detect the trigger associated with trigger-based hardware trojan. A deep feed-forward neural network implemented in the Python programming language was utilized in our initial trials. The PyTorch machine learning library was utilized to describe the neural network in order to easily access deep learning functionality [54]. The network consists of an input layer, two hidden layers and an output layer. The activation function utilized was the Rectified Linear Unit (ReLU) available through PyTorch's libraries. The built in cross entropy loss function was used because the training data would be unbalanced. The Adam optimizer was also used.

4.5.2 Training Data

In order to train the neural network to recognize hardware trojan triggers, a set of

triggers and original circuit data was needed for training and verification purposes. The original circuit data was created using the methods depicted in Section 4.4. As there is no large database of trigger-activated hardware trojans we needed to generate this data. In order to remain as unbiased as possible, we started from the same adjacency matrices used to generate the original circuit data. We created an algorithm that generated random trigger structures, only constraining it to the number of inputs the trigger would have. `AND` gates and `NOT` gates were selected randomly and the structure of the trigger was built until it met the criteria of the pre-determined number of inputs. Those completed triggers were then inserted into the original circuit adjacency matrices by adding rows and columns and creating the appropriate connections, as well as inserting the appropriate node types into the list at the bottom. In order to keep the insertion unbiased we selected random points within each circuit to attach the triggers. Some were attached directly to inputs while others were buried deep within the circuit. These new, trigger-inserted circuit adjacency matrices were then used as the starting point to create inverse node fanin data as per the normal procedure.

The inverse node fanins collected from real circuits that were not trigger-inserted varied from as few as a single one-hot encoding, up to 893. The inverse node-fanins generated from the trigger-inserted circuits varied in length from 5 to 897 one-hot encodings. To train the neural network, the data was broken into three sets: a training set, an inter-batch verification set, and a final verification set. Care was taken to ensure that no duplicates between the sets existed. The network was trained using a batch size of 32 over twenty training epochs.

Training Data For Graphical & Recurrent Models

We modified the input data slightly to experiment with learning better, more focused representations for the recurrent and graph neural network models. Let us consider the set $\mathcal{S} = \{\mathcal{G}_1, \mathcal{G}_2, .., \mathcal{G}_n\}$ to be the database of graphs such that a subset of graphs $\mathcal{S}' \subset \mathcal{S}$ has graphs with triggers embedded within them. For a given graph $\mathcal{G}_i \in \mathcal{S}$, we generated a set of trees by considering the *inverse fanin* of a node up until depth d, to produce a set $S_i^T = \{T_i^1, T_i^2, .., T_i^k\}$. By repeating this procedure for all graphs in $\mathcal{S}$, we obtain a set of

k-ary trees $\mathcal{S}^T = \{T_1^1, T_1^2, .., T_1^j, ..., T_n^1, T_n^2, ..., T_n^k\}$. The set $\mathcal{S}^T$ is used as the dataset for all the recurrent and graphical model experiments. The depth d for the curtailed *inverse fanin* was selected based on the size of the largest trigger in the dataset so that the largest trigger can be completely contained in a tree $T_i^j \in \mathcal{S}^T$. Each node in a tree T_i^j is denoted by a gate type. Our circuits contained 16 types of gates each denoted using a 4 dimensional binary vector, including one dummy gate which was denoted by the special 4-d vector $[0, 0, 0, 0]$. Each tree $\mathcal{T}_i^j$ was further standardized structurally by injecting dummy nodes to make the tree a *perfect* k-ary tree. In our case, each tree was standardized to form a *perfect* binary tree i.e a tree with exactly $2^{d+1} - 1$ nodes. We chose $d = 8$ and hence each tree T_i^j has 511 nodes (gates) in our case, each node represented by a 4-d feature vector denoting gate type as described previously. We note here that as a function of this type of data generation, a tree T_i^j can have a complete or partial trigger or no trigger at all. A tree T_i^j is said to have a complete or partial trigger when a certain percentage of nodes of a known trigger are present in T_i^j, else it is said to have no trigger. This tree-based dataset was generated to test the robustness of models to datasets containing partial triggers, i.e modeling in the presence of contaminated training data to mimic real-world settings.

4.5.3 Graphical & Recurrent Models

Since a hardware circuit can be inherently viewed as a directed acyclic graph, we naturally also inspected the representation learning capacity of more sophisticated learning architectures compared to the feed-forward neural network, namely the Graph Convolutional Network (GCN) and the Recurrent Neural Network (RNN).

GCN

Graph convolutional networks generalize the well known *convolution* operation from traditional grid based applications like images (2D grids) to irregularly structured objects like graphs. A graph $\mathcal{G}$ can be represented by a triple $(\mathcal{V}, \mathcal{E}, \mathcal{A})$ where $\mathcal{V}$ represents the nodes in the graph (in our case a node is a gate in the hardware circuit), $\mathcal{E}$ represents the number of edges in the graph (connections between the gates in the circuit) and these connections and

gates can be represented as an adjacency matrix $\mathcal{A}$. If we assume $|\mathcal{V}| = N$, the adjacency matrix can be represented as $\mathcal{A} \in \mathbb{B}^{N \times N}$ where $\mathbb{B} \in \{0, 1\}$. Each node in the graph can be associated with its own attributes. The attribute matrix $\mathcal{X} \in \mathbb{R}^{N \times k}$ is supplied to the GCN in addition to the adjacency matrix. Each row $\mathbf{x}_i \in \mathbb{R}^{1 \times k}$ of $\mathcal{X}$ represents the node attributes of node v_i. For simplicity, we consider the circuit and the corresponding adjacency matrix to be undirected.

The goal of GCNs is to learn a feature representation for each node v in the graph $\mathcal{G}$. These feature representations are then aggregated to form an overall representation for the graph which is then supplied to a classification mechanism, trained to recognize whether or not the input graph contains a trojan trigger. In our case, the classification layer is an extension of the GCN pipeline and is a simple linear classifier.

Similar to other deep learning models like convolutional neural networks and feedforward neural networks, GCNs learn a new feature representation for the feature vector $\mathbf{x}_i$ for each node v_i in the input graph $\mathcal{G}$. Each graph convolutional layer takes in a node feature matrix H as input and returns an updated node feature matrix (with the same dimensions as H) as output. For the l^{th} convolutional layer, the input feature matrix is denoted as $H^{l-1} \in \mathbb{R}^{N \times h}$ where h denotes the user specified hidden size. H^{l-1} is updated to produce the output representation H^l.

$$H^0 = \mathcal{X} \tag{4.1}$$

Eq. 4.1 shows that the representation of all nodes in the GCN for a particular input graph $\mathcal{G}$ is initialized (i.e. input to the first GCN layer) by the attribute matrix $\mathcal{X}$.

Node Feature Propagation: Every node v_i in the GCN has associated with it, a representation vector $\mathbf{h}_i \in \mathbb{R}^{1 \times h}$. The network is trained by feature propagation of representation vectors from each node to all its neighbors governed by the network structure as specified by the adjacency matrix $\mathcal{A}$. Every node receives the feature representation from its neighbors and updates its own representation by aggregating the set of feature representations governed by some function f [55].

$$h_i^l = \frac{1}{d_i + 1} h_i^{l-1} + \sum_{j \in \mathcal{N}(v_i)} \frac{a_{ij}}{\sqrt{(d_i + 1)(d_j + 1)}} h_j^{l-1} \tag{4.2}$$

Eq. 4.2 is a simple form of the feature aggregation function f for a node v_i in the graph $\mathcal{G}$. In this case, the aggregation is a weighted sum of the hidden state of node v_i from the previous layer and hidden states of the neighbors of v_i (represented by $\mathcal{N}(v_i)$) in the equation.

If $\mathcal{D} \in \mathbb{R}^{N \times N}$ represents the degree matrix of $\mathcal{G}$, then let $\bar{\mathcal{D}} = \mathcal{D} + I$ represent the degree matrix with self-loops added where I is the identity matrix of the same size as $\mathcal{D}$. We can similarly define $\bar{\mathcal{A}} = \mathcal{A} + I$ to be the modified adjacency matrix. The node representation update process in the GCN can now be expressed in matrix form by Eq. 4.3.

$$\bar{H}^l = \bar{\mathcal{D}}^{-\frac{1}{2}} \bar{\mathcal{A}} \bar{\mathcal{D}}^{-\frac{1}{2}} \tag{4.3}$$

The process of feature propagation defined in Eq. 4.3, encourages neighboring nodes to have similar representations in the graph thereby enforcing a smoothing effect. The motivation behind using a GCN is that if a trigger circuit were to be connected sparsely to the rest of the circuit and relatively densely connected within itself, all the trigger nodes would have similar representations, different from the rest of the circuit.

Nonlinear Feature Transformation: After feature propagation, a linear combination of the feature matrix H^i in each GCN layer i is performed with a learned weight matrix $W^i \in \mathbb{R}^{h \times h}$. This linear combination is then subjected to a non-linear transformation as indicated in Eq. 4.4.

$$H^l = \text{ReLU}(\bar{H}^l W^l) \tag{4.4}$$

Graph Classification: After the feature propagation and transformation steps through all the GCN hidden layers, we obtain the matrix $H^{final} \in \mathbb{R}^{|\mathcal{V}| \times h}$ which represents the node representations of the $|\mathcal{V}|$ nodes in $\mathcal{G}$. These representations are then transformed into a graph representation vector $h_{\mathcal{G}} \in \mathbb{R}^{1 \times h}$ by some transformation function $g(H^{final})$. We use

mean-pooling as our choice for function g but more sophisticated functions may be employed as well. Let this new hidden representation be $h^{\mathcal{G}} \in \mathbb{R}^{1 \times h}$. We perform the final classification with a linear classifier as defined in Eq. 4.5.

$$\hat{Y_{\mathcal{G}}} = \text{softmax}(W^{\mathcal{G}} h^{\mathcal{G}}) \tag{4.5}$$

RNN

We also consider treating each directed graph $\mathcal{G}$ as a sequence of components and using this approach for graph representation learning. Recurrent neural networks like gated recurrent units (GRU), long short-term memory units (LSTM) [56,57] have been successfully used to model sequential data in many domains. Hence, we employ recurrent architectures for our purpose of sequential modeling and representation learning of each input graph $\mathcal{G}$. We specifically use the GRU unit for our purposes.

The GRU consists of two gates, the *reset* and *update* gates. Eq. 4.6

$$\begin{aligned} z_t &= \sigma(x_t U^z + h_{t-1} W^z) \\ r_t &= \sigma(x_t U^r + h_{t-1} W^r) \\ \tilde{h}_t &= \tanh(x_t U^h + (r_t * h_{t-1}) W^h) \\ h_t &= z_t * \tilde{h}_t + (1 - z) * h_{t-1} \end{aligned} \tag{4.6}$$

The *reset* gate r_t governs the amount of information from the historical sequence to be incorporated into the current sequence step representation and the *update* gate z_t blends this updated historical representation with the current hidden representation.

In the case of the recurrent neural network model, each input graph was converted into a *perfect* binary tree $\mathcal{T}$ and all the nodes at a particular depth of the tree, were supplied at each step of the recurrence (including the root node, each tree in our case is 9 levels deep

and hence the recurrent network had 9 steps). The final hidden representation h^{final} at the end of the recurrence is passed into a linear classifier as described in Eq. 4.7.

$$\hat{Y}_{\mathcal{T}} = \text{softmax}(W^{\mathcal{T}} h^{final}) \tag{4.7}$$

In addition to deep learning models, we also evaluate the performance of other state-of-the-art classification models like Gradient Boosting Classifier (GBC), Random Forest Classifier (RF) and the standard logistic regression classifier on the trigger detection task.

4.6 Results

We conduct three experiments to evaluate model performance. In the first experimental setting, we treated all circuits containing a complete trigger as anomalous instances and ensured that the non-anomalous circuits were free of any complete or partial trigger circuits. Although filtering out all circuits with partial triggers, drastically reduced the number of available instances for the GRU and the GCN models, the rest of the models (which were trained on full circuit data and not on tree based sub-circuits as described in sec. 4.5.2) were still able to perform relatively well in this setting with purely anomalous and non-anomalous data. The results in Table 4.1 show the precision, recall and F1 scores of all models in the trigger detection task, and omit the corresponding performance scores of the non-anomalous circuit detection task as that is the majority class and hence is the easier of the two tasks and most models achieve an F1 score greater than 95% on that task. We notice that the feed-forward neural network model and the Gradient Boosting Classifier (GBC) models perform marginally better than the other models. Another curious result is the underperformance of the Gated Recurrent Unit based model (GRU) and GCN which can be attributed primarily to the lack of adequate training instances as well as the imbalance in the number of instances belonging to the positive (trigger) and negative (non-trigger) class in the dataset.

In the second experimental setting, circuits with less than 40% of a full trigger embedded within them are considered non-anomalous instances while all circuits with greater than or equal to 40% of a full trigger embedded within them are considered anomalous. This setting

Model	Trigger Precision	Trigger Recall	Trigger F1
Feed-forward	**0.99**	**1.0**	**1.0**
GRU	0.05	0.92	0.10
RF	0.98	1.0	0.99
GBC	**0.99**	**1.0**	**1.0**
Logistic Reg.	0.99	1.0	0.99
GCN	0.0	0.0	0.0

Table 4.1: Trigger Graph Detection Performance 100% Trigger Anomaly Dataset.

was chosen to test the performance of the learning models in the trigger detection task when the training data was contaminated by partial trigger circuits more representative of a real world setting. We must note however that this has two effects, (1) the contaminated training data makes the learning problem a little more challenging by allowing partial triggers to be present in non-anomalous instances, (2) the number of instances deemed anomalous increase relative to the first experimental setting containing only full trigger circuits labeled as anomalous. The result of the second effect is readily apparent in the case of the GRU and GCN models which have drastic improvements in performance (although they still under-perform relative to the other models) due to the increased number and variety of anomalous instances available during training. This enables the GRU and GCN models to learn better representations of what constitutes a trigger in our case. Overall, the feed-forward neural network model yields the best performance even in the 40% partial trigger case as shown in Table 4.2 indicating that it is able to exhibit the robustness required in real-world settings with contaminated training data containing partial triggers. We also characterize the classification results of the feed-forward neural network model by varying the partial trigger percentage in the experimental setting from 10% to 90% and record the classification performance in each case. Fig. 4.5 shows the results of this experiment, wherein we can observe an increasing trend in the precision, recall and F1 scores as we decrease the trigger threshold (i.e de-contaminate the non-anomalous circuit instances), with a 0% trigger threshold indicating that the circuits labeled non-anomalous are completely free of any partial triggers.

In the two experiments so far, we see that the feed-forward neural network models are

Model	Trigger Precision	Trigger Recall	Trigger F1	Percentage Change In F1
Feed-forward	**0.91**	**0.9**	**0.9**	-10.0
GRU	0.3	0.85	0.44	+340.0
RF	0.9	**0.9**	**0.9**	-10.0
GBC	0.87	0.8	0.83	-17.0
Logistic Reg.	0.88	0.76	0.81	-18.2
GCN	0.85	0.4	0.54	-

Table 4.2: Trigger Graph Detection Performance 40% Partial Trigger Anomaly Dataset with an additional characterization of the percentage change in classification performance measured with the F1 score, with respect to the performance in the 100% trigger case depicted in Table 4.1.

able to showcase good performance and learn good representations of trigger-infested and clean circuit instances. We see a deterioration in performance of the logistic regression and the GBC models (depicted in the last column in Table 4.2) indicating that these models are relatively less robust to contamination compared to the feed-forward neural network and random forest models.

Our third experimental setting aims to test the model performance in the presence of an *adaptive* adversary. In section 4.7, we discuss the experimental details and model classification performance in this setting.

Through our various experiments we have found that as circuit sizes increase our methods scale linearly. This allows us to examine large or small circuits for trojans without seeing a huge increase in the time.

4.7 Adaptive Attacker

In order to bypass detection by any of the deep learning models utilized in this paper, an adaptive attacker may attempt to modify the trigger. One option would be to change the trigger to be a logical equivalent. They may change the gate types of the trigger without affecting the functionality. For example, an attacker may choose to replace `AND` gates with the logical equivalent shown in Fig. 4.6 of three `NOT` and an `OR` gate. This `AND` gate equivalent could be substituted for any number of the gates in the trigger. An adaptive attacker may also attempt to change the size of the trojan triggers in an attempt to thwart detection.

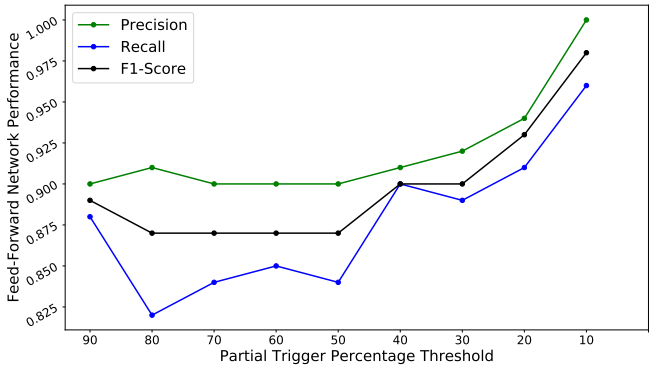


Fig. 4.5: Performance characterization of 2 layer Deep Feed-Forward Neural Network model with change in partial trigger percentage threshold. A particular trigger percentage threshold indicates that all circuits which contained less than the specified percentage of a trigger were marked as clean instances while the circuits with partial (or complete) triggers greater than the specified trigger percentage were marked to be anomalous instances. We notice that the model performance increases with decrease in partial trigger percentage threshold, indicating that the model learns better representations when trained on a greater variety of partial / complete triggers which occurs at lower trigger percentage thresholds.

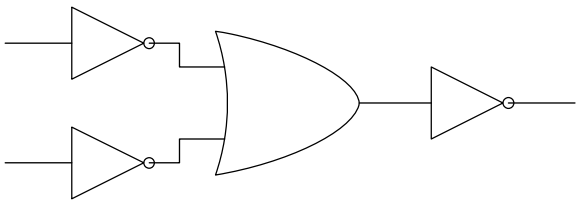


Fig. 4.6: Depiction of a logical equivalent to an `AND` gate by three `NOT` gates and an `OR` gate.

Both of these scenarios are considered.

4.7.1 Logical Equivalent Gates

In order to determine how successful and adaptive attacker would be we created trigger-inserted feature sets containing `AND` gates replaced by `AND` equivalents. Initially these features sets were tested on the previously trained feed-forward neural network. A second trial was conducted in which a second feed-forward neural network was trained using only the `AND` equivalent triggers as the positives for being trigger-inserted. Then a third feed-forward neural network was trained using the original training data augmented with the trigger-inserted inverse node fanins with `AND` equivalent gates. The results of the separate verification set held off until after training can be seen in Table 4.3.

Training	Trigger. Precision	Trigger. Recall	Trigger. F1
Original	1.00	0.15	0.22
`AND` Equiv.	1.00	1.00	1.00
Both	1.00	0.99	1.00

Table 4.3: Trigger detection performance against an adaptive attacker utilizing `AND` equivalent logic in the triggers.

The results show that the original feed-forward neural network would not be able to detect `AND` equivalent logic placed within the triggers. Although in these trials the triggers it labeled as such were correct, it missed far too many for it to be considered effective. When the network was trained with the `AND` equivalent triggers as indicated by the `AND` Equiv. row, it was able to be trained and identify all `AND` equivalent triggers. What is even more interesting is that when the training data contained both the original triggers with `AND` gates and the `AND` equivalent triggers it was able to successfully identify either type as a trigger.

4.7.2 Trigger Size Manipulation

In order to determine the likelihood of successfully bypassing detection by deep learning an adaptive attacker may attempt to modify the size of the triggers. We created several test sets including triggers containing four, eight, sixteen, and thirty-two inputs. These were all tested against the original feed-forward neural newtork and the results can be seen in Table 4.4.

Input Size	Trigger. Precision	Trigger. Recall	Trigger. F1
Four	0.99	1.00	1.00
Eight	1.00	1.00	1.00
Sixteen	1.00	1.00	1.00
Thirty-two	1.00	0.99	1.00

Table 4.4: Trigger detection performance against an adaptive attacker utilizing varying sized triggers.

As can be seen by the results in Table 4.4, size did not affect the ability for the neural network to efficiently label the triggers as such. Note that no further training was required,

but the neural network was able to identify larger triggers without being trained specifically to do so.

4.8 Conclusions and future work

In this paper we have proposed a methodology that allows arbitrary VHDL or Verilog circuits to be converted into a circuit adjacency matrix and then into the inverse node fanin form. We have found the inverse node fanin representation of circuit data to be an excellent way to present data to deep learning models. Others may find additional ways to take the data found in the circuit adjacency matrix form and transform it into useful representations for consumption by a deep learning model.

We have also shown by demonstration that several deep learning models are capable of identifying hardware trojan triggers. As discussed in the Results section, the feed-forward, RF, GBR, and Logistic Regression models were all able to identify hardware trojan triggers comprised of `AND` and `NOT` gates.

Using deep learning to identify hardware trojans does not rely on either expert feature selection or on golden model verification. The training data used in these experiments can be further enhanced to produce better results and more broad detection of hardware trojans in the future. We will also enhance the modeling architecture to perform effectively in the presence of an adaptive adversary by incorporating an adversarial learning approach using Generative Adversarial Networks (GAN).

REFERENCES

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, vol. 27, no. 1, 2010.

[2] U.S. department of commerce bureau of industry and security office of technology evaluation, "Defense industrial base assessment: Counterfeit electronics," http://www.bis.doc.gov/defenseindustrialbaseprograms/osies/defmarketresearchrpts/final_counterfeit_electronics_report.pdf, 1999, accessed: 2012-05-01.

[3] Committee on Armed Services, United States Senate, "Inquiry into counterfeit electronic parts in the department of defense supply chain," http://www.armed-services.senate.gov/Publications/Counterfeit%20Electronic%20Parts.pdf, 1999, accessed: 2012-05-01.

[4] J. Robertson and M. Riley, "The big hack: how china used a tiny chip to infiltrate us companies," *Bloomberg Businessweek*, vol. 4, 2018.

[5] J. Cross and M. Staff, "Apple strongly denies bloomberg's chinese hacking report, call for retraction," *Macworld*, 2018.

[6] J. Robertson and M. Riley, "The big hack: Statements from amazon, apple, supermicro, and the chinese government," *Bloomberg Businessweek*, 2018. [Online]. Available: https://www.bloomberg.com/news/articles/2018-10-04/the-big-hack-amazon-apple-supermicro-and-beijing-respond

[7] M. Dorning, "U.s. agency backs tech firms that deny china hacked their system," *Bloomberg Businessweek*, 2018. [Online]. Available: https://www.bloomberg.com/news/articles/2018-10-07/dhs-backs-u-s-tech-companies-denying-china-hacked-their-systems

[8] G. Faulconbridge and J. Menn, "Uk cyber security agency backs apple, amazon china hack denials," *Reuters*, 2018.

[Online]. Available: https://www.reuters.com/article/us-china-cyber-britain/uk-cyber-security-agency-backs-apple-amazon-china-hack-denials-idUSKCN1MF1DN

[9] N. K. Brar, A. Dhindsa, and S. Agrawal, "Prevention of hardware trojan by reducing unused pins and aes in fpga," in *Recent Findings in Intelligent Computing Techniques*. Springer, 2019, pp. 105–113.

[10] C. Dong, G. He, X. Liu, Y. Yang, and W. Guo, "A multi-layer hardware trojan protection framework for iot chips," *IEEE Access*, 2019.

[11] Q. Bi, N. Wu, F. Zhou, J. Zhang, M. R. Yahya, and F. Ge, "Fault attack hardware trojan detection method based on ring oscillator," *IEICE Electronics Express*, pp. 16–20 190 143, 2019.

[12] A. Malekpour, R. Ragel, D. Murphy, A. Ignjatovic, and S. Parameswaran, "Hardware trojan detection and recovery in mpsocs via on-line application specific testing," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2019, pp. 1–6.

[13] K. S. Subramani, A. Antonopoulos, A. A. Abotabl, A. Nosratinia, and Y. Makris, "Demonstrating and mitigating the risk of a fec-based hardware trojan in wireless networks," *IEEE Transactions on Information Forensics and Security*, 2019.

[14] V. Jyothi and J. J. Rajendran, "Hardware trojan attacks in fpga and protection approaches," in *The Hardware Trojan War*. Springer, 2018, pp. 345–368.

[15] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware." *Leet*, vol. 8, pp. 1–8, 2008.

[16] J. Zhang and Q. Xu, "On hardware trojan design and implementation at register-transfer level," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2013, pp. 107–112.

[17] A. Waksman, M. Suozzo, and S. Sethumadhavan, "Fanci: identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the 2013 ACM*

SIGSAC conference on Computer & communications security. ACM, 2013, pp. 697–708.

[18] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 153–166.

[19] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "Veritrust: Verification for hardware trust," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, 2015.

[20] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 652–660.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, 2012, pp. 1097–1105.

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 91–99. [Online]. Available: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf

[23] P. Fortuna and S. Nunes, "A survey on automatic detection of hate speech in text," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 85, 2018.

[24] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *AAAI 2015*, 2015.

[25] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut, "Text summarization techniques: a brief survey," *arXiv preprint arXiv:1707.02268*, 2017.

[26] M. Gambhir and V. Gupta, "Recent automatic text summarization techniques: a survey," *Artificial Intelligence Review*, vol. 47, no. 1, pp. 1–66, 2017.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[28] S. Bhasin and F. Regazzoni, "A survey on hardware trojan detection techniques," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2021–2024.

[29] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware trojan attacks: threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[30] H. Li, Q. Liu, J. Zhang, and Y. Lyu, "A survey of hardware trojan detection, diagnosis and prevention," in *2015 14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)*. IEEE, 2015, pp. 173–180.

[31] H. Li, Q. Liu, and J. Zhang, "A survey of hardware trojan threat and defense," *Integration*, vol. 55, pp. 426–437, 2016.

[32] M. Abramovici and P. Bradley, "Integrated circuit security: new threats and solutions." *CSIIRW*, vol. 9, pp. 1–3, 2009.

[33] F. Courbon, P. Loubet-Moundi, J. J. Fournier, and A. Tria, "A high efficiency hardware trojan detection technique based on fast sem imaging," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 788–793.

[34] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using ic fingerprinting," in *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007, pp. 296–310.

[35] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 396–410.

[36] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *2009 IEEE International high level design validation and test workshop*. IEEE, 2009, pp. 166–171.

[37] N. Yoshimizu, "Hardware trojan detection by symmetry breaking in path delays," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014, pp. 107–111.

[38] X. Cui, E. Koopahi, K. Wu, and R. Karri, "Hardware trojan detection using the order of path delay," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 3, p. 33, 2018.

[39] S. Narasimhan, X. Wang, D. Du, R. S. Chakraborty, and S. Bhunia, "Tesr: A robust temporal self-referencing approach for hardware trojan detection," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2011, pp. 71–74.

[40] X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga, "Ensemble deep learning for regression and time series forecasting," in *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*. IEEE, 2014, pp. 1–6.

[41] P. Filonov, F. Kitashov, and A. Lavrentyev, "Rnn-based early cyber-attack detection for the tennessee eastman process," *arXiv preprint arXiv:1709.02232*, 2017.

[42] P. Filonov, A. Lavrentyev, and A. Vorontsov, "Multivariate industrial time series with cyber-attack simulation: Fault detection using an lstm-based predictive data model," *arXiv preprint arXiv:1612.06676*, 2016.

[43] Y. Miao, M. Gowayyed, and F. Metze, "Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2015, pp. 167–174.

[44] M. L. Seltzer, D. Yu, and Y. Wang, "An investigation of deep neural networks for noise robust speech recognition," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 7398–7402.

[45] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[46] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[47] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *(ICML)*. JMLR. org, 2017, pp. 1263–1272.

[48] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, "Spectral networks and locally connected networks on graphs," in *ICLR, CBLS, April 2014*, 2014.

[49] M. T. C. Wang, *Introduction to Hardware Security and Trust*. 233 Spring Street, New York, NY 10013: Springer, 2012.

[50] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE, 2008, pp. 15–19.

[51] H. Salmani and M. Tehranipoor, "Trojan benchmarks," https://www.trust-hub.org/benchmarks/trojan, accessed: 2019-06-28.

[52] P. P. Chu, *RTL hardware design using VHDL: coding for efficiency, portability, and scalability.* John Wiley & Sons, 2006.

[53] D. Lampret. (2019) Opencores. [Online]. Available: https://opencores.org/

[54] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch," *Computer software. Vers. 0.3*, vol. 1, 2017.

[55] F. Wu, T. Zhang, A. H. d. Souza Jr, C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," *arXiv preprint arXiv:1902.07153*, 2019.

[56] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[57] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *(EMNLP) 2014*, Oct. 2014.

CHAPTER 5

CONCLUSION

The work in this dissertation has furthered the field of hardware security in several ways. Specifically, it has dealt with vulnerabilities of ICs. It has covered the topics of vulnerabilities in a Tiva C brought about by an onboard ROM containing factory-installed libraries which allowed for a return-oriented programming attack. These libraries contain a Turing-complete gadget set which can be utilized to erase and reprogram the Flash memory of the device. This would allow an attacker to rewrite the program of the microcontroller. The work in this dissertation also covers the topic of hardware trojans, another potential vulnerability of ICs today. It was shown that an important paper which describes a method to manufacture ICs in a way to prevent hardware trojan implantation did not cover all classes of circuits. In fact a class of circuits exists, namely highly redundant circuits, for which the methods proposed by that paper do not provide the security that was claimed. It is important for everyone to know the limitations of excellent methods of hardware trojan prevention so that they can be used where appropriate, and not used when they are found lacking. Lastly, a new form of hardware Trojan detection was discussed. This is a new application for state of the art methods of deep learning. Deep learning models were used to detect hardware trojan triggers. In order to use the deep learning models new methods to create data sets were described. These datasets were made publicly available.

The contributions made in each of the papers included in this dissertation are shown below:

- Chapter 2 made the following contributions:

    1. The ability to create a gadget set capable of erasing flash memory. This is the first step in taking control of a microcontroller and could result in a denial of service.

2. The ability to create a gadget set capable of programming the region of flash memory that was previously erased. This is the second step in taking control of a microcontroller.
3. The ability to create a Turing-complete gadget set from the TivaWare ROM. This allows for arbitrary code execution with ROP.
4. That modern energy-efficient embedded devices lack sufficient security assurances for mission-critical applications.
5. Demonstration of actual use of the Turing-complete gadget library found in the ROM. This goes beyond the theoretical and demonstrates that these gadgets actually work when used as part of an ROP procedure.

- Chapter 3 made the following contributions:

1. A discussion demonstrating how redundant circuitry weakens the defenses proposed in [1].
2. A demonstration showing that the wire lifting procedure from [1] does not provide the intended amount of security to highly redundant circuits, particularly cryptographic ciphers. This is shown on both DES and AES circuits. These circuits were chosen to show that the wire lifting procedure does not frustrate the implantation of a hardware Trojan for either a Feistel structure or an SPN.
3. An approach which allows a hardware Trojan to be inserted into a DES circuit that has undergone the wire lifting procedure is explained. This is a modification to the Trojan described in [1]. This new approach entails attacking all portions of the circuit that are indistinguishable from one another at the same time, instead of choosing one portion and only attacking it, or attacking each portion one at a time.
4. A second approach is introduced which allows a hardware Trojan to be inserted into an AES circuit that has undergone the wire lifting procedure. This method is similar to the approach outlined for inserting a Trojan in a DES circuit, but

does not attack every indistinguishable portion of the circuit at the same time. It attacks enough of the indistinguishable portions of the circuit to be able to recover the key through an exhaustive search. This allows the size of the Trojan to be less than it would have to be if every portion of the circuit indistinguishable from another were to be attacked. This method can also be applied to a DES circuit, increasing the probability of success of against a DES circuit to 100%.

- Chapter 4 made the following contributions:

  1. A methodology to create datasets for hardware trojan research is presented. This allows for research to be continued in the field of hardware trojan detection in general, not only while applying deep learning models to trigger detection.
  2. A dataset of trigger-inserted circuit adjacency matrices is provided. This dataset contains 14,628 individual instances of hardware trojan triggers inserted into various circuits. The triggers vary in size, trigger conditions, and location they are inserted in the circuit. The next best currently available database contains 5 instances of a trigger based hardware trojan.
  3. A methodology to create feature vectors from circuit adjacency matrices is set forth. The creation of feature vectors is so important when attempting to apply deep learning models to any field. The work to determine how to represent the data has been set forth so that future researchers can concentrate on other ways to improve the results.
  4. The groundwork for a new application of deep learning: using state-of-the-art models to identify hardware trojans is presented. This truly is the groundwork in this field and it will act as the baseline for future researchers to expand upon.

REFERENCES

[1] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara, "Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 495–510.

CURRICULUM VITAE

Nathanael R. Weidler

Published Journal Articles

- Weidler, N.R., Brown, D., Mitchell, S.A., Anderson, J., Williams, J.R., Costley, A., Kunz, C., Wilkinson, C., Wehbe, R. and Gerdes, R., 2019. Return-oriented programming on a resource constrained device. Sustainable Computing: Informatics and Systems, 22, pp.244-256.

Published Conference Papers

- Weidler, N.R., Brown, D., Mitchel, S.A., Anderson, J., Williams, J.R., Costley, A., Kunz, C., Wilkinson, C., Wehbe, R. and Gerdes, R., 2017, August. Return-Oriented Programming on a Cortex-M Processor. In 2017 IEEE Trustcom/BigDataSE/ICESS (pp. 823-832). IEEE.

Submitted Papers

- On the Limitations of Obfuscating Redundant Circuits in Frustrating Hardware Trojan Implantation. Submitted to the Journal of Hardware and Systems Security June, 2019. Order of Authors: Weidler, N.R., Gerdes, R., and Chantem, T.

- Hardware Trojan Detection Without a Golden Model Using Deep Learning. Submitted to ASHES - Attacks and Solutions in Hardware Secutiry July, 2019. Order of Authors: Weidler, N.R., Muralidhar, N, Gerdes, R.