The Pennsylvania State University

The Graduate School

AN EXPLORATION INTO THE SECURITY VIABILITY OF RISC-V SYSTEMS AND SUPPLY CHAIN

A Dissertation in

Computer Science and Engineering

by

Asmit De

© 2021 Asmit De

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2021

The dissertation of Asmit De was reviewed and approved by the following:

Swaroop Ghosh
Joseph and Janice Monkowski Career Development Associate Professor
Computer Science and Engineering, Penn State
Dissertation Advisor
Chair of Committee

Trent Jaeger
Professor
Computer Science and Engineering, Penn State

John Sampson
Associate Professor
Computer Science and Engineering, Penn State

Peng Liu
Raymond G. Tronzo, MD Professor of Cybersecurity
College of Information Sciences and Technology, Penn State

Chitaranjan Das
Head of the Department and Distinguished Professor
Computer Science and Engineering, Penn State
Chair of the Graduate Program

Abstract

RISC-V is a promising open-source architecture that have gained a lot of traction in the recent years. The architecture is primarily targeted towards low-power embedded devices and SoCs, however, due to its flexibility and customizability it can also be adapted to other applications such as machine learning accelerators and data-center microprocessors. Due to the open platform, RISC-V SoCs allow rapid prototyping and faster time-to-market. This is made possible due to the distributed nature of the semiconductor supply chain, such as sourcing certain components from IP vendors and using third party foundries. However, this also poses security concerns. For embedded devices such as IoTs, RISC-V often allows programs and applications to run bare-metal on the system. Such applications have full access to the underlying memory subsystem, and thereby opens up the risk of memory vulnerabilities, if not designed with proper security practices. Moreover, the involvement of several third parties also makes the supply chain vulnerable to IP theft and IC piracy. In this dissertation, we explore different hardware accelerated security designs for mitigating several system security vulnerabilities. To further secure the design development stack of SoCs, we explore hardware security principles for preventing reverse engineering and IP design theft.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Throughout my research journey I have received a lot of help and guidance, for which I am extremely grateful. I would first like to thank my advisor Dr. Swaroop Ghosh, for mentoring me and helping me achieve my goal. Without his patience, constant motivation, and support this would not have been possible.

My sincerest thanks and gratitude go to my fellow labmates and collaborators who helped in shaping parts of this thesis. To Md Nasim I Khan, thank you for being my research buddy from the very first day. It was wonderful sharing this journey together, and it goes without saying that your contributions were invaluable. To Aditya Basu, thank you for the interesting discussions and for sharing your Linux expertise, which helped me overcome quite a few technical roadblocks.

I would like to acknowledge my colleagues from my internship at SiFive, particularly my mentor Ryan Macdonald, who helped me understand CHISEL the RISC-V SoC architecture. Thank you for providing a fulfilling learning experience, it had a direct impact on my research work.

Thank you to my friends Abhianshu, Abhishek, Anirban, Kushagradhi, Samik and Souransu for their encouragement and inspiration, and for providing happy distractions to rest my mind outside of work.

Lastly, I would like to thank my parents Asit Kumar De and Banhisri De for their unwavering support and encouragement. You helped me become who I am and thank you for always being there.

Funding Acknowledgements: This dissertation is based on work supported in parts by Semiconductor Research Corporation (SRC) [#GRC-2727.001, #GRC-2847.001, #GRC-3011.001], National Science Foundation (NSF) [#CNS-1722557, #CNS-1801534, #CCF-1718474, #DGE-1723687 and #DGE-1821766], and Defense Advanced Research Projects Agency (DARPA) Young Faculty Award [#D15AP00089].

The opinions, findings and conclusions in dissertation are that of the authors and do not necessarily reflect the views of SRC, NSF, or DARPA.

To my parents

Chapter 1

Introduction

There has been an exponential growth in the number of computing systems and connected devices in the past few decades. Internet of Things (IoT) devices and low-power embedded System-on-Chip (SoC) devices are one of the fastest growing compute segments. These devices will change the way we interact with the environment, thereby spawning a whole array of new application domains like home automation such as, smart bulbs, automated temperature controllers and voice assistants, wearable technology such as fitness bands and smart watches, healthcare such as, medication dispensing systems, industries such as weather and climate monitoring, and, agriculture machinery and so on. To cater to these vast array of application areas, these devices need to be small, fast and energy efficient. Vast majority of embedded SoCs are energy constrained and operate with limited resources.

With the growth of such devices, there has also been an exponential growth in the number of cyber-attack incidents resulting in significant financial loss and national security concerns. Broadly, the security threats are targeted on software programs, operating system (OS), network and the underlying hardware, with the intention to steal information, install virus or possess control over the target machine. Software security threats include control flow hijacking using Return Oriented Programming (ROP), buffer overflow, memory

safety violation, pointer corruption and string manipulation, to name a few. Network security threats include SQL injection and Denial-of-Service (DoS), and operating system threats include illegitimate access, information leakage and integrity violation. Hardware security threats include information leakage using hardware side-channels, hardware trojans and intellectual property (IP) theft.

Several security solutions have already been built to counteract such threats. For example, to counteract software and system security threats, software-based memory protection and access control techniques have been explored. These solutions although effective in mitigating such threats, often are designed for high performance devices such as server-scale or complex microprocessors. As such, they are often too complex or not suitable to be deployed on low power embedded devices due to performance and power budget constraints.

Embedded SoCs can be based on different architectures. For example, Intel has Atom and Quark SoCs based on the x86 architecture. Qualcomm provides Snapdragon SoCs based on the ARM architecture. Recently, there has been significant development RISC-V, a new open and modular architecture. RISC-V has been gaining popularity since 2011 and is already being adopted for commercial SoC development. For example, Huami has developed smartwatches running on RISC-V based SoC. Seagate has developed their disk controllers based on RISC-V, and Western Digital also has developed RISC-V based cores for use in their own products. SiFive, one of the pioneers behind the development and maintenance of the RISC-V architecture has developed their proprietary RISC-V core IPs that are being used by several companied in their compute products. RISC-V is a fully modular architecture and can be scaled with ISA extensions as required by the domain

application. RISC-V is also open-source, which allows anyone to make domain specific customizations to the ISA and develop SoCs based on it. Due to the flexibility provided by the architecture, RISC-V based cores can cater to a wide range of application domains – from tiny, embedded controllers to server-scale microprocessors. However, as of now, RISC-V is a viable alternative to ARM for low power embedded SoCs because of its performance benefits and non-proprietary ISA.

It is thereby essential that there is a low overhead security design for RISC-V based SoCs. Given the wide attack surface, the development of RISC-V SoCs need to have a complete security solution. Security solutions can be designed in the software layer as well as in the hardware layer. The most common security solutions are designed in software. The benefits of a software solution are ease of development, flexibility, and portability. However, software solutions also tend to be performance intensive, and needs to rely on the security provided by the underlying hardware. A pure hardware solution offers native performance and can offer greater security since it is built from the ground up. However, these also tend to be less flexible, as they target specific hardware and architectures. In hardware there is a tradeoff between scalability and performance. Application specific hardware such as ASICs provide the best performance with little room for flexibility, while more generic hardware such as FPGAs are less performant with a high degree of customizability. In this dissertation, we explore hardware assisted design solutions to provide end-to-end security guarantees for the SoC system and hardware, with the goal of high flexibility as well as high performance.

1.1 Thesis Statement

Although RISC-V systems are showing promising growth potential, security solutions in RISC-V SoCs are still in its infancy. There are several challenges to designing end-to-end security solutions. First, most software-based system security solutions that are ported from other architectures tend to be performance intensive, since they might not be natively developed for RISC-V. Second, embedded SoCs are power constrained, hence running intensive security operations are impractical. Lastly, because of the open nature and easy design-to-market cadence of RISC-V SoCs, there is an inherent lack of IP and chip integrity due to the distributed semiconductor supply chain.

The research presented in this dissertation addresses these system security and hardware security challenges in a practical and low overhead manner. This requires exploring the hardware security primitives for use in system security applications, exploring hardware assisted security architecture implementations of well-known system-security solutions and laying out a design implementation strategy that is scalable. This leads to the following thesis statement:

The design and use of hardware accelerated security modules for embedded RISC-V systems enables protection against memory corruption vulnerabilities at low performance overheads. The design principles needed for achieving this goal are customizability, modularity, and decoupled security architectures from the primary core. Ensuring the integrity of supply chain for RISC-V SoCs by unique circuit level hardware camouflaging designs provides the foundation for a secure and trustworthy RISC-V system.

The focus of this thesis is on security design implementation strategies on RISC-V systems. For system memory protection, we primarily explore hardware-assisted implementation in modular decoupled fashion that allows rapid design-to-market on RISC-V SoCs, without changes to the core. For supply chain integrity, we explore custom circuit-level design strategies using non-conventional gate designs.

1.2 Contribution and Dissertation Outline

In this dissertation, our primary contributions are towards the security of SoCs encompassing two major areas: (i) software and (ii) hardware. In the software domain, we explore SoC memory corruption vulnerabilities (chapters 3 and 4). In the hardware domain, we explore SoC supply chain vulnerabilities (chapters 5 and 6). The following is a summary of contributions in this dissertation based on the above thesis statement:

- *In Chapter 3, we identify one of the primary causes of a common memory corruption vulnerability in an SoC – buffer overflow on the stack.* We present PUFCanary, a stack canary design using hardware security primitives, such as Physically-Unclonable Functions and True Random Number Generators. PUFCanary can protect buffer boundaries and detect overflow of stack buffers in a fine-grained and lightweight manner. We also present FIXER, a hardware assisted shadow stack and policy matrix design decoupled from the primary core. FIXER is implemented as a security coprocessor extension, that can mitigate Return Oriented Programming (ROP) attacks originating from a buffer overflow.

- *In Chapter 4, we explore protection of the SoC heap memory from heap buffer overflows and use after free attacks.* We present HeapSafe, a hardware assisted pointer protection and bounds checking design. HeapSafe, implemented as a security coprocessor, uses pointer tagging and metadata propagation with metadata saving and validation in hardware to provide protection against heap memory corruption.
- *In Chapter 5, we delve deeper into the underlying hardware of SoCs and explore potential hardware Trojans during SoC fabrication that can eventually lead to information leakage causing data privacy issues.* We present HarTBleed, a hardware Trojan design and a class of exploits that can obtain sensitive information from the address space of a running process. We use a capacitor based Trojan trigger that exploits the virtual addressing of L1 cache to activate a Trojan payload on the TLB.
- *In Chapter 6, we explore a few gate design techniques to prevent reverse engineering (RE) and intellectual property (IP) theft of hardware designs in SoCs.* We present Threshold Voltage based and Charge Trap based Logic Camouflaging techniques that can obfuscate logic design in the SoCs. Such gate camouflaging techniques increase the reverse engineering effort of an adversary significantly, thereby resulting in IP protection and SoC integrity.

Before we detail our specific technical contributions as outlined above, we begin with Chapter 2 presenting the relevant background information. We explain the system and hardware security threats we are trying to mitigate, as well as existing solutions in literature. We further describe the RISC-V SoC system Rocket-Chip, which is our primary implementation platform for our security architecture designs.

Chapter 2

Background Information

In this chapter, we discuss the background information and relevant concepts that form the basis of our work in security design. In the first section, we look at the design of System-On-Chip devices. Next, we explore common system security and hardware security vulnerabilities, and their existing solutions in literature. Finally, we detail the design of the RISC-V Rocket-Chip platform and explore how we can leverage the SoC subsystem to implement our security designs.

2.1 System-On-Chip Architectures

A System-On-Chip (SoC) is essentially an integrated circuit (IC) that integrates all parts of a compute system onto a small single platform. This is an alternate to assembling and interconnecting different electronic components on a printed circuit board (PCB) or a motherboard. One of the primary motivators behind SoC development is the ability to build tightly coupled circuits with a small area and power footprint, while retaining high performance. This results in reduction of energy waste and save on development costs.

With the growth of IoT devices, SoCs have seen widespread use. They are now present in almost every technological product we use in our everyday lives, such as,

smartphones, smartwatches, cameras, networking devices, game consoles, automobiles, point-of-sale terminals, to name a few.

Fig. 2-1 shows a typical SoC layout. A standard SoC is driven by a microprocessor, although some simpler designs may use a microcontroller instead. The microprocessor is based on a specific microarchitecture, such as ARM, x86, RISC-V, etc. The architecture ISA determines the instructions that the applications need to be compiled to. The SoC also has an internal random-access memory (RAM) system, and read-only memories (ROM). Other than these basic components, SoCs provide a wide variety of input and output peripheral connectivity, such as I2C, SPI, UART, GPIO, CAN, USB, EMAC, etc. on

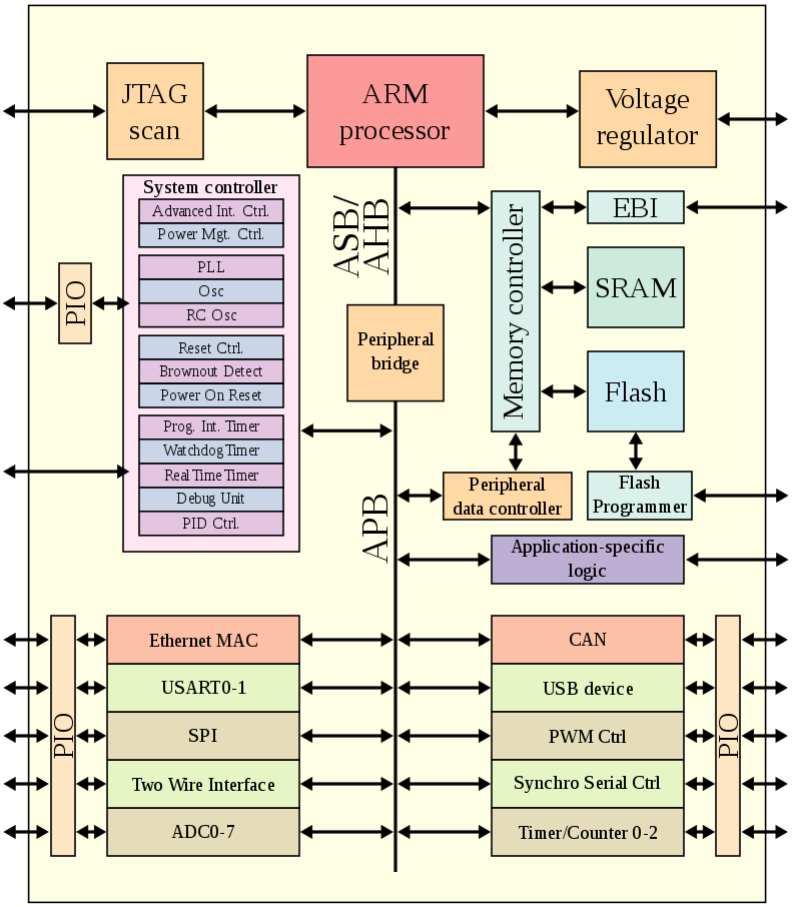


Figure **2-1**: System-on-chip layout. *[Source: Wikipedia]*

peripheral buses. It may also provide display outputs and dedicated graphics processing engine. It can house JTAG interface for debugging and tracing, DMA and memory controllers for management of data transfers. There may also be analog devices, ADC controllers, application-specific custom logic blocks, cryptographic blocks, etc. All these components communicate with each other over a shared interconnect network using a specific protocol. The SoC components are internally managed with system controller that consists of voltage regulators, power management modules, clock generators and timers, debug modules, etc.

An SoC manufacturer is often an SoC ‘integrator’. It means that a manufacturer most often does not develop all the SoC components in-house. The microprocessor cores may be licensed and purchased from a core IP vendor such as ARM. The SRAM/Flash memories may be sourced from some other 3rd party vendors such as Synopsys. Hence, when a complete SoC design RTL is generated, it can contain a mix of in-house proprietary IPs, open-source IPs, and 3rd party commercial IPs. Furthermore, the actual fabrication of the SoC may again be done by an external foundry, such as TSMC. This is what is termed as the ‘distributed semiconductor supply chain’.

2.2 Security Vulnerabilities

Traditional computing systems are inherently vulnerable to a wide attack surface from the topmost application level to the systems architecture and even hardware level, leading to serious security and integrity concerns such as leaking private SSH keys, launching Denial-of-Service (DOS) attacks, malicious Trojan insertion, and IP theft.

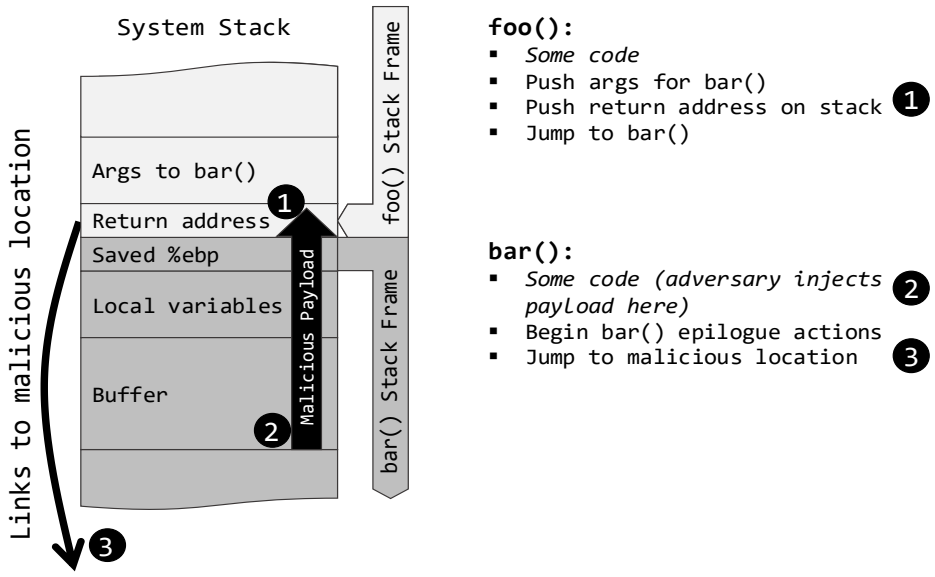


Figure **2-2**: Buffer overflow exploit.

2.2.1 System memory vulnerabilities

Programming languages like C which are closer to the hardware, provide a lot of flexibility in terms of memory and IO access to facilitate system and device level programming. However, this also means that such languages which are weakly typed, often tend to have inherent security deficiencies and can lead to vulnerabilities if not used with proper and secure practices. Buffer overflow [1] is the most commonly exploited vulnerability that can cater to a wide attack surface. In a program without bounds checking, an adversary can overload a user input with excess data that can overrun the buffer capacity and overwrite nearby memory locations with potentially malicious data (**Error! Reference source not found.**. 2-2). Buffer overflows can occur in a process's stack, or in the heap. A buffer overflow in a process's address space can lead to several exploits such as Control-Flow Integrity violations, Data-Flow Integrity violations, Return-oriented-Programming

attacks [2-5], etc. Although stack-based buffer overflows are more common, heap buffer overflows can also occur [6-8], and it is more difficult to protect against. A heap buffer overflow occurs when a pointer used to write data to the buffer goes beyond the allocated region of the heap and overwrites the critical data, potentially allowing an adversary to launch attacks, such as, write-what-where, malicious shellcode execution, etc. This is possible due to the lack of bounds checking (a technique that allows validation of a pointer's access bounds) in pointers. Unlike a stack-buffer overflow-based attack, which occurs in tandem with the rolling/unrolling of the process's stack, heap-buffer overflows are arbitrary and can happen anywhere at any point during the program's execution.

Defense mechanisms

Stack canaries [9] are *sacrificial* words placed on the stack at stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer in order to overwrite the return address, the canary word will also be overwritten. Before returning in call stack, canary word is checked, and if modified, the return address is assumed to be compromised, and the program is halted.

Data Execution Prevention (DEP) [10] is employed to prevent an adversary from injecting malicious code onto the stack. Memory pages are marked W⊕X, meaning, a page can either be executable (code) or be writable (stack, heap), but not both. This prevents an adversary from executing malicious code from the stack. However, an adversary can return to existing code in the program or functions in the linked library using gadget chains (return-to-libc attack).

Address Space Layout Randomization (ASLR) [11] randomizes the code, stack, heap, and shared library locations on the address space, to make it difficult for the adversary to determine the specific addresses and launch attacks. However, buffer overread and side-channel vulnerabilities can be used by an adversary to reverse engineer the randomized address.

Control Flow Integrity (CFI) [2] involves statically computing a valid control flow graph (CFG) of the program and ensuring that during runtime, the program abides by that CFG. A coarse-grained approach to ensuring control flow integrity while returning from functions is the use of a shadow stack (a separate stack residing in a secure memory location) [12]. On each function call, the return address is saved on the shadow stack alongside being put on the stack normally. While returning from a function, the return address on the stack is validated against the one on the shadow stack. On mismatch, it is assumed that the return address has been compromised and the program execution is halted. However, a shadow stack can be expensive and can hurt performance since the pages housing the shadow stack may not be present in cache and will require hundreds to thousands of cycles to bring the page onto the cache and perform the validation. Several software and compiler level systems have been proposed in literature for supporting shadow stack [13-14].

Even with the presence of a shadow stack, an adversary can bend the control flow of a program. To prevent such incorrect control flows for indirect calls, the program is first analyzed to compute a coarse-grained or fine-grained CFG [2]. A control-flow policy matrix can then be created from the CFG that specifies the allowed call targets for each call site. During execution of the program, for each indirect call, the policy matrix is looked

up to determine the validity of the call target. However, this approach still suffers from similar performance degradation if the policy resides in memory. Compile-time and runtime enforcement of CFI have been shown in [15-16]. Lazy CFI [17-18] can somewhat alleviate the performance loss, but that leaves room for generating false negatives.

Pointer protection is a common technique that provides protection to pointers based on bounds checking of allocated memory [19-20]. Such techniques associate additional metadata structures with the pointers and provide software runtimes to perform access validation. A similar metadata association approach has also been explored for objects instead of pointers [21]. A better alternate approach to pointer-based protection is low-fat pointers [22], where instead of using additional metadata structures with pointers, the authors have utilized the native pointer itself for storing the metadata, thereby preserving backwards compatibility. A few hardware based memory allocation and runtime monitoring approaches have also been explored in [23-24].

Secure hardware platforms e.g., ARM TrustZone [25] and Intel Software Guard Extensions (SGX) [26] isolate the hardware so that the access to systems assets is restricted. Hardware acceleration of security validation has been proposed to address the performance impact partially while covering a subset of security threats e.g., Intel CFI Enforcement Technology (CET) [27] to protect against control-flow hijacking. Intel Memory Protection Extensions (MPX) [28] with extended instruction set architecture is developed to prevent memory safety violations such as buffer overflow, heap overflow and pointer corruption. Intel Transactional Synchronization Extensions (TSX) [29] exposes and exploits hidden concurrency in multi-threaded applications. Intel PT [30] logs TSX events when a transaction begins, commits or aborts. It has been shown in [31] that tagging of code and

data using software-defined metadata and processing the tag using custom designed processor can detect ROP, code reuse, buffer overflow, code injection, memory safety violation and pointer corruption. Although effective, this new architecture cannot be readily deployed due to lack of re-configurability, and area, energy and performance overhead. Other hardware-assisted techniques to protect forward and backward edges in control flow are proposed in [32-35]. Data flow protection in stack and heap using hardware assistance is also proposed [23,36]. Specialized hardware stack redundancy systems have also been developed for embedded systems [37-40], however these are architecture dependent and cannot be updated post-deployment.

The common challenges associated with the existing secure hardware platforms include design overhead, lack of provisions to patch the design and keep pace with rapidly evolving threats, need of code changes or instrumentation of the program binaries, compiler modifications, and lack of adaptability to adjust the security level in runtime as needed. Furthermore, these platforms are associated with performance impact. To alleviate these issues, a decoupled architecture using hardware performance monitors implemented on a RISC-V coprocessor has been proposed in [41].

2.2.2 Hardware supply chain vulnerabilities

Semiconductor Intellectual Property (IP) is recognized as a driving force to accelerate the semiconductor market, allowing large scale Integrated Circuit (IC) design and manufacturing. However, current semiconductor market is plagued with counterfeit chips. This is primarily due to the distributed nature of the IC supply chain, which breaks

the line of trust, leading to IP theft [42-43]. This not only harms the economy of the semiconductor industry [44], but also poses a serious risk, especially in the security and defense sector [45].

Data Leakage with Hardware Trojans

Data leakage prevention (DLP) techniques usually offer protection of sensitive data by content monitoring and detection, tracking anomalies in data behavior [46], or apply cryptographic techniques such as encryption to secure the data [47-49]. However, these techniques only work at the application level. Data-flow analysis and dynamic taint tracking [50] are employed at the system level to ensure data protection of running processes. User-level processes are protected from code injection and code reuse by techniques such as CFI [2], DEP [10], ASLR [11], etc. Even with such protections in place, a bug was recently found in the OpenSSL cryptographic library that leaked data from memory using a buffer overread vulnerability, allowing attackers to eavesdrop on SSL/TLS secured communication channels [51]. User processes are separated from the OS kernel by employing protection rings in the CPU, which ensures that user space code cannot access data in kernel space. However, recent vulnerabilities have been found in the systems architecture, namely, Spectre [53] and Meltdown [54], which can take advantage of the speculative execution of out-of-order processors to gain unauthorized access to data from the CPU cache, and even leak kernel data. Existing OS level patches to handle Meltdown incur 0-30% performance overhead whereas cleaning branch predictors and branch target

buffers, Lfence etc. [54] for Spectre are expected to impose more serious performance overheads. Architecture design changes are also planned to prevent such vulnerabilities.

The underlying assumption of the above threats and their mitigation techniques is that the hardware itself is free from malicious tampering. However, the recently surfaced news such as tampering of server motherboards by Chinese manufacturers that affected top US companies like Amazon, Apple etc. [55] emphasize that this assumption might not be true due to involvement of untrusted third parties in the semiconductor manufacturing supply chain. Another popular incident is the hardware fabricated with hidden-backdoor to disable radars in Syria [56] to facilitate an attack.

In light of the above threat, the US Defense Advanced Research Projects Agency (DARPA) identified the trusted and untrusted steps of a supply chain [57] and initiated a Trust-in-IC (Integrated Circuit) program to develop tools and techniques to ensure that ICs are authentic and free of Trojans post-manufacturing. The Australian Department of Defence also raised an awareness of the threat and proposed broad classes of Hardware Trojans and countermeasures [58]. Research has also been conducted to develop efficient DLP frameworks for the semiconductor industry [59]. In spite of above efforts, the industry heavily relies on untrusted third parties for Intellectual Property (IP) and manufacturing to preserve the cost benefits.

Hardware Trojan [60] is a malicious modification in a circuit that is introduced during design and/or manufacturing process to force a chip to perform undesirable operations. Ideally, these modifications made to an IC should be detected during pre-Silicon verification and post-Silicon testing. In order to evade the structural and functional testing, an adversary designs the Trojan to activate only under certain rare conditions and

to remain undetected during the test phase. Hardware Trojan has two main components, namely, i) Trojan trigger, and ii) Payload [61]. Trojan Trigger is designed in a way that it gets activated under a certain unique condition which might not be possible to generate in test/validation phase, and thereby, remains undetected. Once triggered, the Trojan can cause Denial of Service (DoS), fault injection or information leakage [61-62].

Reverse Engineering and Intellectual Property Theft

Reverse Engineering (RE) of ICs is one of the most prominent methods of stealing the confidential IP to produce counterfeit ICs. It is a process of identifying its design, functionality, and structure. In the invasive RE process, the adversary de-packages the Integrated Circuit (IC) and takes pictures of each layer. The images of metal layers provide connectivity information whereas the image of base layer is employed to identify the gate functionality. Finally, the information obtained from images are merged to prepare a netlist for automatic synthesis unlocking the IP (Fig. 2-3). RE has been originally used by industries with the mindset of gathering information on its competitors such as, process

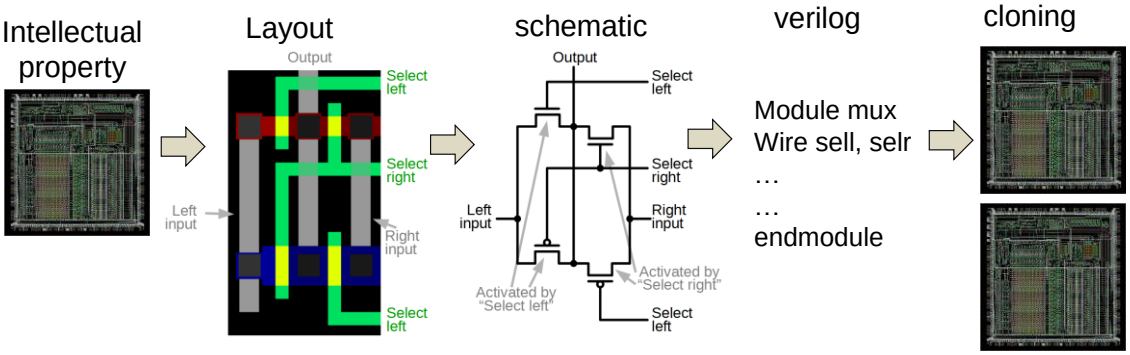


Figure **2-3**: Reverse engineering of IP: the chip is de-layered to identify the gate functionality and their connectivity which is used to reconstruct the schematic and netlist.

parameters (e.g., channel length, pitch of poly and metals), to confirm or debug the functionality of their own design, and to ensure the legitimacy of circuits from piracy. However, the advanced adversaries can exploit this technique with an ill intention to illegally steal the design to clone the IC from the netlist and siphon off the profit margin of the IP provider.

Defense Mechanisms

Split manufacturing [63] is a technique to protect the IP. However, it addresses the RE of IP and Trojan insertion during manufacturing process. In contrast to the regular manufacturing procedure, the front-end (transistors) is manufactured in an untrusted foundry whereas the back end (interconnect) is manufactured in trusted facility. This makes the RE and Trojan insertion more challenging for the adversary since the connectivity information is hidden in the untrusted foundry. Furthermore, since the front-end fabrication cost is higher than the back end, the cost benefit of outsourcing the fabrication is still preserved without increasing the security risks.

Obfuscation is another technique to hide functionality of a logic design. This exploits software/hardware to realize functionally original but difficult to reverse engineer IP designs. For sequential circuits, additional logic (black) states are introduced in the finite state machine [64-67], which allow the design to reach a valid state only using the correct key. In combinational logic, XOR/XNOR gates are introduced to conceal the functionality [68-69]. Watermarking and passive metering techniques are also proposed to detect IC piracy [70-71].

Gate camouflaging can affordably hide the logic functionality and make the RE process economically non-profitable or extremely difficult. The primary objective behind camouflaging is to hide the functionality of a few chosen gates (since the camouflaged gates are typically area, delay, and power intensive) to increase the RE effort while keeping area/performance/power overhead minimal. We note that careful camouflaging, albeit with a ~10-40% overhead, can increase the RE effort significantly [72]. Several camouflaging techniques have been proposed in prior work [72-78]. Gate camouflaging has been proposed using dummy contacts [72-73] which can realize three functions (AND/OR/XOR). Although dummy contacts hide the functionality, it comes at the cost of ~5X area/power overhead and requires a process change (hollow via). Additionally, the RE effort is not exhaustive. Programmable standard cells using control signals [74] can hide functionality. However, it incurs significant area overhead (control signal routing for each camouflaged gate) and requires a non-volatile memory to hold the configuration bits for control signals which can be compromised. The work in [75] uses filler cells to intentionally fill up unused space in the ASIC design. By connecting the filler cells with metals and vias, it creates a robust camouflaging to protect ASICs from RE. However, it leaves layout trace and careful RE reveals the fake gates. In [76] however, the silicide layer (coated over the source/drain of the transistors) on a selected substrate area are used to interconnect the source/drain active areas to camouflage multiple gates. Due to the sheer thinness of the silicide layer, observing it optically is exceedingly difficult, and RE by etching inadvertently etches the silicide layer as well.

2.3 Rocket-Chip SoC Platform

Our security architectures are based on Rocket Chip [77] (written in CHISEL [78]), an open-source parameterized system-on-chip (SoC) design generator. It generates synthesizable RTL for a RISC-V SoC. We use the RocketChip generator to generate the standard Rocket Core, a six-stage single-issue in-order pipeline processor that executes the 64-bit scalar RISC-V ISA (Fig. 2-4). The Rocket Tile consists of the scalar core, the L1 instruction and data caches, and the Rocket Custom Coprocessor (RoCC). The RoCC acts as an accelerator for the core and can be triggered by a set of custom instructions capable of mediating communication between the core and the RoCC over the RoCCIO interface.

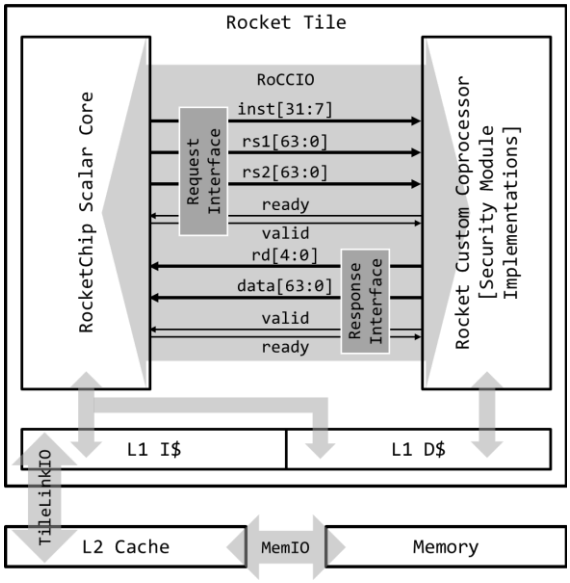


Figure **2-4**: RocketChip SoC architecture with Rocket Custom Coprocessor.

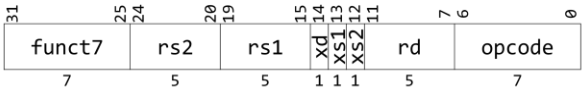


Figure **2-5**: RoCC instruction encoding.

Table **2-1**: RoCC Instruction opcodes.

RoCC Instruction	Opcode
custom0	0001011
custom1	0101011
custom2	1011011
custom3	1111011

The Rocket Custom Coprocessor (RoCC) instructions are processed by the Rocket Core and requests are passed to the coprocessor over the RoCCIO. The Rocket Tile communicates with the shared L2 cache over the TileLinkIO interface and with the DRAM memory over the MemIO interface.

RoCC Instructions: In general, 32-bit RoCC instructions extend the RISC-V ISA and are encoded as shown in Fig. 2-5. There are four custom instructions supported by the Rocket Chip as shown in Table 2-1.

The *xs1*, *xs2*, and *xd* bits control read and write processor of the base integer registers by the coprocessor instruction. If *xs1* is 1, then the 64-bit value in the integer register specified by *rs1* is passed to the coprocessor. If the *xs1* bit is clear, no value is passed over the RoCCIO interface. The *xs2* bit similarly controls whether a second integer register specified by *rs2* is read and passed to the RoCC interface. If the *xd* bit is a 1 and *rd* is not 0, the core will wait for a value to be returned by the coprocessor over the RoCC interface after issuing the instruction to the coprocessor. The value is then written to the integer register specified by rd. If the *xd* is 0 or *rd* is 0, the core will not wait for a value from the coprocessor. In all cases, bits 31-7 of the instruction are also passed to the coprocessor for further decoding, along with a 2-bit field indicating the major opcode (0/1/2/3). The coprocessor is responsible for signaling illegal instructions back to the core.

RoCCIO Interface: The coprocessor interacts with the Rocket core and the shared memory system via the standard RoCCIO interface (Fig. 2-4). Each portion of the interface is decoupled by standard queues and ready-valid signals.

The Rocket core initiates a coprocessor command by passing most of the RoCC instruction directly to the coprocessor, as well as the relevant register values.

- *inst* includes all but the last bits of the RoCC instruction.
- *rs1* and *rs2* contain the contents of the registers specified in the instruction if *xs1* and *xs2* are high, respectively. If the control signals in the instruction are low, then these fields will not contain valid data, even though a single valid signal is responsible for the entire interface.

If the instruction supplied to the RoCC set the *xd* bit, then the RoCC must eventually supply a response over the RoCC response interface.

- *rd* is the destination register supplied by the original instruction.
- *data* is the 64 bits of data to be written to that register.

In the following chapters, we design and implement several security solutions on the RocketChip SoC platform. Specifically, we leverage the Rocket Custom Coprocessor (RoCC) and use it as a hardware accelerator in our security designs. The modular design of RocketChip allows us to decouple our security design from the primary core, requiring no modifications to the core. The customizable RoCC allows us to easily implement it as a security coprocessor to work in tandem with the core. Finally, the parameterization provided by CHISEL in RocketChip allows us to generate different SoC configurations by plugging in and out different security coprocessor (RoCC) modules keeping the rest of the system same.

Chapter 3

Protecting the SoC Memory: Stack Exploit Mitigation

We will explore the security designs in a top-down manner, starting at the system memory, and then later down to the physical hardware. In this chapter, we start off with exploring memory corruption vulnerabilities on the stack memory. Stack is the most used memory in a process's address space. The stack houses a process's local data, aids in the rolling and unrolling of functions in the process's code. Stack buffers are contiguous memory locations in the stack memory that can be read or written using an iterative operation. For function calls, each function gets allocated a stack frame, with the return address of the function caller placed at the top of the stack frame. Once the function body is executed and the program needs to return to the caller, the stack frame is destroyed and the return address on the top is used to transfer the control flow to the caller. However, this layout also poses the risk of stack buffers overflowing with excessive data that can overwrite the return address, leading to a corrupt stack frame and potential control flow attacks.

Stack canaries [9] are *sacrificial* words placed on the stack at stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer in order to overwrite the return address, the canary is also overwritten. Before returning in the program's execution stack, the canary is checked, and if modified, the return address is

assumed to be compromised. This approach works if the adversary's target is to overflow a buffer to overwrite the return address. However, there are scenarios where the adversary can skip over the canary using a vulnerable pointer reference, guess the canary, or learn the canary using a disclosure vulnerability.

Unfortunately, stack canaries cannot detect a buffer overflow if the attack payload does not actually overwrite the canary value. In a data-oriented attack, the adversary can overflow the buffer just enough to overwrite some sensitive variable above the buffer, but

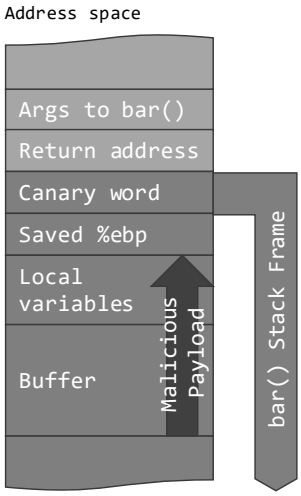


Figure **3-1**: A vulnerable stack frame layout with stack canaries.

```
1. int authenticated = 0;
2. char packet[1000];
3.
4. while (!authenticated) {
5.     PacketRead(packet);
6.
7.     if (Authenticate(packet))
8.         authenticated = 1;
9. }
10.
11.if (authenticated)
12.    ProcessPacket(packet);
```

Figure **3-2**: A vulnerable C code.

not cross stack frame boundaries. Here, the canary will not be overwritten; hence, it will not be able to detect the attack. Fig. 3-1 shows a typical layout of the stack frame, where such an attack is possible.

Fig. 3-2 shows an example of vulnerable code for a data-flow attack [4]. In this example, an adversary can send more than 1000 bytes of data which the `PacketRead` function writes to the `packet` variable. If the adversary's payload is large enough, it will overwrite the return address which will be detected by the canary when the stack rolls back. However, if the payload is carefully crafted, adversary can just overwrite the `authenticated` variable above the `packet` buffer. Then the check `Authenticate(packet)` can be bypassed and the packet will be processed, without being detected by the canary.

The obvious solution is to detect a buffer overflow as soon as it happens, and not wait for the function to end and the stack to rollback in order to validate the canary. One possible way is to place canaries at the top of every buffer. This leads to some challenges. Canaries are held in specialized canary registers or in a protected memory location in the address space. If the canaries are randomized, they take up some memory space or several registers. This is expensive in terms of memory space, especially if we try to put multiple canaries in the same stack frame to protect every buffer. Moreover, a vulnerability in saved canary locations can also lead to disclosing the canaries. The canaries also need to be validated after every write to a buffer. This can be expensive if implemented in a fine-grained manner. Existing solutions to protect programs from data-flow integrity are software based, e.g., performing reaching definitions analysis [4], or enforcing compile-time memory safety constraints [81], while others use specialized hardware or architecture

to perform tagging and metadata processing [31,36]. However, these techniques are expensive in terms of performance and/or memory requirements or require hardware or architectural modifications to support them.

In this chapter, we present PUFCanary, a fine-grained yet lightweight hardware generated stack canaries to protect buffer boundaries and can detect overflow of the buffers using Physically Unclonable Function (PUF) [82]. The PUF generates randomized canary words in a secure manner based on the address in use. We implement our design in RocketChip [79] based on the RISC-V architecture. The Rocket Custom Coprocessor (RoCC) of RocketChip allows a flexible hardware design implementation of our Canary Engine without modifying the core processor architecture. Compared to existing stack canary, our design provides the following key benefits: (i) secure and randomized canary word generation using PUFs, (ii) fine-grained individual buffer protection, and (iii) no need to save canaries in memory/ registers.

We further design FIXER (Flow Integrity Extensions for Embedded RISC-V), a low energy, low overhead security coprocessor that ensures integrity of backward and forward edge control flow of programs running on a RISC-V core. FIXER decouples the security architecture from the RISC-V core architecture, enabling a highly flexible security design. In the target deployment platform, the unmodified RISC-V core will be a hard IP, while the dynamically reconfigurable FIXER coprocessor will be implemented on an on-chip FPGA. Such an approach has the potential to be scaled to hybrid processor designs e.g., a Xeon + FPGA core [83]. The FPGA also provides the flexibility to change and update the security architecture in demand to new threats, without a complete redesign of the primary computing core. With the number of vulnerabilities rapidly increasing, it

demands an efficient low-power flexible and scalable security solution that is sustainable for long periods of time. FIXER potentially unlocks the design capability to protect our systems from such cybersecurity threats. Software based CFI techniques are also limited by the size of the address space, which can be overcome by FIXER's flexible FPGA implementation. Compared to NILE [41], FIXER achieves better performance. Although NILE uses an unmodified RISC-V core similar to FIXER, the core-coprocessor interface is modified for the coprocessor to tap into more resources of the core. Note that, even though PUFCanary and FIXER both aim to protect memory, they approach the solution in different ways. PUFCanary tries to proactively detect one of the fundamental causes of memory exploits – the buffer overflow itself. This requires compiler support and incurs more overhead than FIXER, especially if protecting multiple buffers. However, this allows protection against both control-flow and data-flow attacks originating from buffer-overflow. Hence security critical systems may opt to use PUFCanary, trading in some performance. FIXER, on the other hand, implements a shadow stack and policy memory for preventing control flow violations, but it cannot prevent data-flow attacks. The tradeoff is simpler design and better performance than PUFCanary, hence this can be used for resource-constrained embedded systems. Table 3-1 shows a qualitative comparison of PUFCanary and FIXER with the state-of-the-art memory protection solutions. Both PUFCanary and FIXER maintain high-performance with low energy, the difference being PUFCanary can also detect data-flow attacks. PUFCanary and FIXER are both hardware agnostic, however, special compiler support is required for PUFCanary implementation. FIXER also has the added benefit of being dynamicaly updated due to it's flexible FPGA implementation.

Table **3-1**: Qualitative comparison of PUFCanary and FIXER with replated works.

	Canary [4]	ASLR [3]	CFI [1]	PUMP [18]	HAFIX [27]	GRIFFIN [29]	HDFI [31]	NILE [10]	**PUFCanary**	**FIXER**
Control flow hijacking protection	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data flow hijacking protection	✖	✓	✖	✓	✖	✖	✓	✖	✓	✖
Maintains high-performance	✖	✖	✖	✖	✓	✖	✓	✓	✓	✓
Low energy overhead	✖	✖	✖	✖	✖	✖	✓	✓	✓	✓
No architecture modifications	✓	✓	✓	✖	✖	✓	✖	✓	✓	✓
No source code pre-processing	✓	✓	✓	✓	✓	✓	✖	✖	✖	✖
No compiler modifications	✖	✓	✖	✖	✖	✓	✖	✖	✖	✓
Software flexibility	✓	✓	✓	✓	✓	✓	✓	✓	✖	✓
Hardware flexibility	✖	✖	✖	✖	✖	✖	✖	✓	✓	✓
Dynamic patching	✓	✓	✓	✖	✖	✖	✖	✖	✖	✓

3.1 PUFCanary: Hardware Canary Design

We first explore PUFCanary, a hardware canary generation scheme to protect buffer boundaries.

3.1.1 Physically Unclonable Functions (PUF)

PUFs are a secure hardware fingerprint applied in hardware crypto systems. PUF generates the response (key) to a particular challenge from physical properties of the chip. The exhaustive set of challenge-response pairs (CRP) serves as the fingerprint of the PUF chip, which is fixed (even across power cycles) for a particular chip but varies chip-to-chip. Several flavors of PUFs exist in literature [84-86], etc. Specially crafted circuit structures e.g., SRAM and flip-flops are used to amplify physical randomness for the PUF signature.

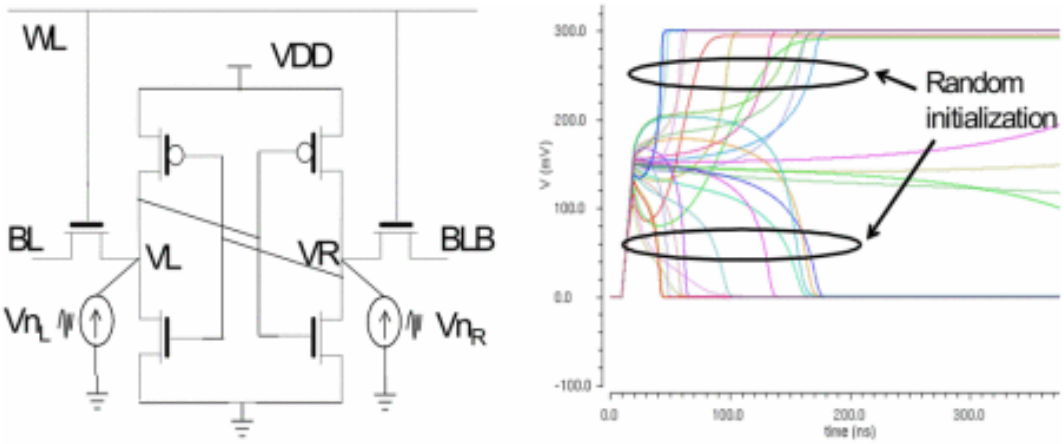


Figure **3-3**: SRAM based PUF design.

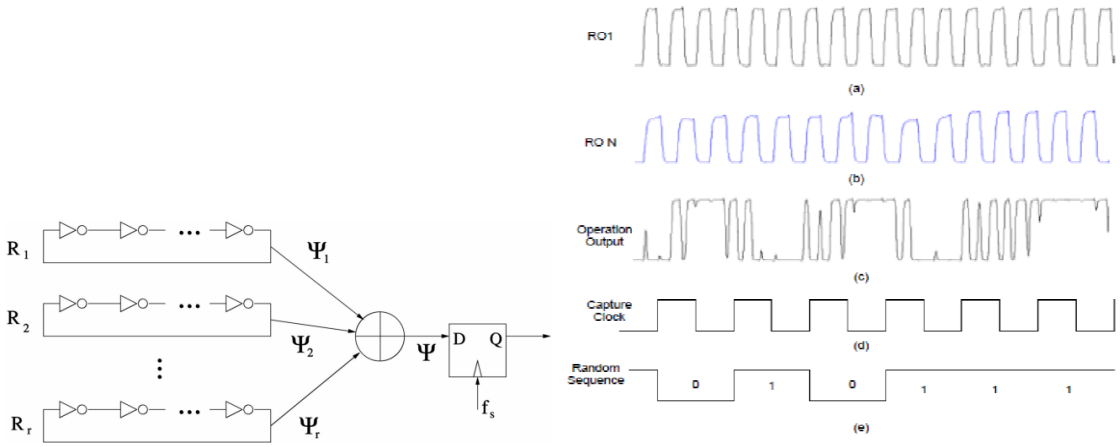


Figure **3-4**: Clock jitter based TRNG design.

Traditionally, PUFs have been used for chip authentication and deter IC counterfeiting. In this work, we use SRAM PUF for ease of implementation however, other types of PUFs can also be used. Fig. 3-3 shows the design of an SRAM based PUF, that we have used in our design.

3.1.2 True Random Number Generator (TRNG)

TRNG harnesses the natural entropy present in the system e.g., thermal noise [87], shot noise, Brownian motion or nuclear decay [88]. Techniques to harvest the noise in the operational amplifier [89], jitter of coupled oscillators [90], state of bi-stable elements [91] and oxide breakdown of transistors [92] have also been proposed. The challenges involved in designing TRNG include exploiting new entropy sources, efficient harvesting mechanisms, and careful post-processing. In this work, we use FPGA's oscillator jitter for TRNG, although other variants can also be used. Fig. 3-4 shows a design of a clock jitter based TRNG.

3.1.3 Generating Randomized Canary Words

In our proposed design methodology, we generate and place one canary per buffer in the program's execution stack. This is in contrast to the standard canary implementation where only one canary is placed at the return boundary of an execution stack. Furthermore, we randomize the canaries such that all the canary words in use for the current process are unique. This is to mitigate any attacks resulting from disclosure vulnerabilities. We design a Canary Engine using a PUF and a TRNG (Fig. 3-5) to generate random canaries. Since the canaries are placed at specific locations in a program's address space, we randomize the canaries based on the address where the canary will be placed. The PUF in the Canary Engine works in challenge-response mode, where the *address location for the canary is used as challenge*. The PUF response r_a is a partial canary word based on the challenge address a. The $\{a \rightarrow r_a\}$ mapping is obtained from the PUF signature (CRP). This partial

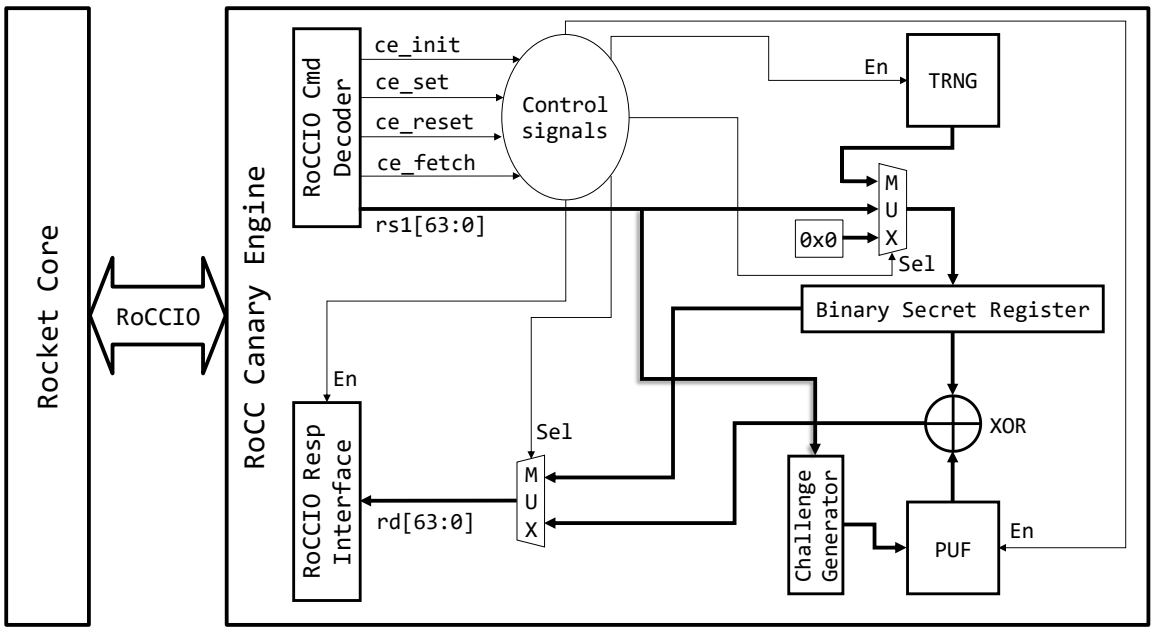


Figure **3-5**: PUFCanary hardware design in RoCC.

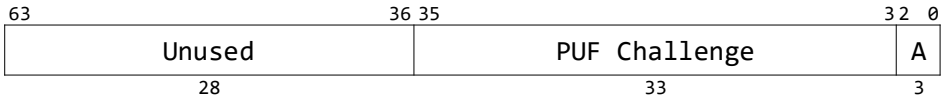
word is used in conjunction with a binary secret value s, kept in a dedicated binary secret register in the Canary Engine. This is crucial in order to make the canary word truly unpredictable for the adversary, since the PUF by itself is not a secret due to its fingerprint nature. We use the TRNG in the Canary Engine to create the secret value s_p for a process p when it is spawned for the first time. The secret value needs to remain constant for the lifetime of a process, hence it is backed up in the process's Process Control Block (PCB) by the operating system, so that it can re-populate the register with the value when the process is switched back in (context switch). The partial word r_a from the PUF is XORed with the binary secret s_p to generate the final canary word w for the requested address a:

$$w = r_a \oplus s_p \tag{3-1}$$

Note that, in the canary generation procedure, PUF serves the purpose of randomization within the same process, while the binary secret makes the canary unique across processes.

```
00007ff3ee05b000     24K rw---   [ anon ]
00007ff3ee078000      4K r---- ld-2.23.so
00007ff3ee079000      4K rw--- ld-2.23.so
00007ff3ee07a000      4K rw---   [ anon ]
00007ffc0bcd3000    132K rw---   [ stack ]
00007ffc0bd90000     12K r----   [ anon ]
00007ffc0bd93000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
        total            20840K
```

(a)



(b)

Figure **3-6**: (a) A partial `pmap` result of a process showing the addresses used by the stack, and (b) the address bits chosen for the PUF challenge ('A' represents the 3 unused alignment bits).

PUF optimization: A 1:1 PUF mapping produces a unique response for every challenge. In a 64-bit architecture, we have 2^{64} addressable locations, where segments such as kernel space and code segment will not be writable and will never be used for canary placement. Moreover, if a word is 8 bytes, and the data is word aligned, only 2^{61} addresses will actually be used for addressing and the lower order 3 bits can be ignored. To optimize our design for this, we can use part of the address bits as challenge instead of all 64 bits. For example, the higher order bits (kernel space) and the lower order bits (for alignment) can be ignored. Fig. 3-6(a) shows a partial `pmap` output for a 64-bit process. We can see that the stack starts at address `0x7ffc0bcd3000` and can grow up to `0x7ff3ee07a000` from where the memory mapping segment begins. Hence, we can practically ignore the higher order 28 bits (conservatively). Fig. 3-6(b) shows the chosen bits from an address for the PUF challenge. However, if this is not chosen conservatively, it can lead to duplicate

challenges. It should be noted that the chosen bits are architecture specific and needs to be tuned for different architectures, which determines the size and region of memory that can be protected. Furthermore, this optimization is only applicable for user-space stack buffer protection only. For kernel space protection, the higher order addresses are also required, hence there is less scope of optimization. For protecting the entire address space, other physical optimization measures can be taken, such as downsizing the PUF.

3.1.4 Security benefits and implications

Our canary design system provides several benefits over the conventional canary implementation. We can generate multiple canaries in the same stack frame, protecting each buffer in the stack frame. Due to PUF usage, the canaries need not be saved in a register or in the address space (typically referenced by an offset from the segment register *%gs* in GCC). The PUF signature (CRP) itself acts as a repository for the canaries to use. Whenever the canary at a particular address needs to be placed or validated, the PUF can be queried with the address as the challenge, and it will respond with the correct partial canary word. This makes it more secure and less prone to disclosure.

The PUF response is XORed with the binary secret to eliminate a brute-force disclosure of all the canaries from the PUF. Since the PUF signature is fixed over multiple power cycles, it is not a strong secret by itself. It generates different responses for each challenge, but for the same challenge, it generates the same response every time. This holds for challenges within the same process, across multiple executions of the same process, and even across different processes. Note that the responses change chip-to-chip.

Therefore, a successful attack on one system cannot be deployed globally. Without the binary secret in place, an adversary can generate all possible challenges (addresses) and retrieve corresponding responses (canaries) for those addresses. Thus, the binary secret allows us to obfuscate the PUF response. To generate a truly random value for the secret, the hardware TRNG (much faster than software pseudo-random generator) is used. This provides a far more efficient and secure random number as the secret value for each invocation of the same program, or for different programs. This ensures that an adversary cannot re-compute the canary values using the secret.

We assume that the OS kernel is secure, and the PCB information cannot be disclosed from the user space. This is important since the binary secret is a critical information for the particular process and must be stored securely in the PCB of the process in the kernel space to handle context switches. We also assume that the Canary Engine is secure, i.e., there is no instruction that can be used to directly query the PUF and obtain the response. Hence, there is no direct way of obtaining the exhaustive CRPs of the PUF. Also, there are no unprivileged instructions to read the binary secret register, preventing any information leakage from the Canary Engine that can be potentially leveraged to obtain the secret value in order to reconstruct the canaries. Since the secret value is 64-bit wide, brute-forcing it will require 2^{64} tries. For further securing the secret, it can be encrypted before storing in the PCB, however this may increase context-switching time due to the encryption/decryption process.

For a particular process p_1, the different canary words will be generated as $w_1 = r_{a_1} \oplus s_{p_1}, w_2 = r_{a_2} \oplus s_{p_1}$, etc. following (3-1). In case of a disclosure vulnerability, if w_1

is known for address a_1, it is not possible for the adversary to compute w_2 for a_2, since neither r_{a_1} nor s_{p_1} can be individually determined.

In our implementation, we have performed the XOR operation of the binary secret value with the PUF response. However, the secret can also be XORed with the address and used as PUF challenge. Both cases provide the same security guarantees. In both cases, the raw CRPs for the PUF remain undisclosed to the adversary due to the XOR operation, since only the challenge or the response is transparent to the adversary, but never both. To completely hide the PUF signature, the XOR operation can be performed on both sides, however, it will reduce performance. The number of PUF CRPs needs to be more than the size of the address space used for canary placement to avoid collisions in canary words.

3.1.5 Canary Usage and Design Flow

A system with our proposed canary design will be modified as follows: The kernel scheduler is modified to include a few extra instructions to configure the Canary Engine for the protection enabled process. Initially, when the process is to be scheduled for the first time, the kernel sends `ce_init` instruction to the Canary Engine. This is a privileged instruction and cannot be used from the user space. The TRNG in the Canary Engine generates the random word and populates the binary secret register. It also sends the value back to the kernel. The kernel saves the value in one of fields of the PCB for the particular process. This is shown in Fig. 3-7(a).

During a context switch, when the protected process is to be switched out, the kernel sends `ce_reset` instruction (privileged) to the Canary Engine. After decoding, a

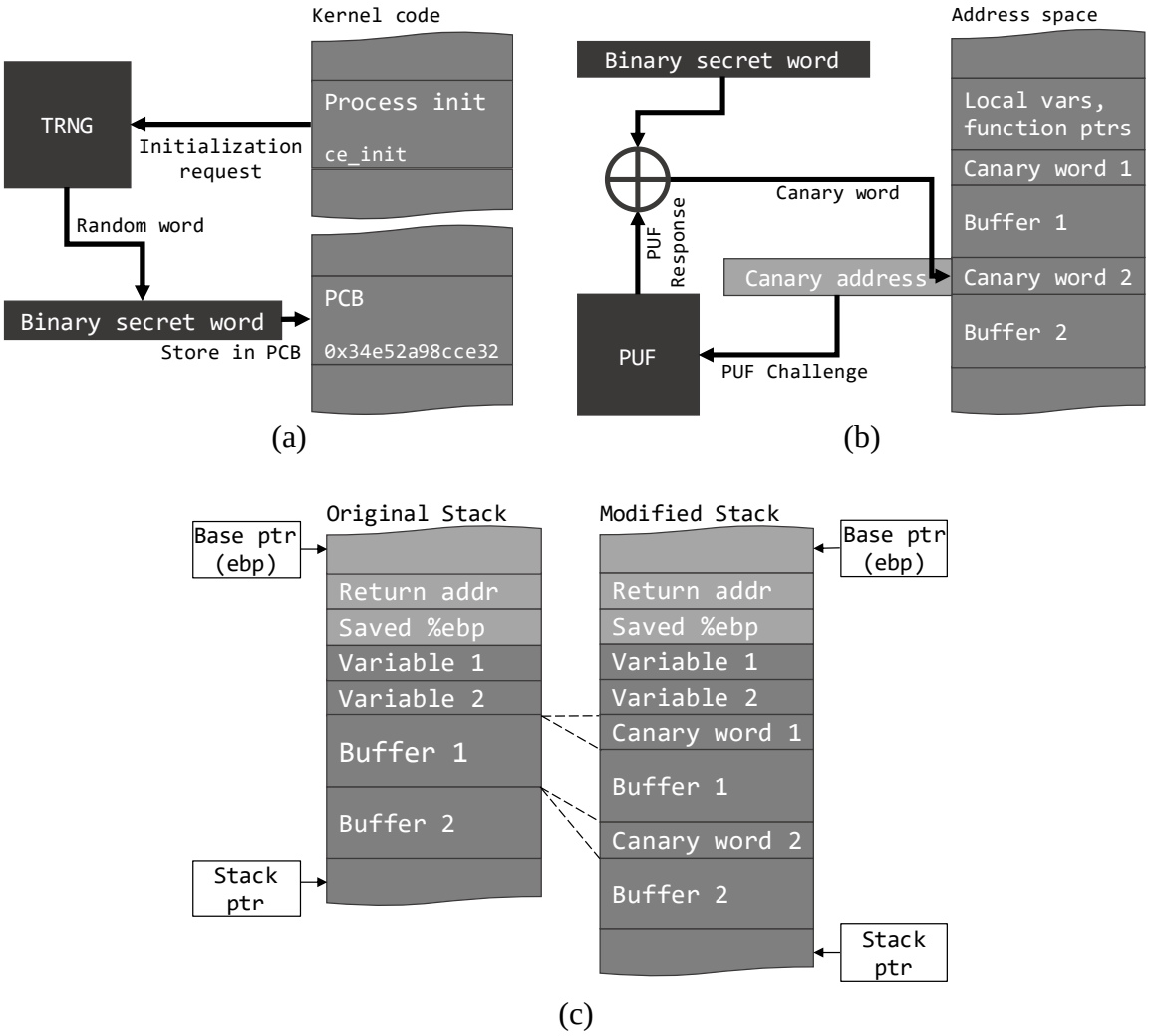


Figure **3-7**: (a) Canary Engine initialization, (b) Canary generation, and (c) Stack frame modification for canary placements.

control signal is sent to the multiplexer to write a `0x0` (reset) value to the binary secret register to prevent disclosure of the secret.

When the process is about to be scheduled again, the kernel sends `ce_set` instruction (privileged) to the Canary Engine along with the secret word stored in the PCB. This writes the secret word and repopulates the binary secret register.

The program to be protected with canaries is compiled as follows: A static analysis is performed on the assembly code to identify the buffer locations in each function. A `ce_fetch` instruction is placed per buffer in the function prologue. This is the only unprivileged instruction that can be sent to the Canary Engine. The instruction sends the address of the canary location to the Canary Engine which decodes the address and sends the challenge to the PUF. The PUF responds with the partial canary word which is XORed with the binary secret register value to generate the final canary word. The canary word is sent back to the process to be placed on the memory location. The canary generation process is depicted in Fig. 3-7(b).

Placing the canaries on the stack frame also requires some readjustment of the stack boundaries due to the extra memory locations taken up by the canaries. This is accomplished by readjusting the stack pointer and base pointers. Furthermore, since the original stack layout has been altered, the locations of the variables and buffers in the stack also needs to be readjusted. This is done by changing the references to all the variables and buffers in the stack after taking into consideration the canary placements. The unmodified and altered stack frames are shown in Fig. 3-7(c).

3.1.6 Canary Engine Implementation using RoCC

The Canary Engine implementation in the RoCC is shown in Fig. 3-5. The program binary runs on the Rocket Core and sends RoCC instructions over the RoCCIO whenever a canary generation or validation is required. The RoCC instruction is first passed through the Cmd decoder, which extracts the individual fields of the RoCC instruction, and the

contents of the two registers *rs1* and *rs2* if specified. The opcode field is decoded to the *custom0* instruction in our implementation. The *funct7* field is decoded to interpret `ce_init`, `ce_set`, `ce_reset` and `ce_fetch` instructions.

When the kernel sends the `ce_init` instruction to the Canary Engine, after decoding, a control signal is sent to the TRNG to generate a 64-bit random word. The word is sent to a multiplexer input. The appropriate select signal is sent to a multiplexer to select and write the random word to the binary secret register. At the same time, the word is sent to another multiplexer to be written to the *rd[63:0]* response interface and sent back to the core register *t0* to be saved in the process PCB.

When decoding the `ce_reset` instruction, a control signal is sent to the multiplexer, which selects the `0x0` (reset) value and writes it to the binary secret register. This resets the Canary Engine when the protected process is not in context.

For a `ce_set` instruction, the contents of the *t0* (the secret value held in the PCB of the process) is sent through the *rs1[63:0]* field of the RoCCIO interface to the canary engine. After the instruction is decoded, the value is read from the *rs1[63:0]* field and sent to a multiplexer. The required select signals are sent to the multiplexer and the value is copied to the binary secret register. This reconfigures the Canary Engine for the current process.

For the `ce_fetch` instruction, the contents of the core register *t0* (the address for the canary) is sent through the *rs1[63:0]* field of the RoCCIO. The PUF is implemented as a SRAM memory initialized to random default values of 64-bit wide words. For our proof-of-concept implementation, we have used a PUF with 1024 unique challenge-response pairs. This was sufficient for our implementation since our RocketChip’s program memory

configuration was generated with a usable stack memory of less than 1024 bytes for the sake of simplicity. However, in practical applications, PUFs with larger CRPs are required since all 64 bits of the address or the optimized 32 bits will be used. When a `ce_fetch` instruction is interpreted the appropriate read control signals are sent to the PUF and the binary secret register. The address in *rs1[63:0]* is sent to the Challenge Generator, which prepares the PUF challenge by selecting the appropriate bits from the address and sends it to the PUF. The default random word in the memory-PUF for that particular address (challenge) serves as the response. It is to be noted that since this is a PUF, the random default values in the 1024 locations are static and is a signature of the PUF, and hence it will not change. The response value is read and sent to the XOR gate. Simultaneously, the value in the binary secret register is also read and sent to the XOR gate. The output of the XOR gate is the resultant canary word and is fed to the *rd[63:0]* field of the response interface. The response is sent back to the core by writing the value in *rd[63:0]* to the register *t0* on the core, as indicated by the RoCC instruction.

3.1.7 Software Design with PUF Canaries

A program that needs to be protected with PUF-based canaries is compiled in the following fashion:

Step 1 – Generating assembly: We initially compile the program to an intermediate assembly code which allows us to scan the code to identify the target buffers to protect. This is demonstrated with the example C function in Fig. 3-8 and the corresponding assembly code in Fig. 3-9. Note that this function is for demonstration purposes only, it is

not actually a vulnerable function. We analyze each stack frame and find the individual buffer locations based on the stack pointer or base pointer offset. We calculate the extra memory space required in the buffer to introduce the canaries, and the specific locations in the stack frame where those words will be placed. We also recalculate the references for variables and buffers in the stack.

Step 2 – Modification of assembly: The assembly code modification involves two major operations – canary placement and canary validation. The modified assembly code is shown in Fig. 3-10. For the canary placement, we first expand the stack frame by modifying the function prologue. In the example shown in Fig. 3-9, we will be placing 2 canaries, hence we subtract 2 x 8 bytes = 16 bytes to the stack pointer. Hence we replace `add sp,sp,-96` with `add sp,sp,-112`. We update all the references that use the stack pointer, such as the location where the return address is saved. For each buffer we place the canary in the following manner. First the address where the canary is to be placed is loaded on to the *t0* register. In our example, for the first buffer, we choose the canary location as -32(s0). This address indicates the address on top of the buffer. We use `add t0,s0,-32` to load the address onto *t0*. Now, we craft our `ce_fetch` custom instruction. There are four custom RoCC instructions available (*custom0-3*) that are identified by the 7-bit opcode field. The *funct7* field can be used to further specify a particular function of the RoCC instruction. We use *custom0* to implement the canary instructions. We set the *funct7* field to b'0000000 (0) for `ce_fetch`. The *rs1* field is set to the *t0* register (b'00101), and the *rd* field is also set to *t0*. The corresponding *xs1* and *xd* fields are set to 1. The final crafted `ce_fetch` instruction is represented by `0x1714b`. The `ce_init`, `ce_set` and `ce_reset` instructions are also crafted similarly with *funct7* set to 1, 2 and

3 respectively (the details are omitted for brevity). We repeat the same process for the second buffer as shown in the example. Next, we readjust the location of the buffers by subtracting 8 and 16 bytes from the original addresses. This is accomplished by changing the buffer references for the `memcpy` function as `add a4,s0,-64` (for the first buffer). Immediately after the `memcpy` function returns, we place our canary validation code. This needs to be done for any copy to buffer function, such as `memcpy`, `strcpy`, etc. First we load the canary value on the stack onto the register *t1* by `lw t1,-32(s0)`. Next, we follow the same steps as before to fetch the actual canary value for that location from the Canary Engine into the register *t0*. We compare the values in registers *t0* and *t1* and proceed or halt depending on a match or mismatch. The same process is repeated for all the buffers on the stack. Finally, at the function epilog, we update the references for fetching the return address, and the stack and base pointers (Fig. 3-10). In the validation process, our design only scans for standard library buffer copy functions. However, it will not be able to detect manual writes to a buffer using a loop. Such cases may be handled using LLVM compiler toolchains where the copy operations can be parsed from intermediate representations (IR). The assembly modification requires identifying the number of buffers, their size and their offset from the symbol used to reference the buffers. The canary address for a buffer can be calculated using the size and offset for the buffer. The stack pointer needs to be calculated accordingly to make space for the canaries. The references to the buffers also need to be calculated considering the extra space for the canaries. For the canary validation,

```
1. void func1()
2. {
3.     int var1;
4.     char buffer1[32];
5.     int var2;
6.     char buffer2[32];
7.     var1 = 1;
8.     var2 = 2;
9.     memcpy(buffer1,"hello",5);
10.    memcpy(buffer2,"world",5);
11.}
```

Figure **3-8**: Example C program.

```
1. func1:
2.      add     sp,sp,-96
3.      sd      ra,88(sp)
4.      sd      s0,80(sp)
5.      add     s0,sp,96
6.      li      a5,1
7.      sw      a5,-20(s0)
8.      li      a5,2
9.      sw      a5,-24(s0)
10.     add     a4,s0,-56
11.     li      a2,5
12.     lui     a5,%hi(.LC1)
13.     add     a1,a5,%lo(.LC1)
14.     mv      a0,a4
15.     call    memcpy
16.     add     a4,s0,-88
17.     li      a2,5
18.     lui     a5,%hi(.LC2)
19.     add     a1,a5,%lo(.LC2)
20.     mv      a0,a4
21.     call    memcpy
22.     nop
23.     ld      ra,88(sp)
24.     ld      s0,80(sp)
25.     add     sp,sp,96
26.     jr      ra
```

Figure **3-9**: Disassembled code.

```
1. func1:
2.      add     sp,sp,-112
3.      sd      ra,104(sp)
4.      sd      s0,96(sp)
5.      add     s0,sp,112
6. # Place canary @ -32(s0)
7.      add     t0,s0,-32
8.      .word   0x1714B
9.      sw      t0,-32(s0)
10.# Place canary @ -72(s0)
11.     add     t0,s0,-72
12.     .word   0x1714B
13.     sw      t0,-72(s0)
14.     li      a5,1
15.     sw      a5,-20(s0)
16.     li      a5,2
17.     sw      a5,-24(s0)
18.     add     a4,s0,-64
19.     li      a2,35
20.     lui     a5,%hi(.LC1)
21.     add     a1,a5,%lo(.LC1)
22.     mv      a0,a4
23.     call    memcpy
24.# Validate canary @ -32(s0)
25.     lw      t1,-32(s0)
26.     add     t0,s0,-32
27.     .word   0x1714B
28.     bne     t0,t1,_die
29.     add     a4,s0,-104
30.     li      a2,5
31.     lui     a5,%hi(.LC2)
32.     add     a1,a5,%lo(.LC2)
33.     mv      a0,a4
34.     call    memcpy
35.# Validate canary @ -72(s0)
36.     lw      t1,-72(s0)
37.     add     t0,s0,-72
38.     .word   0x1714B
39.     bne     t0,t1,_die
40.     nop
41.     ld      ra,104(sp)
42.     ld      s0,96(sp)
43.     add     sp,sp,112
44.     jr      ra
```

Figure **3-10**: Modified assembly code. Canary placement and validation code are shown in bold.

the symbol referencing the buffer can be used to identify the location of the canary. A table can be maintained with the IR to match the buffer to its canary location.

Step 3 – Final Compilation: The final PUFCanary enforced assembly code is passed to the compiler to assemble, link, and generate the final executable binary of the program. No compiler modifications are necessary to embed the instructions in the final binary since we provided the custom instruction as a binary instruction word, and the RoCC instruction format is already supported by the RISC-V GNU toolchain. Clang/LLVM can be used to automate the entire process. The automation involves emitting the LLVM IR from the source, writing compiler passes for the IR to modify the assembly by adding canary placement and validation codes, and compiling the IR after the passes into machine code using Clang.

3.1.8 Experimental Results

We tested our Canary Engine in the C++ cycle accurate emulator of the RocketChip Generator, and on the Xilinx Zybo FPGA. The hardware architecture of the Canary Engine is coded in CHISEL and is translated to synthesizable Verilog code using the available tools in the RocketChip Generator. We evaluated the security of our design and the performance overheads using the Wilander Buffer Overrun Suite [93]. We compiled the tests using the RISCV GNU GCC compiler in two versions: (i) the *baseline* code without canary protections, and (ii) *PUFCanary* code with our canary protection. Since our design only protects the stack, we considered only the 12 stack-based test cases in the test suite. However, due to the limitations in the RISC-V toolchain, we were able to port only 8 test

Table **3-2**: Wilander Test Cases for Stack Corruption.

Case	Description
T-4	Overflow all the way to FUNCTION PTR as PARAM
T-2	Overflow of a PTR, then pointing at FUNCTION PTR as PARAM
T1	Overflow all the way to RETURN ADDRESS
T2	Overflow all the way to OLD BASE POINTER
T3	Overflow all the way to FUNCTION PTR as local variable
T7	Overwrite of a PTR to point at RETURN ADDRESS
T8	Overwrite of a PTR to point at BASE POINTER
T9	Overwrite of a PTR to point at FUNCTION PTR as variable

cases. The 4 test cases (with LONGJMP) could not be ported. For the 8 working test cases (shown in Table 3-2), our *PUFCanary* was able to prevent all 8 attack cases.

Fig. 3-11(a-b) shows the performance overheads of *PUFCanary* over the *baseline* code. The corresponding instruction overheads are shown in Table 3-3. The execution time overhead of *PUFCanary* over *baseline* is 2.3% on average across the 8 test cases. There is none or negligible effect on CPI (cycles per instruction), where our tests reveal 0.97X average overhead. Few of the test cases show improvement in execution time; this may be due to architectural optimizations such as better cache performance. Our results are better than the original StackGuard [9] which shows 6% overhead in the best case. Furthermore, our results are comparable to HDFI [36] which also has ~2% overhead. To further support our claim of multiple buffer protection performance, we modified the test case T1 to include multiple buffers. We enforced canary protection on 1, 3, 5, and 10 buffers in the same vulnerable stack of T1. The performance trends in case of multiple buffer protection are shown in Fig. 3-11(c-d). The execution time overhead trend is mostly linear with 10-buffer protection having 1.5X the overhead of 1-buffer protection. Similar trend can be observed with CPI overhead where 10-buffer protection overhead is 1.4X that of 1-buffer

protection. This shows that a fine-grained protection with our Canary Engine does not have a significant performance impact. The Canary Engine RoCC module with a 1024 CRP PUF exhibited ~2.9% area overhead over a vanilla RocketChip with the default configuration.

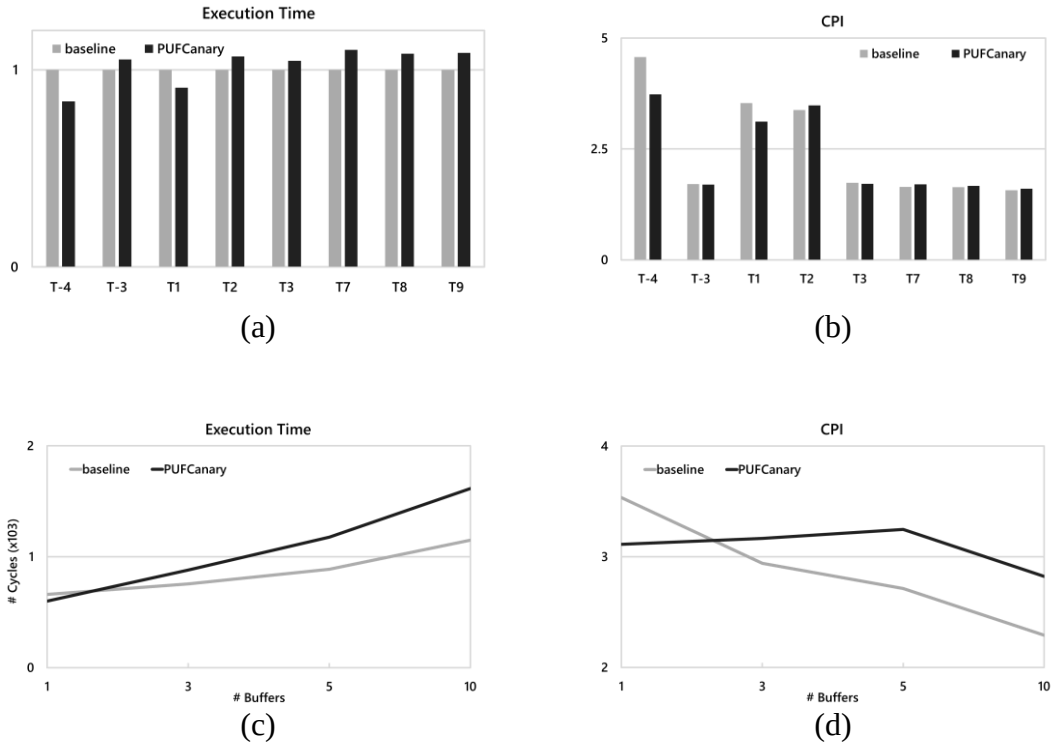


Figure **3-11**: Wilander benchmark evaluation w.r.t. (a) execution time (normalized), and (b) effective CPI; Performance overhead trends for multiple buffer protection w.r.t (c) execution time (cycles), and (d) CPI.

Table **3-3**: Instruction overheads.

Case	**Overhead %**
T-4	3.00
T-2	5.98
T1	3.21
T2	3.57
T3	5.88
T7	6.05
T8	6.26
T9	6.07

3.1.9 Limitations and opportunities

3.1.9.1 PUF design decisions

We have implemented a simplified version of PUF. The security of the design is dependent on the number of CRPs that the PUF can generate. A smaller PUF with limited CRPs can lead to duplicate canaries, potentially allowing the attacker to guess the canary for different addresses. To get around this security limitation, the output of the XOR gate can be combined with the original address and hashed to generate a 64-bit canary. For higher security, a larger SRAM PUF can be used for a 1:1 address-to-canary mapping at the cost of area and power overheads. To optimize the overhead, we can down-size the SRAM footprint, however, this can cause read disturb failures during query. It has been shown in literature that PUFs can be used to reliably generate random numbers by targeted NBTI aging [94]. If raw SRAM PUF responses are not uniformly random, PUF responses can be transformed to high-entropy random values by fuzzy extraction [95]. This may add to the performance overhead if the PUF is queried frequently. Cryptographic engines in processors often provide hardware PUFs and TRNGs which can be reused in the Canary Engine. Our proposed technique is not limited by the choice of the PUF used, and is also applicable to other PUF flavors e.g., arbiter PUF [84], flip-flop PUF [86], etc. MRAM/STTRAM PUFs [96-97] that guarantee uniform randomness may also be used to eliminate the need for post-processing. In our SRAM PUF, we have a synchronous interface between the core and the Canary Engine, which contributes to some of its performance overhead. In practical implementations, the Canary Engine does not need to be stateless and can be pipelined to improve performance.

3.1.9.2 Canary validation decisions

In our design, we have validated all the canaries in a function stack frame after every write to a buffer. Although more complex and performance intensive, it can detect data-oriented attacks proactively. However, the canaries can also be validated all at once in the function epilogue to reduce design complexity at the cost of attack detection when the function returns. It is possible that a non-control data attack may succeed before detection in this case. Control-flow bending attacks using a buffer overflow vulnerability can only be detected if the control-data is used after the validation of the corrupted canary takes place. If the canaries are validated after the control data is used to bend the control flow, the attack may still succeed.

3.1.9.3 Buffer overread protection:

PUFCanary can only detect buffer overwrites and not overreads. This leaves the system open to memory disclosure and can potentially leak the canaries. However, since all the canaries are random and unique, disclosure of one canary may not provide the adversary enough opportunities to launch a control-flow or data-flow attack unless there is a buffer-overflow and a buffer-overread vulnerability on the same or nearby buffers. This is possible, for example, in a loop which contains a buffer vulnerable to both overread and overwrite. If, due to a memory disclosure attack at that location, the adversary learns the canary, he can reuse that canary for the same location for the overflow attack. Aside from this scenario, the adversary cannot reuse the same canary found from disclosure to attack a different buffer, since they are protected by different canary words. However, in case of

a smaller PUF design with repeated canaries, it may be easier for an adversary to reuse canaries in case of memory disclosures.

3.1.9.4 Advanced canary design with temporal security

The PUFCanary design presented above only provides spatial security. Hence, through some disclosure vulnerability, if the adversary can learn the generated canary word for a particular address, he can reuse the canary as part of his payload to exploit an overflow vulnerability at the same location. This can be alleviated by using a more advanced canary generation process to provide temporal security. The primary limiting factor to providing temporal security in the original design is the fixed binary secret for the process, which leads to the same generated canary word for a particular address. We propose to update the generation process by using an access counter in conjunction with the secret, to provide temporal randomness.

In this design, we use a dedicated content-addressable memory (CAM) with two fields (i) address (ii) counter, that is populated every time a canary for a particular address is generated. If the canary is being generated for the first time for an address, a new entry is created in the CAM with the address and a counter value of 1. For any subsequent requests of canary generation at the same address, the address entry is looked up quickly and the counter value is incremented. The canary generation process is as follows: When a canary is to be generated, the `ce_fetch` instruction is sent to the canary engine. The address is decoded and searched in the CAM. If the address is not present, a new entry is created with the counter as 1. If the address entry exists, the counter is incremented. The

counter value (c_a) is XORed with the binary secret value (s_p) and hashed to a 64-bit random number (s_c):

$$s_c = hash(s_p \oplus c_a) \tag{3-2}$$

The address is simultaneously sent to the PUF to generate the partial word (r_a). The final canary word for the address is generated by XOR of the generated random number (s_c) and the partial word (r_a):

$$w_a = r_a \oplus s_c \tag{3-3}$$

This provides the guarantee that every time a canary is generated for the same address, it is unique. However, in this case, we cannot use the same instruction to validate the canary. We craft a new instruction `ce_validate` to regenerate the canary word for validation. This works similar to `ce_fetch`, except that it does not modify the address counter in the CAM. The reasoning behind this split logic is that a canary may be validated multiple times (once for each copy operation to the same buffer), however, it will only be generated once for a specific buffer. By using this method, we can ensure that even if the adversary learns the canary word for the buffer when it is first created, he will not be able to use the same canary for subsequent creations of the same buffer at the same location.

3.1.9.5 Data-oriented attacks

PUFCanary can detect data-oriented attacks that originate from a buffer overflow vulnerability. However, other data-oriented attacks that originate from memory disclosures, format-string vulnerabilities or integer overflows cannot be detected by PUFCanary.

3.2 FIXER: Hardware Enforced CFI

In this section, we explore the design of hardware assisted control flow integrity enforcement using hardware shadow stacks and policy memories.

3.2.1 FIXER Design for Backward Edge CFI

The first security primitive implemented in FIXER to prevent a memory corruption vulnerability is a Shadow Stack. C programs compiled with the GNU GCC Toolchain for RISC-V target architecture do not provide any protection against memory corruption vulnerabilities such as, buffer overflow. An adversary can provide malicious inputs to a program and can overwrite the return address of a function and redirecting the control flow of the program. The Shadow Stack security primitive can enforce the CFI at the backward edge (return to functions). The RoCC is used to implement the Shadow Stack, thus preventing the need to modify the core system architecture. The Shadow Stack is designed as a hardware memory on the RoCC. Fig. 3-12 shows the steps for detecting CFI violation using a Shadow Stack. The return address is pushed on the system stack by default when a function call is made in the program. During this time, same return address is sent using a RoCC custom instruction to the RoCC to push it on the Shadow Stack as a backup. The return address is popped from the system stack to the instruction pointer register for execution when returning from a function. During this return the RoCC Shadow Stack is queried to retrieve the backup return address and compare against the one from the system stack. If they match, the program proceeds with normal execution. If they do not match, then a potential memory corruption is detected, and program execution is stopped. Note

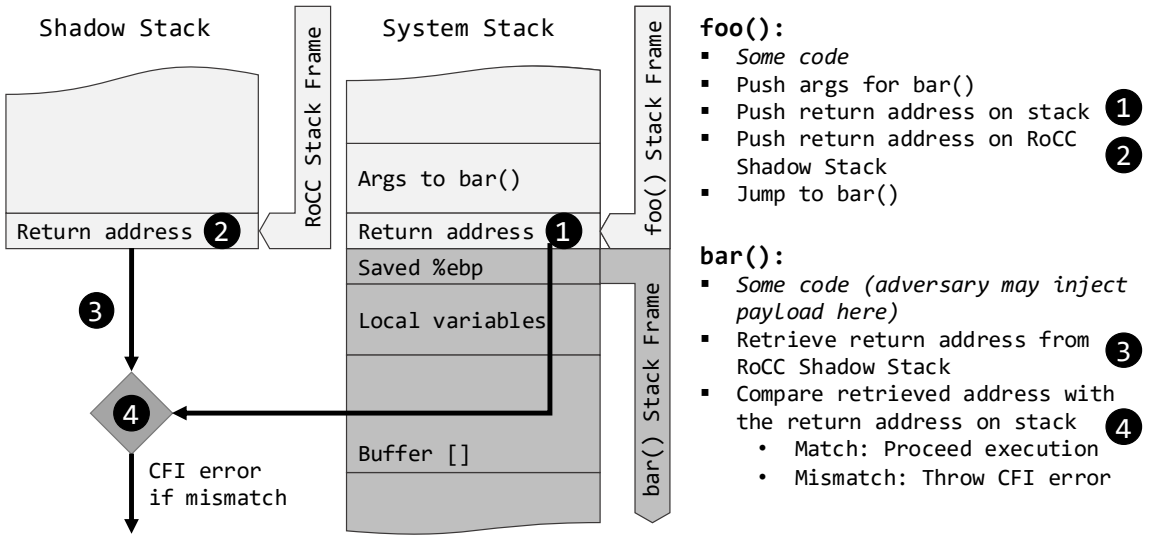


Figure **3-12**: CFI violation detection using a Shadow Stack.

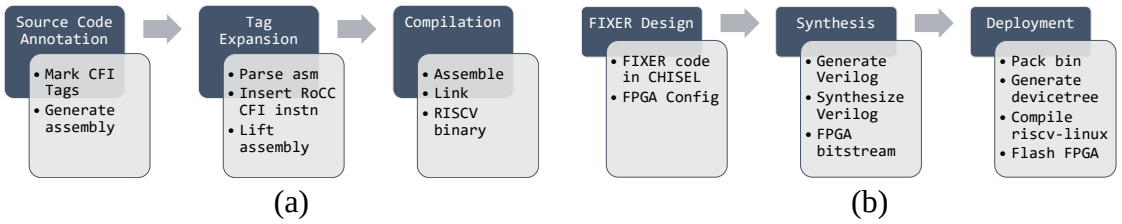


Figure **3-13**: FIXER design flow in (a) software and (b) hardware.

that compared to HAFIX [33] where Shadow Stack is part of core, FIXER implements it in the coprocessor leaving the core architecture untouched. It is to be noted that FIXER is complementary to existing DEP protection since the FIXER instructions must be tamperproof to ensure protection.

Fig. 3-13(a) details the software design flow for FIXER. The source code is first marked with CFI tags and compiled to an intermediate assembly code using the RISC-V GNU toolchain. The assembly code is parsed by expanding the tags and injecting the

required RoCC instructions in the assembly. The lifted assembly code is generated using a custom parsing script or a compiler pass and then assembled and linked to produce the fully compiled RISC-V binary.

Fig. 3-13(b) shows the hardware design flow for FIXER (coded in CHISEL [80] as a RoCC). The hardware implementation of FIXER in RoCC is described in Section 3.2.3. The relevant configuration files for RoCC targeting the FPGA platform are also written. The RocketChip with the RoCC is then compiled with the RocketChip Generator to output the synthesizable Verilog code, from which the FPGA bitstream is compiled. The required RISC-V Linux system image, the FPGA devicetree and the generated bitstream is then deployed to the FPGA to run the RocketChip system. This FIXER assisted RocketChip system can successfully protect against CFI violations on the RISC-V programs compiled with the FIXER assisted compilation process.

3.2.2 RISC-V Software Design with FIXER

Any program that needs to be backward-edge CFI enforced, is compiled and processed by the following steps:

Step 1 - Source code annotation: We annotate the function calls and returns with a special tag to indicate the sites where the enforcement needs to take place. We use `CFI_CALL` tag before a function call and a corresponding `CFI_RET` tag just before a return from the called function, as shown in Fig. 3-14.

Step 2 – Tag expansion: We expand the CFI tags to actual RISC-V assembly instructions. During compilation, we intercept the intermediate assembly code of the

```
void main () {
    ...
    CFI_CALL
    myFunc();
    ...
}
```

```
void myFunc() {
    ...
    CFI_RET
    return;
}
```

Figure **3-14**: FIXER source code annotation.

```
# CFI_CALL
auipc   t0,0
add     t0,t0,14
.word   0x0002a00b
call    myFunc
```

```
# CFI_RET
.word   0x0200428b
bne     t0,ra,_cfi_error
jr      ra
```

Figure **3-15**: FIXER tag expansion.

program and inject the RoCC custom instructions to communicate with the RoCC. Fig. 3-15 shows the assembly instructions corresponding to `CFI_CALL` and `CFI_RET`, that are placed just before the call and `jr ra` (return) instructions respectively.

For `CFI_CALL`, we first retrieve the current value of the program counter from the instruction pointer register using the `auipc` instruction and add 14 bytes offset (instructions are variable length) to calculate the target return address. We save the computed return address in a temporary register *t0*. Then we craft the RoCC instruction `cfi_call` to push the return address from *t0* to the Shadow Stack. There are four RoCC instructions available (*custom0-3*) that are identified by the 7-bit opcode field. The *funct7* field can be used to further specify a particular function of the RoCC instruction. We use *custom0* to implement the CFI instructions. We set the *funct7* field to b'0000000 (0) for `cfi_call` and to b'0000001 (1) for `cfi_ret`. We use the *rs1* field to set it to use the t0 register (b'00101), where we temporarily stored the computed return address and set the

corresponding *xs1* bit to 1. The final crafted instruction word for `cfi_call` is represented by `0x0002a00b` in hex.

For `CFI_RET`, we set the *funct7* field to b'0000001 (1) and set the *rd* field to use the *t0* temporary register (b'00101) along with *xd* bit as 1. The final crafted instruction word for `cfi_ret` is represented by `0x0200428b` in hex. During a return from a function, the saved return address is popped from the system stack on to the link register *ra*. We then use the `cfi_ret` custom instruction to retrieve the backup return address from the RoCC Shadow Shack on to the temporary register *t0*. The value in *t0* is then compared against the value in the register *ra* using the `bne` instruction. If they match, the execution proceeds by completing the return (`jr ra`: jump register). Otherwise, we throw a CFI error.

Step 3 – Compilation: The final CFI enforced assembly code is passed to the compiler to assemble, link and generate the final executable binary of the program. No compiler modifications are necessary to embed the instructions in the final binary since we provided the custom instruction as a binary instruction word, and the RoCC instruction format is already supported by the GNU toolchain.

3.2.3 FIXER Hardware Implementation in RoCC

Fig. 3-16 shows the FIXER implementation in the RoCC. The program binary runs on the Rocket Core and sends RoCC instructions over the RoCCIO whenever a security validation is required. The RoCC instruction is first passed through the Cmd decoder, which extracts the individual fields of the RoCC instruction, and the contents of the two

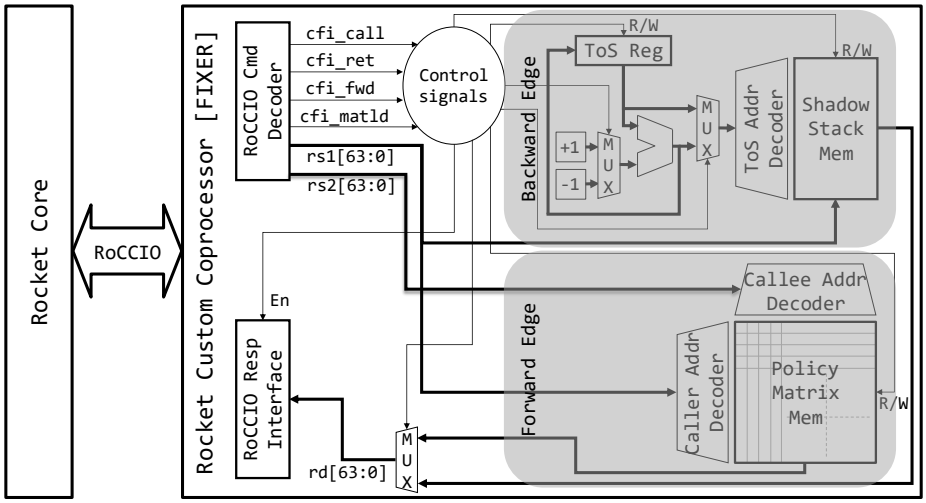


Figure **3-16**: CFI violation detection using a Shadow Stack.

registers *rs1* and *rs2* if specified. The opcode field is decoded to the *custom0* instruction in our implementation. The *funct7* field is decoded to interpret a `cfi_call` or a `cfi_ret`.

For `cfi_call`, the contents of core register *t0* (the return address) is sent through the *rs1[63:0]* field of the RoCCIO interface. The shadow stack is implemented as a SRAM memory with 64-bit wide words. A top-of-stack register (ToS) holds the address of the top of the shadow stack. If a `cfi_call` is interpreted, the content of the ToS register is incremented by 1. The updated value in the ToS register is used to decode the write address for the shadow stack. The value in the *rs1* field is written to this address on the shadow stack. This operation is non-blocking, so the core can continue execution after issuing the `cfi_call` instruction. There is a command queue at the RoCCIO interface to prevent race conditions. If the instruction function is interpreted as `cfi_ret`, then the ToS register is read to obtain the address for the shadow stack. This address is used to read the saved return address from the shadow stack memory. The value is then sent back to the core by

writing to the *rd[63:0]* field of the response interface of the RoCCIO, which writes the value to the *t0* register on the core as indicated by the RoCC instruction. Our proof-of-concept implementation of the shadow stack can accommodate 1000 addresses. However, this can be updated on demand by simply reconfiguring the FIXER module on the FPGA, a benefit exclusive to our implementation. The size of the shadow stack will be limited by the memory available on the target FPGA.

3.2.4 Forward-edge Protection with FIXER

A shadow stack only protects control flow on return boundaries. However, programs often use function pointers to jump to multiple function addresses. To ensure the validity of such function calls using function pointers, a pre-computed call policy is enforced. A static analysis is performed on the program to construct a control flow graph (CFG). The CFG is represented as a policy matrix that indicates the valid call targets for each function call made using a function pointer. The policy matrix is loaded in memory and at runtime, it is queried to validate the call target for every indirect function call. This forward-edge protection is implemented as another FIXER security module (Fig. 3-16). The policy matrix memory is created in the RoCC along with peripheral caller and callee address decoders. Our proof-of-concept implementation has 64 rows (each represents an originating call site address) in the matrix and each row holds a 64-bit policy vector (each bit represents a call target address). A set (unset) bit indicates that the call is valid (invalid) for that (caller, callee) pair. A RoCC instruction `cfi_matld` is used to load the policy bitmap into the FIXER module prior to the program execution. A RoCC instruction

`cfi_fwd` is inserted before every indirect function call in the source code. The `cfi_fwd` instruction sends the caller and the dereferenced function pointer (callee) addresses to the RoCC for validation. The forward-edge FIXER module then validates the action using the policy matrix and sends back a 1 or 0 indicating allow or disallow respectively. Similar to the shadow stack implementation, the policy matrix size can also be updated post-deployment by reconfiguring the FPGA.

3.2.5 Security Implications and Benefits

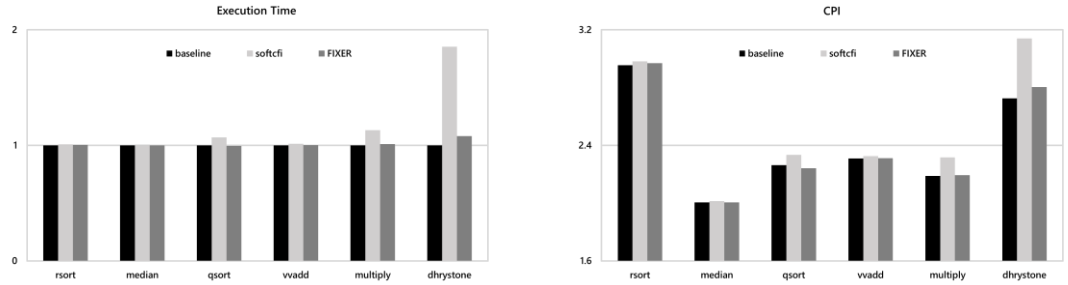
FIXER can be scaled to hybrid architectures, e.g., CPU+FPGA, or ASIC+FPGA. It is true that if the FPGA is off chip, there could be performance degradation (due to speed gap between CPU and FPGA) if the checking is performed in a synchronous and fine-grained manner. Performance issues can be alleviated by making the checking asynchronous using interrupts. In such cases, the program can continue execution, until the FPGA raises an interrupt to halt the program. However, it cannot be guaranteed that the adversary has not been able to take control of the system before the FPGA detects the attack. When the FPGA is on-chip, e.g., Intel Xeon with embedded FPGA, performance overheads can be alleviated due to QuickPath Interconnect (QPI) interface between the core and the FPGA for fast communication.

FIXER implemented on the FPGA offers benefits compared to other core based or system level protection schemes. Designs e.g., NILE which use the virtual address space to house the shadow stack cannot scale based on the branch sequence depth. HAFIX has a separate limited memory on the core to store the CFI tags. However, in case of FIXER, the

design can be scaled up or down based on the actual workload of the system. Typically, embedded devices e.g., IoTs have a limited set of workloads, and FIXER module on the IoT's SoC can be scaled appropriately based on the workload. For example, if a new workload introduced to the system requires a larger shadow stack, the FPGA can be reconfigured to accommodate that (the maximum size being limited by available LUTs).

3.2.6 Experimental results

The hardware architecture of FIXER is coded in CHISEL and translated to synthesizable Verilog using the available tools in the RocketChip generator. We prepared a FPGA system image using the generated Verilog and ran it on a Xilinx Zynq FPGA. A sample program is written with 1 billion iterations of function calls and returns. One version of the code implemented a simple software version of the shadow stack (*softcfi*). The software shadow stack is created as a regular stack in the address space. During function calls, the return address is simultaneously placed on the system stack as well as the shadow stack. Another version instrumented the code with the RoCC CFI instructions (*FIXER*). We compiled the *baseline* (no CFI checks), the *softcfi* and *FIXER* versions using the RISC-V GNU toolchain. The three versions of the program were run on the system running on the FPGA. The base code takes 19 seconds to execute, whereas the *softcfi* takes 74 seconds. *FIXER* takes 29 seconds resulting in ~1.5X overhead over *baseline* and ~2.55X lower overhead compared to *softcfi*. The FPGA on idle draws 370mA current, while on load (with the program running) draws 420mA current, resulting in 1.13X increase. The corresponding energy overhead is 3.89X for *softcfi* and only 1.53X for the *FIXER* (60.52%



(a) (b)

Figure **3-17**: RISC-V benchmark evaluation for FIXER backward-edge protection w.r.t. (a) execution time (number of cycles), and (b) effective CPI.

Table **3-4**: Benchmark Instruction overheads.

	Backward-edge	**Improvement over *softcfi***
rsort	1.000019X	0.0126%
median	1.000305X	0.2310%
qsort	1.00434X	3.1770%
vvadd	1.000622X	0.5080%
multiply	1.008037X	5.7140%
dhrystone	1.068607X	32.7930%

improvement). The FIXER RoCC module incurs only 2.9% area overhead over the vanilla RocketChip without RoCC.

We evaluated FIXER performance by enforcing it on RISC-V architecture benchmarks. The benchmarks are modified to create three versions for comparison: (i) *baseline* with no CFI enforcement, (ii) *softcfi* with the software-based CFI enforcement, and (iii) *FIXER* with RoCC based CFI protection. We ensured that the benchmark code remains the same across all the three versions except the CFI enforcement code. We compiled the benchmarks with the RISC-V GNU toolchain without compiler optimizations

and ran the compiled binaries on the Zynq FPGA. Fig. 3-17 show the evaluation results for backward-edge FIXER. The instruction overheads are shown in Table 3-4. With the backward-edge protection, the execution time overhead with *softcfi* is ~18% on average across the six benchmarks compared to 1.5% with *FIXER*. The *softcfi* increases the CPI (cycles per instruction) by 4.6% over the *baseline*, while the *FIXER* increases the CPI by only 0.5%. With the forward-edge protection, the execution time overhead with *softcfi* is ~2% on average across the six benchmarks compared to 0.61% with *FIXER* and CPI reduces 0.4% on average, which is negligible.

3.2.7 Limitations and opportunities

3.2.7.1 Multi-process protection

Our implementation of FIXER enforces protection for a single process only. For a simultaneous multi-process protection, the FIXER design can be expanded to accommodate multiple shadow stacks and policy memories for different processes. A round-robin scheduler on the FIXER module can assign the shadow stacks and policy memories to each process based on the process ID. RISC-V has support for hardware threads identified by their `hartID`. For multi-process support, multiple fixer modules can be instantiated each reserved for a hart. When multiple processes simultaneously require protection, each process is associated with a specific FIXER module using the process's `hartID`.

3.2.7.2 Tail-call optimization support

By default, FIXER assumes that there are no tail call optimizations being performed while generating the assembly, since, it performs the necessary `CFI_CALL` and `CFI_RET` tag expansions automatically without validating if there is a `call` or `ret` instruction following the assembly tag. However, compilers may perform tail-call optimizations which skip the `call` instruction and jump to the function to avoid creating a new stack frame. In such cases our naïve implementation will still expand the `CFI_CALL` tag and will result in a return address mismatch during validation. One way to alleviate this issue is by automating the tag expansion process with an IR parser, such as LLVM. Here, the LLVM compiler pass can be written to check if the instruction following the assembly tag `CFI_CALL` is a `call` instruction creating a new stack frame. If it is, then the tag expansion is performed otherwise it is skipped. Hence, the FIXER shadow stack is always in sync with the stack frame.

3.2.7.3 Process fork support

Since the FIXER module is designed for providing protection to a single process by default, it does not support forks natively. However, this can be implemented similar to multi-process support (Section 3.2.7.1). When a fork is created for a process, it gets a new process ID, and hence is allocated as a new hardware thread with a new hartID. When the new process is created using the `fork()` system call, an additional instruction is sent to the parent process's FIXER module to copy the shadow stack and policy memory to the child process's FIXER module.

3.2.7.4 Tamper protection

The FIXER module on the FPGA also needs to be protected from tampering or data leaks. The current RocketChip implementation allows the entire code containing custom RoCC instructions to be run with supervisor privileges. This can be restricted via system calls so that RoCC instructions are first verified and then run with supervisor privileges.

3.2.7.4 Buffer overread protection

It should be noted that FIXER is still vulnerable to buffer over-reads. Similar to HAFIX and NILE, FIXER cannot enforce security if the adversary can modify binary to skip the custom instructions.

Key takeaways

In this chapter, we presented randomized stack canaries for fine grained buffer overflow detection. Following are the key takeaways from this work:

- Hardware security primitives such as PUFs and TRNGs can be employed for assisting in the mitigation of system security exploits.
- Our unique PUFCanary approach allows multiple canaries to be placed in the stack frame, providing a lightweight, yet secure way of detecting buffer overflow vulnerabilities.

- For a more performance-friendly low-power security design, our FIXER design uses a reconfigurable CFI security architecture to implement a shadow stack and a policy memory in a RISC-V coprocessor for uninterrupted program flow.
- FIXER provides fast and efficient CFI checking whereas PUFCanary provides better protection against overflow vulnerabilities at the cost of design complexity and slight performance loss.
- Decoupled security architecture implementation in PUFCanary and FIXER allows the hardware-based security engine to be attached to any RISC-V core, without invasive core pipeline modifications.

Chapter 4

Protecting the SoC Memory: Heap Exploit Mitigation

The Heap memory in a process's address space is another critical region that is used for dynamic memory allocation and can be potential target for memory corruption exploits. A heap buffer is a memory space dynamically allocated on a process's heap. In the userspace, a heap is created using GNU C library (glibc) functions such as, `malloc()` or `calloc()` as shown in Fig. 4-1. The function returns the memory address of the first byte of the allocated space and is used as the pointer to the heap. Data is written to the heap's allocated bytes with the pointer using glibc functions such as `memset()`, `memcpy()`, `strcpy()`. In the following paragraphs, we explain the basics of heap-based attacks using known vulnerabilities from the CWE database.

Buffer overflow (CWE-122): Fig. 4-2(a) shows a simple code demonstrating a buffer overflow vulnerability on the heap. Here, a heap buffer is allocated on the process's memory and a string from the command-line is copied to the buffer. However, it is easy to overflow the buffer if the size of the string is more than `SIZE`, cause memory access out-of-bounds (OOB). This can potentially overwrite process data on the heap in other allocated buffers and may lead to memory exploits. The scope of such attacks is quite large, since, in real applications, a lot of complex constructs such as objects, structs, function pointers, etc. are allocated on the heap.

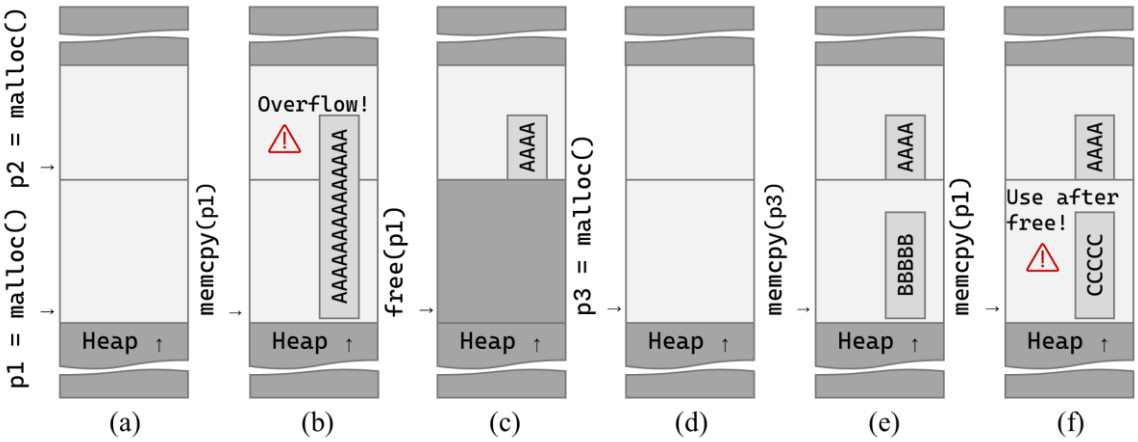


Figure **4-1**: (a) Dynamic memory allocation on a heap; (b) Buffer copy and resulting overflow; (c) Memory de-allocation; (d) New allocation in freed location; (e) New data copy to buffer; (f) Data corruption using dangling pointer (use-after-free).

Use after free (CWE-416): This vulnerability occurs when, heap memory is reallocated after the data on a heap is freed. A previously leftover reference (dangling pointer) to that memory can potentially access newly created data from that heap location. Fig. 4-2(b) shows an example of this vulnerability. In this case, location pointed by `p1` gets freed if an error occurs. It is possible that when memory referred by `p2` is created, it is at the same freed location as `p1`. In such a situation, referring to `p1` later in the code can inadvertently leak or even corrupt data at that location.

In this chapter, we describe HeapSafe, a hardware assisted heap protection engine built on the RocketChip SoC running the RISC-V ISA. We leverage the Rocket Custom Coprocessor (RoCC) to design the HeapSafe module for protection from heap-buffer overflows and use-after-free attacks. Over a traditional software-based approach, HeapSafe incurs no performance losses due to context switching or cache replacement, no manual secure memory management for bare-metal applications and no compiler dependent

```
#define SIZE 32

int main(int argc, char **argv)
{
  char *buffer;

  buf = (char*)malloc(SIZE);
  strcpy(buffer, argv[1]);

}
```

(a)

```
char* p1 = (char*)malloc(SIZE);
if (err) {
  abrt = 1;
  free(p1);
}
char* p2 = (char*)malloc(SIZE/2);
...
if (abrt) {
  logError("Aborted", p1);
}
```

(b)

Figure **4-2**: Vulnerable code showing (a) heap buffer overflow weakness; (b) use-after-free weakness.

pointer analysis. HeapSafe can achieve high backwards compatibility and completeness, while retaining good performance.

Recently, there has been some developments on a secure and memory safe processor [98-99], which utilizes the fat-pointer scheme implemented in the architecture pipeline. The primary difference between Shakti-T [98] and our work is that Shakti-T is a processor with memory safety built into the pipeline, and hence is not scalable or customizable. The metadata field width is also fixed. Similarly, [22] explores a low-fat pointer design for heap protection. However, the default implementation in [22] is performance intensive, cannot detect use-after-free (UAF) errors, and requires a custom memory allocator. Our implementation is a decoupled implementation which can be attached to any RISC-V core interfacing with the RoCCIO. Since our implementation is on a custom coprocessor, the design can be tuned and scaled according to needs, without modification of the core pipeline architecture. We can detect OOB as well as UAF errors, while maintaining high performance because of usage of dedicated hardware. Table 4-1

Table **4-1**: Qualitative comparison of Heap Protection Approaches.

Type	Compatibility	Completeness	Performance	Area
Fat pointer	↓	↑	↓	↓
Low-fat pointer	↑	↓	↑	↓
Object tagging	↑	↓	↓	↓
Hardware allocators	↑	↓	↑	↑
HeapSafe	↑	↑	↑	↑

provides a qualitative analysis of the different heap protection approaches currently explored in literature.

4.1 HeapSafe Protection Scope

In this work, we aim to protect dynamically allocated buffers on the heap that are accessed using pointers derived from the allocation pointer. We enforce metadata propagation between pointers, and the system is able to trace the correct metadata for all pointer arithmetic operations. HeapSafe can protect against buffer overflow on heap, and also prevent inadvertent use-after-free accesses. Any program targeted for the RISC-V system can be updated to use the safe heap functions from the HeapSafe library. Each pointer used to allocate a heap buffer will be converted to a *safe_pointer* by the HeapSafe library. Any other pointers derived from the *safe_pointer* is also tagged as a *safe_pointer*. We enforce tag propagation between pointers for all pointer assignments and pointer arithmetic operations. The program is compiled by including the HeapSafe library and while running on the core, the HeapSafe hardware is responsible for storing and validating heap metadata.

Table **4-2**: HeapSafe Tag Propagation.

Case	Code
Memory allocation	`safe_ptr = safe_malloc(size);`
Assignment	`safe_ptr2 = safe_ptr1;`
Pointer arithmetic	`safe_ptr2 = safe_ptr1 + offset;` `safe_ptr2 = safe_ptr1 – offset;`
Type cast	`safe_ptr2 = (type*) safe_ptr1;`

Since we are reusing the same pointer to store the tag, referencing pointers is trivial, without having to process the pointer information. This also allows us to easily enforce tag propagation in the following scenarios (Table 4-2):

(a) *Memory allocation:* When allocating a heap buffer, a new *tag* is generated. The *safe_pointer* is created using the *tag* and the base pointer (*raw_pointer*).

(b) *Pointer assignment:* During a pointer assignment, the *tag* from the original *safe_pointer* is propagated to the new *safe_pointer* alongside the *raw_pointer* value.

(c) *Pointer arithmetic:* During a pointer arithmetic operation such as array access at a specific index, the new *safe_pointer* created by adding/subtracting the offset receives the same *tag* as the original *safe_pointer*.

(d) *Pointer type conversion:* When a *safe_pointer* is cast to a new type, the *tag* is propagated to the *safe_pointer* of the new type.

Aside from these four cases, storing and retrieving *safe_pointers* from memory are trivial and same as storing and retrieving normal pointers from memory. The *tag* in the *safe_pointer* is retained throughout the store and retrieve operations. Passing *safe_pointers* as function arguments and returning *safe_pointers* from functions are also same as with normal pointers, and the *tag* is retained in the process. Two other cases that need special

mention are the null pointer and manual pointer creation from integer values. In both cases, the *tag* is set to 0. The *tag* 0 is also indicates that the pointer is not protected and any safe heap operations using the HeapSafe library with the pointer will result in an error.

4.2 HeapSafe Design

In this section we describe the implementation of the HeapSafe engine, the associated HeapSafe library and the usage of HeapSafe in standard C programs for heap buffer protection.

4.2.1 HeapSafe library

The HeapSafe protection engine is accompanied by a library containing safe implementations of critical heap buffer functions such as `safe_malloc()`, `safe_copy()`, `safe_free()` and `safe_read()`/`safe_write()`, that utilize the HeapSafe hardware. We describe the operation of these helper functions below.

`safe_malloc()`: This function allocates a buffer in the process's heap similar to `malloc()` in the GNU C library. The allocated memory is referred to by the address of the first byte of the heap called the *raw_pointer*. We create a *safe_pointer* from the *raw_pointer* by using the topmost significant byte as a tag reference. The bit allocation is

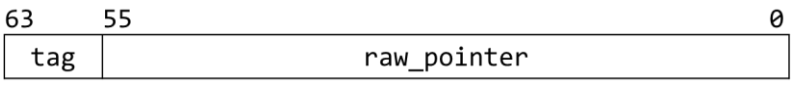


Figure **4-3**: HeapSafe `safe_pointer` bit allocation scheme.

shown in Fig. 4-3. We assign the *tag* from a static list of available tags, which are local to the process. Since we are using 1 byte to represent the *tag*, we can use HeapSafe for a maximum of 255 simultaneous heap allocations in the process. We exclude 0 as a *tag* to maintain compatibility with pointers that are not using the HeapSafe engine. Furthermore, this also excludes the higher order 256-byte memory region in the address space, however, this allocation scheme is sufficient for standard RISC-V applications running bare-metal. For RISC-V systems with user-space and kernel-space separation in memory, HeapSafe can protect user-space processes only. It is to be noted that, even though we have used 1 byte for representing the *tag*, this bit allocation is customizable in the library. While compiling the HeapSafe library, the bit allocation for the *tag* can be set as required to match with the HeapSafe hardware.

After allocating the *tag*, we send a custom RoCC instruction `HS_STORE` to the HeapSafe hardware to write the pointer metadata. We send the *safe_pointer* and the size of the allocated buffer encoded in the RoCC instruction.

The `HS_STORE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the *safe_pointer*, *rs2* is set to the register containing the size of the heap buffer, *xs1* and *xs2* fields are set to 1, and *funct7* is set to *hs_store* (b'0000000). The instruction is non-blocking, and the program proceeds without waiting for a response from the HeapSafe engine.

`safe_copy()`: This function enables a safe copy operation from a source buffer to the destination buffer which guarantees that the buffer will not be overflowed. To perform the copy operation, we check the pointer being used to refer to the destination heap

buffer. We send the pointer with the `HS_VALIDATE` instruction to the HeapSafe engine to perform an out-of-bounds validation.

The `HS_VALIDATE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the pointer, *rd* is set to a register to receive the validation outcome, *xs1* and *xd* fields are set to 1, and *funct7* is set to *hs_validate* (b'0000001).

After performing the out-of-bounds validation, the HeapSafe engine returns a 0 or 1 indicating in-bounds or out-of-bounds respectively. The `safe_copy()` function can then proceed or halt based on the validation outcome.

`safe_free()`: This function is complementary to `safe_malloc()` which allows a clean de-allocation of the memory space and metadata removal from the HeapSafe engine. We first parse the *safe_pointer* to extract the *raw_pointer*. We then send the *safe_pointer* to the HeapSafe engine with `HS_FREE` instruction to perform the metadata removal.

The `HS_FREE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the *safe_pointer*, *xs1* is set to 1 and the *funct7* field is set to *hs_free* (b'0000011). The instruction is non-blocking, so the program continues to execute on the core without waiting for a response from the HeapSafe engine. After sending the `HS_FREE` instruction, the memory de-allocation is performed normally, similar to `free()` in glibc.

`safe_read()` / `safe_write()`: By re-purposing the MSB bits of the original pointer to store the *tag*, we get the benefit of enforcing easy tag propagation. However, it precludes us from performing pointer de-referencing to read/write data in the standard way.

This is because, the *safe_pointer* by itself is not a valid memory address due to the inclusion of the *tag* bits, and de-referencing in the usual way, e.g., `data = *safe_ptr;` will raise a memory access exception. To circumvent this issue, we have also provided `safe_read()` and `safe_write()` functions, that can safely extract the *raw_pointer* from the *safe_pointer*. It then sends a `HS_VALIDATE` instruction to HeapSafe engine to validate the read/write access, and then performs the de-referencing to read/write data in memory based on the *raw_pointer*:

```
addr = extractRawPointer(safe_ptr);
data = *addr; // For read
*addr = data; // For write
```

4.2.2 HeapSafe Hardware

The HeapSafe engine is a custom designed accelerator that is decoupled from the processor core and connected over the RoCCIO interface. The engine consists of a metadata parser, a metadata table, and a validation engine as shown in Fig. 4-4.

The HeapSafe engine receives commands over the RoCCIO request interface. The Cmd Decoder decodes the RoCC instruction to read the *opcode*, the *rs1* an *rs2* data fields, and the *funct7* function field, and asserts the required control signals. In our implementation the opcode field is always decoded to *custom0*. The *rs1* and *rs2* data fields contains pointer metadata as requires for a specific function. The *funct7* field is decoded to functions such as, *hs_store*, *hs_validate* and *hs_free*.

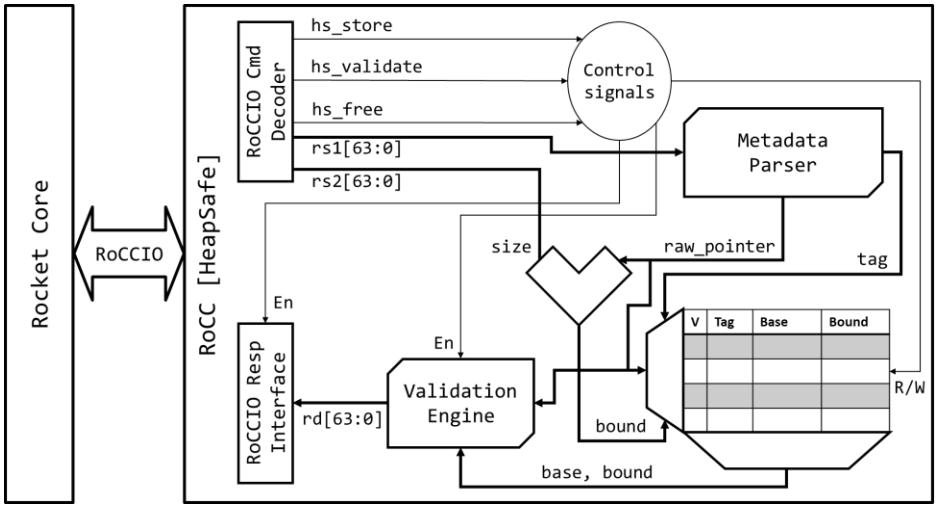


Figure **4-4**: HeapSafe architecture in RoCC.

***hs_store*:** This function indicates a heap buffer creation and instructs the HeapSafe engine to store the associated metadata which is received in the form of the *safe_pointer* on the rs1 field and the size on the rs2 field. The metadata parser processes the *safe_pointer* and extracts the tag and the *raw_pointer* values based on the specified bit encoding.

The metadata table is implemented as a hardware content-addressable memory for parallel search and fast lookup. The table consists of three fields for storing the metadata - (i) Tag, (ii) Base, and (iii) Bound. Each row in the table is designed as a vector of the three 64-bit wide fields. In addition to the metadata, each row in the table also contains a valid bit to indicate the validity of the metadata. The size of the table is customizable as part of the design and can be set while instantiating the hardware. In our implementation for testing, we have used a table consisting of 256 rows, allowing metadata storage for a maximum of 256 heap buffers.

A write signal is automatically issued on decoding the *hs_store* as the function, which writes the pointer metadata at an available location in the metadata table. These

locations are indicated by the valid bit set to 0. The parsed tag is stored in the Tag field and the *raw_pointer* is stored in the Base field. Instead of storing the size of the heap buffer, we pre-compute the bound address value as:

$$Bound[Tag] = raw\_pointer + size \quad (4\text{-}1)$$

Since the metadata store instruction is non-blocking, we improve performance by saving the calculated bound address value in the Bound field. Finally, the valid bit is set to 1 to mark the row as active.

***hs_validate*:** This function indicates a heap buffer write operation on the core and instructs the HeapSafe engine to validate the pointer's access bounds. The pointer being used to access the heap is received on the *rs1* field. The metadata parser processes the pointer on the *rs1* field and extracts the current *tag* and the *raw_pointer* (memory address) values.

A read signal is issued on decoding the *hs_validate* as the function, which performs a parallel search of the current *tag* on the *tag* field of the metadata table. Once a tag match is found, the *Base* and *Bound* fields are read. The access bounds for the current pointer (*ptr*) are validated as:

$$isOOB = (ptr < Base) \,||\, (ptr \geq Bound) \quad (4\text{-}2)$$

The out-of-bound signal (*isOOB*) is asserted when the current pointer (memory address) is either less than the lower bound (base) or is greater than or equal to the upper bound of the heap buffer. The value of the *isOOB* signal is held at 0 if the current pointer is within bounds. Since the validation is a blocking operation by default, having the upper bound address of the heap buffer in the metadata table speeds up the validation time. The

value of *isOOB* signal is placed on the *rd* field of the response interface and is sent back to HeapSafe library to take the required action - (i) proceed or (ii) terminate.

hs_free: This function indicates a heap buffer de-allocation on the core and instructs the HeapSafe engine to clear its corresponding metadata. However, instead of removing or zeroizing the metadata from the table, we set the valid bit for the row to 0 to invalidate the metadata entry and mark it as available for future use. The *hs_free* operation is non-blocking and the program on the core continues to run without waiting for a response from the HeapSafe engine. Invalidating metadata entries allow us to mitigate inadvertent use-after-free vulnerabilities, since the *tag* for the dangling pointer will be invalidated after free.

4.2.3 HeapSafe Usage in C Programs

We demonstrate a basic use of HeapSafe in a simple C program (Fig. 4-5(a)). Let us consider a function that receives a lowercase string data, converts each character to uppercase and stores to a buffer allocated on the heap. The function then returns the pointer to the heap buffer storing the uppercase string.

In this program, the `upper` buffer is vulnerable to overflow, and hence, let us modify the code to use HeapSafe to protect the buffer. We assume that `lower` buffer does not need to be protected. We perform the modification as follows:

We first replace the `malloc()` function call with `safe_malloc()`, that returns a tagged `safe_pointer` to `upper`. When iterating over the characters from the source buffer, after converting to lowercase, we need to de-reference the `upper` pointer to store

```
char* convert_case(char *lower)
{
  char *upper;
  upper = malloc(SIZE);
  while (*lower != '\0') {
    char u = *lower - 32;
    *upper = u;
    upper++;
    lower++;
  }
  return upper;
}
```

(a)

```
char* convert_case(char *lower)
{
  char *upper;
  upper = safe_malloc(SIZE);
  while (*lower != '\0') {
    char u = *lower - 32;
    safe_write(upper, u);
    upper++;
    lower++;
  }
  return upper;
}
```

(b)

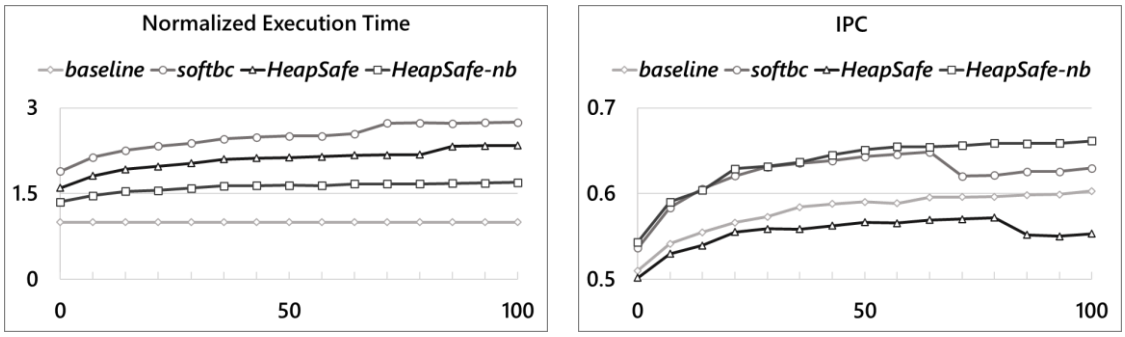
Figure **4-5**: (a) Source code with heap buffer overflow vulnerability; (b) HeapSafe protected code to prevent overflow.

the uppercase character. We modify the de-referencing code with the `safe_write()` function that performs the correct write to memory operation. Towards the end of the loop, we update the destination heap pointer (`upper`) by increasing the pointer by 1. The pointer arithmetic propagates the original tag for `upper` to the new `upper`. The HeapSafe protected code is listed in Fig. 4-5(b).

4.3 Evaluation

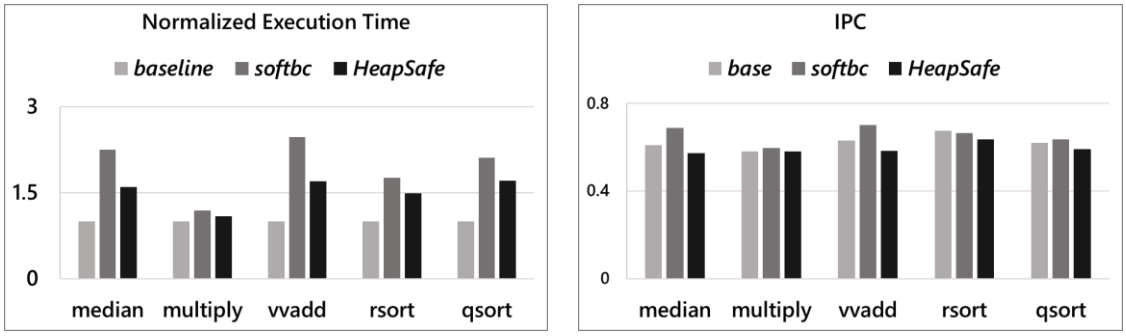
We evaluated HeapSafe by generating a RocketChip SoC design config with the HeapSafe module. We tested the HeapSafe security architecture in the C++ cycle accurate emulator built from the config. The hardware architecture of HeapSafe is coded in CHISEL and synthesizable Verilog is generated using the RocketChip generator. To evaluate performance, we created sample workloads that perform multiple buffer copy operations on the process's stack and heap. We compiled three versions of the code: (i) *baseline* with

no protection, (ii) *softbc* with in-process software-based bounds checking, and (iii) *HeapSafe* with our HeapSafe library and protections. We swept the workload balance between the stack and the heap and evaluated the trend as shown in Fig. 4-6. We note that *HeapSafe*'s execution time overhead is low and similar to *softbc* when there are more stack workloads; however, as the heap workload increases compared to the stack workload, *HeapSafe* tends to perform better. At around 75% heap workload, *HeapSafe* performs 20% faster than *softbc*. However, IPC suffers in *HeapSafe* compared to *softbc*, since *HeapSafe*



(a) (b)

Figure **4-6**: (a) Execution time trend normalized to baseline; (b) IPC trend. X-axis represents the percentage of buffer copy operations occurring on the heap.



(a) (b)

Figure **4-7**: RISC-V tests benchmarks protected with *softbc* and *HeapSafe*. (a) Execution time normalized to baseline; (b) IPC.

runs less instructions in total. We have also implemented a fully non-blocking version of HeapSafe which outperforms *softbc* in both execution time and IPC at the cost of delayed heap corruption detection. At high heap workloads, *HeapSafe-nb* is 38% faster than *softbc*. We estimated the area of HeapSafe by generating a E300 Arty FPGA bitstream and found it to have a nominal 1.59% overhead (number of cells) over the default configuration. We further updated the RISC-V ISA test benchmarks with *HeapSafe* and *softbc* and evaluated their performance (Fig. 4-7). *HeapSafe* incurs a 1.5X overhead over *baseline* on average, while being 22.4% faster than *softbc*. Our results are better than a similar software-based approach in [22], which shows an average 2.5X overhead. Average IPC is slightly low at 0.59 compared to 0.62/0.65 (*baseline*/*softbc*). A qualitative evaluation of HeapSafe has been shown in Table 4-1 as well.

4.4 Design Improvements

HeapSafe implements fine-grained approach for instant detection of heap overflow at the cost of some performance penalties. However, such high security guarantees are not needed in less critical systems. We propose the following alternate flavor of HeapSafe system design:

4.4.1 Non-blocking validation

`HS_VALIDATE` instruction can be made non-blocking to avoid wait by the core for a response from HeapSafe. In this design, HeapSafe will asynchronously raise an exception

on the core when it detects an out-of-bounds error (instead of sending the value of the *isOOB* signal on the response interface). An exception handler is implemented in the HeapSafe library will terminate the application in such cases. This approach will improve performance by 27.6% (Fig. 4-6(a)) at the cost of slight delay in attack detection.

4.4.2 Compiler support

Replacement of unsafe heap operations with safe variants using library can be automated by adding compiler support. In this design, we compile the source program to be protected using LLVM/Clang for RISC-V. The LLVM generates an intermediate representation (IR) of the source code. An LLVM compiler pass is written to parse the IR to scan for heap pointers and replace them with the tagged *safe_pointers*. The custom HeapSafe instructions are also inserted for the required operations to communicate with the HeapSafe engine. The IR is then compiled to generate the ELF binary to run on the system. This improvement is more design friendly to the source code programmer.

4.4.3 Top byte ignore

Pointer de-referencing requires additional library functions to perform the correct pointer extraction in HeapSafe. This can be avoided if the core is set to ignore the top byte for any memory address in the userspace. This can be achieved by conditionally masking the top byte of the address in the address decoder in the core pipeline. This will guarantee

that any tagged *safe_pointer* is seen as a normal *raw_pointer* in the pipeline, and all load/store operations will automatically be performed using the *raw_pointer*.

4.4.4 Multi-process support

Due to the decoupled coprocessor-based design, HeapSafe can be scaled up to support protection of multiple processes simultaneously. RISC-V cores view each running process as a hardware thread (hart). In the scaled-up implementation, multiple instances of the HeapSafe coprocessor are instantiated in the same tile along with the core. Each instance of HeapSafe engine is associated with a `hartId`. When a process using the HeapSafe library is running on the core, the system hardware selects the specific HeapSafe engine to use with the hart for that process.

4.5 Discussions

4.5.1 Security of HeapSafe hardware

In order for HeapSafe to guarantee protection, it needs to ensure that the HeapSafe hardware is not compromised. Hence, we need to ensure the integrity of the RoCC HeapSafe engine, so that no malicious code can overwrite metadata entries in the metadata table. This is guaranteed to some extent by appropriately setting the RocketChip configuration to run RoCC operations in machine (M) mode only, while rest of the code runs in user (S) mode. This prevents any malicious code to run the RoCC instructions while in user mode. The security of the hardware can be further improved by running the

HeapSafe library functions in machine mode only. This requires some additional mediation logic in the application code utilizing traps that requires a switch to machine mode from user mode when calling a HeapSafe function, and then exit to user mode after returning from the function. This can mitigate code-reuse attacks that might try to run the HeapSafe library functions maliciously.

4.5.2 PMP vs. HeapSafe

The RISC-V architecture provides some basic memory protection as part of the ISA. There are 16 Physical Memory Protection (PMP) registers in the base architecture, which can be utilized to perform access control on different memory regions. The PMP registers allow machine (M) mode to specify which memory regions are available during user (U) mode operations. This is an easy and low-cost way to implement memory protection in user mode for simple systems. However, due to the limited number of registers available, it imposes a restriction on the number of regions it can protect. Hence this is not scalable to more complex applications. Furthermore, PMP protected regions need to be contiguous in physical memory, it can lead to memory fragmentation.

In contrast, HeapSafe can be applied in a more granular manner to individual pointers pointing to memory locations. The number of regions to be protected is not restricted by the core architecture but set by the configurable HeapSafe engine. Thus, HeapSafe is more scalable and versatile than PMP.

4.5.3 Backwards compatibility

Our HeapSafe implementation is fully backwards compatible with standard unprotected pointers. This allows the source code programmer to use a mix of protected and unprotected pointers. The programmer can opt to use HeapSafe protected pointers only for security critical heap regions. Although this is less secure, it improves the performance since the program is not being slowed down due to unnecessary validations.

We achieve backwards compatibility by assuming a *tag* value of 0, since pointers in user-level code has the MSB bits as 0. Since this is inherent to the design, we simply exclude 0 as a *tag* for HeapSafe pointers. If we encounter a pointer with its *tag* bits as 0, we treat the pointer as a non-protected pointer and exclude it from validation.

4.5.4 Limitations

HeapSafe has a few limitations in its current implementation. It is unable to protect pointers derived from a null pointer, since its *tag* is set to 0. For example, a NULL[offset] can be used maliciously to access a random address. Developing compiler support to scan for pointer arithmetic from NULL pointers can overcome this limitation.

HeapSafe cannot protect against type-confusion attacks. Since we propagate the tag for type-casted pointers, an adversary can re-cast a pointer for an object and access memory by using the object members. A compiler pass can be written to sanitize type-casts in such cases; however this may reduce performance.

```
class WithNHeapSafe(n: Int) extends Config((site, here, up) => {
  case BuildRoCC => List.tabulate(n)( i =>
    (p: Parameters) => {
      val heapsafe = LazyModule(new HeapSafe(OpcodeSet.custom0,
                                             mtSize = 256,
                                             hartId = i)(p))
      heapsafe
})})
```

Figure **4-8**: RocketChip config class for configurable HeapSafe module generation.

4.6 HeapSafe System Generation

Due to the flexibility in HeapSafe's design configuration, we can customize the size of the metadata table to be generated on the hardware. We can also create multiple instances of the HeapSafe coprocessor to support simultaneous multi-process protection. The code in Fig. 4-8 demonstrates the easy configurability of HeapSafe design.

When generating a system configuration with HeapSafe, we can set the parameter `n` to specify the number of HeapSafe modules to instantiate. We associate a `hartId` to each HeapSafe module. The size of the metadata table can be set through the `mtSize` parameter. In this config, it is set to 256. In the HeapSafe module implementation, we calculate the bit allocation scheme for the tag in the *safe_pointer* as $\log_2 mtSize$, e.g., if `mtSize` = 256, we set the tag bit allocation to the MSB 8 bits in the *safe_pointer*.

During the SoC generation, the HeapSafe configuration changes are independent from the core configurations and the rest of the SoC configuration. This provides an advantage that the HeapSafe module can be hooked up to any RISC-V core configuration

implementing the RoCCIO interface. As long as the application running on the core is compiled including the HeapSafe library, protection can be provided by the HeapSafe hardware.

Key Takeaways

In this chapter, we presented HeapSafe, a customizable and lightweight heap protection hardware engine for the RISC-V SoCs. Following are the key takeaways from this chapter:

- Heap memory exploits are difficult to detect compared to stack memory.
- For low-power resource constrained SoCs, software-based protection approaches tend to be expensive.
- HeapSafe provides pointer protection by tagging and bounds validation on heap memories using hardware assistance.
- Decoupled security architecture implementation in HeapSafe allows the hardware-based metadata storage and validation engine to be attached to any RISC-V core.

Chapter 5

Hardware Trojans: A Hidden Threat

In the previous chapters, we explored different system level threats and countermeasures for RISC-V SoCs. The underlying assumption of the explored threats and their mitigation techniques is that the hardware itself is free from malicious tampering. However, this assumption may not be true always. RISC-V SoC architectures are highly configurable and customizable due to its open ISA and extensible design. These SoCs platforms allow rapid design prototyping and manufacturing by sourcing certain intellectual property (IP) designs from 3rd party vendors. In such hybrid design environments, the SoC designer may choose to develop only the core as an in-house IP, while licensing other component IPs such as memories from other IP vendors. Most RISC-V SoC developers are also fabless, and hence need to outsource the SoC fabrication jobs to 3rd party foundries. Such a distributed approach is highly efficient in terms of cost and time-to-market. However, it is also a very weak approach in ensuring the integrity of the semiconductor supply chain.

In this chapter, we will explore Hardware Trojans, and how they can be leveraged to launch system level attacks. Hardware trojan is a malicious circuit modification in the design or manufacturing process with the goal of compromising the functionality. Although hardware Trojan is a well-explored topic in the VLSI design communities, most

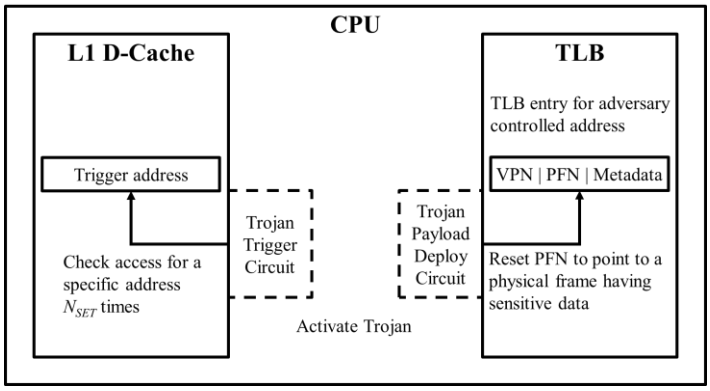


Figure **5-1**: Overview of HarTBleed Trojan.

of the attack vectors are confined to low-level circuits and implications such as, DoS and leakage of cryptographic keys. The potential of hardware Trojans in compromising system assets and launch system security attacks have received much less attention.

In this work, we assume an untrusted manufacturing house located outside the US that can alter the chip GDS-II file to introduce the malicious Trojan trigger and payload. This assumption is widely accepted in the hardware security community because of large filler areas present in the chip and the adversary's access to the raw design. We propose the use of a capacitor-based hardware Trojan trigger [62] and novel payload circuits and perform detailed analysis to guarantee that the Trojan is, (i) triggered even under worst-case process and temperature conditions with correct inputs; (ii) able to bypass conventional post-manufacturing test. The Trojan is activated if a particular preselected address of L1 Cache is accessed for ~1800 times. Note that the proposed Trojan trigger directly taps the wordline of the preselected address to leverage the existing decoder design framework and hence, does not incur any overhead for address decoding. Once activated, the trigger will reset the preselected TLB (Translation Lookaside Buffer) entry to preselected bit pattern. A TLB entry is composed of a tag (Virtual Page Number (VPN))

and mapping information (Physical Frame Number (PFN), and other metadata bits). The tag is implemented using a Content Addressable Memory (CAM) for high-speed lookup operation, while the mapping information is implemented on a 6T SRAM array [100]. The proposed Trojan payload resets the SRAM portion of the TLB.

In order to simulate the effect of the Trojan, we design HarTBleed, an attack with the hardware Trojan trigger embedded in the CPU cache and the TLB as the victim, as shown in Fig. 5-1. The goal of the Trojan payload is to manipulate the TLB mapping information to access data from a process's address space. The Trojan is triggered when a specific address of L1 is accessed multiple times. This changes the address mapping in a target TLB line to a specific physical page with sensitive information.

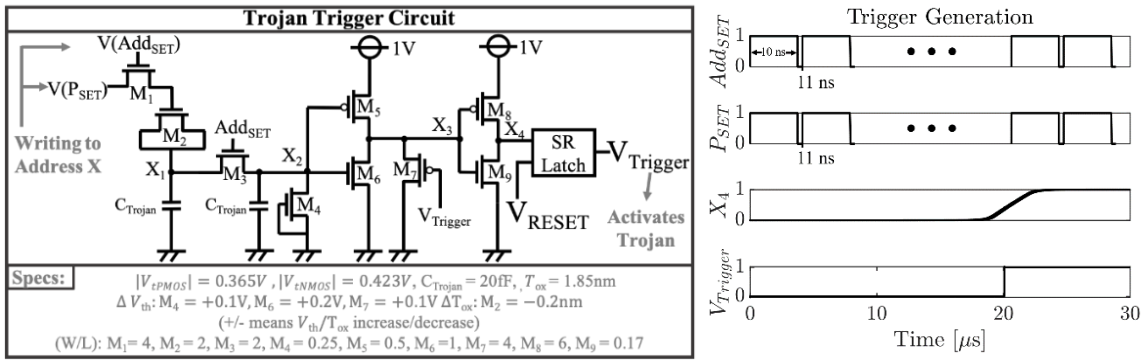
In the following sections, we will first introduce the design of the Trojan trigger and payload that is used in HarTBleed. We then explain the effect of the attack payload in the context of a system level data leakage attack.

5.1 Hardware Trojan Design for HarTBleed

In this section, we will present the Trojan trigger and payload circuit designs. We explain the trigger features and the payload deployment mechanisms.

5.1.1 Trojan trigger

The trigger circuit (Fig. 5-2a) is designed to be activated if a particular memory address (chosen during design phase, let's call it Add_{SET}) is accessed for at least N_{SET}



(a) (b)

Figure **5-2**: (a) Trojan trigger circuit (specifications provided at the bottom). $V(P_{SET})$ and $V(Add_{SET})$ are two inputs, (b) Trojan trigger waveform.

times. The trigger has two inputs namely, $V(Add_{SET})$ and $V(P_{SET})$. $V(Add_{SET})$ (= 1V in this work) is the wordline enable signal of $V(Add_{SET})$ and $V(P_{SET})$. is a constant voltage source of 1V. For a more complex Trojan with a superior stealthiness, $V(P_{SET})$. can be programmable.

Whenever Add_{SET} is accessed, $V(Add_{SET})$ is asserted and MOSFETs M_1 and M_3 is activated. M_2 has a thinner gate oxide compared to other MOSFETs and its source and drain are shorted. Therefore, M_2 works as a capacitor and charges C_{Trojan} from the P_{SET} source through Fowler Nordheim (FN) tunneling [101] if $V(Add_{SET})$ is asserted. M_4 is an OFF transistor which offsets gate leakage of M_5 and prevents unwanted charging-up of node X_2. M_7 keeps node X_3 as low as possible until node X_2 charges up sufficiently. The node X_4, that is charged up during the hammering process, is used as the SET input of a SR latch. The output of the SR latch ($V_{Trigger}$) transitions from 0 → 1 when X_4 charges up to 0.5V. The signal $V_{Trigger}$ is then used to activate the Trojan. Fig. 5-2b shows the Trojan trigger waveform.

Table **5-1**: Features of the trojan trigger.

Parameter	**Value**
Dynamic Power (μW)	3.781
Static Power (μW)	0.589
Energy/hammer (nJ)	2.059
Area ((μm^2))	42.97
Target $\boldsymbol{N_{SET}}$	1837
Worst Case $\boldsymbol{N_{SET}}$	1502

The charge at node X_2 leaks away (due to capacitance leakage of C_{Trojan}) once the hammering is discontinued. However, $V_{Trigger}$ will still be asserted due to the SR latch. To deactivate the Trojan, V_{RESET} needs to be asserted. V_{RESET} can be generated by accessing a different address (let us say Add_{RESET}) for at least N_{RESET} times and using a circuit similar to the trigger one. Note that a smaller C_{Trojan} (~1fF) can be used in the RESET circuit to minimize the area overhead which leads to $N_{RESET} = 92$. However, the AND'ed output of $V(Add_{RESET})$ and $V(P_{RESET})$ can also serve as V_{RESET} which further reduces the area overhead.

Table 5-1 summarizes the key performance metrics of the Trojan trigger. The absolute area and static power of proposed trigger are 42.95μm^2 and 0.589μW, respectively which are $5.34 \times 10^{-5}\%$ and $6.24 \times 10^{-5}\%$ of a typical memory chip area and static power [102], respectively. Therefore, the overhead due to Trojan trigger is negligible to be detected via optical inspection or side channel analysis.

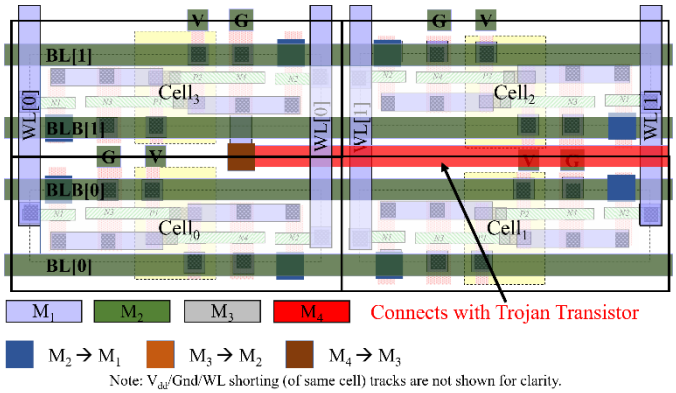


Figure **5-3**: Layout showing 4 SRAM bitells and metal tracks allocated for bitlines (horizontal) and wordlines (vertical). One M_4 track can be stolen to connect the SRAM data node to Trojan payload transistor (located in the column area).

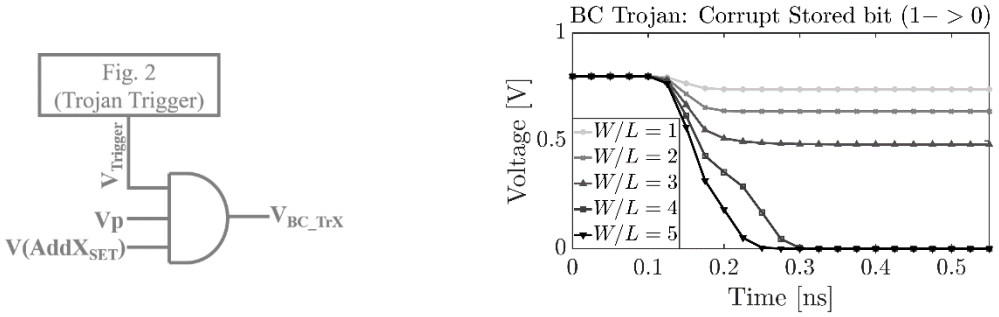
5.1.2 Resetting single and multiple TLB entries

The Trojan payload resets TLB entries after the trigger is activated. We have chosen the TLB as the victim as it provides a greater attack surface to leak data from multiple sources such as, kernel, process space, shared memory etc. Furthermore, the payload circuit is placed in the SRAM portion of the TLB itself, as it is easy to conceal the transistors in the SRAM layout and peripheral circuits (as shown in Fig. 5-3 and Fig. 5-4).

A single TLB entry can be reset to preselected data pattern (i.e., 0 or 1) as shown in Fig. 5-4a-b. If the adversary needs to reset multiple TLB entries (say, 10), he needs to select 10 L1 cache addresses. Let's call them $AddX_{SET}$, where X = 1, 2, ..., 10. For each of the target TLB entries the adversary designs one AND gate with $V_{Trigger}$ and $V(AddX_{SET})$ as inputs (Fig5-5a). When the adversary accesses address $AddX_{SET}$ after the Trojan is activated, V_{BC_TrX} will be asserted. This will reset the stored bits in the corresponding TLB entry to 0/1 as designed during Trojan insertion. Fig. 5-5b shows the simulation result of

(a) (b)

Figure **5-4**: TLB entry reset to (a) 0; and (b) 1. The sizes of SRAM pull-up, pull-down and access transistors are shown. The back-to-back inverters in SRAM are omitted for clarity.



(a) (b)

Figure **5-5**: (a) Logic circuit to generate V_{BC_TrX}; (b) data node voltage discharges as the (W/L) of Trojan transistor increases. It discharges to 0V for a minimum (W/L) of 4. This means that the stored data flipped i.e., 1 → 0 fault occurred.

flipping the stored bit to 0 with respect to (W/L) of Trojan transistor. For a successful flip operation, minimum (W/L) is 4. We have used (W/L) = 5 for a faster switching. Fig. 5-6 shows the layout of 4 SRAM cells and metal tracks allocated for bitlines (horizontal) and wordlines (vertical). One M_4 track per global column of SRAM array connects the target SRAM data node to the Trojan payload transistor which is co-located with sense-amp and other peripherals in the column area. To reset a n-bit wide PFN in the TLB, n Trojan payload transistors are required.

5.1.3 Evading test

During conventional memory testing such as March test, each address is written with different data patterns (e.g., block-0/1, stripe, checkerboard and so on) [103] and then read to verify memory functionality and correctness (i.e., free of faults such as stuck-at faults, coupling faults etc.). The test pattern consists of a finite sequence of March elements. Each element consists of increasing or decreasing address order of read/write operations covering all memory cells. In a 32-bit wide memory, a striping pattern requires 32 accesses per address, whereas a checkerboard (alternating) pattern requires 16 accesses per address. Therefore, each address may be accessed for a maximum 16 to 32 times. This method of testing ensures linear test-time complexity. The Trojan proposed in this paper requires 1837 accesses which is significantly higher than 32 and therefore, evades the test. Considering the overall test time, the proposed trigger requires approximately $1837 \times 64 \times 1024 = 120389632$ accesses in the worst case per 64KB byte-addressable memory. Even if high temperature and V_{dd} is used during burn-in test, the trigger does not get activated. An advanced trigger with a unique data pattern makes detection more difficult.

5.1.4 Bypassing error detecting codes

TLBs typically employ parity-based error detecting codes which can detect the fault injection by the Trojan. The parity bits can also be reset to match with the new data to avoid detection. If Error Correcting Code (ECC) is employed, the Trojan payload could reset the ECC appropriately to bypass detection at the cost of few extra transistors.

5.2 Systems Architecture

In this section, we describe the architecture for HarTBleed deployment and the threat model under consideration.

5.2.1 Overview of the systems architecture

We consider a standard RISC-V 32-bit microprocessor architecture with a single core. The memory system is configured with L1 and L2 caches, where the L1 cache is further separated into L1 instruction cache and L1 data cache. The L1 cache is virtually indexed and physically tagged to improve performance. The processor is connected to a DDR system memory. The system runs a standard Linux kernel with paging enabled, with a fixed page size of 4KB. The memory management unit (MMU) in the microprocessor also has a fully associative TLB to speed up address translation.

The physical memory in a system is very limited in size, and much less than the range of addresses the CPU can reference. All applications running on a specific CPU architecture is designed for the addressable memory space that the specific CPU architecture can reference. For example, a 32-bit CPU can address 2^{23} memory locations, which is a 4GB address space. This is known as the virtual address space. However, the installed physical memory (and the physical address space) can be different for different system configurations. Hence the virtual address space is divided into smaller chunks known as virtual pages (Fig. 5-6a). Typically, the virtual pages are 4KB each in a 32-bit address space. Each process has its own view of the entire virtual address space. When an application process runs, the required virtual pages are mapped to the physical memory as

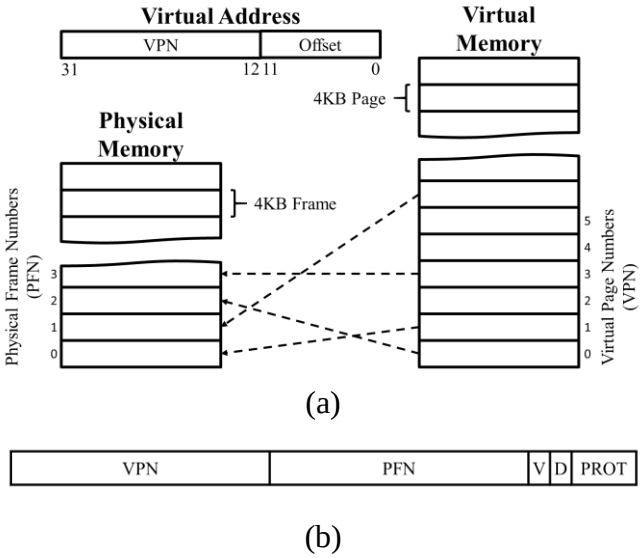


Figure **5-6**: (a) Paging in a 32-bit address space with 4KB pages; (b) an example TLB entry configuration showing VPN, PFN and the associated metadata bits.

physical frames. The physical memory can have frames belonging to multiple processes simultaneously. The corresponding mapping information (virtual page number → physical frame number) is stored in a per-process page table in the OS kernel. The page table entries (PTE) also store some associated metadata bits signifying protection status, caching information, etc. When the CPU accesses a memory address, the corresponding page mapping is pulled from the page table in the kernel and cached onto the TLB in the CPU.

The TLB is implemented as a small, fully associative SRAM memory. Each entry in the TLB contains the virtual to physical memory mapping information of a page of the current process in context. TLB entries are configured as shown in Fig. 5-6b. In a 32-bit address space with 4KB pages, the Virtual Page Number (VPN) is the MSB 20 bits of the virtual address. The Physical Frame Number (PFN) bits signify the frame number on the physical memory where that page is mapped. Aside from VPN and PFN, the TLB may also

have protection bits specifying read/write/execute permissions, a valid (V) bit indicating if the frame is present in physical memory, and other bits such as global (G), cached (C), dirty (D), etc. When a process goes out of context, the TLB is flushed or invalidated.

5.2.2 Threat model

We describe the threat model under consideration as follows: We assume a standard multi-user multi-process system. The microprocessor hardware used in the system is compromised with a hardware Trojan (introduced during manufacturing process) that was undetectable during post-silicon testing. We assume that the adversary has the knowledge of the Trojan hardware and the specific scenario in which the Trojan can be triggered and activated. This is very likely if the adversary is foundry-sponsored. After the deployment of the chip in the market, the adversary can launch a malicious program to trigger the Trojan for the desired payloads. The adversary has standard user privilege i.e., no administrator or superuser privileges. The adversary can interact with the existing applications running on the system and can also compile and run his own malicious program with user privileges. It is assumed that the OS kernel is generally secure and cannot be tampered by a normal non-adversarial user. The adversary need not have physical access to the system and can access the system over a network. However, the adversary is knowledgeable about the hardware details of the system in use.

We assume that the hardware Trojan has deep access to the system hardware at the physical layer. The Trojan trigger can tap into and monitor the CPU address and data buses and the payload can reset the TLB entries and their associated metadata to the adversary-

known values. The Trojan can only be triggered after a specific event, or a series of events have taken place. This trigger event may not be an event which occurs during normal operation but can be forced to happen by the knowledgeable adversary. The adversary can also deactivate the Trojan after carrying out the attack to prevent further detection or to prevent system faults. It should be noted that the Trojan hardware is dormant, and it cannot proactively engage and cause data-leaks or failures. It only serves as a hidden system backdoor, which a knowledgeable adversary can leverage to launch his exploits.

5.3 Designing HarTBleed Exploits

In this section, we describe and demonstrate the design methodology for HarTBleed exploits.

5.3.1 Attack design

We describe a simple proof-of-concept attack scenario demonstrating the use of the hardware Trojan. In this attack we focus on extracting data from a process's heap. Let us consider a single victim process which accesses a certain memory location T that is initialized (written) with a specific bit pattern, and then repeatedly accessed. The number of accesses to the location is controlled by a user input N. Let us also assume that the process stores some `secret' data D, which is kept in the process's heap at location S. This ‘secret’ data is not available for the user to view in its raw form. This can happen for example, in a program which performs encryption/decryption in software by using some

cryptographic keys. The keys will not be available to the user, but it may still be loaded into the process's memory. The goal of the adversary posing as the user is to obtain the secret cryptographic key, using some existing input/output operation in the program. The process also reads back and prints data from location P to the user. The goal of this attack is to interact with the victim program by controlling N, and force it to print the secret D using the code which normally prints from location P.

It should be noted that this is a very controlled attack where we make certain assumptions about the program structure, which may not be possible in real attacks. However, this attack example shows how a hardware Trojan can be used to perform unauthorized data access. More sophisticated attacks can stem from this simple example. We make following assumptions in this attack: (i) memory locations T, S, and P reside in different pages, i.e., accesses to each of these addresses creates different page table entries (and TLB entries); (ii) addresses S and P have the same offsets into their pages; (iii) all these pages are mapped to the physical memory frames and hence have valid and present entries in the TLB; iv) the adversary has knowledge about the cache line that maps to trigger address T, the TLB entry for P, and the Trojan payload used to modify the TLB entry to maliciously map to S.

The malicious hardware Trojan is designed as shown in Fig. 5-7. The trigger for the Trojan is placed in a L1 d-cache line (in this case, the cache line which maps to T), that monitors its access for N_{SET} times. This action activates the Trojan trigger which generates a signal for the Trojan payload. The Trojan payload performs a malicious mapping in the TLB. It first accesses the TLB entry that stores the mapping for the page of P. The initial

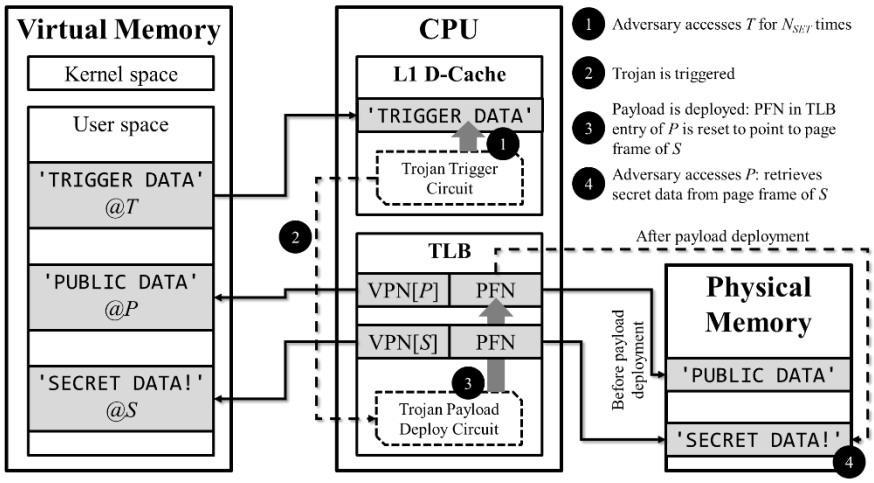


Figure **5-7**: Trojan design in the system architecture. The attack steps are annotated.

mapping is VPN[P] → PFN[P]. It then resets the physical frame number for P to that of S. Thus, the effective mapping becomes VPN[P] → PFN[S].

We launch the attack as follows: We first provide N as a large value. This makes the process access the location T for N times. This action creates N repeated accesses to the same cache line that is mapped to T, which serves as the trigger for the Trojan. The activated Trojan performs the malicious mapping as detailed above. Now, when the process reads from virtual address P, the TLB translates this address to the physical address that stores the secret data D, and the process unknowingly prints out D.

This attack assumes that the victim program coincidentally happened to work in favor of the hardware Trojan, which may not always be the case. However, the adversary can use the same attack methodology to write his own malicious program with the knowledge of the address (T) which the Trojan uses as a trigger, the number of accesses required to trigger the Trojan (N), and the location of the data of the victim process in physical memory which he is trying to leak. The secret data may be in the kernel (e.g., page

tables), or in other processes, that have a presence in the physical memory. We discuss these attacks in Section 5.4.2.

5.3.2 Attack demonstration

We demonstrate the attack with the help of a simple C program shown in Fig. 5-8. We assume that the program itself is the victim and contains some sensitive 'secret' data that the adversary is trying to leak. In the code, `addr_t` is used as the trigger address location (T). This is first initialized (written) with the data `'TRIGGER DATA'`. The location (S) for the sensitive data (D) is allocated in the heap as `addr_s`. The data to be leaked (D) is represented as `'SECRET DATA!'` and is copied to `addr_s`. A public data location (P) is also allocated on the heap as `addr_p`, and the data `'PUBLIC DATA'` is copied to it. The sizes of the allocated memory pointed to by `addr_s` and `addr_p` are large enough so that they are in different pages but at the same offset in the pages, and thus have different TLB entries.

We initially loop over and read the Trojan trigger address `addr_t` 2000 times, which is greater than the minimum accesses required by the Trojan trigger as set by N_{SET}. This Trojan trigger monitors the L1 data cache line mapped to `addr_t`. After N_{SET} (= 1837) accesses, the Trojan is triggered, and a signal is sent to the TLB to perform the malicious mapping. Once triggered, the signal stays asserted for several microseconds to facilitate the attack. In the TLB, the Trojan payload is placed at the entry corresponding to `addr_p` by resetting the PFN field in the entry to the PFN for `addr_s`. The frame for `addr_s` stores the secret in the physical memory. After the TLB receives the signal to deploy the

payload, it performs this malicious mapping. Now when we read from `addr_p`, it is able to read the secret. We can see the output of the program in Fig. 5-9. Before the Trojan is triggered, reading `addr_p` prints `'PUBLIC DATA'`. After the Trojan is triggered, `addr_p` can access and read back `'SECRET DATA!'`.

```
void main() {
      char *addr_t, *addr_s, *addr_p;
      char data;
      int i, len;

      /* N: Number of accesses to trigger address, minimum 1837 */
      int n = 2000;

      /* T: Address of trigger; value initialized */
      addr_t = (char*)malloc(sizeof(char) * 4080);
      strcpy(addr_t, "TRIGGER DATA");

      /* S: Address of secret data */
      addr_s = (char*)malloc(sizeof(char) * 4080);
      strcpy(addr_s, "SECRET DATA!");

      /* P: Address of public data */
      addr_p = (char*)malloc(sizeof(char) * 4096);
      strcpy(addr_p, "PUBLIC DATA");

      /* Before Trojan activation, P points to public data */
      printf("## Before Trojan is activated ##\n");
      printf("addr_t = %p\tvalue = %s\n", addr_t, addr_t);
      printf("addr_s = %p\tvalue = %s\n", addr_s, addr_s);
      printf("addr_p = %p\tvalue = %s\n", addr_p, addr_p);

      /* Trigger the Trojan by N accesses to T */
      for (i = 0; i < n; i++) {
            data = addr_t[0];
      }

      /* After Trojan activation, P points to secret data */
      printf("## After Trojan is activated ##\n");
      printf("addr_t = %p\tvalue = %s\n", addr_t, addr_t);
      printf("addr_s = %p\tvalue = %s\n", addr_s, addr_s);
      printf("addr_p = %p\tvalue = %s\n", addr_p, addr_p);
}
```

Figure **5-8**: Example attack code using the HarTBleed Trojan.

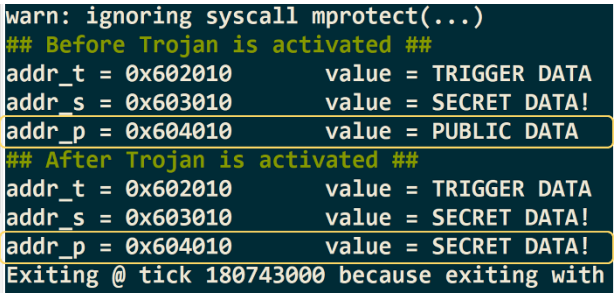


Figure **5-9**: GEM5 console output for the attack code in Fig. 5-8.

5.3.3 Evaluation

We model the effects of the hardware Trojan in the GEM5 architectural simulator [104]. The system is designed with an CPU model running a single core at 1GHz, connected to a DDR3 physical memory of size 512MB. Two levels of cache memory are configured, L1 and L2. L1 I-Cache is 32KB and L1 D-Cache is 64KB, while L2 cache is 2MB. The MMU in the CPU is configured with a 64 entry TLB.

To simulate the Trojan trigger, we added code to the `readMem()` function in the CPU code to monitor accesses to trigger address `addr_t` (`0x602010`). Since the L1 cache is virtually indexed, we can just monitor the cache line mapped to the trigger address. For the Trojan payload insertion, we modified the TLB class code to handle Trojan activation and deactivation states. We added additional class methods to deploy the Trojan payload when activated. In the payload deployment method, a lookup is performed to access the handle for the TLB trie entry corresponding to the ‘public’ data address `addr_p` (`0x604010`) with VPN `0x604`. Using the trie handle the PFN field (`paddr`) in the entry is reset to `0x44d`, which is the PFN targeted to leak the `secret' data from. The `secret' data is in `addr_s` (`0x604010`), which is mapped to the physical address `0x44d010`. The

payload deployment method is called from the CPU code once the trigger condition is satisfied. The attack is evaluated by running the compiled binary in the GEM5 simulator. As seen in Fig. 5-9, we were successfully able to retrieve the secret data from the process's address space.

5.4 Discussions

5.4.1 Advanced hardware Trojan

The Trojan hardware described in this paper only exploits the decoded address for trigger. To lower the probability of accidental triggering further, a simple logic circuit can be implemented to generate $V(P_{SET})$. input of Fig. 5-2a which outputs logic 1 (1V) only if a specific data pattern (let us say P_{SET}) is sent to the data bus. This guarantees that the trigger capacitor gets discharged if the L1 data doesn't match the trigger data. For example, let us consider that the data bus width is 8-bits. Assume that we take four specific data bits to design the trigger logic e.g., `data[0]`, `data[3]`, `data[4]` and `data[6]`. The logic circuit will output 1 if these bits are asserted except `data[4]` which should be de-asserted (Fig. 5-10). In practice, data bits with low activation probabilities should be used to design the trigger logic to lower the overall probability of assertion unless intended. Note that, even if the Add_{SET} is asserted N_{SET} times during normal/test conditions, the Trojan will not be activated since a specific data pattern is required to assert $V(P_{SET})$. Note that in this case, the adversary has to write P_{SET}.data pattern to Add_{SET} for the first time and then keep reading it to charge the trigger capacitor incrementally. To detect such triggers, the test

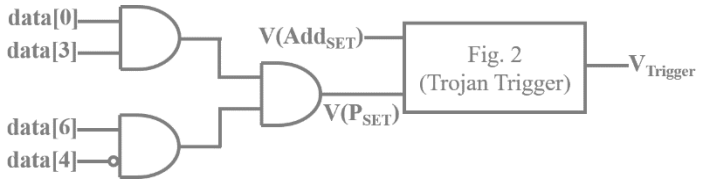


Figure **5-10**: An example of the logic circuit to generate $V(P_{SET})$ from a specific data pattern which serves as an input of the proposed Trojan Trigger.

methodology becomes very complex, since all possible bit combinations must be tried for repeated times for each location of memory. This results in an exponential test-time complexity.

5.4.2 Attack opportunities

Our simple simulated HarTBleed attack can be extending to other possible exploits crafted using the hardware Trojan.

5.4.2.1 Reading data from other processes

In this attack, the goal of the adversary is to extract data from processes that he does not have access to. For example, in a multi-user system, a user (victim) may be running his personal financial application. The adversary, as a second user on the system does not have access to the victim user's financial application, but he wants to gain access to that application process when it is used by the victim user. To design the exploit, the adversary can write his own attack program similar to Fig. 5-8 with the knowledge of the Trojan trigger and payload specifications. The adversary's attack program and the finance

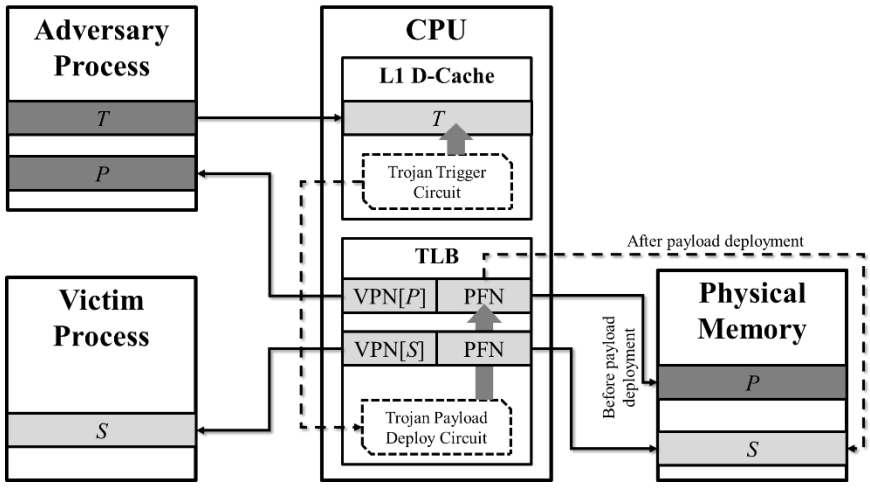


Figure **5-11**: Using HarTBleed to leak data from other processes. T and P is in adversary's process, while S is in the victim process. Depending on the memory segment (stack, heap, mmap, kernel) where the page of S is located, the TLB entry of P can be made to point to the frame of S in physical memory to gain access to data from the page of S.

application process pages simultaneously reside in the physical memory. If the Trojan payload PFN maps to the pages of the finance application, then the adversary can trigger the Trojan using his attack program and then force the mapping of one of his own virtual pages to a physical page of the finance application. In this way, he will be able to read out data from address space of the finance application process. This is shown in Fig. 5-11, where S may be in a page located in the stack or heap of the victim process. The trigger address T and the adversary-controlled address P is in the adversary's malicious process.

5.4.2.1 Reading data from shared memory locations

In this attack, the goal of the adversary is to gain access to memory locations which are shared between processes. Assuming ASLR is off, it is possible to know the location

of the shared memory. For example, dynamic linked libraries and files read from disk are mapped to the `mmap` region of the address space. The pages from the `mmap` region are shared between multiple processes. At the design time, the Trojan payload can be designed to map to the pages of the `mmap` region in physical memory. A victim process may request access to a sensitive database file. The request is served by the kernel by mapping the contents of the file to the `mmap` region. The adversary in this case can write his attack program to trigger the Trojan and map one of his own pages to the `mmap` page containing the contents of the database file in physical memory, thereby gaining access to the sensitive data. As shown in Fig. 5-11, S may be residing in the mmap segment. Adversary can use the Trojan to map his P to point S in the TLB.

5.4.2.2 Reading data from system kernel

In this attack, the adversary tries to gain access to sensitive kernel data such as page tables from kernel memory. The kernel space in the virtual memory is mapped to a fixed location in the physical memory. For example, in a 32-bit address space, the higher order 1GB used as the kernel space is mapped exactly to the higher order 1GB in the physical memory. Thus, it is possible to determine the location of kernel data in physical memory. The Trojan payload can be designed to point to the pages corresponding to kernel space, and the adversary can write his attack code and launch the exploit as before. In Fig. 5-11, S may be an address in the kernel space. Although in the figure, S is shown to be in the victim process, it should be noted that the kernel space is shared across all processes since they map to the same region in the physical memory. However, there is high a possibility

that this attack may be thwarted since a user mode process is trying to gain access to data which is only accessible in kernel mode. To circumvent this issue, the Trojan can be made to change or bypass the CPU status register that sets the protection rings or execution modes.

5.4.2.3 Resetting multiple TLB entries

The adversary can be provided with greater control of page frames to leak from the memory by using a hardware Trojan capable of conditionally resetting multiple TLB entries (refer to Section 5.1.4 for hardware details). The Trojan is designed with multiple trigger addresses (T_1, T_2, etc.), and corresponding target victim addresses (S_1, S_2, etc.). Then, depending on the page frame in the physical memory where the sensitive data is mapped (for S_1, S_2, etc.), the adversary can craft his attack by choosing one of his controlled address (P_1, P_2, etc.) for deploying the Trojan payload and triggering the Trojan using the corresponding trigger address. The payload deployment circuit can then choose the required TLB entry based on the trigger address and map to the victim frame. If the adversary is unsure of the location of the sensitive data, he can hit-and-try all the triggers to reset multiple TLB entries at the same time.

5.4.3 Possible issues or limitations

HarTBleed has few limitations that may present challenges to an adversary in launching the exploits.

5.4.3.1 Context switching

We have assumed that the Trojan payload in the TLB remains active while the adversary is carrying out the attack. However, there may be a context switch after the payload is deployed and before the adversary is able to access the sensitive data. During the context switch, if the TLB is flushed, the deployed payload may be removed and later reverted to the original mapping from the corresponding PTE when the adversary's process comes into context. In this case the adversary may not be able to read the secret data. The adversary may have to run his attack multiple times to be successful. This limitation is somewhat alleviated when the TLB uses the ASID fields to cache PTEs of multiple processes at the same time, since the TLB may not be flushed or invalidated as frequently.

5.4.3.2 Detection/faults during attack

Gaining access to kernel data may also prove difficult, as mentioned before. The CPU runs processes in different protection rings or modes of operation. For example, in X86 architecture, kernel runs in ring 0 (highest privilege), while user processes run in ring 3 (least privilege). These protection rings are enforced in hardware using registers, which specify the current mode. This is to ensure that code executing in user mode cannot do something outside its purview, such as, accessing data from kernel pages. Such actions result in a trappable exception being thrown, which terminates the process. These protection rings can be bypassed if a Trojan is designed specifically do that transparently in the hardware.

5.4.3.3 Caching effects

We have assumed a virtually indexed, physically tagged (VIPT) L1 data cache, since that is most commonly used in modern cache architectures [105,106]. Here, the virtual page number of the address is used to index the cache and lookup the data for faster access. Simultaneously, the virtual address is also sent to the TLB for address translation, so that the page can be retrieved from the lower levels of memory hierarchy in case the data is not already present in L1. This is an advantage that works in favor of the Trojan trigger, since it is easier to infer the cache line mapped to the trigger address. However, the L1 cache may already have a valid mapping for the adversary-controlled (public data) address. In such a scenario, even after deploying the Trojan payload to reset the PFN of public data address to map to that of the victim (secret data) address, that mapping may not be used to retrieve the secret data from physical memory, if the adversary-controlled address already has a valid mapping in L1, containing the original data in that address. In this case the adversary may not be able to read the secret data. However, L1 caches are typically very small, which will eventually replace the cache line mapped to the adversary-controlled address with a different mapping to accommodate data from other memory locations. Once that cache line is remapped, accessing the adversary-controlled address will force the use of the maliciously mapped TLB entry, and retrieve the secret data from the lower levels of memory hierarchy (such as the L2 or L3 caches which are physically indexed, or the physical memory). However, if the L1 data cache is not VIPT, it becomes difficult for the adversary to gain access to his controlled addresses. In such cases, the adversary has to infer the virtual-to-physical translation using other means [107-108].

5.4.3.4 Address space randomization

In order to effectively obtain the secret data from a process, the adversary needs to know the location of the data in the address space. However, Address Space Layout Randomization (ASLR) is a commonly deployed technique that randomizes the locations of the different memory segments in the address space. Typical ASLR deployments randomize the memory mapped segment. This can create a challenge for the adversary, since it will be difficult to know the location of the secret data, if the goal is to obtain the data from the `mmap` segment. OS kernels may also enable KASLR [109], which randomizes the pages in the kernel space. ASLR may be circumvented if there is an existing disclosure vulnerability in the process. Sophisticated attacks [110,111] have also been shown to thwart ALSR in modern systems.

Key takeaways

In this chapter, we presented HarTBleed, an attack methodology that uses hardware Trojans to gain access to sensitive data from a process's memory. Following are the key takeaways from this chapter:

- Semiconductor supply chain is distributed, which can lead to supply chain vulnerabilities such as IC hijacking using Hardware Trojans in the design or manufacturing process.
- Hardware Trojans can not only cause malicious functional issues, but also lead to system level data leakage attacks.

- A capacitor-based trojan trigger on an SRAM based L1 cache can be effectively hidden from post-silicon tests.
- A HarTBleed type trojan attack payload is possible that can cause fault injection on memories to leak memory data.

Chapter 6

Protecting the SoC IP: Reverse Engineering Prevention

In the previous chapter we explored hardware compromising using Hardware Trojans due to supply chain vulnerabilities. Semiconductor supply chain is also plagued with IC piracy and intellectual property (IP) theft. Modern SoC architectures such as RISC-V that are reliant on the distributed supply chain can be especially vulnerable to IP theft Hence securing the IP from theft and counterfeiting is an important goal towards trustworthy computing.

Reverse Engineering (RE) is an invasive process of de-laying and optically inspecting the ICs to reconstruct the functionality of the chip. Reverse engineering of intellectual property has become increasingly more efficient with sophisticated imaging and probing techniques. Gate camouflaging (Fig. 6-1) is a well-known technique used to prevent an adversary from deciphering the chip design and stealing the IP. Several flavors of camouflaging have been previously proposed to thwart RE such as, dummy vias and logic obfuscation. However, these techniques are either costly or remain vulnerable to backside probing and sophisticated optical attacks.

In this chapter, we explore a threshold-voltage (VT) based gate camouflaging design to prevent reverse engineering of IPs. The VT-based camouflaged gate can hide six Boolean functionalities. Since the threshold voltage of a transistor is opaque to the

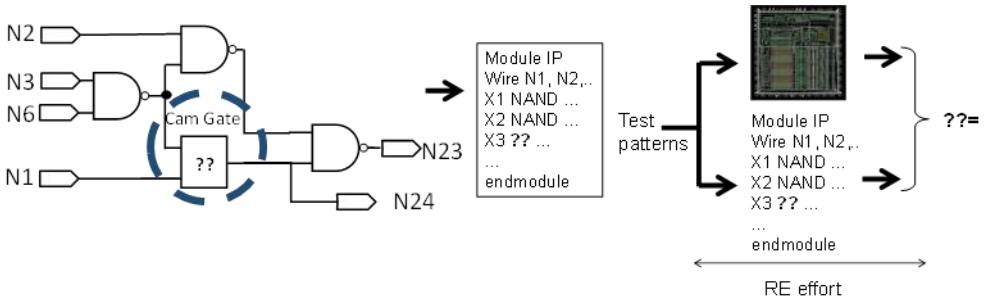


Figure **6-1**: When a gate is camouflaged, the adversary extracts the partial netlist, guesses the missing gate functionality ("??") and applies specific test pattern to match the output against actual chip to confirm the guess. The RE effort is the time invested by adversary to find appropriate test pattern and identify the camouflaged gate functionality.

adversary, it becomes difficult to guess the functionality of the circuit even if the adversary can visually see the layout of the camouflaged gate. Therefore, the adversary is forced to resort to RE-intensive trial-and-error approach. Note that, it is possible for the major adversary to identify each of the transistors VT value through rigorous I-V measurements. However, such effort will require expensive equipment and nanometer-scale probe capability. Therefore, in this paper we assume that the adversary is unable to identify the VT value of transistors. Furthermore, VT modulation (mixing normal VT (NVT), high VT (HVT) and low VT (LVT) transistors in a circuit) is a well-known technique that is extensively utilized [114,115] in semiconductor industries for the trade-off between power, performance, and robustness.

In the SoC manufacturing process, an untrusted fabrication house may be involved. For such scenarios, we further explore a charge-trap based camouflaging technique (implemented by exploiting the gate capacitance of a transistor to store charge) which is impervious to most RE techniques. We first achieve a rudimentary charge trapping by

utilizing two transistors with their gate terminals connected to each other at a node and their source/drain terminals set to a pre-determined voltage. The charge at the node is injected through Fowler Nordheim (FN) tunneling. After injection, the voltages are lowered to prevent de-trapping. Using this trapped charge, we selectively activate/deactivate various functions in a camouflaged gate. To lower overhead and eliminate the possibility of leakage of trapped charges, we also propose an alternative design by replacing the 3-transistor design with a single Non-Volatile Ferroelectric FET (NV-FeFET) [115-118]. The NV-FeFET can be polarized to retain charges in a non-volatile fashion. Therefore, we selectively activate/deactivate the different functions in the camouflaged gate by positively/negatively polarizing the NV-FeFET access transistor. FeFET process is CMOS compatible which makes integration practically feasible [119]. The camouflaged gate is designed using dynamic logic with multiple pull-down networks (PDNs), each of which serves a particular gate function.

6.1 Threshold Voltage Defined Camouflaged Gate

In this section, we present the concept of VT-defined switch. Next, we explore the VT-defined 3-input and multi-input camouflaged gates.

6.1.1 Threshold voltage defined switch

We use the programmable switch proposed in [120] that turns ON/OFF based on VT asserted on it, to implement the camouflaging technique (Fig. 6-2a). The switch is

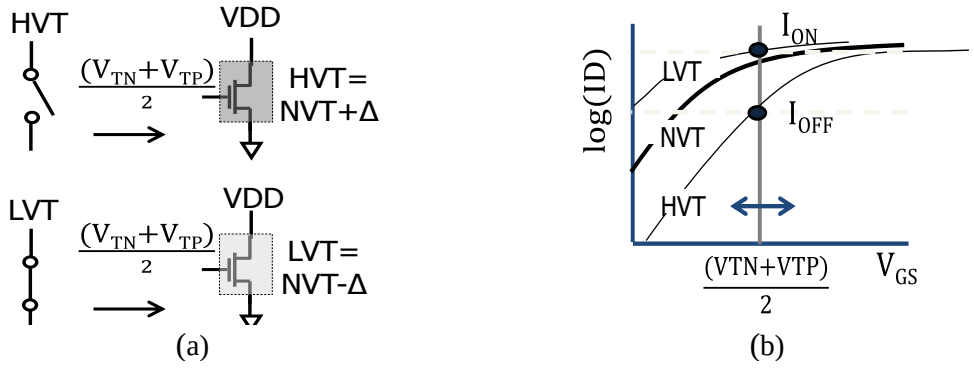


Figure **6-2**: (a) V_T programmable switch. HVT: OFF, LVT: ON. PMOS switch works similarly; and (b) cartoon of I-V curves of NVT, HVT and LV transistors. The I_{ON} and I_{OFF} depends on the LVT and HVT values as well as on gate voltage biasing.

realized by using conventional NMOS transistors with the gate biased at the mid-point between nominal NMOS and PMOS threshold voltages i.e., $0.5(VTN + VTP)$ (Fig. 6-2b). Therefore, the switch conducts when low VT (LVT) is assigned during manufacturing. This is due to the fact that $VGS = 0.5(VTN + VTP) > LVT$. The switch stops conducting when high VT (HVT) is assigned (VGS < HVT). A good VT defined switch should offer high ON current and low OFF current. The gate voltage, HVT, LVT values and transistor sizes are tuned to maximize the ION/IOFF ratio. For NMOS-switch, higher HVT values and lower gate voltage is good for IOFF (leakage) whereas lower LVT and higher gate voltage is good for ION (performance).

6.1.2 Proposed 3-input camouflaged gate

The proposed 3-input camouflaged gate is shown in Fig. 6-3. The design consists of 6 inverters, 4 pass transistors, 2 level restorers and 11 threshold voltage defined switches.

It is a two-stage design with the first stage generating a Boolean output based on the first two inputs, A and B. The third input C is attached to the second stage. The first stage essentially can perform 2-input AND, OR and XOR functions. The second stage is capable of performing 2-input NAND, NOR, XOR and XNOR functions. Combining the two stages in series generates the different 3-input Boolean functions. The threshold voltage defined switches, numbered from S1 to S11 can be selectively set to HVT or LVT to configure the gate to dynamically modify the circuit to behave as a particular function. Table 6-1 shows the six different functions that the camouflaged gate is capable of performing along with the corresponding switches that needs to be turned ON to configure the circuit. The switches are considered to be in an OFF (ON) state when it is set to HVT (LVT).

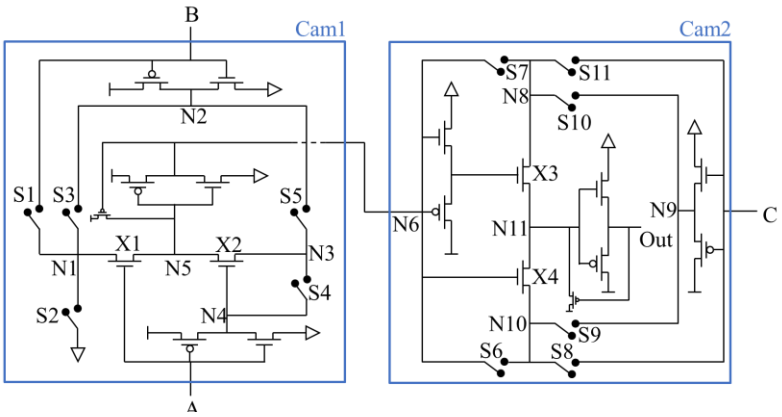


Figure **6-3**: 3-Input 6-Function Camouflaged Gate.

Table **6-1**: 3-Input Camouflaged Gate Functions.

Gate	Function	LVT Switches
NAND	(A·B·C)’	3, 4, 7, 8
NOR	(A+B+C)’	2, 5, 6, 11
AOI	(A·B+C)’	3, 4, 6, 11
OAI	((A+B)·C)’	2, 5, 7, 8
XOR	A ⊕ B ⊕ C	1, 5, 8, 10
XNOR	(A ⊕ B ⊕ C)’	1, 5, 9, 11

The six functions performed by the proposed gate are NAND, NOR, AOI, OAI, XOR and XNOR. All the switches are initially set to HVT. In case of the NAND gate, in the first stage, switches S3 and S4 are set to LVT to turn them ON. The inputs to the first stage are the first two inputs A and B. This makes the first block function as an AND gate performing AB. In the second stage, switched S7 and S8 are set to LVT to turn them ON. The inputs to the second stage are the third input C and the output of the first stage, designated as node N6. This makes the second block function as a two input NAND gate using the aforementioned two inputs. Thus, the final output from the second stage is (ABC)', i.e., NAND on the three inputs A, B and C. The other five functions are generated in a similar manner using the switches as mentioned in Table 6-1.

The logic blocks are designed based on a 2-input pass transistor logic. The inputs are used in normal and complemented form by using CMOS inverters. The outputs in each stage are obtained in complemented form by using additional inverters and level restorers as the output at nodes N5 and N11 do not generate the full swing from 0 to 1. The four pass transistors (X1, X2, X3 and X4) are NMOS transistors set to normal VT.

6.1.3 Proposed multi-input camouflaged gate

The proposed design of the 3-input camouflaged gate can be extended for any number of inputs. In the 3-input camouflaged gate, two flavors of camouflaged logic blocks placed in series, as shown in Fig. 6-4. The first flavor, $Cam1$ is capable of performing three functions: AND, OR and XOR. The second flavor, $Cam2$ is capable of performing four functions: NAND, NOR, XOR and XNOR. Both $Cam1$ and $Cam2$ are two-input gates. By

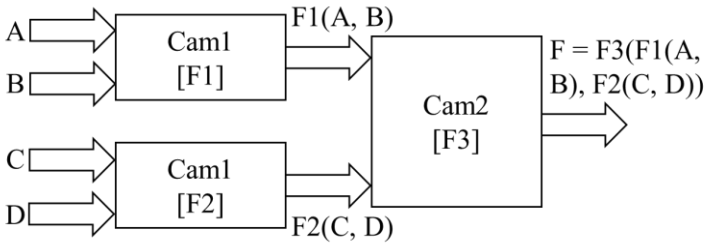


Figure **6-4**: 4-Input Camouflaged Gate formed by Cam1 and Cam2.

using different combinations of $Cam1$ and $Cam2$ type gates, we can design a multi-input camouflaged gate by following the same principle used in the 3-input camouflaged gate design. For example, a four input NAND gate with inputs A, B, C and D can be designed using the following combination of $Cam1$ and $Cam2$ type gates: $Cam2(Cam1(A\ AND\ B)\ NAND\ Cam1(C\ AND\ D)$. In this example one instance of $Cam1$ calculates $O1\ =\ A\ AND\ B$ with inputs A, B and intermediate output $O1$. Another instance of $Cam1$ calculates $O2\ =\ C\ AND\ D$ with inputs C, D and intermediate output $O2$. Finally, one instance of $Cam2$ calculates $O\ =\ O1\ NAND\ O2$ with inputs $O1$, $O2$ and final output O. The same functionality can also be achieved by using the modules in series, such as $Cam2(Cam1(Cam1(A\ AND\ B)\ AND\ C)\ NAND\ D)$, however the former design is preferred because of faster speed due to parallelizing of the $Cam1$ instances. Fig. 6-4 shows a generic example of a 4-input camouflaged gate using $Cam1$ and $Cam2$ instances. The above example of 4-input NAND can be configured in the same structure where functions $F1$ and $F2$ of $Cam1$ evaluates AND, and $F3$ of $Cam2$ evaluates NAND.

Similarly, a 4-input AND-OR-INVERT (AOI22) gate performing $f\ =\ (AB\ +\ CD)'$ can be designed as $Cam2(Cam1(A\ AND\ B)\ NOR\ Cam1(C\ AND\ D))$. The corresponding

switches for the $Cam1$ and $Cam2$ modules need to be set to be LVT to configure the gates to the desired functions.

In general, an n-input camouflaged gate for any of the six Boolean functions can be designed using the following structure of $Cam1$ and $Cam2$ gates:

$$f^N = Cam2\left(Cam1^{\lceil N/2 \rceil}\ op\ Cam1^{\lfloor N/2 \rfloor}\right) \tag{6-1}$$

Where, f^N is a Boolean function of N inputs, and $Cam1^K$ is a recursive representation of a $Cam1$ module of K inputs defined as:

$$Cam1^K = Cam1\left(Cam1^{\lceil K/2 \rceil}\ op\ Cam1^{\lfloor K/2 \rfloor}\right) \tag{6-2}$$

$$Cam1^2 = Cam1(\ I_0\ op\ I_1) \tag{6-3}$$

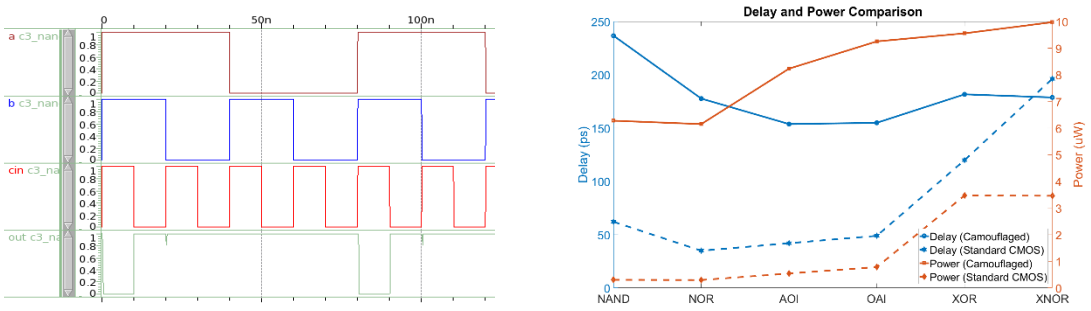
$$Cam1^3 = Cam1(Cam1(\ I_0\ op\ I_1)\ op\ I_2) \tag{6-4}$$

The above technique for a n-input camouflaged gate can be further extended to implement any camouflaged Boolean function of n inputs. Any Boolean function can be expressed in a Sum-of-Products (SOP) or Product-of-Sum (POS) form. To camouflage a n-input Boolean function, the function can be hierarchically represented, as SOP or POS where each node in the hierarchy represents a 2-input Boolean function. These functions can be implemented by using $Cam1$ or $Cam2$ instances. Furthermore, since the $Cam2$ camouflaged block can be configured to behave as 2-input Universal Logic Gates such as, NAND and NOR, it is possible to implement any n-input Boolean function. However, it should be noted that maximum obfuscation is achieved only when each instance of $Cam1$ and $Cam2$ are set to behave differently. The flavors of Boolean functions realized by $Cam1$ and $Cam2$ provide sufficient opportunity to achieve good camouflaging for a multi-input logic function.

6.1.4 Simulation Results

The proposed design has been simulated using HSPICE with 45nm Nangate technology. The normal threshold voltage of NMOS and PMOS transistors are set at 0.623V and -0.587V respectively. HVT and LVT are set as +0.35V and -0.35V over NVT. The inverters are sized with a β-ratio of 2.25, where the NMOS width is chosen as 1.6μm. The NMOS pass transistor widths are set to 3μm. The width of the VT defined switches is set to 3.2μm. Due to the incomplete swings at the outputs of Cam1 and Cam2 blocks, PMOS level restorers are used to generate a complete swing. The level restorers for Cam1 and Cam2 blocks are sized at 0.2um and 0.4μm respectively. The simulations are done assuming an operating temperature of 25C. Fig. 6-5a shows the timing waveform when the circuit is configured as a NAND gate with switches S3, S4, S7 and S8 set to LVT (ON). The calculated delay and power for the six functions of the 3-input camouflaged gate compared to standard 3-input CMOS implementations are shown in Fig. 6-5b. The average delay is 178ps (3.03X overhead compared to standard CMOS) and average dynamic power is 8.282μW (12.33X overhead compared to standard CMOS) for the 6 configurations. The total transistor count for the 3-input camouflaged gate is 29, compared to the standard 3-input CMOS implementations having transistor counts of 6 for NAND, NOR, OAI, AOI (5X overhead), and 30 for XOR and XNOR (no overhead). Since the proposed gate fuses 6 functionalities, the area benefit with respect to cumulative sum of 6 discreet normal gates is approximately 65%.

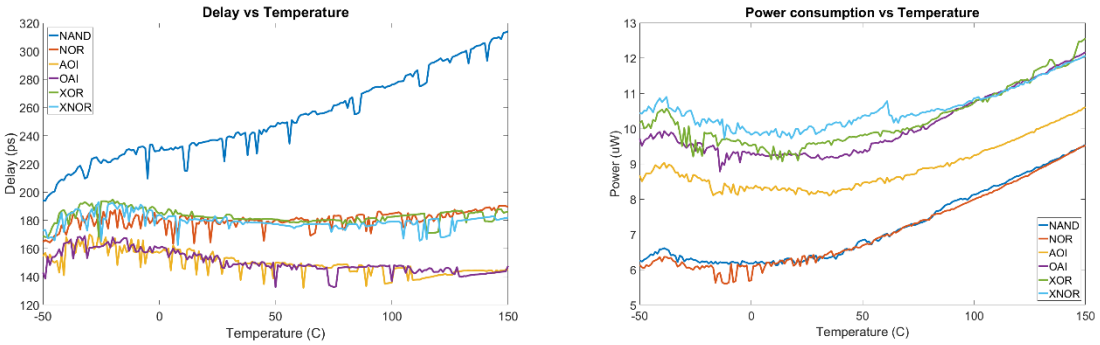
For thermal analysis, we swept the temperature from -25C to 150C, to account for non-normal operating temperature (assuming normal operating temperature to be between



(a) (b)

Figure **6-5**: (a) Waveform for 3-input camouflaged gate configured as NAND; and (b) delay and power comparisons for the 6 camouflaged configurations with standard CMOS gates.

-10C and 90C) that can be exploited by adversary to create side channel. Fig. 6-6a shows the plot for the gate delays with temperature variations for each of the six configurations. The power consumption with temperature variations is shown in Fig. 6-6b. From the graphs, it can be seen than the NAND configuration delay gradually increases with temperature. At higher temperatures, the gate experiences glitches due to higher delays. The remaining five configurations have consistent delays, reaching the highest at approximately -25C. The average delay at high operating temperature (100C) is similar to that at normal temperature (25C). The power profiles are nearly identical for all the six functions, first reaching a peak around -40C, then stabilizing around 20C, and finally increasing steadily above 50C. The average power consumption at 100C is 1.18X higher than at 25C.



(a) (b)

Figure **6-6**: (a) Delay; and (b) power profile with temperature variations of the 3-input camouflaged gate configurations.

6.2 Charge-Trap based Camouflaged Gates

In this section, we describe the charge-trap circuit and the basics of CTCG circuits. We also perform the design space exploration to achieve desired robustness and performance.

6.2.1 Charge-Trap Circuit

Fig. 6-7 illustrates the charge-trap circuit, where the two charge trap transistors (CTA & CTB) are connected to an access transistor (T_X). This is to model a functionality similar to how charges are stored in a non-volatile manner in the gate oxide of a NAND-Flash. We leverage FN Tunneling to charge the node 'P' via gate leakage. Since the oxide of CT_A is very thin, it is susceptible to quantum mechanical tunneling effects. While programming the node, connecting the source and drain of CT_A & CT_B to V_{DD} creates a strong electric field that bends the energy band at the gate oxide making it sufficiently

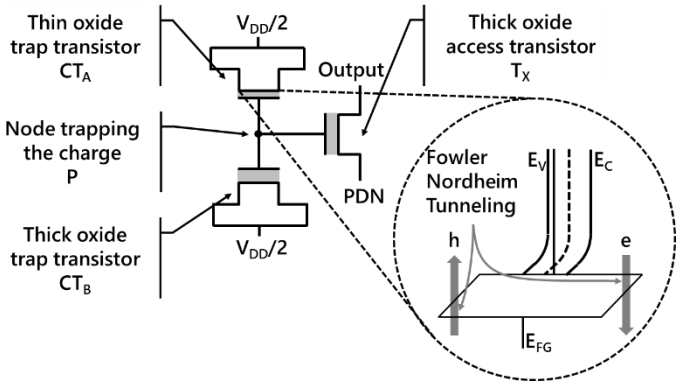


Figure **6-7**: Charge-Trap circuit.

penetrable. In such a situation, electrons and holes can tunnel through the gate oxide and charge up the node to $V_{DD.}$ An 'ON' ('OFF') state is realized at node P when it is charged (discharged) to V_{DD} (gnd). The fundamental idea behind this technique is that T_X is selected (turned ON) only when the trapped charge voltage is greater than its threshold voltage. This is achieved by carefully sizing the transistors CT_A & CT_B, with CT_A having a thin gate-oxide (to allow quicker charging via FN tunneling current), and CT_B having a thick oxide (to minimize gate leakage i.e., high charge retention).

One of the concerns with this methodology is the retention time of the trapped charge. Although it has been shown in past that charge trap circuit data is non-volatile [121,122], the charge stored in the node P can potentially leak over time and pull the node below V_{TH} of T_X. To ensure retention of the charge in the node, we keep the source and drain terminals of CT_A & CT_B biased at a certain voltage. Since we need to make all the charge trap circuits in a gate identical to prevent leakage of information about the gate configuration, the same biasing is also applied to the charge trap circuits of the unselected branches. A potential problem for this is that the unselected nodes can also charge up via

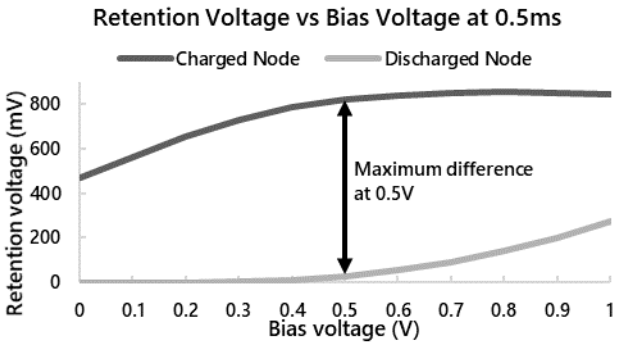


Figure **6-8**: Retention voltage at 0.5V to determine the best bias voltage for non-volatility.

gate leakage due to the biasing. We sweep the bias at CT_A and CT_B to obtain the best bias voltage (Fig. 6-8) that ensures the charged node retains voltage above V_{TH} and the discharged node retains voltage below V_{TH} and observe the 'retention voltage' (voltage at the charged node and the discharged node after 0.5ms). It can be noted from Fig. 6-8 that, for a high bias voltage, the discharged (charged) node begins to charge up (down). We observed that $V_{DD}/2$ provides the best-case retention voltage. Therefore, the analysis of CTCG in this work is performed at bias voltage of $V_{DD}/2$. Additionally, the gate oxide thickness of CT_A, CT_B and T_X can be optimized to further improve the retention voltage. It should be noted that a thicker CT_A gate oxide makes it hard to charge the node while a thinner CT_B/T_X gate oxide increases gate leakage. Further, process optimization can completely eliminate the leakage concerns [121,122].

6.2.2 Camouflaged gate design in dynamic logic

CTCGs are designed using dynamic logic, where the output node is pre-charged to VDD during pre-charge phase and observed at the evaluate phase of the clock (Fig. 6-9). The camouflaging is attained by having multiple PDNs connected to the same pull-up network (PUN) through the proposed charge trap circuits. The drain terminal of the access transistor is connected to the output node and the source terminal is connected to the PDN of that function branch. Identical charge trap circuits control the gate of each access transistor. By doing so, we can set the camouflage gate's functionality before test and shipment of the chip, making it impossible to determine the functional logic using RE methods. A keeper is connected to the output node to prevent charge re-distribution during the evaluate phase. Fig. 6-9a shows the implementation of a 2-input camouflaged gate (CTCG2) capable of performing two functions – NAND2 and NOR2. Depending on the access transistor programming, only one of the PDNs will be active. Fig. 6-9b shows a similar camouflaged gate (CTCG4) design with 3 inputs that provides four functionalities (NAND3, NOR3, AOI21 and OAI21). Although we have shown CTCG2/4 implementation, any complex (inverting) logic of n inputs can be implemented with the proposed approach. The camouflaging logic is implemented in the PDN of dynamic logic to optimize area, hence only inverting logic can be implemented with this approach.

The proposed dynamic logic is scalable and robust to be implemented in a domino chain as shown in Fig. 6-10. Inverters are inserted in between each stage of domino chain to ensure a proper 0-to-1 transition at the inputs for the cascaded stages. Each stage of the domino consists of camouflaged gates hiding different functionalities.

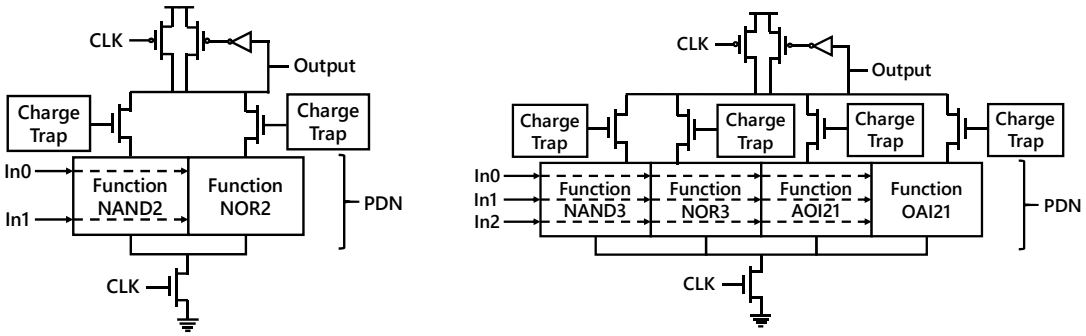


Figure **6-9**: CTCG design flavors: (a) 2-input 2 functions (CTCG2); (b) 3-input 4 functions (CTCG4).

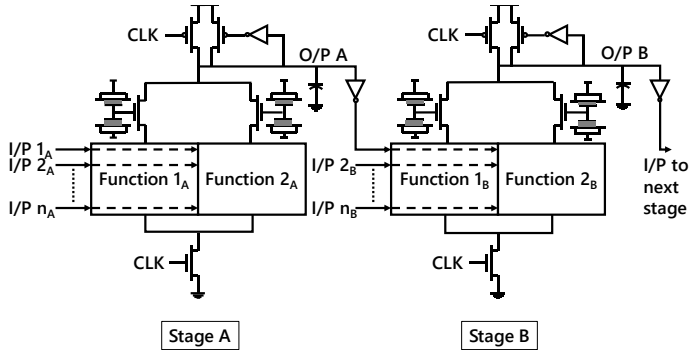


Figure **6-10**: Two back-to-back CTCG gates are shown in domino logic although other regular domino gates could also be cascaded.

6.2.3 Operational Analysis

The standard operational procedure of the proposed camouflaged gate from fabrication house to end-user is shown in Fig. 6-11. During fabrication, a selected set of gates from the circuit is designed using the CTCG versions replacing the standard ones. The choice could be based on metrics such as, controllability and observability [77-78] to maximize the RE effort. This will ensure that certain paths in the circuit are obfuscated due

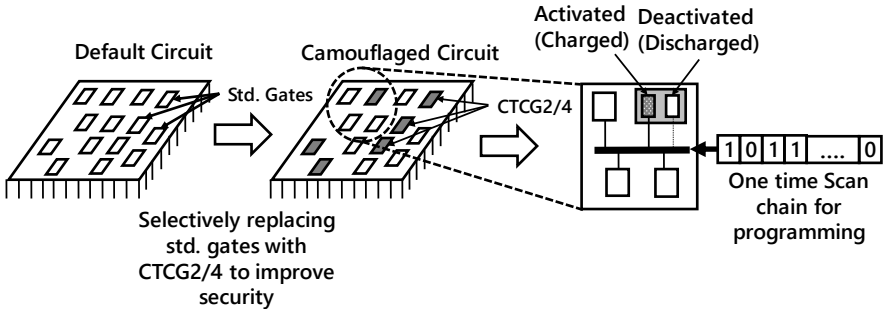


Figure **6-11**: Operational procedure of CTCG.

to the CTCG blocks. Before testing the chip and shipping to the end-user, the CTCG blocks are programmed to enable the desired functionality and disable the undesirable functions. The programming is done by loading appropriate pattern in the scan-chain, where each bit in the bitstream correspond to the program information of a charge trap circuit. The bit (1 or 0) indicates how the charge trap circuits should be programmed (VDD or 0). For a netlist with two CTCG2 gates, the scan chain will contain 4 bits. Pattern 4'b1001 will program the charge trap circuit to enable NAND2 in first CTCG2 gate and NOR2 in second CTCG2 gate. Once the programming is completed, scan chain will be flushed to prevent extraction of CTCG identity information.

The trap circuit is initialized by *charging-up* the trap node of the required function to V_{DD}, thereby activating the corresponding access transistor. The other trap nodes are kept discharged to *gnd*, which ensures their corresponding access transistors are deactivated. Only one out of a total *n* possible functions are active at any given time by this charge trap methodology. These nodes are ideally non-volatile and is expected to retain the preset voltage across power cycles.

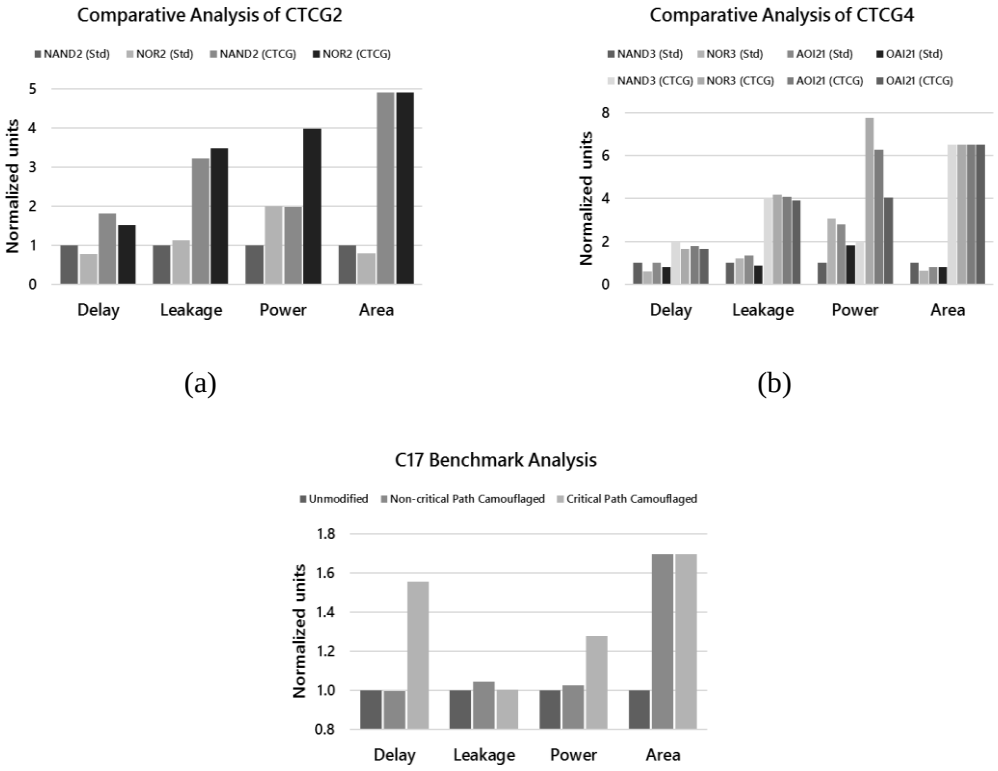


Figure **6-12**: Comparative overhead analysis of (a) CTCG2 and (b) CTCG4; (c) C17 benchmark with single camouflaged gate in critical and off-critical path, respectively.

Although the camouflage gate offers higher security, it comes at the expense of power, delay, leakage, and area. We compare the overheads incurred with this design against standard dynamic logic gates. The proposed CTCG2/4 were simulated in SPICE using 45nm ASU Predictive Technology Models [123]. Fig. 6-12 illustrates the comparison. From Fig. 6-12a-b, it is found that the CTCG versions of the different gate functionalities incur an average delay, leakage, total power, area overhead of 2X, 3.5X, 2.2X, 7.4X respectively. However, it should be noted that since CTCG2/CTCG4 are hiding

2/4 functions together, the cumulative area overhead compared to the standard dynamic (NAND2 + NOR2) and (NAND3 + NOR3 + AOI21 + OAI21) gates is only 2X and 2.7X respectively.

Proposed CTCG functionality are tested by using them in standard dynamic implementation of C17, an ISCAS85 benchmark. We replaced one of the gates of the C17 circuit with CTCG2 on a critical path and a non-critical path. Fig. 6-12c shows the associated overhead compared to the standard implementation. Replacing gates on the critical path incurs a much higher delay penalty (expected), whereas, on the off-critical path, it incurs almost no penalty in terms of delay, leakage, and power.

6.2.4 Process Variation and Temperature analysis

To ensure robustness, the camouflaged gates should be resistant to process variation and temperature (PV & T). We analyze robustness of CTCGs with respect to PV & T below.

6.2.4.1 Temperature analysis

Fig. 6-13a-c shows the impact of temperature on the delay, leakage and total power consumption of the CTCGs. Although the leakage seems to increase considerably, overall leakage power is in the order of nanowatts (not significant to cause functional failures). However, functionality dependent leakage behavior could be potential side channel for RE. It should also be noted that the gate delay degrades significantly at higher temperature

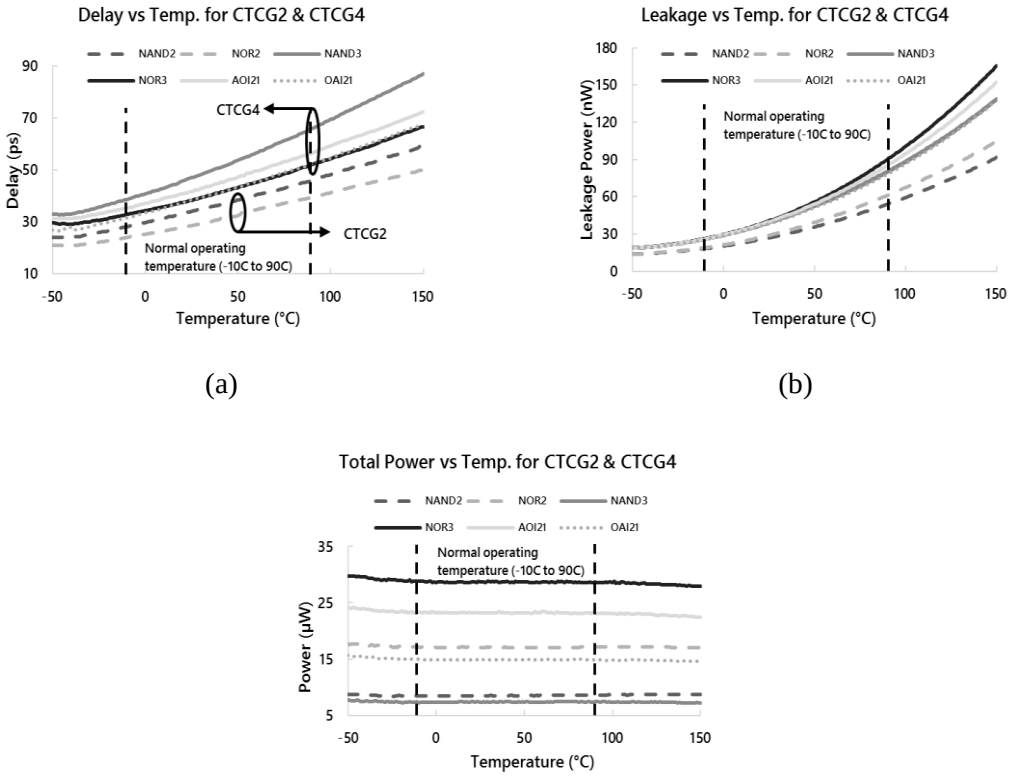
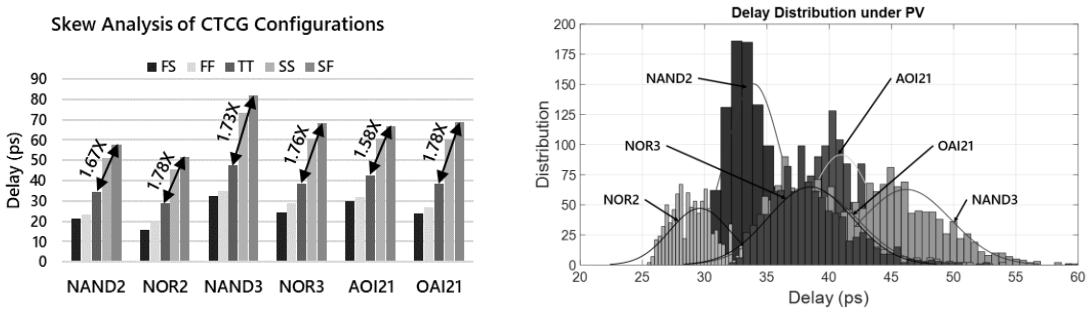


Figure **6-13**: Temperature analysis of CTCG2 and CTCG4.

(1.35X on average). This is due to poor drive current owing to mobility loss at high temperature. The total power consumption of the gates maintains a linear profile with temperature variations.

6.2.4.2 Inter-die process variation

Due to process variation at the wafer level, the transistors are skewed to either typical (T), fast (F) or slow(S) corners. We test the CTCG2/CTCG4 gates against TT, FF, FS, SF and SS corners. The fast corner is modeled by lowering the threshold voltage of the



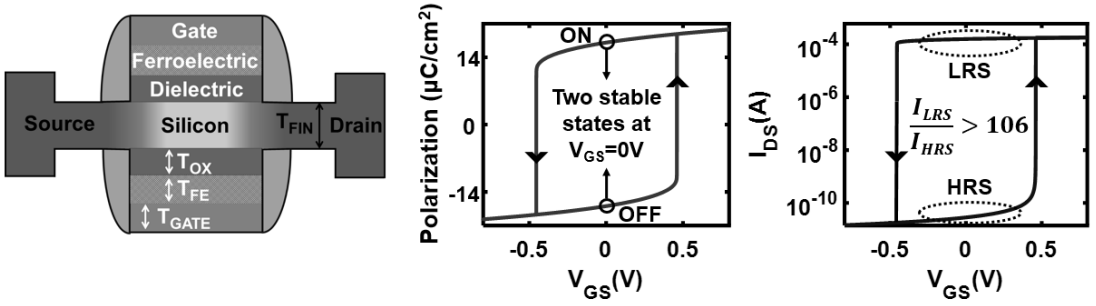
(a) (b)

Figure **6-14**: (a) Process skew analysis of CTCG gate flavors, (b) Delay distribution of CTCG gate flavors under process variations of the access transistors.

transistor by 150mV. The slow corner is modeled by increasing the threshold voltage by 150mV. Fig. 6-14a shows the impact these corners on the performance of CTCG2/CTCG4. The delay is least at the FS corner since the PDN is stronger than the PUN and is highest at the SF corner since the PDN becomes weaker than the PUN. The variations from TT case in the extreme corners are within ~30ps (which amounts to 1.72X increase on average).

6.2.4.3 Intra-die process variation

We study the effects of random process variation on the two proposed camouflage gates. The threshold voltage of the access transistors is modeled using $V_{TH}+\Delta$ where Δ is varied using a Gaussian distribution with mean = 0V and sigma = 50mV. Fig. 6-14b shows the delay distribution for different functionalities of the CTCG2/CTCG4 under V_{TH} variation at each function branch. The delay distribution can be seen to follow a normal distribution.



(a) (b) (c)

Figure **6-15**: (a) FeFET device model; (b) Polarization hysteresis; (c) I_D-V_G characteristics showing non-volatile operation.

6.3 Improved camouflaged gate design with NV-FeFET

The above CTCG circuits, although effective, suffer from a few limitations. The 3-transistor charge trap circuit incurs a significant area overhead. Furthermore, programming the charge trap requires additional on-chip circuitry (area overhead can be significant when designing with large number of camouflage functions). The sizes of the transistors and process also need to be carefully tuned to make the charge trap non-volatile. To overcome these limitations, we explore an alternative design where 3-transistor charge-trap circuit can be replaced with a single NV-FeFET.

6.3.1 Basics of NV-FeFET

FeFETs are designed by depositing a ferroelectric (FE) layer in between the dielectric layer and the metal gate of a field effect transistor as shown in Fig. 6-15a. The capacitance of FE couples with the capacitance of the underlying MOSFET leading to

unique polarization vs gate voltage (P vs V_{GS}) characteristics [116]. The hysteretic behavior of FE is reflected in the P vs V_{GS} characteristics of FeFET (Fig. 6-15b) if the thickness of FE is optimized properly [22]. The low resistance state (LRS) or the ON state of FeFET is achieved by applying a positive V_{GS} such that, the voltage developed across FE (V_{FE}) is greater than the coercive voltage (V_C) of FE in order to switch the polarization from a negative to positive value. The positive value of polarization is retained after removing the applied V_{GS} leading to inversion of channel even at V_{GS} = 0. The high resistance state (HRS) or the OFF state is achieved by applying a negative V_{GS} corresponding to a V_{FE} with magnitude greater than V_C in order to switch the polarization from a positive to negative value. The negative value of polarization is also retained after the removal of the applied V_{GS}. This leads to unique non-volatile operation in FeFETs (Fig. 6-15b) as there exists two possible stable states (LRS and HRS) of the FeFET at V_{GS} = 0V. The current ratio in LRS and HRS of the FeFET is greater than 10^6 (as shown in the I_{DS}-V_{GS} characteristics in Fig. 6-15c), which makes it attractive for non-volatile circuit applications.

We have employed a physics-based NV-FeFET model [118] based on time dependent Landau Khalatnikov (LK) equations [124] which captures the non-volatile behavior of FE. The FE model is calibrated with experiments [118] and integrated with the 10nm FinFET model [123]. The FE parameters used are: $\alpha=-1.05x10^9 m/F$, $\beta=1x10^7 m^5/F/C^2$, $\gamma=6x10^{11} m^9/F/C^4$ *and* $\rho=0.025\Omega$-*m.*

6.3.2 Camouflaged Gate design with NV-FeFET

Replacing the 3-transistor charge-trap with a single NV-FeFET improves the retention capabilities considerably. The programming circuit is also simplified, since now only one connection (to the gate of NV-FeFET) is required for programming using the scan-chain. Fig. 6-16a shows the implementation of CTCG2 using NV-FeFET access transistors. To program the CTCG block, a positive gate voltage ($+V_{DD}$) is applied for a short duration to the T_X for the branch to be activated. This positively polarizes the ferroelectric material in NV-FeFET and lowers the V_{TH}. The positive polarization holds the charges at the gate, thereby keeping the transistor ON. For the remaining branches, a negative gate to source voltage ($-V_{DD}$) is applied to the gate of the T_Xs to negatively polarize the FET. The negative polarization lowers the V_{TH} of the transistors and ensures that it is kept OFF. Subsequently, removing the gate voltages or setting the gate voltages 0V will make the T_Xs retain their ON/OFF state. The ON access transistor determines the actual gate function of the CTCG block. Since polarization of the ferroelectric layer does not have any visible changes, it makes it optically indistinguishable between the activated and the deactivated branches. Furthermore, unlike NAND-flash, FeFET is CMOS process compatible. Hence, incorporating FeFET in the proposed design is practically feasible.

6.3.3 Retention testing of NV-FeFET based CTCG

In order to prove the retention capabilities of the NV-FeFET based CTCG, we first test the CTCG2 (NV-FeFET based) functionality for extended period of time. We monitor the currents at the source terminal of the positively polarized (ON) and negatively polarized

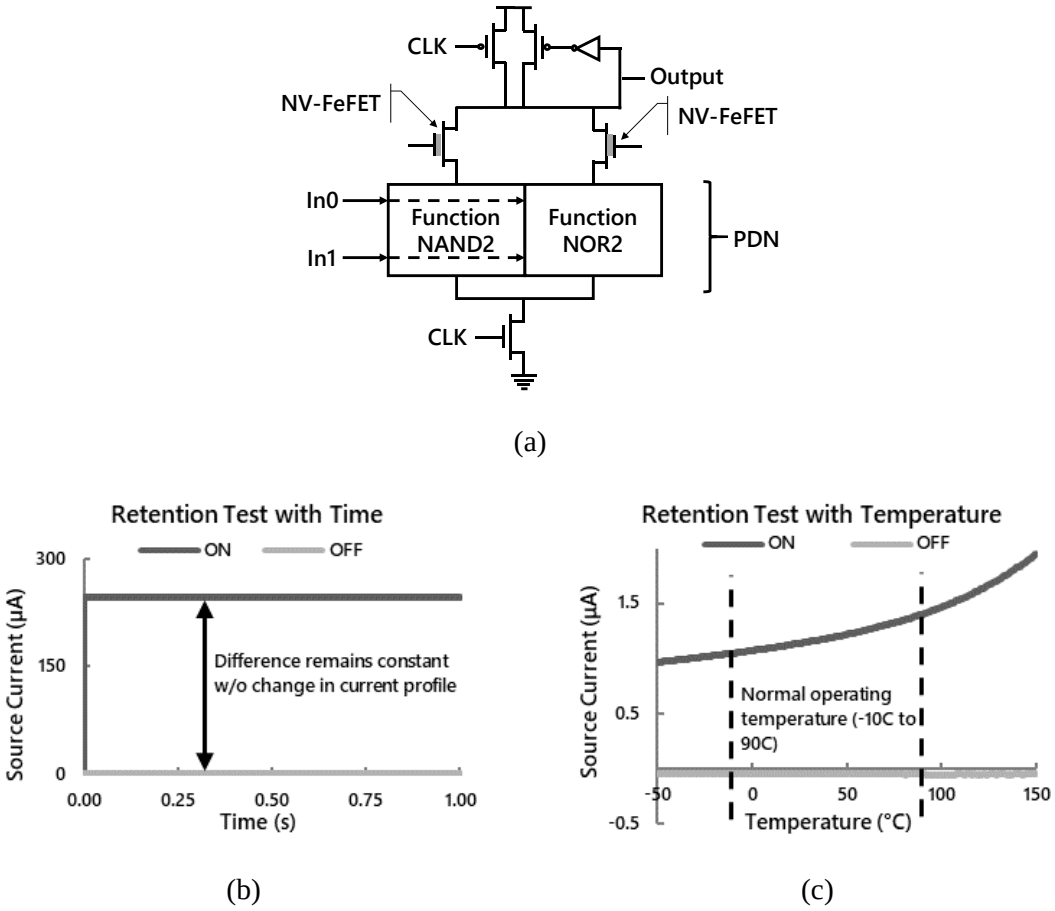
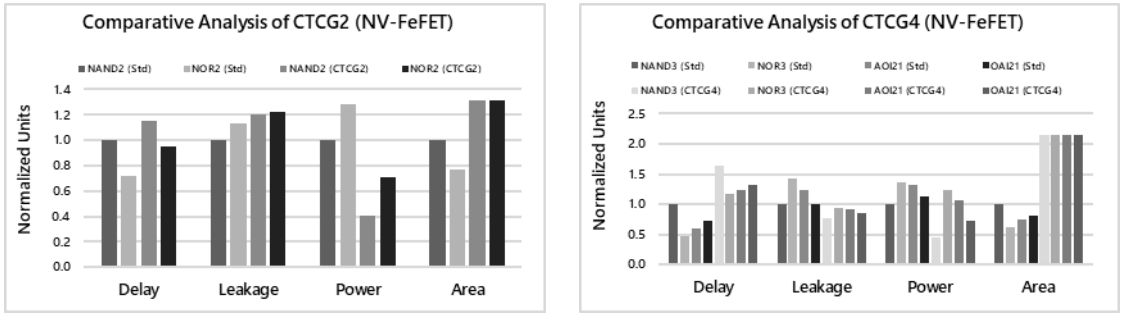


Figure **6-16**: (a) CTCG2 implementation using NV-FeFET access transistors; (b) Retention test for 1 sec; (c) Retention test under temperature.

(OFF) NV-FeFET access transistors. Fig. 6-16b shows the currents at the selected and unselected branch for a simulation runtime of 1 sec. It can be seen that the source current at the ON transistor remains constant at ~250µA. Similarly the source current at the OFF transistor remains constant at ~0A. We can guarrantee the retention capability of the NV-FeFET for extended period of time by extrapolating the source curent profiles.

We also test NV-FeFET retention capability when subjected to temperatures variation. We sweep the temperature from -50C to 150C and monitor the currents at the source terminal of of the positively/negatively polarized (ON/OFF) NV-FeFET. From Fig.



(a) (b)

Figure **6-17**: Overhead analysis of NV-FeFET based (a) CTCG2 and (b) CTCG4.

6-16c we can see that the source current at the OFF transistor remains constant and very close to 0A (only the negligible leakage current is present). The source current at the ON transistor does not decay at higher temperure, rather it increases. The circuit functionality is tested at 150C and found to be correct. Commercial FRAM chip (by Cypress) which stores data with the same mechanism (charge trap) validates the retention to 151/38/10 years @65/75/85C [125] which further confirms our observation.

6.3.4 Comparative analysis

We compare the delay, leakage, power and area overheads incurred with this NV-FeFET based CTCG2 and CTCG4 designs against standard dynamic logic gates. The simulations use 10nm ASU Predictive Technology Models [123] and a physics based 10nm custom calibrated NV-FeFET model [118]. Fig. 12 illustrates the comparison. From Fig. 6-17a-b it is found that the NV-FeFET based CTCG versions of the different gate functionalities incur an average delay, leakage, total power, area overhead of 1.7X, 0.9X, 0.6X, 2.3X respectively.

Table **6-2**: Comparative analysis of camouflaging techniques.

Feature / Technique	**Hollow Via [63]**	**Obfuscation [64-67]**	**MUX Gate [126]**	**V_T Dynamic Gates [127]**	**V_T Static Gates**	**CTCG2**	**CTCG4**	**NV-FeFET CTCG2**	**NV-FeFET CTCG4**
# of functions	3	Varies	2^{2^m} for 2^m:1 MUX	2	6	2	4	2	4
Area overhead	3.06X	1.15X	1.15X	4.25X	10.5X	4.9X	6.51X	1.52X	2.76X
Delay overhead	1.32X	0.99X	1.48X	11.48X	1.51X	1.8X	2.01X	1.24X	1.99X
Power overhead	3.67X	1.16X	N/A	3.83X	8.75X	1.98X	2X	0.48X	0.7X
Static / dynamic	Static	Static	Static	Dynamic	Static	Dynamic	Dynamic	Dynamic	Dynamic
Security weaknesses	Backside probing	Design integrity	Layout traces	Optical attacks	Optical attacks	Retention, thermal tampering	Retention, thermal tampering	Tampering w/ electric field	Tampering w/ electric field

Note: All comparisons are done with respect to a standard NAND gate

Table 6-2 shows a comparison of the proposed CTCG gates with respect to existing camouflaging techniques in terms of overhead, logic style and security vulnerabilities. It can be observed that CTCG gates are better than existing dynamic camouflaged gate from [127]. It also provides competitive overheads compared to static camouflaged gates.

6.3.5 Security Analysis

6.3.5.1 Security guarantees

CTCG: The proposed CTCG2/CTCG4 gates are secure against typical RE practices: i) optical attack: it is difficult to find the functionality since there is no physical trace of the intended functionality on the circuit; ii) design leak during fabrication: it has an advantage over other methods of camouflaging, since the gates will not be tied to a

particular configuration even during fabrication; iii) back-side probing: is also ineffective since the stored charge in trap circuit cannot be identified; iv) side channel: it is also secure against potential side channels since the delay, leakage and power profiles do not have any distinct signature and are very hard to distinguish. Side channel analysis becomes even more unreliable with increasing number of camouflaged gates and normal gates in the circuit (obfuscates the signature).

The adversary in such a scenario will resort to guessing the circuit functionality by brute-force. The brute-force RE effort is exponential in nature. An *n*-stage domino circuit with each stage camouflaging just two functions requires *2n* tries in the worst case, to guess the correct functionality. For Boolean Satisfiability (SAT) attacks, we have found that that reverse engineering effort time increases exponentially with increasing number of camouflaged functions. An ISCAS85 c2760 benchmark circuit with 15% random mux-based camouflaging takes 74456 seconds to solve. Some larger benchmarks such as c5315 and c7512 are unsolvable (running over 10^6 seconds). Moreover, the SAT solver requires the original netlist for generating the I/O pairs which will require additional RE effort time when trying to generate I/O pairs exhaustively from an actual chip. This effort time is not accounted for in the SAT solver.

NV-FeFET based CTCG: Polarization of the ferroelectric layer also does not leave any physical traces which makes it resilient to optical attacks/backside probing. The design will provide the same security guarantees as original CTCG.

6.3.5.2 Security vulnerabilities

CTCG: It must be noted that, in non-ideal scenarios, the rudimentary 3-transistor CTCG may suffer from gate leakage, thus the validity of the function is maintained as long as the active node's voltage is greater than T_x's V_{TH} (opposite must be maintained for inactive nodes). Therefore, a 'charge retention time' is used to quantize the amount of time the trap circuit is operating correctly. Based on our current simulations of CTCG for the worst-case, a retention time of 500µs gives a good voltage difference between active/inactive node, where the active (inactive) node retains a voltage higher (lower) than V_{TH} of T_x. We believe that low retention exhibited in our simulation could be an artifact of modeling inaccuracy of gate leakage in the PTM model. This is likely since gate leakage is not typically critical for circuit functionality. The trapped charge in the trap circuit could potentially be made non-volatile by optimizing the gate leakage through process options.

NV-FeFET based CTCG: The aforementioned security concern regarding the rudimentary 3-transistor charge-trap is mitigated by the use of the high-retention NV-FeFET. Experimental simulation shows that NV-FeFET retains the polarization in a non-volatile fashion. However, the polarization of the FE layer can be affected by strong external electric fields, which can be used by the adversary to hinder circuit functionality or launch Denial-of-Service attacks. Application of electrical shielding can protect the circuit from interference of such external electric fields.

Key takeaways

In this chapter, we presented several gate camouflaging techniques to thwart adversaries from reverse engineering chips and stealing IPs. The following are the key takeaways from this chapter:

- Semiconductor supply chain is distributed, which can lead to supply chain vulnerabilities such as IP theft and IC piracy.
- RISC-V SoCs can be affected with supply chain vulnerabilities since most RISC-V based companies are fabless and need to rely on several third-party vendors and foundries for sourcing different IPs and fabricating the chips.
- Gate camouflaging is an effective way to prevent malicious reverse engineering (RE) efforts.
- Threshold voltage of a transistor can be used as a knob for designing camouflaged gates, which can prevent low-cost optical RE.
- Charge-trap based transistor access control is effective in designing camouflaged gates to prevent RE attempts even when an untrusted foundry is involved.
- Advanced process technologies such as NV-FeFET can be used alongside CMOS processes for designing low-overhead transistor access control.

Chapter 7

Conclusions

In this dissertation, we explored the security viability of RISC-V based systems and supply chain. RISC-V SoC architectures are well suited for low-power resource constrained devices, and as such leave little room for expensive security solutions to common vulnerabilities. Software vulnerabilities are the most common security issues in such systems, and we looked at enabling stack and heap memory protection from buffer overflows using hardware security primitives and dedicated hardware accelerators. Our design emphasizes the core-decoupled implementation, which allows customization and adaptation of the security modules with modification of the primary RISC-V core. To this end, we presented PUFCanary, FIXER and HeapSafe security modules, each of which mitigates a type of memory corruption vulnerability. These modules are designed to be a direct drop-in in the SoC configuration template and can be used with any RISC-V core.

Solutions to software and system vulnerabilities are only secure if the underlying hardware is guaranteed to be secure. Unfortunately, as we saw, due to the distributed nature of the semiconductor supply chain, especially in open architectures such as RISC-V, it is difficult to ensure the trust of the hardware. We explored hardware trojans, a malicious hardware modification that can evade detection and can be used to cause data and information leakage. We presented HarTBleed, a class of data leakage attacks crafted in

conjunction with a capacitor-based hardware trojan, that can be used to maliciously retrieve data from a process's address space. Furthermore, SoCs often contain confidential IPs from different vendors, and it is crucial that such IPs are protected from theft and piracy. Hence, guaranteeing the security and integrity of the hardware is essential towards the goal of trustworthy computing. We explored gate camouflaging, a unique approach to circuit level gate design, that can effectively hide the logic functionality from reverse engineering attempts. We presented threshold-voltage defined camouflaged gates, and CTCG, a non-conventional camouflaged gate design that uses different charge-trap and NV-FeFET processes alongside CMOS circuits. Although we presented these design solutions in light of RISC-V based SoCs, these are equally applicable to other architectures.

Do these solutions guarantee a comprehensive security protection of RISC-V systems? Unfortunately, hardware and software are both developing at a rapid pace, and with it comes new security challenges. RISC-V architecture is constantly evolving and getting more powerful with new features such as vector extensions, making it lucrative for high performance and data-center applications. These will come with new challenges requiring security attention. Machine Learning is already being used to detect network level intrusion in modern systems. However, it will be interesting to explore the application of Machine Learning techniques to detect system vulnerabilities such as control flow violations. One future direction is to use architectural performance counters such as cache metrics, branch predictor metrics and hardware sensor counters such as power and temperature metrics to profile vulnerable programs running normally or under attacks. For real-time detection of such attacks, the RISC-V extensible architecture is already at an

advantage due to its capability to easily create custom hardware accelerators, that can be designed to aid in simultaneous profiling and detection of attacks.

As we see, security design is an ever-evolving area of work. This dissertation, we have provided the foundational concepts to getting started with security designs of SoCs. We have touched upon designing hardware extensions to provide software memory security, as well as approaching hardware design from the ground up keeping security and integrity of the hardware in mind. We believe that these will prove beneficial to future researchers and developers to come up with even more complex and secure SoC designs.

References

[1] J. Deckard, *Buffer overflow attacks: detect, exploit, prevent.* Elsevier, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, pp. 1– 40, 2009.

[3] W. Wu, Y. Chen, X. Xing, and W. Zou, “KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities,” in 28th USENIX Security Symposium (USENIX Security 19), pp. 1187–1204, 2019.

[4] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in Proceedings of the 7th symposium on Operating systems design and implementation, pp. 147–160, 2006.

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” ACM Transactions on Information and System Security (TISSEC), vol. 15, no. 1, pp. 1–34, 2012.

[6] M. Daniel, J. Honoroff, and C. Miller, “Engineering heap overflow exploits with javascript.,” WOOT, vol. 8, pp. 1–6, 2008.

[7] N. Huang, S. Huang, and C. Chang, “Analysis to heap overflow exploit in linux with symbolic execution,” in IOP Conference Series: Earth and Environmental Science, vol. 252, p. 042100, IOP Publishing, 2019.

[8] B. Hawkes, “Attacking the vista heap,” Blackhat USA. (Aug. 2008), 2008.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.,” in USENIX security symposium, vol. 98, pp. 63–78, San Antonio, TX, 1998.

[10] “Data Execution Prevention. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/memory/dataexecution-prevention. Accessed: Feb 26, 2019,” 2018.

[11] “Pax address space layout randomization (ASLR). [Online]. Available: https://pax. grsecurity. net/docs/aslr.txt. Accessed: Feb 26, 2019,” 2003.

[12] Park et al. "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks." *IEEE Micro*, 2006.

[13] Nishiyama et al. "SecureC: control-flow protection against general buffer overflow attack," C*OMPSAC*, 2005.

[14] Sinnadurai et al. "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[15] Zeitouni et al. “ATRIUM: runtime attestation resilient under memory attacks.” ICCAD 2017.

[16] Iwainsky et al. "Compiler Supported Sampling through Minimalistic Instrumentation," ICPPW, 2014.

[17] Pappas et al. "Transparent ROP exploit mitigation using indirect branch tracing." In USENIX SEC, 2013.

[18] Cheng et al. "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack." NDSS Symposium 2014.

[19] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy software,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 27, no. 3, pp. 477–526, 2005.

[20] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, “Cyclone: A type-safe dialect of c,” C/C++ Users Journal, vol. 23, no. 1, pp. 112–139, 2005.

[21] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.,” in USENIX Security Symposium, pp. 51–66, 2009.

[22] G. J. Duck and R. H. Yap, “Heap bounds protection with low fat pointers,” in Proceedings of the 25th International Conference on Compiler Construction, pp. 132–142, 2016.

[23] O. Arias, D. Sullivan, and Y. Jin, “Ha2lloc: Hardware-assisted secure allocator,” in Proceedings of the Hardware and Architectural Support for Security and Privacy, pp. 1–7, 2017.

[24] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, “Secure embedded processing through hardware-assisted run-time monitoring,” in Design, Automation and Test in Europe, pp. 178–183, IEEE, 2005.

[25] Alves et al. "TrustZone: Integrated hardware and software security." ARM white paper, 2004.

[26] McKeen et al. "Innovative instructions and software model for isolated execution." In HASP@ ISCA, 2013.

[27] Intel: Control-Flow Enforcement Technology Review, 2016.

[28] Ramakesavan et al. "Intel memory protection extensions (intel mpx) enabling guide," 2015.

[29] Yoo et al. "Performance evaluation of Intel transactional synchronization extensions for high-performance computing." SC-Intl Conf for HPC, Networking, Storage and Analysis. 2013.

[30] Kasikci et al. "Failure sketching: a technique for automated root cause diagnosis of in-production failures." In SOSP, 2015.

[31] Dhawan et al. "Architectural support for software-defined metadata processing." SIGARCH Computer Arch News, 2015.

[32] Wang et al. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In IEEE S&P, 2010/

[33] Davi et al. "HAFIX: hardware-assisted flow integrity extension." in DAC, 2015.

[34] Jin et al. "Hardware control flow integrity." *The Continuing Arms Race*. ACM and Morgan & Claypool, 2018.

[35] Ge et al. “GRIFFIN: Guarding Control Flows Using Intel Processor Trace” In ASPLOS, 2017.

[36] Song *et al*., "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy (SP), 2016.*

[37] Bresch et al. "A red team blue team approach towards a secure processor design with hardware shadow stack," *IVSW*, 2017.

[38] Bresch et al. "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," in *IEEE ESL*, 2018.

[39] Panis et al. "Scaleable shadow stack for a configurable DSP concept," in *IWSOC, 2003.*

[40] Ming et al. "Shadow Stack Scratch-Pad-Memory for Low Power SoC," in *IEEE Intl Symposium on Embedded Computing*, 2008.

[41] Delshad Tehrani et al. "Nile: A Programmable Monitoring Coprocessor," in *IEEE Computer Architecture Letters*, 2018.

[42] Semiconductor Industry Association (SIA), "Winning the battle against counterfeit semiconductor products," http://www.semiconductors.org/document_library_sia/anti_counterfeiting/sia_whitepaper_winning_the_battle_against_counterfeit_semiconductor_products/, August 2013.

[43] Guin et al, "Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain," PIEEE, 2014.

[44] Dignan L., "Counterfeit chips: A $169 billion tech supply chain headache." http://www.zdnet.com/article/counterfeit-chips-a-169-billion-tech-supply-chain-headache/, April 2012.

[45] DoD Report, "Inquiry into counterfeit electronic parts in the department of defense supply chain." May 2012.

[46] J. Schtte, D. Titze, and J. M. D. Fuentes, “Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps,” in 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 370–379, Sep. 2014.

[47] X. Zhang, F. Liu, T. Chen, and H. Li, “Research and application of the transparent data encryption in intranet data leakage prevention,” in 2009 International Conference on Computational Intelligence and Security, vol. 2, pp. 376–379, Dec 2009.

[48] C. Cao, L. Guan, N. Zhang, N. Gao, J. Lin, B. Luo, P. Liu, J. Xiang, and W. Lou, “Cryptme: Data leakage prevention for unmodified programs on arm devices,” in Research in Attacks, Intrusions, and Defenses (M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, eds.), (Cham), pp. 380–400, Springer International Publishing, 2018.

[49] Z. Ma, “Cpsec dlp: Kernel-level content protection security system of data leakage prevention,” Chinese Journal of Electronics, vol. 26, no. 4, pp. 827–836, 2017.

[50] Y. Wen, J. Zhao, and H. Chen, “Towards thwarting data leakage with memory page access interception,” in 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, pp. 26– 31, Aug 2014.

[51] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, “The matter of heartbleed,” in Proceedings of the 2014 Conference on Internet Measurement Conference, IMC ’14, (New York, NY, USA), pp. 475–488, ACM, 2014.

[52] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in 40th IEEE Symposium on Security and Privacy (S&P’19), 2019.

[53] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in 27th USENIX Security Symposium (USENIX Security 18), 2018.

[54] “Software Techniques for Managing Speculation on AMD Processors [Online]. Available: https://developer.amd.com/wpcontent/resources/managing-speculation-on-amd-processors.pdf. Accessed: Feb 26, 2019,” 2018.

[55] “The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. [Online]. Available: https://bloom.bg/2owldii. Accessed: Oct 28, 2018,” 2018.

[56] “The Hunt for the Kill Switch. [Online]. Available: https://spectrum.ieee.org/semiconductors/design/the-hunt-for-thekill-switch. Accessed: Oct 28, 2018,” 2008.

[57] “Trusted Integrated Circuits (TRUST). [Online]. Available: https://www.darpa.mil/program/trusted-integrated-circuits. Accessed: Oct 28, 2018,” 2018.

[58] B. H. M. Beaumont and T. Newby, “Hardware Trojans-Prevention, Detection, Countermeasures (A Literature Review),” in Technical Report, 2011.
[59] S. Zhu, E. Guo, M. Lu, and A. Yue, “An efficient data leakage prevention framework for semiconductor industry,” in 2016 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), pp. 1866–1869, Dec 2016.
[60] S. Bhunia and M. M. Tehranipoor, The Hardware Trojan War: Attacks, Myths, and Defenses. 2018.
[61] M. Tehranipoor and F. Koushanfar, “A Survey of Hardware Trojan Taxonomy and Detection,” IEEE Design Test of Computers, vol. 27, pp. 10–25, Jan 2010.
[62] M. N. I. Khan, K. Nagarajan, and S. Ghosh, “Hardware Trojans in Emerging Non-Volatile Memories,” in Design, Automation Test in Europe Conference Exhibition (DATE), 2019, 2019.
[63] Imeson, Frank, Ariq Emtenan, Siddharth Garg, and Mahesh 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), pp. 495-510. 2013.
[64] Chakraborty et al. "Hardware protection and authentication through netlist level obfuscation." ICCAD, 2008.
[65] Chakraborty et al. "Security against hardware Trojan attacks using key-based design obfuscation." JET, 2011.
[66] Chakraborty et al. "Security against hardware Trojan through a novel application of design obfuscation." ICCAD, 2009.
[67] Chakraborty et al. "HARPOON: an obfuscation-based SoC design methodology for hardware protection." ICCAD, 2009.
[68] Rostami et al. "A Primer on Hardware Security: Models, Methods, and Metrics." PIEEE, 2014.
[69] Rajendran et al. "Security analysis of logic obfuscation." DAC 2012.
[70] Baumgarten et al. "Preventing IC piracy using reconfigurable logic barriers." IEEE D&T, 2010.
[71] Kahng et al. "Watermarking techniques for intellectual property protection." DAC, 1998.
[72] Rajendran et al, "Security analysis of integrated circuit camouflaging." ACM SIGSAC, 2013.
[73] Imeson et al, "Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation." USENIX Security, 2013.
[74] Cocchi et al, "Building block for a secure CMOS logic cell library." U.S. Patent 8,111,089, issued February 7, 2012.
[75] Chow et al. "Camouflaging a standard cell based integrated circuit." U.S. Patent 8,151,235, issued April 3, 2012.
[76] Baukus et al. "Method and apparatus using silicide layer for protecting integrated circuits from reverse engineering." U.S. Patent 6,117,762, issued September 12, 2000.
[77] Nirmala et al. "A novel threshold voltage defined switch for circuit camouflaging." ETS, 2016.
[78] Jang et al. "Recent Trends in Intellectual Property (IP) Protection from Reverse Engineering." MTV, 2016
[79] Asanovic et. al, “The Rocket Chip Generator”, technical report, www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[80] Bachrach *et al*., "Chisel: Constructing hardware in a Scala embedded language," In *DAC, 2012*.

[81] Nyman et al. “HardScope: Hardening Embedded Systems Against Data-Oriented Attacks”, in DAC 2019.
[82] R. Pappu, “Physical one-way functions," PhD thesis, Massachusetts Institute of Technology, 2001.
[83] www.extremetech.com/extreme/184828-intel-unveils-new-xeon-chip-with-integrated-fpga-touts-20x-performance-boost
[84] Lim et al, "Extracting secret keys from integrated circuits." IEEE Trans. on VLSI Sys., vol 13, no. 10 (2005).
[85] G. E. Suh, S. Devadas, Physical unclonable functions for device authentication and secret key generation," DAC, 2007.
[86] Zheng et al, "ScanPUF: robust ultralow-overhead PUF using scan chain," ASP-DAC, 2013.
[87] Petrie et al, “A noise-based IC RNG for applications in cryptography,” IEEE Trans. Circuits Syst. I, vol. 47, no. 5, pp. 615–621, May 2000.
[88] Sunar et al, “A provably secure TRNG with built-in tolerance to active attacks,” IEEE Trans. Comput., vol. 56, no. 1, pp. 109–119, Jan. 2007.
[89] Brederlow et al, “A low-power TRNG using random telegraph noise of single oxide-traps,” in IEEE ISSCC Dig. Tech. Papers, 2006.
[90] Schellekens et al, “FPGA vendor agnostic TRNG,” in Proc. 16th Int. IEEE Conf. Field Programmable Logic and Applications, 2006.
[91] Kinniment et al, “Design of an on-chip random number generator using metastability,” in Proc. ESSCIRC, 2002, pp. 595–598.
[92] Yasuda et al, “Physical RNG based on MOS structure after soft breakdown,” IEEE J. Solid-State Circuits, vol. 39, no. 8, 2004.
[93] Wilander et al, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention." in *NDSS*. Vol. 3. 2003.
[94] Mathew *et al*., "16.2 A 0.19pJ/b PVT-variation-tolerant hybrid physically unclonable function circuit for 100% stable secure key generation in 22nm CMOS," *in ISSCC*, 2014.
[95] Suzuki et al., "Efficient Fuzzy Extractors Based on Ternary Debiasing Method for Biased Physically Unclonable Functions," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 66, no. 2, pp. 616-629, Feb. 2019.
[96] X. Zhang *et al*., "A novel PUF based on cell error rate distribution of STT-RAM," *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Macau, 2016, pp. 342-347.
[97] Das et al., "MRAM PUF: A Novel Geometry Based Magnetic PUF With Integrated CMOS," in *IEEE Transactions on Nanotechnology*, vol. 14, no. 3, pp. 436-443, May 2015.
[98] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan,“Shakti-t: A risc-v processor with light weight security extensions,” in Proceedings of the Hardware and Architectural Support for Security and Privacy, HASP ’17, (New York, NY, USA), Association for ComputingMachinery, 2017.
[99] S. Das, R. H. Unnithan, A. Menon, C. Rebeiro, and K. Veezhinathan,“Shakti-ms: A risc-v processor for memory safety in c,” in Proceed-ings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, (New York, NY, USA), p. 19–32, Association for Computing Machinery, 2019.
[100] S. Manne, A. Klauser, D. C. Grunwald, and F. Somenzi, “Low powertlb design for high performance microprocessors,” 1997.

[101] N. M. Ravindra and J. Zhao, “Fowler-Nordheim Tunneling in Thin SiO2 Films,”Smart Materials and Structures, vol. 1, pp. 197–201, sep 1992.

[102] M. Chang, P. Rosenfeld, S. Lu, and B. Jacob, “Technology Comparison for Large Last-Level Caches: Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM,” in 2013 IEEE 19th International Symposium on High Performance Computer Architecture(HPCA), pp. 143–154, Feb 2013.

[103] M. Bushnell and V. Agrawal,Essentials of electronic testing for digital,memory and mixed-signal VLSI circuits, vol. 17. Springer Science &Business Media, 2004.

[104] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell,M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,”SIGARCH Comput. Archit. News, vol. 39, pp. 1–7, Aug. 2011.

[105] M. Cekleov and M. Dubois, “Virtual-address caches. part 1: problemsand solutions in uniprocessors,”IEEE Micro, vol. 17, pp. 64–71, Sep.1997.

[106] “Arm Cortex-A77 Core Technical Reference Manual. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0201d/i21752.html. Accessed:Nov22,2019,” 2019.

[107] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson,K. Lai, and O. Mutlu, “Flipping bits in memory without accessingthem: An experimental study of dram disturbance errors,” in2014ACM/IEEE 41st International Symposium on Computer Architecture(ISCA), pp. 361–372, June 2014.

[108] B. Hopkins, J. Shield, and C. North, “Redirecting dram memory pages:Examining the threat of system memory hardware trojans,” in2016IEEE International Symposium on Hardware Oriented Security andTrust (HOST), pp. 197–202, May 2016.

[109] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard,“Kaslr is dead: Long live kaslr,” inEngineering Secure Software andSystems(E. Bodden, M. Payer, and E. Athanasopoulos, eds.), (Cham),pp. 161–176, Springer International Publishing, 2017.

[110] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, “CAIN: Silentlybreaking ASLR in the cloud,” in9th USENIX Workshop on OffensiveTechnologies (WOOT 15), (Washington, D.C.), USENIX Association,2015.

[111] Y. Jang, S. Lee, and T. Kim, “Breaking kernel address space layoutrandomization with intel tsx,” inProceedings of the 2016 ACM SIGSACConference on Computer and Communications Security, CCS ’16, (NewYork, NY, USA), pp. 380–392, ACM, 2016

[112] F. Liu, H. Wu, K. Mai, and R. B. Lee, “Newcache: Secure cachearchitecture thwarting cache side-channel attacks,”IEEE Micro, vol. 36,pp. 8–16, Sep. 2016.

[113] “22nm PTM Technology File [Online]. Available:http://ptm.asu.edu/modelcard/lp/22nmlp.pm. Accessed: Feb10,2019,” 2018.

[114] Hamzaoglu, Fatih, Umut Arslan, Nabhendra Bisnik, Swaroop Ghosh, Manoj B. Lal, Nick Lindert, Mesut Meterelliyoz et al. "A 1 Gb 2 GHz 128 GB/s Bandwidth Embedded DRAM in 22 nm TriGate CMOS Technology." IEEE Journal of Solid-State Circuits 50, no. 1 (2015): 150-157.

[115] Wang, Yih, Umut Arslan, Nabhendra Bisnik, Ruth Brain, Swaroop Ghosh, Fatih Hamzaoglu, Nick Lindert et al. "Retention time optimization for eDRAM in 22nm tri-gate CMOS technology." In 2013 IEEE International Electron Devices Meeting. 2013.

[116] Khan, et al. "Ferroelectric negative capacitance MOSFET: Capacitance tuning & antiferroelectric operation." IEDM, 2011.

[117] Wang et al. "Ferroelectric transistor based non-volatile flip-flop." ISLPED, 2016.

[118] Aziz et al. "Physics-Based Circuit-Compatible SPICE Model for Ferroelectric Transistors." IEEE EDL, June 2016.
[119] Degans H., "Breakthrough in CMOS-compatible ferroelectric memory." https://phys.org/news/2017-06-breakthrough-cmos-compatible-ferroelectric-memory.html.
[120] Iyengar, Anirudh, and Swaroop Ghosh. "Threshold Voltage Defined Switches for Programmable Gates." GOMACTech, 2015.
[121] Wang et al. "A fully logic cmos compatible non-volatile memory for low power IoT applications." IOT, 2015.
[122] Lu et al. "A floating-gate analog memory with bidirectional sigmoid updates in a standard digital process." ISCAS, 2013.
[123] ASU PTM. http://ptm.asu.edu.
[124] Song T. K., "Landau-Khalatnikov simulations for ferroelectric switching in ferroelectric random access memory application." Journal of the Korean Physical Society, January 2005.
[125] FRAM. http://www.cypress.com/products/f-ram-nonvolatile-ferroelectric-ram.
[126] Wang et al. "Secure and Low-Overhead Circuit Obfuscation Technique with Multiplexer." GLSVLSI, 2016
[127] Collantes et al. "Threshold-Dependent Camouflaged Cells to Secure Circuits Against Reverse Engineering Attacks." IEEE Computer Society Annual Symposium on VLSI, 2016

Vita

Asmit De

Asmit De received his Bachelor's degree in Computer Science and Engineering in 2014 from National Institute of Technology Durgapur, India. He worked as a Software Engineer in the enterprise mobile security team at Samsung R&D Institute India for 1 year and 7 months, before joining the Ph.D. program at University of South Florida in 2016. He is currently pursuing his Ph.D. degree in Computer Science and Engineering from The Pennsylvania State University after transferring from USF in 2016.

Asmit's primary research interests include security of systems, architecture, and hardware, and designing energy-efficient and scalable hardware security architectures for system security applications, with a focus on RISC-V based SoC architectures. His research work has culminated in several peer-reviewed journal and conference publications as well as best paper and poster awards. He also facilitated the development of a modular cybersecurity training kit, which was demonstrated to a limited audience in a pilot program.

During the course of his Ph.D., Asmit also had the opportunity to intern twice at SiFive for the 2019 and 2020 summers. He was supporting the platform engineering team in the development of custom memory security modules, interfaces and SoC templates. As part of service to the research community, Asmit has served as a technical reviewer for the IEEE TVLSI journal. Outside of work, Asmit spends most of his time learning about aviation. Asmit's hobbies include flight simulation and photography.