2024-09-16

Towards Reconfigurable Hardware for In-field Hardware Bug Patches

Dharavathu, Anudeep

UNIVERSITY OF CALGARY

Towards Reconfigurable Hardware for In-field Hardware Bug Patches

by

Anudeep Dharavathu

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA
SEPTEMBER, 2024

Abstract

System-on-chip (SoC) designs are becoming increasingly complex, and the ability to detect and address all possible bugs at design time is highly challenging. Thus, to improve the survivability of SoC designs, it is desirable to be able to patch newly discovered design bugs or potential vulnerabilities in the field. Recently, the idea of hardware-based patching, especially of hardware bugs, has emerged as a complementary approach to software/firmware-based post deployment updates. In anticipating potential problems, designers must invest an upfront cost to implement hardware-based patching infrastructures. My thesis investigates the feasibility of incorporating an embedded field-programmable gate array (eFPGA) fabric as an approach to enable hardware-based patching, i.e., reprogrammable hardware to patch hardware bugs, and explores the resource overhead costs for varying patching architectures. We also characterized different eFPGA configurations to help designers decide the eFPGA design parameters.

Preface

Material from Chapters 1 and 4 of this thesis has been published as A. Dharavathu and B. Tan, "Investigating the Feasibility of eFPGA-based Hardware Patching," in the 30th IEEE International Symposium on On-Line Testing and Robust System Design, 2024.

The rest of this thesis is original, unpublished, independent work by the author, Anudeep Dharavathu.

Acknowledgements

I would like to offer my sincere gratitude to my supervisor, Dr. Benjamin Tan, for his guidance throughout my master's studies during the past two years. Through my supervisor's mentoring, I was able to develop insights in Hardware patching.

I would like to thank Semiconductor Research Corporation (SRC) and our SRC liaisons, Alberta Innovates and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their technical feedback and support. I would also like to thank the University of Calgary for providing the financial support and facilities required for my work.

I would like to thank CMC Microsystems and its' volunteers for providing the simulation software tools required for my thesis work.

Finally, I would like to thank my parents and friends for their encouragement.

Table of Contents

Abstract ii

Preface iii

Acknowledgements iv

Table of Contents vi

List of Figures vii

List of Tables viii

List of Symbols, Abbreviations, and Nomenclature ix

1 Introduction 1
- 1.1 Current trends . . . 1
- 1.2 Motivation . . . 1
- 1.3 Research Questions and Objectives . . . 6
- 1.4 Contributions . . . 7
- 1.5 Thesis Layout . . . 7

2 Literature review 8
- 2.1 Overview . . . 8
- 2.2 Background . . . 8
  - 2.2.1 Hardware Patch . . . 9
- 2.3 Prior work . . . 11
  - 2.3.1 Static hardware patching mechanisms . . . 11
  - 2.3.2 Reconfigurable hardware patching mechanisms . . . 15
  - 2.3.3 Observations . . . 17

3 Exploring Hardware bugs 19
- 3.1 Overview . . . 19
- 3.2 Hardware Bugs . . . 19

4 Patching Architectures 27
- 4.1 SoC + eFPGA Architectures for Patching . . . 28
  - 4.1.1 eFPGA-IP . . . 29
  - 4.1.2 eFPGA-DMA . . . 29
  - 4.1.3 eFPGA-REP . . . 29
  - 4.1.4 Patching framework . . . 29
  - 4.1.5 Case study description . . . 31
  - 4.1.6 Patch Architectures and Integration . . . 31
  - 4.1.7 Case Study CWEs and Hardware Bugs . . . 33
- 4.2 Experimental setup and Results . . . 37

5 eFPGA Characterization **40**
5.1 CWE Analysis . . . 41
5.2 Concurrent FSM . . . 45
5.3 Experimental setup and Results . . . 47
5.3.1 eFPGA RTL Generation . . . 47
5.3.2 eFPGA Characterization . . . 48
5.3.3 Concurrent FSM Characterization . . . 49

6 Discussion and Conclusion **51**
6.1 Discussion . . . 51
6.1.1 Answer to RQ1: Hardware Bugs and Vulnerabilities . . . 51
6.1.2 Answer to RQ2: Patching Architectures . . . 51
6.1.3 Answer to RQ3: eFPGA Characterization . . . 52
6.1.4 Key Insights . . . 52
6.2 Conclusions and Future work . . . 53

Bibliography **55**

List of Figures

1.1 An Example SoC . . . 2
1.2 Digital IC Design phases . . . 3

2.1 Hardware Patch . . . 10

3.1 AES Encryption rounds . . . 21
3.2 AES Register Interface . . . 22
3.3 AES Key Exploit Output . . . 23
3.4 AES Exploit Firmware . . . 23
3.5 Lock Bit Modification . . . 24
3.6 Memory Range Overlap . . . 24
3.7 Test Interface with No Access Control . . . 24
3.8 Test Interface Signal Waveform . . . 25

4.1 Patch Location Spectrum . . . 28
4.2 Patch FSM . . . 30
4.3 Three SoC + eFPGA Design Architectures . . . 32
4.4 Access monitoring FSM in an eFPGA . . . 35
4.5 MMIO Implementation in an eFPGA . . . 36
4.6 Access Control bit set by an eFPGA . . . 36
4.7 Design characterization of the three SoC + eFPGA architectures. . . . 38

5.1 Access control Bus Width Mismatch . . . 42
5.2 Exploit for CWE-1220 . . . 42
5.3 Access Control Exploit Output . . . 42
5.4 Patch for CWE-1220 . . . 43
5.5 Patch for CWE-1243 . . . 44
5.6 Concurrent FSM . . . 45
5.7 Interleaved Exploit . . . 46
5.8 Task Configuration file . . . 47
5.9 eFPGA characterization of different CWEs . . . 50

List of Tables

2.1 Prior work comparison. 17

3.1 2021 Top CWE 20
3.2 CWE and SoC location mapping 26

4.1 Patch and eFPGA complexity. 39

5.1 Patch Characterization 48
5.2 Concurrent FSM Fabric Utilization 49

List of Symbols, Abbreviations, and Nomenclature

Symbol	Definition
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
LLM	large language model
NMT	neural machine translation
HLS	High-Level Synthesis
SVA	SystemVerilog Assertion
AST	Abstract Syntax Tree
IP	Intellectual Property blocks
HT	Hardware Trojan
HLL	High-Level Language
RTL	Register-Transfer level
PPA	Power, Performance, and Area
SoC	System-on-Chip
ISA	Instruction Set Architecture
SDL	Secure Development Lifecycle
ROI	Return on Investment
APR	automatic program repair
eFPGA	embedded field-programmable gate array
NoC	Network-on-Chip
DRM	Digital Rights Management
RTL	Register Transfer Level

OTA	over-the-air
I/O	Input/Output
LUT	Lookup Table
FPGA	Field-programmable Gate Array
TOCTOU	time-of-check-time-of-use
IoT	Internet of Things
AHB	Advanced High-performance Bus
AXI	Advanced eXtensible Interface
IC	Integrated Circuit
SDL	Security Development Lifecycle
ASIC	Application-specific Integrated Circuit
CPU	Central Processing Unit
ALU	Arithmetic and Logic Unit
AES	Advanced Encryption Standard
DMA	Direct Memory Access
HMAC	Hash-based Message Authentication Code
CLB	Configurable Logic Block
UCI	Unused Circuit Identification
FRCL	Field-Repairable Control Logic
RSPE	Reconfigurable Security Policy Engine
DfD	Design for Debug
SER	Single Event Radiation
ACW	Access Control Wrapper
MoP	Monitoring and Mitigation Patch
E-IIPS	Extended Infrastructure IP for Security
MMIO	Memory Mapped Input Output
FSM	Finite State Machine
SIG	Special Interest Group
SHA	Secure Hash Algorithm
UART	Universal Asynchronous Receiver Transmitter
JTAG	Joint Test Action Group
ROM	Read Only Memory

LSB	Least Significant Bit
MSB	Most Significant Bit

Chapter 1

Introduction

1.1 Current trends

We live in a world surrounded by billions of computing devices, which identify, track, and analyze some of our intimate personal information, including health, sleep, location, network of friends, etc. The trend is towards an even higher proliferation of such devices. These devices generate, process, and exchange a large amount of sensitive information and data (often collectively referred to as "assets"). The rapid growth of Internet of Things (IoT) has resulted in edge devices accessing and exposing security assets everyday [1]. In addition to private end-user information, assets include security-critical elements, e.g., fuses, cryptographic and Digital Rights Management (DRM) keys, firmware execution flows, and on-chip debug modes introduced during the system architecture design step. Moreover, the devices handling security-critical information can stay long in the field, e.g., a car or a smart home controller can have a decade or more of field life. This is a long time! Security needs can change during this period in response to previously unanticipated use cases, vulnerabilities discovered while the device is in the field [2], and even advances in device technology (e.g., if quantum computers become a reality, then many of the cryptographic protections in the current devices would be compromised [3]). Malicious access to such assets can leak company trade secrets for device manufacturers and cause identity theft or privacy breaches for end users, so they must be protected from unauthorized or malicious access.

1.2 Motivation

Typically, an System-on-Chip (SoC) resides at the heart of these smart devices. SoCs comprise a set of predesigned hardware or software blocks (referred to as Intellectual Property blocks (IP)) that interact either

through a bus protocol like Advanced High-performance Bus (AHB), Advanced eXtensible Interface (AXI) or Network-on-Chip (NoC) communication fabric [4]. The SoC shown in Figure 1.1 has the following IPs: Direct Memory Access (DMA), Universal Asynchronous Receiver Transmitter (UART), Joint Test Action Group (JTAG), Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA), Read Only Memory (ROM) and Hash-based Message Authentication Code (HMAC). AES, SHA and HMAC are cryptographic IPs. Cryptographic IPs are specialized hardware components designed to implement cryptographic functions within a larger system, such as a SoC. These IP blocks are used to ensure the security and privacy of data through encryption, decryption, authentication, and other cryptographic operations. AES [5] IP: Implements AES encryption and decryption algorithms for secure data transmission and storage. SHA [6] IP: Generates hash values for data integrity verification and digital signatures. HMAC [7] IP: Provides message authentication codes using cryptographic hash functions combined with a secret key. The IPs are highly complex design elements optimized for performance, power, and silicon overhead. Adding to the complexity are the communication protocols used in implementing complex system-level use cases. During the last stages of the design process, security assets are sprinkled in different IPs across the design, and complex security measures govern asset access [8]. These measures are defined by system architects and different IP and SoC integration teams and undergo refinement and modification throughout the Security Development Lifecycle (SDL) [9]. Security assets' access restrictions often involve subtle and complex interactions between hardware, firmware, and software associated with these design modules.

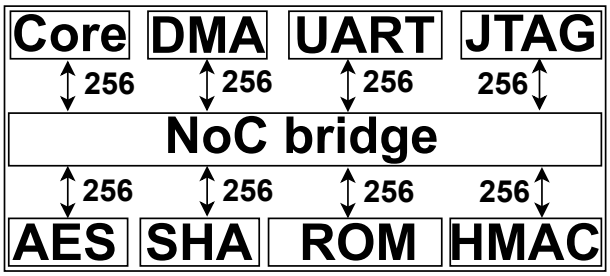


Figure 1.1: An SoC containing IP like AES, DMA, HMAC etc.

In the context of SoC complexity, the design and fabrication of SoC designs are expensive and time-consuming. Any post-manufacturing change requires a new chip design cycle, so the SoCs designs undergo many intricate design phases as shown in Figure 1.2. These phases are in place to minimize the bugs present in the design before being ready for deployment. As part of the design cycle, detecting hardware bugs in SoCs involves various verification and validation techniques [10], including simulation, formal verification, emulation, and prototyping. However, such methods might not achieve complete coverage, especially of security-related bugs [11]. Hardware verification and validation are constantly evolving, and new techniques

are being developed to address these limitations. For example, researchers are exploring the use of machine learning algorithms to improve the efficiency and effectiveness of formal verification [12]. SoCs' complexity and subtle interactions between hardware and software make it challenging to validate a system, develop architectures to provide built-in resilience against unauthorized access, or update security requirements, e.g., in response to changing customer needs.

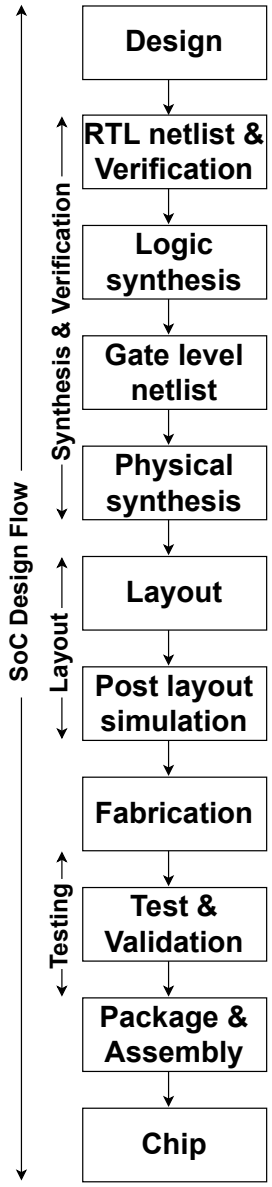


Figure 1.2: Digital IC Design phases. My thesis aims to aid designers in integrating reconfigurable design elements during the design phase and enabling them after fabrication.

Security weaknesses from bugs (vulnerabilities) can arise due to incorrect security specifications, inaccurate translation of design to Register Transfer Level (RTL), human error, and incomplete design verification.

Traditionally, vulnerabilities discovered in the field have been addressed through software and firmware patches. Software bugs can be patched using existing techniques, such as over-the-air (OTA) firmware updates [13]. Furthermore, the ability to reconfigure systems in the field through software and firmware requires that the bulk of the critical functionality of the system be implemented in software/firmware, not hardware. This shift in implementation has significant implications for the power consumption and performance of the system. All other things remaining equal, software implementation typically runs slower than the hardware implementation of the same functionality [14]. For example, a hardware implementation of a cryptographic algorithm might run faster than a software implementation. Consequently, it is not possible to enable systematic software patching at the scale that might be necessary in the course of the extended field life, particularly for devices with security-critical elements (e.g., Smart Home controllers, IoT) as well as devices with high-performance or real-time requirements (e.g., Smartphones, Automobiles). Traditional SoCs have fixed hardware configurations [15], limiting their adaptability to changing requirements or new functionalities.

Implementing digital designs in Field-programmable Gate Array (FPGA) can provide complete flexibility to change the design [16]. A conventional FPGA is a standalone chip that needs to be interfaced with other components of the SoC. The additional packaging and interfacing requirements can result in a larger overall system area. While still highly flexible, conventional FPGAs have a fixed architecture that might not be efficiently tailored to hardware patching applications. Moreover, we may not require all the design components of a conventional FPGA. Conventional FPGAs may require additional security measures to protect against tampering and unauthorized access [17], adding complexity and design overhead costs to the overall system. Alternatively, an embedded field-programmable gate array (eFPGA) can be embedded directly into an SoC, allowing for tight integration with the rest of the system [18]. This improves area efficiency since the eFPGA fabric can be tailored to hardware patching needs and avoid unnecessary overhead. The fabric of an eFPGA can be customized, including the number of Lookup Table (LUT) elements and Input/Output (I/O) blocks. This allows for a more efficient use of resources. Overall, eFPGAs are better suited for hardware patching applications in SoCs.

To improve the survivability of SoC designs and the desire to patch newly discovered design bugs or potential vulnerabilities in the field, prior work [19, 20] proposed hardware-based patching mechanisms to enable patching. These design elements can be used for reconfiguration in post-silicon and in-field. In the field of reconfigurable computing, where systems can be reconfigured and optimized post-fabrication, especially eFPGAs [21] have gained significant traction for their unique ability to provide on-the-fly reconfigurability within Integrated Circuits (ICs). eFPGA integrates the flexibility of FPGA with the compactness and performance of Application-specific Integrated Circuit (ASIC), enabling designers to create hardware that

can adapt to changing requirements and fix bugs without the need for complete redesigns. This hybrid approach leverages the strengths of both programmable and fixed-function hardware.

Hardware patching using eFPGA presents a compelling solution for addressing post-silicon issues and enhancing system security. The ability to reconfigure hardware at a granular level helps extend the life cycle of electronic devices. Moreover, as the complexity of SoCs continues to increase, the traditional approach of firmware/micro-code updates to correct errors becomes increasingly impractical and ineffective [22]. eFPGA-based hardware patching provides an efficient alternative [23], allowing for in-field updates and modifications that can significantly enhance the reliability of electronic systems. While related work [24] used eFPGA for hardware patching, it focused on security policy enforcement on hand-crafted hardware vulnerabilities. Other reconfigurable hardware-based mechanisms for patching [25, 21] are proposed, where hardware components observe and control selected security-relevant signals.

eFPGA can be placed at various locations within an SoC to address various hardware bugs. Potential patching placements include:

- **Central Processing Unit (CPU)**: To patch bugs in pipeline, control, Arithmetic and Logic Unit (ALU) and specific CPU functions [26].
- **Data path**: To modify data processing and secure information transmitted over a bus.
- **I/O**: To adapt to changing I/O protocols or standards and NoC configuration table.
- **IP**: To enable modifications to specific IP blocks [27] and inter-IP interactions.

The placement of the eFPGA determines its flexibility in patching hardware bugs. The designer can take a centralized [27] or decentralized approach [24] when placing an eFPGA in an SoC. The eFPGA location also determines the type of hardware Common Weakness Enumerations (CWEs) [28] it can patch. Hardware CWEs are a standardized list of common security weaknesses in software and hardware maintained by the MITRE [29] Corporation.

There has not yet been a detailed study to characterize the feasibility of different ways to use an eFPGA fabric for patching. While an obvious use for a large eFPGA fabric is to re-implement problematic IPs, we are interested in patching with smaller fabrics. There is not any information on which eFPGA parameters are needed to choose a specific eFPGA fabric size and the investment costs given a fabric size. So the primary focus of my thesis is to explore and demonstrate the potential of integrating and using eFPGA fabrics for patching and provide insights into how much additional hardware-based support needs to be added during the design phase. Several architectural options exist for integrating eFPGAs into SoCs, each with different area and performance trade-offs. We plan to explore some of those potential architectural options. We

focus on upgrades/changes to the IP interface and replacing IPs. Why is that adequate or even interesting? Because most of the in-field updates to the design are in response to vulnerabilities in IPs [25]. Complex patches can also be implemented with one of our proposed design architectures. Bugs inserted in our case study SoC and their patch designs are motivated by Hardware CWEs [28] and characterized by OpenFPGA [30] tool flow. Our case study is based on Open-source SoC [31], and we performed ASIC synthesis on our SoC-eFPGA design architectures. We aim to quantify the cost of a hardware-supported IP bug patching architecture with varying design configurations.

1.3 Research Questions and Objectives

This thesis hypothesizes that it is feasible to use small eFPGAs for patching in SoC designs, offering a flexible and efficient means of addressing hardware bugs while maintaining manageable overhead costs. To explore this hypothesis, we look at three primary research questions:

RQ1: Which design and security bugs inspired by hardware CWEs can be patched?

RQ2: What are the potential eFPGA-based patching architecture options for an SoC given the IPs are accessed through firmware and memory transactions, and what are the overhead costs for area, delay, and latency in each patching scenario?

RQ3: How can a designer select the "best" eFPGA configuration given a set of hardware CWEs that need to be patched?

This thesis aims to investigate the use of eFPGA for hardware patching, focusing on several key objectives:

1. **Analyze the current state of eFPGA-based hardware patching**: Review the existing literature and state-of-the-art implementations of eFPGA-based hardware patching to understand its capabilities, limitations and potential research directions in hardware patching.

2. **Develop a methodology for hardware patching using eFPGA**: Establish a systematic approach for designing and implementing hardware patches, including workflows.

3. **Evaluate the overhead cost of eFPGA-based hardware patches**: Conduct experiments and case studies to assess the impact of inserting eFPGA in an SoC and characterize the area and performance overhead costs.

1.4 Contributions

Our contributions are as follows.

- We analyze selected hardware CWEs and create representative instances. For simulation, we also developed example firmware to exploit them.
- We propose and explore three potential eFPGA-based patching architecture options for an SoC. We characterize the feasibility of using these architectures for patching, showing that our example small eFPGA fabrics as part of a patching infrastructure add reasonable area overhead, ranging from 4.78% to 56.17%.
- We perform a case study using an open-source SoC design and security bugs inspired by a set of weaknesses from the Hardware CWE database [28]. We characterized patches for different CWEs, generated respective fabric sizes and synthesized them to create a database for area and critical path delay overhead estimation. We also developed patches using concurrent FSMs to mitigate interleaved exploits and to improve fabric utilization.

Material from Chapters 1 and 4 of this thesis has been published as A. Dharavathu and B. Tan, "Investigating the Feasibility of eFPGA-based Hardware Patching," in the 30th IEEE International Symposium on On-Line Testing and Robust System Design, 2024.

1.5 Thesis Layout

The remainder of this thesis is structured as follows. Chapter 2 explains hardware patching and reviews existing research on hardware-based patches. Chapter 3 explores hardware bugs motivated by hardware CWEs and directs towards developing effective mitigation architectures. Chapter 4 presents a detailed description of the patching architectures, Hardware CWEs description and the case study with OpenPiton based SoC and OpenFPGA based eFPGA. It also details the patching architectures' experimental setup and discusses design overhead costs. Each patching architecture is characterized by its area, delay, and latency parameters. Chapter 5 describes the eFPGA characterization, where a case study is performed to find out which eFPGA and patch characteristics help decide an eFPGA size given a patch. It analyses the eFPGA characterization results. Patch characteristics, area, and delay overhead costs are modeled as a function of eFPGA size. Chapter 6 discusses the findings of Chapter 4, Chapter 5 and suggests directions for future work, and concludes the thesis.

Chapter 2

Literature review

2.1 Overview

In this chapter, we provide the motivation for hardware patching, what hardware patching means, and ways to implement patching. We also categorize the related work, describe the relevant work and its limitations, and compare our contributions with prior work.

2.2 Background

Recently, industry and academia have proposed hardware CWEs [28] as a taxonomy of potential design weaknesses/bugs that can lead to security vulnerabilities not specific to a specific architecture. Detection and mitigation of the many different CWEs in the field remains an open problem. We studied the CWEs to gain insights into common vulnerabilities encountered in hardware design and the ways to eliminate and mitigate them. The efforts on bug detection and mitigation continue even today and still remain an open problem. We specifically study the weaknesses inadvertently introduced during the design phase or during implementation, focusing on weaknesses with direct security impacts. For example, consider *CWE-1191: On-Chip Debug and Test Interface With Improper Access Control*. Registers corresponding to a particular IP are accessed with a Memory Mapped Input Output (MMIO) address. Some of these registers store encryption keys. When a designer incorrectly implements an access control mechanism in an SoC, encryption keys could be read by an unintended (malicious) user. This issue could be partially solved by restricting the software access to secure memory locations, but privileged software could still exploit the bug.

Integrators can try to patch such issues with firmware or software updates. However, depending on the bug's severity, device recall is sometimes unavoidable [32]. Firmware fixes can be limited in their capability

to fix hardware bugs, especially if there is the risk of compromised (privileged) software; hardware-based patching as a fall-back or fail-safe could be beneficial. Hardware patching can extend the useful life of an SoC by allowing design flexibility and upgradability. By allowing for post-deployment updates, hardware patching can reduce the pressure to perfect the design before release, thus shortening the time-to-market for new products. As explored in our case study (subsection 4.1.5), *CWE-1191* and other CWEs might be addressed by hardware-based patching, as we explain next. We expect post-deployment vulnerabilities to be variants of existing hardware CWEs. We assume the introduced design elements, such as an eFPGA, will be sufficient to patch unforeseen vulnerabilities.

2.2.1 Hardware Patch

What is hardware patching? The ability to change design elements post-deployment, i.e., a *patch*, can help to address vulnerabilities. Patching can be done in response to design bugs found during post-silicon validation, for configuring boot time settings, changing access control policies, or adapting to new security requirements in the field. The latter could arise due to vulnerabilities detected in the field or the deployment of the SoC in different use case scenarios. The hardware patch may involve adding new design elements or manipulating existing designs in the SoC. These could indirectly require extracting more security-critical events from various IP blocks and/or setting additional proactive controls than those considered at design time. It could also be re-configuration post-deployment to change the originally intended behavior.

Why is patching required? With the ever-rising functional capabilities and associated complexities of a modern-day SoC, access to on-chip security-critical assets is also increasing. Typical use cases in an SoC involve subtle interactions between hardware logic, software, and/or firmware of the underlying IP blocks and other SoC components. As industries' attitude is quicker design to deployment [33], only a reasonable amount of time can be invested for verifying and testing the SoCs, so bug-free designs may not be guaranteed. Additionally, the cross-layer interactions and SoC design complexity make it difficult to achieve 100% test coverage. Hence, a key requirement in SoC designs is the flexibility to change the underlying hardware logic rather than just a software patch/upgrade [8]. This requirement to have the flexibility to change design elements post-deployment necessitates the need for patching. To address bugs or attacks detected in the field, new design elements may require to be inserted, or existing ones need to be upgraded or changed. Patching may require detecting new events (outside what had been considered in the design phase), extracting more event information, and/or controlling the IP functionality in response to a trigger. A key challenge arises in detecting bugs and implementing patching when in-field. Our approach to handling this challenge is guided by our industry liaisons, prior work (discussed in section 2.3), and CWE database [28].

A typical approach for implementing hardware patching so that it is reconfigurable is a microcontroller-based implementation where reconfigurable firmware is stored in a non-volatile instruction memory [21]. This firmware would be upgraded based on secure authentication via on-chip keys. However, an alternative approach would be using a eFPGA module [24]. An eFPGA, which constitutes reconfigurable logic and interconnect fabric, would also meet the requirement of the upgradeability feature. The modified configuration bitstream would be uploaded to the internal eFPGA non-volatile memory during secure boot. A simple patching implementation is shown in Figure 2.1, where a programmable input implemented in an eFPGA would aid in modifying the behavior of the circuit. If the AND gate controls the access to the IP, it is connected to, then the programmable input can change the access control in the field. We shall see concrete/specific examples of patching in Chapter 3. We intend to use an eFPGA to implement patching.

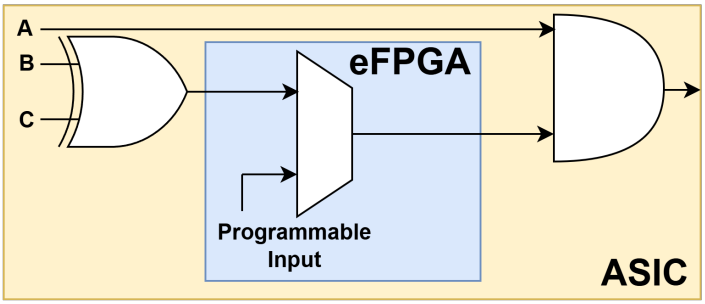


Figure 2.1: Hardware Patch

An eFPGA might be better suited, especially for implementing time-sensitive policies like time-of-check-time-of-use (TOCTOU) [34], liveness policies, etc., due to an eFPGA's normally higher operation speed than a micro-controller. Besides, as the underlying constituent hardware logic and their interconnections are changed during the reconfiguration of an eFPGA, they are more secure from design reverse-engineering-based attacks [35], which an adversary could use to gain knowledge of the IP (proprietary to the IP design house) if possible. This change in the constituent hardware logic of an eFPGA may also aid in complex hardware-level patches, which can even replace a whole IP.

An eFPGA in SoC can either be integrated at the NoC level, interface level, or IP level, which provides reconfigurability, flexibility, and adaptability (some of the requirements of being a hardware patch). The eFPGA can be configured to extract additional (to what was considered during design) internal security-critical events and/or set/disable appropriate controls to govern its functionality, depending on the system state. There are many signals to consider in a modern-day SoC. We rely on hardware CWE description in choosing which signals to monitor/control. We aim to provide in-field programmability to observe and control selected security-relevant signals through dedicated patching hardware implemented on an eFPGA.

2.3 Prior work

Prior work has investigated hardware monitoring mechanisms for specific triggering conditions and responding to a violation by overriding signals, specifically in the context of a processor's internal signals [26, 36, 37]. Other related works like E-IIPS [21] and RSPE [24] can implement security primitives. [25] and [38] consider black-box IPs and patch bugs. The prior work can be classified into two categories based on their capability and implementation. 1) Static hardware patching mechanisms 2) Reconfigurable hardware patching mechanisms. Static mechanisms know what to patch specifically, and once implemented, no active case-specific monitoring changes are required. A static approach enforces a predefined set of checks and corrections, whereas a dynamic patch infrastructure has the flexibility to change what it monitors and corrects, given the relevant signals are made accessible to the patch. In both cases, designers need to consider the programmability, a patching infrastructure offers.

2.3.1 Static hardware patching mechanisms

Blue Chip

Blue chip [39] has two components. They are the design time component and the run time component. The design time component removes erroneous hardware using a Unused Circuit Identification (UCI) algorithm, and the run time component uses software to emulate the missing instructions from the removed hardware. Blue chip replaces the buggy hardware with an exception handling circuit, which gets triggered in the event of a bug detection. The threat model is that a rogue designer adds buggy hardware during the design phase. The Blue chip assumes that the software is trustworthy and non-malicious. In the event of an exception caused by a single instruction, the blue-chip flushes the pipeline and hands over the control to the blue-chip software. For instance, If the hardware implementation of the "MUL" instruction is removed, then the software emulates the "MUL" instruction using "SHIFT" and "AND" instructions.

Implementation: BlueChip uses UCI to detect suspicious circuitry and replaces it with exception generation hardware. If such a circuit is triggered during runtime, the exception handler software emulates the hardware's intended function to maintain system integrity.

Limitations: There will be an implementation fault when the patching instruction depends on the removed hardware, causing recursive exceptions. For instance, when shadow mode [40] attack is patched, the "STORE" instruction is removed, but "STORE" is a fundamental instruction. Further runs of code that used the "STORE" instruction caused a recursive exception. This emphasizes the need to ensure the basic instructions are bug-free.

DIVA

DIVA [26] is "A Dynamic Approach to Microprocessor Verification". Deep submicron process nodes deteriorate the signal integrity, introduce cross-talk, and reduce voltage margins, which affect a microprocessor's reliability. Also, it is intractable to verify a modern-day processor due to its complexity. DIVA proposes a dynamic verification methodology in which a checker unit ensures the processor is always reliable. It contains a DIVA core and a DIVA checker. DIVA core is inserted before the commit stage of a pipelined processor, and the checker unit verifies whether an instruction is correct and finally commits it. The advantages of DIVA are that it eliminates the need to thoroughly verify the core of an SoC as long as the checker unit is always reliable. Circuits other than the DIVA checker can be made on the latest high-performance and low-power process node without worrying about Single Event Radiation (SER) faults as long as the checker unit is fabricated on a reliable process node. Voltage and timing margins on the core can be tightened for optimal performance without reliability concerns.

Implementation: The DIVA architecture divides the processor into a speculative core and a robust checker. The core performs speculative execution, while the checker re-executes program instructions to verify their correctness. The checker ensures correct computation, communication, control, and forward progress of instructions. Faults detected by the checker are corrected, and the processor state is restored using the existing speculation recovery mechanism.

Limitations: The DIVA checker needs to perform the same computation as the DVIA core to verify the result before committing. This redundant computation incurs latency, thereby affecting the performance of heavy-load applications. Structural hazards and exceptions due to the DIVA checker impede the normal functioning of the processor.

Final filter

Final Filter [36] is a run time verification tool that detects security violations post-deployment. It can monitor violations such as "the processor going into superuser mode after a software exception". Normal functioning of the SoC is impeded only in the event of a violation or bug. It only serves as a verification/monitoring tool but cannot fix or workaround an issue, so it must be used in conjunction with other techniques, such as DIVA [26]. Final Filter uses assertions hard-coded into the design to verify the correctness of execution runs. The system is reconfigurable, allowing updates to monitored properties post-deployment.

Implementation: Final Filter monitors the processor's state and events, focusing on privileged instruction set architecture (ISA)-level registers. It consists of several components, including routing blocks, logic blocks, assert blocks, and a merge block. The configuration data provided by trusted software determines

which state elements are monitored and how assertions are defined.

Limitations: The number of signals routed to the final filter limits the coverage metric, as routing cannot be changed post-deployment in light of new vulnerabilities. Final filter is not a standalone solution, so the hardware area overhead cost includes the area occupied by the final filter and the DIVA (if used).

FRCL

Field-Repairable Control Logic (FRCL) [37] addresses the challenge of functional correctness in microprocessors, which are often released with latent bugs due to complex architectures and inadequate verification tools. FRCL is a hardware-patching mechanism designed for in-the-field correction of control logic errors. The FRCL employs a programmable state matcher that identifies erroneous configurations and switches the processor to a formally verified degraded mode capable of executing the full instruction set architecture. This allows for correcting multiple design errors with minimal performance and area impact.

Implementation: Bug signature describing the erroneous control state is generated and sent to customers as a patch and loaded into the state matcher. The matcher monitors the control state and switches to degraded mode if a bug is detected. The processor runs in degraded mode to bypass the error and then returns to high-performance mode.

Limitations: FRCL is confined to patching control bugs, while instruction and microcode patching have broader applicability in addition to control-related bugs. With inadequate selection of critical control signals and insufficient matcher size, FRCL can miss bug detection.

CASPAR

CASPAR [20] is a patching-based runtime validation solution for multi-core processors. It uses checkpointing to log system states and employs programmable hardware detectors to identify errors and trigger recovery. The recovery mechanism operates in a reduced-complexity mode to ensure memory coherence and consistency.

Implementation: CASPAR consists of three main components: Checkpointing System: Periodically records the system's state using a distributed log mechanism. Event Detectors: Programmable circuits that identify errors by comparing cache transitions to known error patterns. Recovery and Bypass Mechanism: Restores the system to a known good state and operates in a mode that prevents further errors by enforcing strict ordering of memory accesses.

Limitations: CASPAR relies on programmable detectors that match state-machine transitions to predefined error patterns. This approach can only detect bugs that can be represented through these transitions.

It is ineffective against errors caused by invalid request timing or those that cannot be captured by simple state changes.

AKER

AKER [41] is a design and verification framework built upon the Access Control Wrapper (ACW), which dynamically manages access to shared resources. It provides property-driven security verification using CWEs to ensure the system is secure and functions correctly.

Implementation: AKER integrates ACWs throughout the SoC, wrapping controller IP cores to enforce local access control policies. The AKER framework is implemented on a Xilinx UltraScale+ programmable SoC, integrated with the OpenTitan [42] hardware root-of-trust.

Limitations: The access control system within a particular design cannot be dynamically modified. This means that once the controllers' addressable regions are set, no additional custom regions can be added, and predefined regions have default read/write permissions without the flexibility to define read-only regions.

Phoenix

Phoenix [32] is a field-programmable system that detects and fixes design defects. It enables hardware patching analogous to software patches. Phoenix monitors control signals to detect concurrent defects. Concurrent defects are triggered by a relatively simple combination of signals and events. These concurrent defects are further classified into pre and post-defects. Pre-defects can be detected before the occurrence of damage, whereas post-defects cannot be detected in advance. When a pre-defect is observed, Phoenix flushes the pipeline and temporarily disables the signal that caused this defect, thereby evading the issue. In the event of a post-defect, a rollback feature is used.

Implementation: Detection: Phoenix taps into key control signals and uses downloaded defect signatures to detect bugs. Recovery: Upon detecting a defect, Phoenix can flush the pipeline and retry the operation or invoke a customized recovery handler. Phoenix hardware includes Signal Selection Units (SSU), Bug Detection Units (BDU), and a Global Recovery Unit. These components work together to monitor, detect, and recover from bugs in real-time. The system is designed to be reprogrammable to handle new defects as they are discovered.

Limitations: Phoenix cannot fix complex defects as such events are difficult to monitor and involve intricate triggering conditions. It also cannot detect novel defects in the field unless a monitor is already programmed. This approach requires internal architectural knowledge of the IP to program the monitoring patch, which may sometimes be unavailable.

2.3.2 Reconfigurable hardware patching mechanisms

MoP

This work explores the detection of IP vulnerabilities in SoC designs using hardware-based design elements, referred to as Monitoring and Mitigation Patch (MoP) [25]. MoP can be a finite state machine-based eFPGA (FSM-MoP) or a reprogrammable hardware block (Re-MoP) that is configured to monitor specific signals and activities within the SoC design that are indicative of vulnerabilities. MoP blocks require memory as monitors involve comparing and processing previous and current data. MoPs can also correct signals, including overriding the output data bus or denying access to an IP. For instance, if an AES IP outputs intermediate text before the complete encryption, the MoP block zeroes the output data bus. When an issue is observed post-deployment, the designer can work around it or override the output signals of an IP using MoPs.

Implementation: MoPs are distributed around security-sensitive and security-aware IPs in an SoC and are programmed only during the secure boot to avoid adversarial manipulation by compromised software. The MoP block probes and monitors IP interface level signals to infer the state of the IP, i.e., sleep state, idle state, or active state. It checks whether an IP violates a security policy and verifies the response of an IP with the expected value and the time taken by an IP before the final response is populated on the data bus. SoC components and IPs are synthesized on Intel FPGAs using the Quartus Lite tool.

Limitations: Distributed monitoring processes may introduce additional overhead, potentially increasing the overall overhead cost of the SoC. Inferring IP state and implementing correction actions with access to only the interface level signals using MoPs is challenging as covertly crafted bugs might go undetected.

E-IIPS

Extended Infrastructure IP for Security (E-IIPS) [21] is a centralized, firmware-upgradable module that interfaces with IP blocks via security wrappers. Security wrappers are integrated with IPs. These wrappers detect security-relevant events and communicate with E-IIPS. They extend existing test/debug interfaces like IEEE 1500 boundary scan and ARM's coresight IP. Security policies are programmed as firmware in the E-IIPS, enabling easy updates and management. Efficient communication links are established between security wrappers and the E-IIPS to minimize performance impact.

Implementation: The key components of this approach are IP Security Wrappers: Extract security-critical events and communicate with E-IIPS using a standardized template-based design. Security Policy Controller (E-IIPS): Analyzes events from security wrappers, enforces policies, and updates based on security state. Secure Firmware Updates: Authenticated updates to E-IIPS firmware using challenge-response keys

generated at power-on.

Limitations: The reliance on a centralized E-IIPS means that communication with this controller could become a performance bottleneck, especially in large SoC designs with numerous IP blocks. The effectiveness of the architecture depends on properly implementing security wrappers around each IP block. This requires additional effort from IP providers and may not be straightforward for all types of IPs.

SoC Security Architecture for Hardware Patch

System on Chip Security Architecture proposed by [24] is a hardware security policy engine that can be updated post-deployment to mitigate attacks and implement new security policies. It addresses two critical design requirements: in-field configurability and low overhead.

Implementation: The building blocks of this architecture are Reconfigurable Security Policy Engine (RSPE), Smart security wrappers, Design for Debug (DfD). RSPE: This centralized engine implements diverse SoC security policies and allows seamless, secure upgrades post-silicon. It is reconfigurable and can adapt to new security requirements or unanticipated attacks. Smart Security Wrappers: These wrappers extend existing test and debug interfaces to monitor and control each IP's internal events, enabling the RSPE to enforce security policies. DfD Infrastructure: This interface provides access to relevant signals inside IP blocks, allowing for efficient policy enforcement without interfering with traditional debug functionalities.

Limitations: Integrating the proposed security architecture with existing SoC designs and workflows might require significant changes and adaptations. Experimental validation in this work is based on representative SoC models and a limited set of security policies. Further validation with more diverse and complex real-world scenarios is necessary to fully understand the architecture's effectiveness and limitations. It requires significant effort from the IP designers. The extent of observability is unspecified and ambiguous.

Hardware-Supported Patching of Security Bugs in Hardware IP Blocks

This approach [38] helps designers maximize the benefits of patchability in IPs while adhering to resource overhead constraints. An integer quadratic program guides the design process, achieving superior patchability compared to baseline approaches at a given cost limit.

Implementation: The paper introduces a formal method to evaluate the trade-offs between patchability and cost. This involves setting parameters and constraints for the patching blocks and using a score function to guide the design process. The goal is to ensure that the added hardware for patching does not exceed a specified resource overhead while providing sufficient coverage for potential vulnerabilities.

Limitations: Crafting a patch for bugs arising from complex situations internal to the IP can be challenging due to the coarse-grain view of IP interfaces and the transactions required for a service request.

The PB might only capture services as a simple three-state finite machine (IDLE, BUSY, DONE), limiting its effectiveness in handling more complex issues.

2.3.3 Observations

We observe that a considerable amount of literature has been researched on patching processor bugs. These processor bugs are motivated by vendor errata documentation and handcrafted bugs. No prior work has patched bugs based on hardware CWEs, so we aim to work on developing architectures and conducting a feasibility study on patching hardware CWE bugs. While some prior work focused on the potential use of eFPGAs for patching, the feasibility characterization was preliminary as the area results are estimated by synthesizing for a conventional FPGA. Moreover, the impacts on SoC overhead were largely speculative and based on the synthesis of patching Finite State Machines (FSMs) for a regular FPGA. A formalized quantification of patchability remains an open problem. These gaps motivate our research questions. Our work explores alternate options for integrating and using eFPGA fabrics for patching and provides insight into how much additional hardware-based support needs to be added a priori. Bugs inserted in our case study SoC and their patch designs are motivated by Hardware CWEs and characterized by OpenFPGA tool flow. Our case study is based on Open-source SoC, and we performed ASIC synthesis on our SoC-eFPGA design architectures. Complex patches can also be implemented with one of our proposed design architectures. We aim to quantify the cost of a hardware-supported IP bug patching architecture with varying design configurations.

Approach	**CWE**	**Evaluated on eFPGA**	**Open-source case study**	**Patch Characterization**	**Complex Patches**
E-IIPS [21]	✗	✗	✗	✗	✗
RSPE [24]	✗	✓	✗	✗	✗
MoP [25]	✗	✗	✗	✗	✗
Proposed	✓	✓	✓	✓	✓

Table 2.1: Prior work comparison.

We tabulated our contribution in Table 2.1 compared to our assessment of previous work. The "CWE" column refers to inserted bugs based on Hardware CWE analysis. CWE descriptions are architecture agnostic, so we examined the demonstrated examples to help us insert bugs specific to our case study SoC. We based our research on hardware CWEs as they are representative of real-world bugs. "Evaluated on eFPGA" column refers to ASIC synthesis with SoC + eFPGA RTL. Resource overhead numbers that we tabulated in section 4.2 are not from conventional FPGA synthesis but ASIC synthesis performed with the Synopsys design compiler. We focus on ASIC synthesis because our use case of using eFPGAs is in an ASIC SoC because those are devices that cannot be reconfigured post-fabrication. "Open-source case study" refers to the

open-source SoC and eFPGA RTL we used in our case studies whereas prior work [24] has used a toy-model SoC. We performed our case study on open-source hardware so the comparison of the results would be fair and the results could be reproduced by other researchers if needed. "Patch Characterization" refers to the patches that we generated and processed with the OpenFPGA framework. We characterized the patch's interface width, utilization percentage and the size of eFPGA needed to implement them. Using eFPGA for simple patches like access control changes would be trivial, so a detailed characterization of patches is needed to take full advantage of using eFPGAs. "Complex Patches" refer to the ability to implement patches with functionality that's comparable to that of a standalone IP. One of the patching architectures that we propose in section 4.1 has the ability to implement a whole IP. This capability to implement a whole IP can also be used to implement RSPE [24].

Chapter 3

Exploring Hardware bugs

3.1 Overview

This chapter discusses hardware bugs based on hardware CWEs and their implications. We provide RTL instantiations of buggy code with waveform reviews and firmware exploits.

3.2 Hardware Bugs

Hardware bugs, also known as hardware defects or flaws, are errors or faults in computer hardware's design, manufacturing, or operation [32, 19, 43]. These bugs can manifest in various forms, including:

Design Flaws: Mistakes in the initial design of hardware components. This can include incorrect logic in the circuitry, design oversights, or misinterpretations of specifications, leading to flawed implementation. Malicious actors can exploit to gain unauthorized access or control over a system

Manufacturing Defects: Issues that arise during the production process, such as defects in the materials used, errors in assembly, or problems with the fabrication process. Examples include broken connections, short circuits, or impurities in semiconductor materials.

Functional Errors: Problems that occur when hardware components do not perform as intended. This can include issues like incorrect calculations, timing problems, or unexpected behavior under certain conditions, such as buggy designs, hardware trojans [44] etc. Over time, hardware components can degrade performance due to device aging [45], environmental factors (such as temperature changes), or electromagnetic interference [46] are also responsible for functional errors.

Compatibility Issues: Problems occur when hardware components do not work well together. This can include issues with drivers, firmware, or other software interacting with the hardware.

To understand more about hardware bugs, we studied some hardware CWEs [28] that could emerge within an SoC design and implemented some of the bugs in our case study SoC. Hardware CWEs are a standardized list of common security weaknesses in software and hardware maintained by the MITRE [29] Corporation. These weaknesses are listed by the Hardware CWE Special Interest Group (SIG), a community forum for individuals representing organizations within hardware design, manufacturing, research, and security domains, as well as academia and government. Therefore, they are a good representative of potential real-world bugs. As a case study SoC, we chose the OpenPiton [47] based SoC provided as part of the HACK@DAC 2021 [31] competition, which has a RISC-V ariane core, 3 AES crypto IPs, HMAC, RSA, SHA, JTAG, DMA and UART IPs. Figure 1.1 illustrates a simplified view of the SoC and its IPs. The SoC contains a NoC bridge and uses the AXI4 protocol.

Given the challenge of predicting future security vulnerabilities, one can consider issues in existing products to identify potentially vulnerable targets and valuable assets in a system. We use the hardware CWEs to inform us of potential scenarios where patching might be required. CWEs also provide Hardware Description Language (HDL) examples agnostic to architectural targets that can lead to exploitable vulnerabilities [28].

CWEs are categorized based on the type of weakness and the context in which they occur. Note that several CWE descriptions are broad and apply to several architectural segments of an SoC (e.g., core, peripherals, DMA). In this study, we chose to explore and model some of the CWEs from the top 2021 Most Important Hardware Weaknesses list (shown in Table 3.1) relevant to the IPs in our case study SoC. SIG picked these top CWEs based on the severity, frequency of occurrence, and applicability/generalizability to a wide range of devices. To understand the type and effects of bugs in hardware designs, we implemented CWE instances in our case study SoC so that we could simulate and observe the implications of bugs. There are other important CWEs in our analysis besides the top 2021 CWEs.

CWE	**Description**
1189	Improper Isolation of Shared Resources on System-on-a-Chip
1191	On-Chip Debug and Test Interface With Improper Access Control
1231	Improper Prevention of Lock Bit Modification
1233	Security-Sensitive Hardware Controls with Missing Lock Bit Protection
1240	Use of a Cryptographic Primitive with a Risky Implementation
1244	Internal Asset Exposed to Unsafe Debug Access Level or State
1256	Improper Restriction of Software Interfaces to Hardware Features
1260	Improper Handling of Overlap Between Protected Memory Ranges
1272	Sensitive Information Uncleared Before Debug/Power State Transition
1274	Improper Access Control for Volatile Memory Containing Boot Code
1277	Firmware Not Updateable
1300	Improper Protection of Physical Side Channels

Table 3.1: 2021 Top CWE

CWE-1240 [48]: *Use of a Cryptographic Primitive with a Risky Implementation.* We chose the AES IP to map the CWE description to a specific implementation. On a high level, the AES [5] algorithm is implemented in four stages, i.e., key expansion, initial round, main rounds, and final round. If a designer skips one of these stages to inappropriately "improve" performance, an adversary can break the AES encryption. For instance, if one of the 12 AES rounds during the main round implementation is skipped (shown in Figure 3.1), then the AES runs faster but is less secure. In the code snippet, only 11 rounds of encryption (r_1 to r_{11}) can be seen, whereas the 12^{th} round is removed, and the output from the 11^{th} round is directly connected to the final output.

```
1   aes_round
2     r1 (clk, s0, k0b, s1),
3     r2 (clk, s1, k1b, s2),
4     r3 (clk, s2, k2b, s3),
5     r4 (clk, s3, k3b, s4),
6     r5 (clk, s4, k4b, s5),
7     r6 (clk, s5, k5b, s6),
8     r7 (clk, s6, k6b, s7),
9     r8 (clk, s7, k7b, s8),
10    r9 (clk, s8, k8b, s9),
11    r10 (clk, s9, k9b, s10),
12    r11 (clk, s10, k10b, s11);
13    //One round of encryption is missing
```

Figure 3.1: AES Encryption rounds

CWE-1262 [49]: *Improper Access Control for Register Interface.* SoCs use memory-mapped I/O registers to access peripherals such as AES, HMAC etc. If secure registers in a particular IP are not protected by register locks, adversaries can run user-level code to extract confidential data. One such bug exists in the AES implementation. AES key register is not register locked (shown in Figure 3.2), thereby letting the user read the secure data. As there are missing case statements for address hits 5, 6, 7, 8, 9, 10, the *rdata* register stores the *key0* value. It retains the value due to the buggy case statement leaking the internal secure data to firmware accesses. Our case study SoC [31] has improper access control implementation, owing to which the AES key can be exploited. It can be seen that AES Key *0x00040003* is extracted as shown in Figure 3.3. The extracted key matches with what is stored in the fuse memory. The AES keys can be exploited by triggering the missing case statements, which are at the memory location `0xfff5209028`. This memory location can be accessed through a DMA transaction. The DMA transaction can be executed in 'C' language as shown in Figure 3.4. There are four arguments to the DMA transfer function: Least Significant Bit (LSB) 32-bit memory location of AES Key, Most Significant Bit (MSB) 32-bit memory location of AES key, LSB memory address of the 'C' language variable, MSB memory address of the 'C' language variable.

```
always @(*)
    begin
      rdata = 64'b0;
      if (en) begin
        rdata = key0[address[6:5]];
        case(address[8:3])
            0:
                rdata = reglk_ctrl_i[0] ? 'b0 : {31'b0, start};
            1:
                rdata = reglk_ctrl_i[2] ? 'b0 : p_c[3];
            2:
                rdata = reglk_ctrl_i[2] ? 'b0 : p_c[2];
            3:
                rdata = reglk_ctrl_i[2] ? 'b0 : p_c[1];
            4:
                rdata = reglk_ctrl_i[2] ? 'b0 : p_c[0];
            11:
                rdata = reglk_ctrl_i[6] ? 'b0 : {31'b0, ct_valid};
            12:
                rdata = reglk_ctrl_i[4] ? 'b0 : ct[31:0];
            13:
                rdata = reglk_ctrl_i[4] ? 'b0 : ct[63:32];
            14:
                rdata = reglk_ctrl_i[4] ? 'b0 : ct[95:64];
            15:
                rdata = reglk_ctrl_i[4] ? 'b0 : ct[127:96];
            default:
                if (ct_valid)
                    rdata = 32'b0;
        endcase
      end // if
    end // always @ (*)
```

Figure 3.2: AES Register Interface

CWE-1231 [50] & **CWE-1233** [51]: *Improper Prevention of Lock Bit Modification & Security-Sensitive Hardware Controls with Missing Lock Bit Protection.* In SoCs, some peripheral registers are programmed during boot and blocked from further modifications. This is implemented using a register lock mechanism. When a register is locked, write access is disabled. In the code snippet shown in Figure 3.5, *reglk_ctrl[1]* controls the write access to register lock memory locations 0 to 5. This allows changes to *reglk_mem[2]* which controls AES Key, although *reglk_ctrl[1]* is meant to control only *reglk_mem[1]*. Incorrect implementation of the register lock mechanism can result in the ability to change sensitive registers by an unprivileged user, allowing them to misuse the system and features that the lock is meant to protect.

CWE-1260 [52]: *Improper Handling of Overlap Between Protected Memory Ranges.* IP registers are isolated from each other by assigning a unique address. There might be multiple registers in an IP. Each register has a specific function. For instance, to start the encryption FSM in an AES IP, the user has to write a non-zero value to the corresponding register. The end of encryption triggers another register, which the user can poll to check the status of the encryption FSM. Finally, an output register is used to store the encrypted data. A designer would allocate all these registers a non-overlapping memory range in a

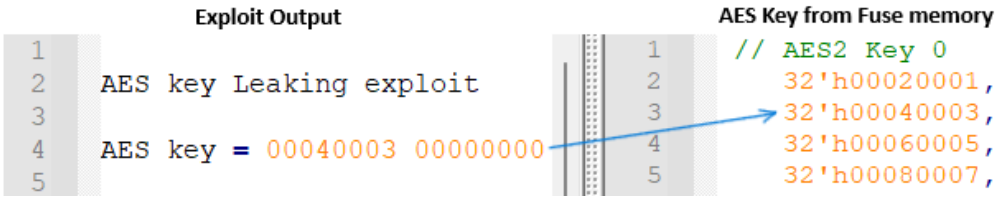


Figure 3.3: AES Key Exploit Output

```c
#include<stdio.h>
#include<stdint.h>

volatile uint64_t* aes2_key_leak = 0xfff5209028;
//MMIO location of AES Key
int main ()
{
    uint64_t rdata[4];//variable to store
                      //the key
    printf("AES key leaking exploit\n");

    dma_transfer(aes2_key_leak,
                 (((uint64_t)aes2_key_leak)>>32),
                 rdata,
                 (((uint64_t)rdata)>>32), 4,1);
    //DMA access to AES Key
    printf("key : %x %x %x %x\n",
                 rdata[0],
                 rdata[1],
                 rdata[2],
                 rdata[3]);
}
```

Figure 3.4: AES Exploit Firmware

non-buggy scenario. As there can be multiple IPs in an SoC, there can be a possibility of overlapping of memory ranges of two IPs. In such a case, the user will have access to privileged registers in a particular IP which are set as unprivileged registers in another IP. If the privileged and unprivileged registers are overlapped erroneously, an unprivileged user can set/unset a privileged register. It can be seen in Figure 3.6, the AES0 memory range overlaps with the AES1 memory range. AES0 memory range is [`0xfff5200000`, `0xfff5201500`]. AES1 memory range is [`0xfff5201000`, `0xfff5202000`]. The memory range [`0xfff5201000`, `0xfff5201500`] is common for the two IPs. If there exists a memory location in the common memory range that is declared as secure and register locked with respect to AES0 but is accessible with respect to AES1, then the secure information stored in the AES0 memory location can be accessed through AES1, violating the security specification.

CWE-1191 [53] & **CWE-1244** [54]: *On-Chip Debug and Test Interface With Improper Access Control & Internal Asset Exposed to Unsafe Debug Access Level or State.* SoCs use access control mechanisms

```
case(address[7:3])
        0:
            reglk_mem[0]  <= reglk_ctrl[3] ? reglk_mem[0] : wdata;
        1:
            reglk_mem[1]  <= reglk_ctrl[1] ? reglk_mem[1] : wdata;
        2:
            reglk_mem[2]  <= reglk_ctrl[1] ? reglk_mem[3] : wdata;
        3:
            reglk_mem[3]  <= reglk_ctrl[1] ? reglk_mem[3] : wdata;
        4:
            reglk_mem[4]  <= reglk_ctrl[1] ? reglk_mem[4] : wdata;
        5:
            reglk_mem[5]  <= reglk_ctrl[1] ? reglk_mem[5] : wdata;
        default:
            ;
endcase
```

Figure 3.5: Lock Bit Modification

```
<!-- AES 192-bit key module-->
<port>
    <name>ariane_aes0</name>
    <base>0xfff5200000</base>
    <length>0x1500</length>
</port>
<!-- AES second module-->
<port>
    <name>ariane_aes1</name>
    <base>0xfff5201000</base>
    <length>0x1000</length>
</port>
```

Figure 3.6: Memory Range Overlap

to restrict unauthorized users from accessing internal registers through test and debug interfaces. The OpenPiton [47] SoC has a JTAG interface, providing access to the internal registers for debug purposes. If the access control or other authentication measures are not implemented correctly, a user may be able to access the encryption keys. The access control mechanism is missing in Figure 3.7. *jtag_hash_o* can be used to debug the internal registers of an IP, but when the access control mechanism is missing, secure internal information can be extracted as shown in Figure 3.8. *key_hash_o* has SHA key values as no access control mechanism is implemented in the test interface of HACK@DAC SoC.

```
assign rdata_o = (addr_q < MEM_SIZE) ? mem[addr_q] : '0;

assign jtag_hash_o = {mem[JTAG_OFFSET-1]};
assign okey_hash_o = {mem[JTAG_OFFSET-9]};
assign ikey_hash_o = {mem[JTAG_OFFSET-17]};
```

Figure 3.7: Test Interface with No Access Control

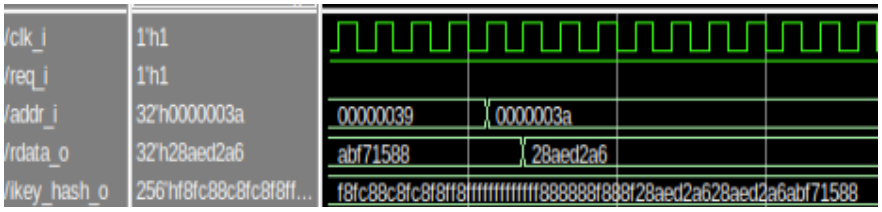


Figure 3.8: Test Interface Signal Waveform

CWE-1189 [55]: *Improper Isolation of Shared Resources on SoC.* SoC designs failing to adequately separate and manage shared resources among different components or subsystems can lead to a range of security issues, including unauthorized access, data leakage, and potential corruption of data or execution flow.

These bugs can be mapped to the following locations in an SoC.

Core-Peripheral Interface: The interface between IP Peripherals and system bus forms the main communication channel between peripherals and the CPU core.

Debug-IP: The debug interface for the processor core. e.g., JTAG.

Access Control, Register Lock: Access Control and Register lock implementation in an SoC to control information and data flow.

MMIO: Memory-mapped interface to access registers of IPs.

Cryptographic cores: Peripheral communication interface to cryptographic accelerator IPs like AES, SHA 256, HMAC etc. and their implementations.

CWE descriptions are mapped to different SoC locations as shown in Table 3.2, but a CWE can be mapped to more than one SoC location. For instance, an illegal attempt to modify an improperly defined access control register could be handled in the Access Control and Register Lock implementation of the Core-Peripheral Interface or in the Cryptographic cores. We mapped CWEs in a way that covers all SoC locations. For instance, CWE-1191 can be mapped to the Access Control and Register Lock SoC location, but we mapped it to Debug-IP to motivate us to develop a patching architecture for a different SoC location.

We explored the nature of hardware bugs, examining their various types and the implications they have on the security and reliability of SoC designs. Through an analysis of hardware CWEs, we gained insights into common vulnerabilities and the necessity of effective mitigation strategies. If hardware bugs are left unaddressed, they can lead to significant security breaches, allowing unauthorized access to sensitive data. Additionally, we discussed the practical impact of these bugs, illustrated through examples such as buggy register lock implementations and faulty cryptographic and debug IPs.

Building on this foundational understanding, Chapter 4 will transition into a detailed examination of patching architectures designed to address the above mentioned hardware CWEs. We will investigate the

specific mechanisms and architectures used to patch hardware bugs, ensuring system security and functionality. By presenting a case study involving the OpenPiton [47] SoC and OpenFPGA [30] eFPGA, we will illustrate the application of these patching architectures in our case study SoC. This chapter will also outline the experimental setup employed to evaluate these architectures, providing a comprehensive analysis of design overhead costs, including area, delay, and latency.

CWE	**SoC location**
1240 [48]	Cryptographic cores
1262 [49]	Access control
1231 [50]	Register lock
1233 [51]	Register lock
1260 [52]	MMIO
1191 [53]	Debug-IP
1244 [54]	Debug-IP

Table 3.2: CWE and SoC location mapping

Chapter 4

Patching Architectures

There are several options when it comes to where a designer could introduce the ability to monitor/respond to events in a system. In other words, there are several locations where a patch could be placed. Given that we want to patch hardware vulnerabilities, the coverage of the patch will be limited in practical terms to parts of a design that are specifically chosen to be patched during the design process. Prior knowledge of what/when must be changed/corrected and how to mitigate a vulnerability will decide the patching implementation and capability. For instance, in [24], an eFPGA-based centralized security policy controller cannot be assumed to patch a wide range of bugs as the programmability (the extent of modifying) of the security wrappers is limited to the signals that are chosen to be security relevant, and the overall patchability (the extent/ability to patch) is limited by the security wrappers and what they are connected to. Observing and controlling all signals is impractical, given constraints on routing and design overhead. When patching a CWE, we rely on our interpretation of the CWE description. We translate the mitigation strategy description to relevant signals in a design. A patch's two main goals are to detect and mitigate a potential vulnerability. In the patching process, the normal operation of the hardware should not be impeded. Assume an instance of an illegal DMA access recognized by the monitoring circuitry of the patch architecture. A naive override approach would be to block the request. However, in a typical SoC communication fabric, this could inadvertently cause a deadlock, as the blocked request will not be removed from the request buffer until the handshake signals are appropriately asserted. In some instances, patching might not even require continuous monitoring, e.g., re-configuring a misconfigured initial IP state.

In Chapter 3, we explored some hardware bugs and mapped them to SoC locations. CWEs mapped to buggy cryptographic cores' SoC location can be patched by entirely replacing the buggy IP. Thus, we propose **eFPGA-REP** architecture that can be directly attached to the SoC bus and implement the desired

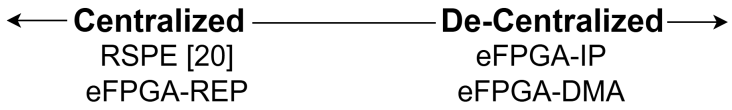


Figure 4.1: Patch Location Spectrum

IP. eFPGA-REP architecture can patch the CWEs mapped to Cryptographic cores, MMIO and Debug-IP.

The Buggy IP can also be patched by closely integrating the eFPGA with the SoC and implementing selected mechanisms. Thus, we propose **eFPGA-IP** architecture for close coupling of eFPGA with chosen IPs. eFPGA-IP can patch the CWEs mapped to Cryptographic cores, Access control, Register lock, and Debug-IP.

As the user makes memory accesses to the SoC through the firmware, some of the illegal accesses can be monitored and patched by manipulating the memory transaction controller/generation. Thus, to monitor transactions, we propose **eFPGA-DMA** architecture.

We investigate each architecture separately and examine if it can patch different CWEs. eFPGA-IP and eFPGA-DMA architectures are integrated close to the location where patching is desired, i.e., a decentralized approach, unlike [24] as shown in Figure 4.1. eFPGA-REP is categorized as a centralized approach as it can act as a centralized security policy engine. We shall see a detailed description of the three architectures with their pros and cons in the subsequent sections.

4.1 SoC + eFPGA Architectures for Patching

A typical SoC comprises one or more processor cores and IPs, e.g., DMA controllers, cryptographic engines, communication peripherals, and JTAG debug infrastructure. IPs are connected to the core via an interconnection fabric, like a NoC, using a well-defined protocol (e.g., AXI4). For hardware patching, designers need to decide where to insert the patching logic for monitoring and intercepting internal signals, as well as which signals to use. To understand the extent of patchability with bus and internal IP signal monitoring, we devised potential design architectures in which an eFPGA can patch hardware bugs: *eFPGA-IP*, *eFPGA-DMA*, and *eFPGA-REP*. As shown in Table 3.2, each of the architectures can address vulnerabilities in different locations. eFPGA-IP can address issues in Cryptographic cores, Access control, and Register locks. eFPGA-DMA is tailored to patch bugs in memory accesses. eFPGA-REP can implement any patch as it can be attached as a standalone IP but in some cases, using an entire eFPGA fabric may be redundant when the utilization percentage is low.

4.1.1 eFPGA-IP

This architecture involves customizing the eFPGA fabric to enable more tightly coupled patching of selected IPs, where the eFPGA's interface is directly connected to the IPs (and internal control signals). Suppose there is a high reliability and security requirement on a crypto engine. To enable patching, we route the crypto engine-related signals to an eFPGA. As this architecture is tightly coupled to patch *selected* IPs, it can lead to a smaller eFPGA fabric footprint than a generic eFPGA fabric. This tight coupling of the eFPGA with specific IPs facilitates monitoring IP-internal states and events. In the case of multiple IPs, designers can extend the I/O size (and, therefore, fabric size) of the eFPGA or attach the IPs to the eFPGA via MUXes for a smaller fabric.

4.1.2 eFPGA-DMA

As most IPs are configured/accessed by a transaction generator/memory access controller through firmware interactions, an eFPGA is added to the transaction generator/memory access controller in this architecture. The goal is to prevent adversaries from exploiting bugs by controlling the transactions generated and restricting unauthorized memory accesses, so having the ability to reprogram the memory controller can help mitigate hardware bugs. This approach is appropriate when the generator/controller has privileged access to a particular IP.

4.1.3 eFPGA-REP

In this architecture, a designer might want the ability to replace an IP outright with an alternate. For example, in the case of a buggy cryptographic engine, an eFPGA, if present, can be programmed with the relevant algorithm. Also, in the event of complete failure of an IP, having an SoC bus protocol-compliant eFPGA can be used to configure a whole IP and replace errant IPs. Therefore, an eFPGA is added to the system with an interface compatible with the bus protocol. In comparison to the other architectures, this solution entails the highest upfront hardware cost.

4.1.4 Patching framework

In line with prior work, we adopt a patching framework to mitigate unforeseen hardware bugs. Patches range from bypassing/overriding the signals to completely curing the cause of a vulnerability. A patch can manipulate the control and data signals, correcting the functionality of a vulnerable IP. Alternatively, patching by replacing a whole IP is possible but likely entails significant complexity and cost.

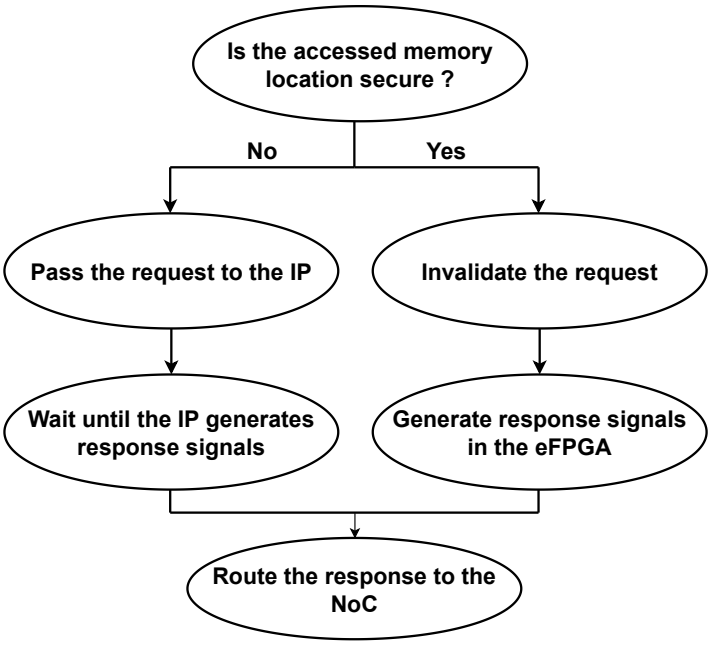


Figure 4.2: An example generalized patch for monitoring and preventing invalid accesses to secure memory location

Concerning our integration design architectures, the patch for the eFPGA-IP and eFPGA-DMA architectures may involve implementing a bus transaction filter, as all the transactions in an SoC go through the bus or boot time implementation of a security policy. Having a bus protocol-compliant eFPGA in an SoC offers the capability to generalize the patching mechanism. The patch for the eFPGA-REP architecture involves Peripheral Communication within a NoC chipset. A generalized patching FSM mechanism to monitor invalid accesses to secure memory locations is shown in Figure 4.2. The FSM either bypasses or routes the signals to the eFPGA based on the memory location accessed. In the event of bypassing the signals, response signals are taken from the IP, and responses to unauthorized accesses are generated within the eFPGA. In this case, the observable signals (eFPGA inputs) are the incoming bus requests, and the controllable signals (eFPGA outputs) are the bus response signals.

In general, designing a patching architecture for a given SoC is an anticipatory/speculative exercise with no standardized approach. Thus, when designing the patching blocks for an SoC, one can first select a CWE to patch and find the set of other CWEs relevant to the one chosen. Next, create a set of "potential" patches for each CWE, as in Figure 4.2. This CWE-based patching block design framework aims to craft a patching infrastructure with as many patchable CWEs as possible. These example patches can help size the required eFPGA fabric and inform the choice of what signals to route to the FPGA and where/how to integrate the eFPGA.

4.1.5 Case study description

To investigate the different SoC + eFPGA architecture options for hardware patching, We performed a case study where we implemented a set of CWE based hardware bugs/vulnerabilities in selected IPs in an SoC. The three eFPGA architectures are used to fix the bugs, and we characterize the area/delay/latency. To model bugs/patches, we examined the CWE database. We implemented them in our case study SoC, identifying important signal interfaces to observe, vulnerable components to control or bypass, critical signals to observe, and critical functions to control. The CWE vulnerabilities we patched are listed in subsection 4.1.7. For each vulnerability, we designed patch triggers and payloads. Note that we model the situation where the bugs somehow escaped detection during design.

We used the open-source OpenFPGA [30] framework to generate the eFPGA fabric and integrated it into the HACK@DAC SoC according to each of the architecture options we discussed in section 4.1. The eFPGA architecture follows a tiled structure containing different tile types, e.g., I/O, Configurable Logic Block (CLB), etc. A .xml input file describes the architecture of the eFPGA. The choice of the architecture file affects the eFPGA's capabilities and size. For instance, a higher input size LUT occupies more area, and the choice of LUT type, for instance, fracturable LUTs, supports efficient implementation of complex logic functions. Finally, we connect all the relevant inputs and outputs from the SoC to the I/O of the eFPGA.

4.1.6 Patch Architectures and Integration

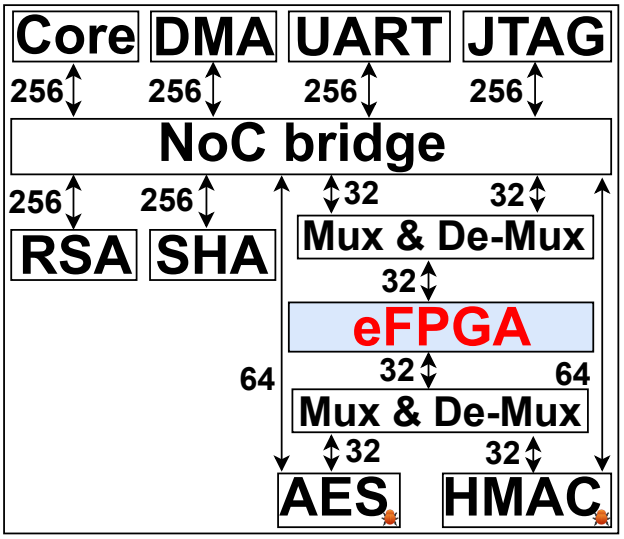
To characterize and compare our proposed architectures, we devised three concrete examples as follows.

eFPGA-IP: eFPGA attached to AES and HMAC

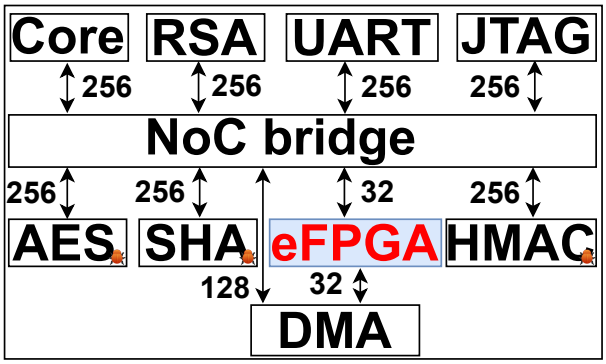
We integrated the eFPGA at the AES and HMAC interface as shown in Figure 4.3a. We tailor the eFPGA to patch bugs in two cryptographic IPs. The eFPGA I/O width is 32 bits to reduce area requirements; this means it can only deal with one IP at a time, so we designed a MUXed architecture that routes one of the active IPs to the eFPGA. So, at any point, only a single IP is attached to the eFPGA. DMA and Non-DMA transactions can be monitored as the eFPGA is closely coupled to the SoC. Adding more IPs to the MUX incurs an increment in MUX size and IP signal routing.

Implementation: Each transaction is checked to see if the region accessed is valid. The eFPGA generates artificial response signals if invalid, as AXI transactions cannot be aborted. If a request is valid, eFPGA acts as a passthrough.

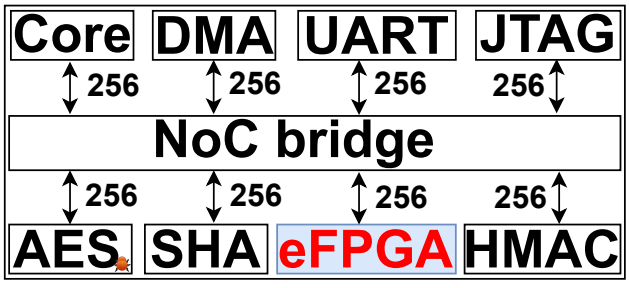
Limitations: Monitoring is limited to the IPs added to the MUXes. As it is a MUXed architecture, interleaved and parallel IP exploits cannot be patched. New IPs cannot be added in the field to be monitored



(a) eFPGA-IP



(b) eFPGA-DMA



(c) eFPGA-REP

Figure 4.3: Three SoC + eFPGA Design Architectures (shaded regions represent our additions to the SoC)

as MUX routing is fixed.

eFPGA-DMA: eFPGA attached to DMA

An eFPGA is integrated at the DMA interface as shown in Figure 4.3b. Given the DMA has complete access to all the IPs, all IP transactions can be monitored and patched in case of unauthorized access via the DMA.

Implementation: This architecture prevents DMA requests to protected regions. The eFPGA fabric is situated between the NoC bridge and the AXI interface of the DMA IP.

Limitations: This architecture is useful in SoCs where the DMA has complete control over all the IPs. However, non-DMA transactions cannot be monitored.

eFPGA-REP: eFPGA attached to NoC

We integrate the eFPGA into the HACK@DAC SoC as shown in Figure 4.3c. As the eFPGA is integrated as an IP to the NoC bridge of the SoC, it can be programmed to act as a fallback for computation or to replace errant/buggy IPs.

Implementation: AXI interface is 256 bits wide, and therefore, eFGPA I/O width is chosen to be 256 bits. In the MMIO space, eFPGA Base is set at `0xfff5212000`. To use the eFPGA as an IP, all the addresses must be remapped with respect to the eFPGA base in the software.

Limitations: The required fabric is complex with a large I/O width and occupies a significant area. eFPGA becomes the bottleneck for faster performance.

4.1.7 Case Study CWEs and Hardware Bugs

In the HACK@DAC [31] SoC, IPs are attached to the NoC bridge, and a NoC-AXI interface translates the NoC signals to AXI signals, which the IP can use. The registers in an IP are memory-mapped to a specific address range. For example, AES IP registers are memory mapped to `0xFFF5209000-0xFFF520A000`. AXI READ and AXI WRITE data are stored in *rdata* and *wdata* registers, respectively. Any IP can be used by populating the *rdata/wdata* registers with appropriate instructions and the correct MMIO address. The MITRE website features a list of the 11 "most important" hardware weaknesses in 2021 [28]. After analyzing the relevant CWEs and studying the potential mitigation techniques, we generalized the observable and controllable signals for monitoring, correcting, overriding, etc, to design the patches. The following bugs can be patched by having a flexible and reconfigurable hardware mechanism in the SoC.

CWE-1240 [48]: *Use of a Cryptographic Primitive with a Risky Implementation.* We chose the AES IP for our case study to implement this weakness. To patch weaknesses in AES, eFPGA-REP architecture can

be used to replace the whole IP with a more secure implementation. The eFPGA-DMA architecture cannot be used as it does not have the capability to re-implement the algorithm. eFPGA-IP might be usable here if the AES core internal logic is tightly coupled to the fabric, but we do not explore this scenario for this CWE as the eFPGA-IP is generic to all the cryptographic cores. The patch for this CWE implements one round of AES encryption. We have implemented the AES algorithm in the eFPGA by routing all the AES IP AXI signals. The missing round in Figure 3.1 (*r12 (clk, s11, k11b, s12)*) is implemented in the eFPGA.

CWE-1262 [49]: *Improper Access Control for Register Interface.* This hardware bug can be mitigated by denying access to the register storing secure information. eFPGA-IP and eFPGA-DMA can filter out insecure read and write access to internal registers. eFPGA-REP would be unnecessary as the issue is at the interface level. The patch for this CWE shown in Figure 4.4, blocks all read accesses to the MMIO registers. The patching FSM is a two-state FSM. In the first state, we invalidate the request signals and generate response signals in the second state. The patch filters the read AXI request by invalidating the *req_addr_valid* signal and generates appropriate response signals to remove stalling of the transaction. In our case study SoC, the byte 0 of the AES key register is memory-mapped to `0xFFF5209028`, so the patching FSM for the CWE-1262 [49] is designed to block read and write accesses from this memory location to the end of the AES key storage (byte 7). *ip_addr_low* and *ip_addr_high* need to be set as `0xFFF5209028` and `0xFFF52090C8` respectively to cover the entire AES key storage memory locations.

CWE-1231 [50] & **CWE-1233** [51]: *Improper Prevention of Lock Bit Modification & Security-Sensitive Hardware Controls with Missing Lock Bit Protection.* eFPGA-IP architecture can be used to mitigate this bug. eFPGA-DMA architecture would also work, given the sensitive registers can be accessed via DMA. eFPGA-REP is not needed here as no IP is broken. The patch for this CWE restricts read and write access to secure registers. Our patching approach issues transactions mimicking the AXI request handshake signals outside the register lock hardware while preventing the write transactions from reaching the register lock module. One possible way to patch the vulnerability is to block any write transactions to the register lock memory after a secure boot sequence. This is a reasonable solution, as the register lock values are not expected to be changed after boot until the system reset. Blocking read/write requests should be performed carefully to avoid deadlock from contention of the AXI request queue. Resource contention happens when appropriate handshake signals are not appropriately handled to pop the request out of the request queue. The patch from CWE-1262 can be reused here by changing the monitoring memory locations.

CWE-1260 [52]: *Improper Handling of Overlap Between Protected Memory Ranges.* MMIO is implemented in the chipset module of the OpenPiton [47] SoC. To patch CWE-1260, MMIO is implemented in the eFPGA as shown in Figure 4.5, where the eFPGA registers storing the MMIO information are read and used in the chipset module. We see that on line 9, the starting address for the MMIO is hardcoded

```
1  always @(posedge clk) begin
2      if (!reset) begin
3          state <= 0;
4      end
5      else begin
6          case (state)
7              2'h0: begin
8                  if (addr_i >= ip_addr_low && addr_i <= ip_addr_high
9                  && req_addr_valid_i == 1'b1)begin
10                     state <= 2'h1;
11                     req_addr_valid_filtered_o <= 1'b0;
12                     // dont forward valid request
13                 end
14                 else begin
15                     state <= 2'h0;
16                     // by default forward requests
17             req_addr_valid_filtered_o <= req_addr_valid_i;
18             resp_addr_ready_filtered_o <= resp_addr_ready_i;
19             resp_r_b_valid_filtered_o <= resp_r_b_valid_i;
20                 end
21             end
22             2'h1: begin
23                 state <= 2'h0;
24                 req_addr_valid_filtered_o <= 1'b0;
25                 resp_addr_ready_filtered_o <= 1'b1;
26                 // make user think request was completed and successful
27                 resp_r_b_valid_filtered_o <= 1'b1;
28                 // make user think request was completed and successful
29             end
30             default: state <= 2'h0;
31         endcase
32     end
33 end
```

Figure 4.4: Access monitoring FSM in an eFPGA

to be `0xfff5202000`. This address cannot be changed later in the field. Setting up the MMIO peripheral-to-address mapping using an eFPGA will allow future modifications in case of overlap or rerouting of the addresses, enabling MMIO reconfiguration. On line 1, we see the HMAC starting memory location is set using an eFPGA which is later used on line 18. eFPGA-REP architecture can mitigate this issue by setting MMIO using an eFPGA but the eFPGA can be customized for a smaller footprint. eFPGA-IP and eFPGA-DMA architectures are unworkable here as this vulnerability affects the MMIO to which the two architectures are not connected.

CWE-1191 [53] & **CWE-1244** [54]: *On-Chip Debug and Test Interface With Improper Access Control & Internal Asset Exposed to Unsafe Debug Access Level or State.* To protect against this weakness, an eFPGA can be used to implement some form of authorization in addition to existing on-chip protections. The patch can act as a secondary layer for debug interface authentication. eFPGA-IP architecture can implement a simple access control mechanism, whereas eFPGA-REP architecture can implement advanced protection mechanisms depending on the use case. eFPGA-DMA architecture would not work here as no bus-based memory access is involved. The patch for this CWE is shown in Figure 4.6, where an eFPGA

```
1  reg [39:0] hmac_begin = eFPGA_Output[0] // `PHY_ADDR_WIDTH'hfff5203000;
2                          // eFPGA controlling the ASIC register
3                          // hmac_begin and hmac_length are
4                          // ASIC registers
5  reg [39:0] hmac_length = eFPGA_Output[1] // `PHY_ADDR_WIDTH'h1000;
6
7  always @* begin
8
9          if ((noc2_filter_data[`MSG_ADDR_] >= `PHY_ADDR_WIDTH'hfff5202000
10                                            // hardcoded address
11         && noc2_filter_data[`MSG_ADDR_] < `PHY_ADDR_WIDTH'hfff5202000
12         + `PHY_ADDR_WIDTH'h1000)
13         & (~uart_boot_en))
14     begin
15         readdressed_flit0[`MSG_DST_X] = `NOC_X_WIDTH'hb;
16     end
17
18         else if ((noc2_filter_data[`MSG_ADDR_] >= hmac_begin
19         // Instead of hardcoding the MMIO start memory address like
20         // in line 6, eFPGA_Output sets the start address.
21         && noc2_filter_data[`MSG_ADDR_] < hmac_begin + hmac_length)
22         & (~uart_boot_en))
23         // Likewise, memory range is also set by the eFPGA.
24     begin
25         readdressed_flit0[`MSG_DST_X] = `NOC_X_WIDTH'hc;
26     end
```

Figure 4.5: MMIO Implementation in an eFPGA

adds the missing access control mechanism. On line 5, we see that the *acct_ctrl_i* input register is set using an eFPGA.

```
1   module fuse_mem # (
2       .MEM_SIZE(FUSE_MEM_SIZE)
3   ) i_fuse_mem         (
4       .clk_i           ( clk_i      ),
5       .acct_ctrl_i     (eFPGA_Output[3]),
6       // access control bit controlled by
7       // an eFPGA
8       .jtag_hash_o     ( jtag_hash  ),
9       .okey_hash_o     ( okey_hash  ),
10      .ikey_hash_o     ( ikey_hash  ),
11      .req_i           ( fuse_req   ),
12      .addr_i          ( fuse_addr  ),
13      .rdata_o         ( fuse_rdata )
14      //eFPGA controls what can be read
15  );
```

Figure 4.6: Access Control bit set by an eFPGA

4.2 Experimental setup and Results

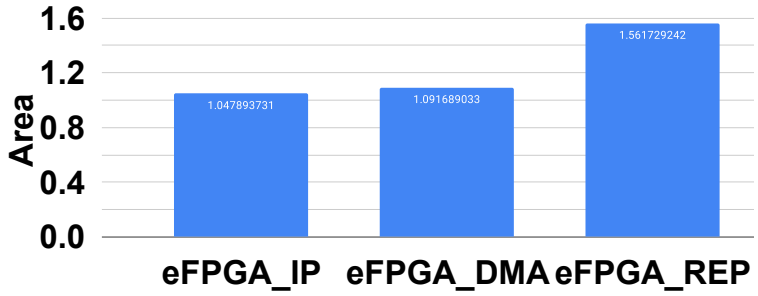
To characterize the feasibility of eFPGA-based hardware patching, we implement the case study SoC and patching architecture options[1]. All the IPs in the HACK@DAC [31] SoC have directional I/O ports, so we generated directional I/O ports instead of OpenFPGA's default bidirectional I/Os. Our goal is to enable patching with minimal overhead, so by removing redundant signals and MSBs of a few control signals, 32 bits of I/O is sufficient to enable patching. This corresponds to 6 x 6 fabric size for the eFPGA-IP and eFPGA-DMA architecture, which is generated using the OpenFPGA [30] tool flow and tested with *Icarus Verilog* [56] before integrating it to the HACK@DAC SoC. Fabric size corresponds to the number of tiles required to implement the eFPGA. The tiles are building blocks of the eFPGA. They are LUTs, I/O blocks and register memories to store the bitstream information. The eFPGA-REP implementation requires a fabric size of 19 x 19 for a 256-bit wide I/O. We need a 256-bit I/O for the eFPGA-REP architecture, as it needs to be added as a standalone SoC, and the I/O width needs to match with other IPs in the HACK@DAC SoC for correct implementation.

Security bugs were inserted in the HACK@DAC SoC by the contest organizers, although we tweaked some bugs to better reflect the CWEs. Initially, patching is disabled to ensure the bugs are exploitable and simulatable with Mentor Questasim 2020.1_1, and simulated with the patches to validate the functionality.

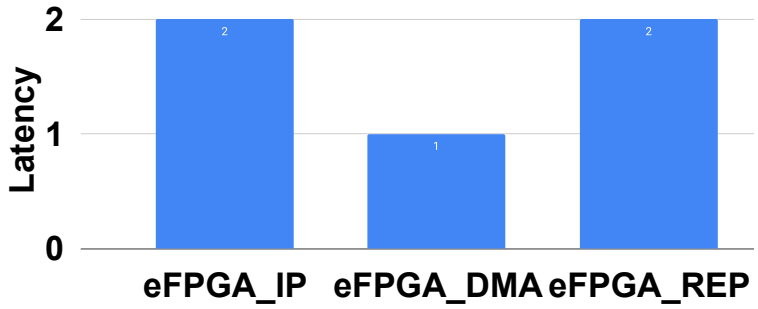
All three design architectures are synthesized and characterized using the Synopsys Design compiler U-2022.12 targeting the 32 nm Synopsys generic academic standard cell library (SAED). We used the updated cva6 [57] Ariane core for synthesis rather than the native Ariane core in the OpenPiton SoC, as the updated cva6 is optimized for clean synthesis results. The three design architectures are synthesized with the same target clock frequency of 0.5 GHz, and the characterization results are shown in Figure 4.7. Note that our synthesis aims to offer a sense of the feasibility of the different architectures, so we did not pursue any aggressive design optimization.

OpenFPGA uses Yosys for synthesis and Verilog Place and Route (VPR) for place and route. It also generates the fabric bitstream for the eFPGA programming. FPGA design parameters like fabric size, I/O width, and utilization percentage are logged after the tool flow run. The parameters from the output log file form the basis for our characterization of patches. Patches for the chosen CWEs are designed by following the CWE potential mitigation description and are characterized using the OpenFPGA [30] flow, which is shown in Table 4.1. The "Architecture" column corresponds to the most appropriate architecture for that patch. The "Patch size" column refers to the number of Verilog lines in the patch Verilog file. "eFPGA size" refers to the number of tiles in the fabric. Each tile corresponds to a specific functionality in the eFPGA. LUTs,

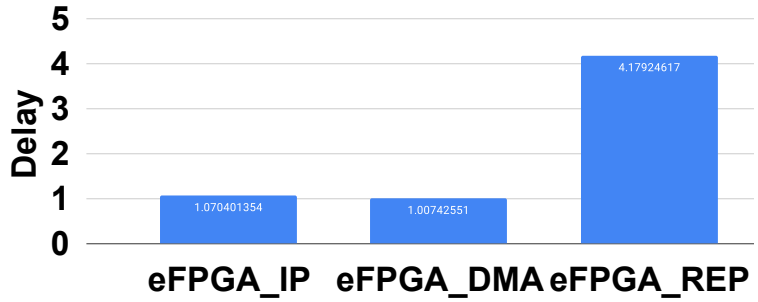
[1]SoC-eFPGA RTL, Patching RTL, and synthesis scripts are available here: https://github.com/CalgaryISH/Openpiton_eFPGA.git



(a) Area of the three design architectures (relative to the standalone SoC area of 1.91 mm^2).



(b) Latency (in terms of clock cycles) incurred due due to the eFPGA.



(c) Worst case critical path delay incurred due to the eFPGA (in terms of Standalone SoC critical path delay of 106.39 ns).

Figure 4.7: Design characterization of the three SoC + eFPGA architectures.

Memory and I/Os are some of the tiles in an eFPGA. With the exception of the patch for CWE-1240 [48] on eFPGA-REP, the fabrics in our case study patches were not fully utilized, suggesting further opportunities to optimize the fabric as we shall see in Chapter 5. Our three design architectures supported all patches with eFPGA-REP having a large fabric size. eFPGA-REP has the capacity to implement any patch, given its' architectural advantage.

The standalone 6 x 6 fabric size eFPGA and HACK@DAC SoC have an area of 0.1 mm^2 and 1.91 mm^2, respectively. The 32-bit wide eFPGA is 5.27% of the area of the standalone SoC (Standalone SoC refers to the SoC alone, excluding the eFPGA). eFPGA-IP architecture occupies an area of 2 mm^2, an increment of 1.04× compared to the standalone SoC as shown in Figure 4.7a. eFPGA-DMA architecture occupies an area of 2.08 mm^2, an increment of 1.09× compared to the standalone SoC, almost the same area as eFPGA-IP

CWE	Architecture	Patch size (# Verilog lines)	eFPGA size	eFPGA Utilization
1240	eFPGA-REP	565	19x19	88%
1231, 1233	eFPGA-DMA	32	6x6	60%
1260	eFPGA-IP	125	6x6	54%
1191, 1244	eFPGA-IP	137	6x6	68%

Table 4.1: Patch and eFPGA complexity.

architecture and comparable to the Standalone SoC. In contrast, eFPGA-REP architecture occupies an area of 2.98 mm^2, an area overhead of 1.56× compared to the standalone SoC. Unless the need to replace a whole IP is required, eFPGA-REP is less feasible, given its area impact.

The latency is dependent on patch complexity. The added latency incurred using eFPGA-REP to patch CWE-1240 is two cycles (one round of AES algorithm takes 2 cycles) as shown in Figure 4.7b. For instance, patching to add 10 rounds of AES may add 20 cycles. The additional latency for simple patches like instruction filtering can be 1 cycle, as seen when we used the eFPGA-DMA architecture. In the case of eFPGA-IP architecture, we observe 2 cycles of latency. One cycle to select an IP to route the appropriate signals and an additional cycle to monitor the accessed memory location.

Worst case critical path timing for standalone SoC is 106.39 ns whereas the timing for standalone eFPGA is 87.2 ns, a decrement of 0.81×. Integrating the eFPGA to the SoC in eFPGA-IP increased the critical path delay by 1.07× as shown in Figure 4.7c, which implies that the eFPGA is constraining the delay requirements. eFPGA-IP architecture's critical path delay (113.88 ns) is slightly higher than that of eFPGA-DMA architecture (107.18 ns) due to the presence of MUX logic in the critical path. Multiple and complex MUXed architecture may further increase the critical path delay in eFPGA-IP architecture. In the case of eFPGA-REP architecture, critical path delay (444.63 ns) increased by 4.17× compared to the standalone SoC. eFPGA-REP architecture requires a larger number of LUT tiles and wider I/O to implement complex patching compared to the other two architectures. Therefore, the fabric's X and Y-dimension expanded, reflected in increased area and critical path delay.

While we identified an eFPGA fabric that could hold one of our patches at a time, designers should consider larger/smaller fabrics for their specific use cases. We shall see eFPGAs holding more than one patch in Chapter 5. Overall, our results suggest that the use of small eFPGAs could be practical in situations where a hardware-based patching fail-safe is desirable at the expense of reasonable area overhead.

Chapter 5

eFPGA Characterization

Chapter 4 provided the feasibility study of eFPGA for hardware patching where an eFPGA is integrated in an SoC to patch hardware bugs. We studied three design architectures that can be used to implement patches in different scenarios, using an OpenPiton [47] based SoC case study. Resource overhead costs for each architecture are determined by ASIC synthesis of SoC + eFPGA RTL. Implementation details of the design architectures with their pros & cons are discussed. For a particular CWE, a patch is generated, and a design architecture is chosen to accommodate the patch. The characterization results focus on the design architectures but not the patches. The designer has information on available patching architectures and the resource overhead for each. However, the designer does not have information on the design parameters that help them decide the eFPGA size for a wide range of patches. Selecting the "best" eFPGA configuration remains an open challenge. In the context of hardware patching, the eFPGA configuration corresponds to the eFPGA design parameters, such as the fabric's I/O width and size. The configuration to use depends on the patches that might need to be implemented and adherence to design specifications.

Note that including a patching infrastructure is an upfront investment where designers anticipate the potential for bugs. However, given that the bugs that might emerge in the field are unknown at the earlier stages of the design lifecycle (otherwise, the bugs should be fixed before manufacturing), designers need strategies to explore what kind of bugs to patch and their associated overhead costs. A potential approach is to consider "historical" bugs to build an idea of what type of patching capabilities are required. However, there is no database currently available that designers can refer to when deciding the eFPGA design parameters. Moreover, SoCs have strict area and critical path delay limitations. An arbitrary eFPGA integration can significantly impact the chip's area, worsen the critical path delay, and may fail to meet design specifications, so prior information on resource overhead costs is required for effective design space exploration.

This necessitates characterizing eFPGA fabrics with various configurations and implementations to support a designer's intent to use eFPGAs for hardware patching.

In this chapter, we do not consider any particular design architecture but instead create various patches for CWEs and develop a method to determine the eFPGA size that needs to be inserted during the design phase. We analyzed additional CWEs to cover a wide range of patches. We provide information on which eFPGA parameters are needed to choose a specific eFPGA fabric size and also provide investment costs given a fabric size. We characterize a wider variety of hardware CWEs to generate a database of CWE and overhead costs, which can then be used for automated design space exploration. This database can help the designer choose an eFPGA fabric that meets the design specifications. We also explore the feasibility of combining individual patch FSMs to better utilize the eFPGA fabric.

5.1 CWE Analysis

To create a database of patches with their eFPGA information, we chose five additional CWEs, implemented instances of them, and designed patches for each one. We synthesized each patch using the OpenFPGA flow [30] to understand which eFPGA design parameters drive the overhead costs. In addition to the patches described in Chapter 3, we created patches for instances of the following CWEs for our characterization.

CWE-1220 [58] & **CWE-1222** [59]: *Insufficient Granularity of Access Control & Address Regions Protected by Register Locks.* Access control and register lock mechanisms restrict unauthorized access to critical assets in a system. In our case study SoC, access to some registers like AES keys, change in privilege levels, etc, are restricted. However, the lack of granularity in access control implementation diminishes the goal of protecting secure information. For instance, using only one register to control read and write accesses is not granular enough to separate accesses for different privilege levels. Similarly, using a few registers to cover broad address ranges may not be possible to separate boot time register programming and runtime register usage. If the AES key is stored at address `0x1000`, plain text is stored at address `0x2000`, and the cipher text is stored at `0x3000`, then two separate register locks must be used to allow the software to transfer the plain text, access the cipher text but not the AES key. The patch for this CWE uses the eFPGA as an additional layer to implement access control policies and register locks. In Figure 5.1, we observe that the access control bus is 96-bit wide, but the output bus is erroneous as only 56 access control bits are used. This may mean that some IPs' access control is not in effect. This is the case for the AES0 IP in our case study SoC. This can enable an adversary to change the default AES0 key and program custom key, after which all the encryptions can be cracked. Exploit for this CWE is shown in Figure 5.2. We see that as there is no access control mechanism in the AES IP, we can make a memory write transaction to the AES

key location and program a custom key. Memory access write is initiated by using the *writeMultiToAddress* 'C' function. AES encryption and decryption can be initiated by using the *aes_encrypt* 'C' function with appropriate arguments. The difference in cipher text for the same plain text after successfully changing the encryption key is shown in Figure 5.3.

Figure 5.1: Access control Bus Width Mismatch

```
1  #include <stdio.h>
2  #include <stdint.h>
3  void main(){
4
5      aes_run(0); //default run
6      aes_run(1); //change the AES key
7  }
8  void aes_run(int change)
9  {
10 uint32_t key[4] = {0x1beef, 0x2beef, 0x3beef, 0x4beef};
11 if(change){
12     writeMultiToAddress((uint64_t *)aes, AES_KEY0_BASE, key, AES_KEY_WORDS);}
13     //exploiting the lack of access control mechanism in AES IP.
14     aes_encrypt(pt, st, ct, key_sel);
15     // call the aes encryption function
16     printf("encrypted data : %x %x %x %x\n", ct[0], ct[1], ct[2], ct[3]);
17     //print cipher text
18 }
```

Figure 5.2: Exploit for CWE-1220

```
1  encrypted data : 8877d7aa 250eab25 704ab443 a8949ebc
2  encrypted data : 348f550b 3d56174e cf1e1901 52c9b5b1
```

Figure 5.3: Access Control Exploit Output

The patch for this CWE is shown in Figure 5.4. The patch is a three-state FSM that filters accesses to AES0. The FSM is triggered when a write access is made to AES0, and the accessed memory location lies in the pre-defined range. In the case of benign access, the signals are routed to AES0. In the event of unauthorized access, response signals are generated by the FSM. The difference between the patch in Figure 5.4 and Figure 4.4 is the type of requests that they can filter and responses they can generate. Figure 5.4 is designed to filter write requests and generate AXI handshake responses for write requests. Figure 4.4 is designed to do the same for read requests.

CWE-1280 [60]: *Access Control Check Implemented After Asset is Accessed.* In Verilog, there are blocking and non-blocking assignments. In a blocking assignment (=), statements are executed sequentially in the order they appear in the code. Non-blocking assignments allow for parallelism in simulation or

```
always @(posedge clk) begin
    if (!reset) begin
        state <= 0;
    end
    else begin
    case (state)
            2'h0: begin
                if(addr_i >= ip_addr_low && addr_i<=ip_addr_high
                && req_addr_valid_i == 1'b1)
                begin
                    state <= 2'h1;
                    req_addr_valid_filtered_o <= 1'b0;
                    // dont forward valid request
                    resp_addr_ready_filtered_o <= 1'b1;
                    // make user think request was completed and successful
                end
                else begin
                    state <= 2'h0;
                    // by default forward requests
              req_addr_valid_filtered_o <= req_addr_valid_i;
              resp_addr_ready_filtered_o <= resp_addr_ready_i;
              resp_w_ready_filtered_o <= resp_w_ready_i;
              resp_r_b_valid_filtered_o <= resp_r_b_valid_i;
                end
            end
            2'h1: begin
                  state <= 2'h2;
                  resp_addr_ready_filtered_o <= 1'b0;
                  resp_w_ready_filtered_o <= 1'b1;
                  // make user think request was completed and successful
            end
          2'h2: begin
              state <=2'h0;
              resp_w_ready_filtered_o <= 1'b0;
              resp_r_b_valid_filtered_o <= 1'b1;
              // make user think request was completed and successful
          end
            default: state <= 2'h0;
        endcase
    end
end
```

Figure 5.4: Patch for CWE-1220

synthesis. When blocking assignments are erroneously used to access an asset and perform access control checks in that order. Assets can be accessed even before access control checks are complete. The patch for this CWE uses the eFPGA as a filter to enforce security measures. The security check is implemented in the eFPGA, then the access to IP is granted.

CWE-1299 [61] *Missing Protection Mechanism for Alternate Hardware Interface.* An asset in an SoC might have access control if accessed via the primary interface. However, if all the paths are not protected, an adversary can compromise the asset through an alternate path. For instance, in our case study SoC, some AES registers are access-controlled when accessed directly, but DMA access to AES is not access-controlled, so secure AES registers can be accessed through DMA. eFPGA can be used as a secondary layer to block

```
case(patch_state)
3'h0: begin
    axi_req_filtered_aw_addr <= 64'hfff5200020;
    //address storing sensitive information
    patch_state <= 3'h1;
end
3'h1: begin
    patch_state <= 3'h2;
    //stall here for a clock cycle
end
3'h2: begin
    axi_req_filtered_w_data <= 64'h0;
    axi_req_filtered_aw_valid <= 1'h1;
    axi_req_filtered_w_valid <= 1'h1;
    patch_state <= 3'h3;
    //clear the register with sensitive
    //information
end
3'h3: begin
    axi_req_filtered_aw_valid <= 1'h0;
    patch_state <= 3'h4;
    //generate AXI handshake signals
end
3'h4: begin
    axi_req_filtered_w_valid <= 1'h0;
    axi_req_filtered_b_ready <= 1'h1;
    patch_state <= 3'h5;
    //ready to receive new request
end
3'h5: begin
    axi_req_filtered_b_ready <= 1'h0;
    patch_state <= 3'h0;
end
endcase
```

Figure 5.5: Patch for CWE-1243

asset access in all possible paths. If the designer intends to use eFPGA-IP architecture, then they must add the IP of interest and DMA IP to the patching architecture to cover all possible paths.

CWE-1243 [62]: *Sensitive Non-Volatile Information Not Protected During Debug.* Certain security-critical data, such as encryption keys, ROM, and access control values, are stored in fuses within a chip. During the early-boot process or at runtime, this data is read from the fuses and temporarily stored in secure locations, like registers or local memories. Access to these locations is restricted to trusted agents, who are only granted read access. However, these access restrictions are lifted during debug operations in buggy implementations, enabling users to retrieve sensitive information. For instance, in our case study SoC, AES IP temporarily stores the key data in registers. These registers can be accessed during debug mode if the implementation is buggy. eFPGA is used to clear out sensitive information in registers during debug mode. The patch for this CWE is a six-state FSM as shown in Figure 5.5. The patch is triggered whenever debug mode is ON. We write 'NULL' data to registers storing sensitive information when debug mode is ON.

5.2 Concurrent FSM

In Chapter 4, single FSM patches are implemented in the eFPGA. With single FSM patch implementation, we observe that the eFPGA fabric is under-utilized, so we plan to implement concurrent patch FSMs in the eFPGA as shown in Figure 5.6 as an attempt to increase the fabric utilization. We aim to increase fabric utilization by having minimal effect on resource overhead costs. As the I/O size of the patch increases the fabric size, we will keep the fabric I/O width the same as that of the single patch fabric implementation. To keep the I/O size the same, we would have to use Mux & De-Mux architecture. The MUX helps reduce the eFPGA size and routes the relevant signals from multiple IPs to a single eFPGA based on the active status of an IP. Additionally, the select signal of the Mux is given to the eFPGA, which would trigger the corresponding FSM for a particular IP. At time t_1, FSM_1 can be triggered, and at time t_2, FSM_2 can be triggered. When more than one FSM is triggered, it transitions to different states, and at any single point, more than one IP may wait for response signals. So the De-Mux in the eFPGA routes the relevant FSM signals to the corresponding IP.

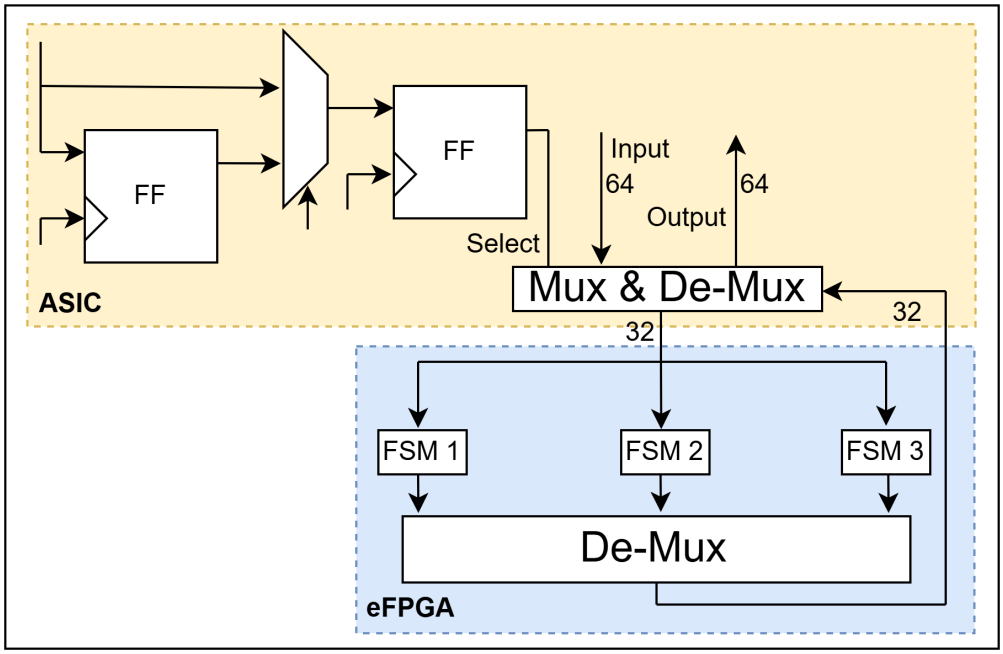


Figure 5.6: Concurrent FSM

In our case study, we combined the FSMS for patches CWE-1262 [49] and CWE-1220 [58]. As explained in subsection 4.1.7 and section 5.1, the patches get triggered whenever there is a read/write operation, so this signal must be routed to the eFPGA. This *RD/WR* signal internal to the eFPGA determines which FSM to trigger and route the corresponding response signals from the De-Mux to the external I/O. We observe that the fabric is better utilized when patches are combined, as shown in section 5.3.

Concurrent FSMs also help in mitigating interleaved exploits. The interleaving of exploits for CWE-1262, Figure 3.4 and CWE-1220, Figure 5.2 is shown in Figure 5.7. The X-axis shows time, and the Y-axis shows different scenarios. Case A: As only AES0 IP is accessed at all time instances, the patch gets triggered, and mitigations will be in effect if needed. The same is true for Case B, where the difference is that AES1 is accessed at all times instead of AES0. In Case C, the accesses to both AES0 and AES1 are interleaved, so the patch, if triggered, will be futile in mitigating any vulnerability as the patch expects the firmware accesses to the same IP at all time instances. In multi-core processors, firmware is run in concurrent threads [63], where the accesses to memory and IP are interleaved to maximize the processing bandwidth. In such cases, having a single patch in the eFPGA may not serve the purpose of patching/mitigating hardware bugs. Therefore, the eFPGA should be programmed to act in a more practical scenario like Case C. Combining the patches for CWE-1262 and CWE-1220 would serve two purposes. One is to maximize the fabric utilization, and the other is to patch bugs in case of multi-thread/concurrent exploits.

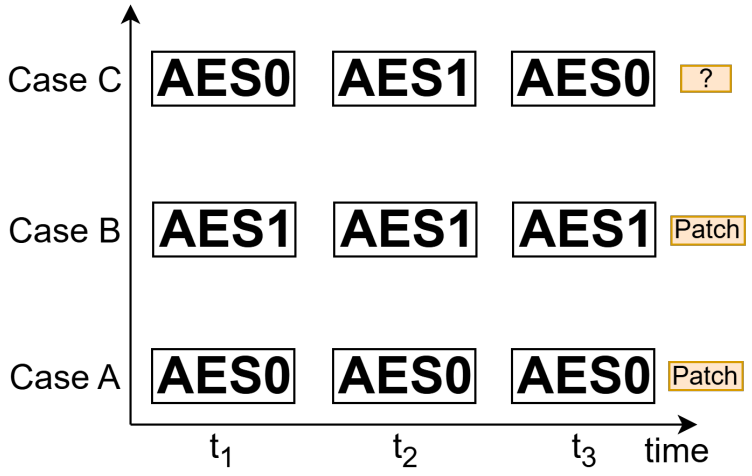


Figure 5.7: Interleaved Exploit

With the increase in the number of IPs that need to be patched, the MUXed architecture shown in Figure 5.6 needs to be changed to accommodate the increase of signals while still keeping the I/O bus width of the eFPGA constant. For instance, to patch two IPs, a 2:1 MUX needs to be implemented. The Verilog code that describes the hardware of a MUX is the same when the size increases, but the register width of signals/bus changes with a pattern. This pattern can be captured in a computer program to automate the task of generating verilog code for increased MUX size. Automated MUX generation script can be found here [1].

[1] https://github.com/CalgaryISH/MUX_Gen.git

5.3 Experimental setup and Results

5.3.1 eFPGA RTL Generation

We chose the open-source OpenPiton[47] based HACK@DAC[31] SoC and open-source FPGA framework OpenFPGA[30] for our eFPGA characterization. The patches for CWEs described in Chapter 4 and Chapter 5 are synthesized using the OpenFPGA flow. Generating an eFPGA design from Verilog code involves several steps, typically using the provided toolchain that includes synthesis, placement, routing, and bitstream generation. Verilog code refers to the patch that the designer plans to implement on the eFPGA. OpenFPGA provides push-button scripts for users to run design flows. We consider an existing design flow script *full_testbench*, which generates eFPGA RTL, a bitstream file, and a testbench file containing the patch I/O to eFPGA I/O mapping. This mapping information will be used in the SoC + eFPGA Integration.

The toolchain first generates a Verilog netlist from the patch file for which it uses a task configuration file located at

basic_tests/full_testbench/configuration_chain/config/task.conf. This configuration file, shown in Figure 5.8, specifies the XML-based FPGA architecture, which determines the parameters and settings for FPGA architecture. We used the following architecture file *k4_N4_40nm_IoSubtile_cc_openfpga.xml* (line 2, 5) in the

```
1  [OpenFPGA_SHELL]
2  openfpga_arch_file=${PATH:OPENFPGA_PATH}/k4_N4_40nm_IoSubtile_cc_openfpga.xml
3
4  [ARCHITECTURES]
5  arch0=${PATH:OPENFPGA_PATH}/openfpga_flow/vpr_arch/k4_N4_tileable_IoSubtile_40nm.xml
6
7  [BENCHMARKS]
8  bench0=/home/anudeep/Openpiton_eFPGA/patches/eFPGA_filter_fsm.v
9
10 [SYNTHESIS_PARAM]
11 bench0_top = eFPGA_filter_fsm
```

Figure 5.8: Task Configuration file

configuration file. As all the IPs in the HACK@DAC SoC have unidirectional I/O ports, we generated unidirectional I/O ports instead of OpenFPGA's default bidirectional I/Os. This architecture file also generates 4 input size LUTs (k4), 4 logic elements in a CLB (N4). The I/O Subtile is used to generate more compact and higher-density I/Os. The flow then uses `yosys` for internal synthesis. The Verilog patch file location needs to be provided in the task.conf file (line 8), followed by the name of the Verilog module (line 11). After updating the *task.conf* file, run the design flow script as follows: *"Python3 mux_gen.py 2"*. The last argument in the Python command above determines the number of IPs the MUX architecture needs to be generated. After the flow run is executed, FPGA RTL and bitstream can be found in the *latest* folder.

5.3.2 eFPGA Characterization

The eFPGA RTL for patches is generated with the process described in subsection 5.3.1. We extract the number of LUTs needed to implement a patch from the `yosys` output log at the end of a successful run. We characterize each patch by the number of Verilog lines and I/O width as an initial measure of relative complexity. Patch characterization for different CWEs is shown in Table 5.1. "Patch size" refers to the number of lines in the Verilog patch file. "Fabric size" refers to the number of tiles placed in a particular dimension during the synthesis process. "Input size" is the I/O Bus width of the eFPGA interface. Multiple copies of the same CWE correspond to variations of the patch implementation. For instance, CWE-1240 is implemented with varying Input sizes to understand the effect of changes in input size on the area and delay overhead costs. Note, however, that a patch might have fewer Verilog lines but can implement complex functionality and vice versa, so the number of Verilog lines cannot be used as a parameter to decide fabric size. The I/O size of our patches will affect the number of I/O tiles that the fabric needs, and the LUT number represents the patch's functionality and complexity, so we chose these two parameters to fix the FPGA size. These parameters are plotted against different FPGA sizes as shown in Figure 5.9a. It can be seen that with an increase in I/O width and LUTs, a bigger FPGA fabric is needed to implement the patch. To estimate the investment cost for various CWE-based patches, we synthesized the corresponding eFPGA RTL using Synopsys Design Compiler on the 32 nm Synopsys generic academic standard cell library.

CWE	Patch size	Fabric size	Input size	LUTs	Area	Delay
1262	125	6	32	42	0.09	87.2
1240	565	38	288	4400	4.61	4546.46
1240	52	11	72	64	0.31	107.85
1231	33	6	32	19	0.1	93.23
1220	151	16	112	625	0.85	308.67
1280	33	6	32	19	0.1	93.23
1299	125	6	32	42	0.09	87.2
1243	182	17	120	749	0.85	418.84
1191	137	19	136	247	1.07	444.63

Table 5.1: Patch Characterization. The area is in mm^2. Delay is in ns.

With an increase in FPGA size, the area overhead cost increases quadratically (an equation modeling this is given as $0.0032x^2 - 0.0015x - 0.0395$), as shown in Figure 5.9b. The critical path delay also increases quadratically (Equation is given as $5.19x^2 - 87.4x + 427$), as shown in Figure 5.9c. The equations are trendlines extrapolated with the data points generated after synthesis. Using the three plots Figure 5.9, a designer can pick a particular FPGA size to meet their design requirements. For instance, if the designer intends to implement a patch with [I/O, LUT]=[192,350], the designer can choose an FPGA of size 30. Then, the Area/Delay vs. FPGA size plot can be used to estimate the investment cost. In our example case, the

area and delay investment cost for an FPGA of size 30 can be estimated using the respective best-fit equation for area and delay. The estimated area overhead cost is $0.0032 * (30^2) - 0.0015 * (30) - 0.0395 = 2.795\ mm^2$. The estimated delay overhead cost is $5.19 * (30^2) - 87.4 * (30) + 427 = 2476\ ns$. If these investment costs meet the desired requirements, then the designer can fix the eFPGA size and insert it. If not, the designer can iterate until their requirements are met. Similar models for other technology libraries can be created by profiling the synthesis results for our example patches.

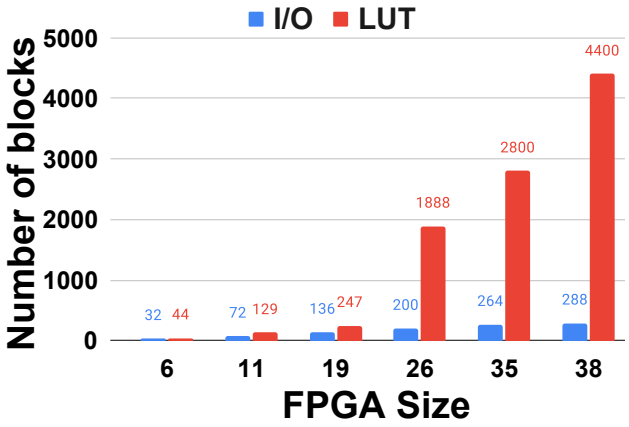
5.3.3 Concurrent FSM Characterization

The eFPGA fabric for the patch, CWE-1262, is 6 x 6, with a utilization fraction of 0.45. The fabric size for the patch, CWE-1220, is 6 x 6, with a utilization fraction of 0.46. The 1% utilization fraction increment is attributed to the patch's increase in FSM states. When patches for both the CWEs are combined as explained in section 5.2, the fabric size remained the same, whereas the utilization fraction increased to 0.54 as shown in Table 5.2, a 9% increase which shows the fabrics are better utilized. This is because the unused LUTs in the case of a single patch are utilized when the patch complexity increases. The patch "complexity" here refers to a higher requirement for LUTs in case of combined patch implementation. Similarly, individual patches can be combined to run concurrently and maximize fabric utilization.

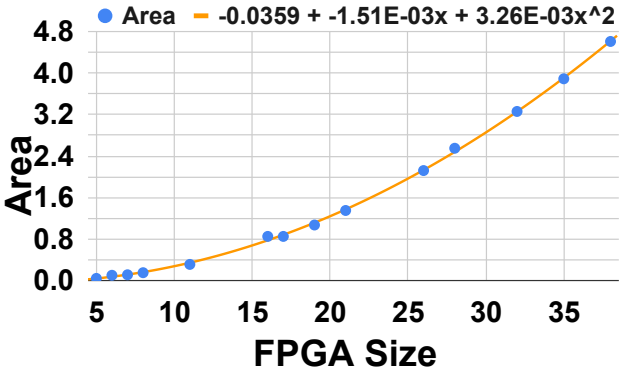
Patch	**Fabric size**	**Utilization fraction**
CWE-1262 [49]	6 x 6	0.45
CWE-1220 [58]	6 x 6	0.46
Combined Patch	6 x 6	0.54

Table 5.2: Concurrent FSM Fabric Utilization

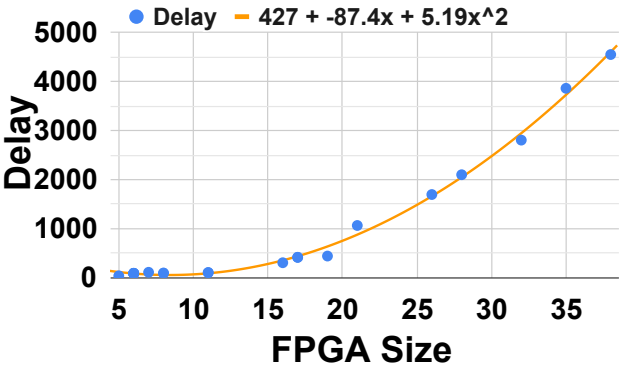
In this chapter, we characterized different eFPGA fabrics to generate a database of fabric sizes and their overhead costs. We hope that this database will help designers fix fabric sizes during the design phase. We also observe that combining patches for individual CWEs will maximize fabric utilization and help in patching interleaved exploits.



(a) FPGA size vs Number of blocks (I/O and LUT)



(b) Area (in mm^2) for different eFPGA sizes



(c) Delay (in ns) for different eFPGA sizes

Figure 5.9: eFPGA characterization of different CWEs.

Chapter 6

Discussion and Conclusion

This chapter synthesizes the findings from Chapter 3, Chapter 4 and Chapter 5. It highlights the key insights, evaluates the proposed solutions, and suggests future research directions to patch hardware bugs using eFPGA-based patching mechanisms.

6.1 Discussion

6.1.1 Answer to RQ1: Hardware Bugs and Vulnerabilities

RQ1: Which design and security bugs inspired by hardware CWEs can be patched?

Answer: The exploration of CWE-based hardware bugs in Chapter 3 identified several critical vulnerabilities in SoC designs. These CWEs are mapped to different SoC locations and implementations, such as improper access control, buggy register lock implementation, and faulty cryptographic and DebugIPs. These bugs can be patched by having a reconfigurable mechanism in the SoC. Their CWE description emphasized that these vulnerabilities, if left unaddressed, could lead to significant security breaches, allowing unauthorized access to sensitive data and malicious use of electronic devices. Buggy code examples from the HACK@DAC SoC and Firmware exploits were provided, and waveforms were reviewed during the exploitation.

6.1.2 Answer to RQ2: Patching Architectures

RQ2: What are the potential eFPGA-based patching architecture options for an SoC given the IPs are accessed through firmware and memory transactions, and what are the overhead costs for area, delay, and latency in each patching scenario?

Answer: Chapter 4 delved into various patching architectures designed to mitigate hardware vulnerabilities. Three primary architectures were evaluated: eFPGA-REP, eFPGA-DMA, and eFPGA-IP. Each architecture offers different benefits and trade-offs in terms of implementation complexity, area overhead, and patching capabilities.

eFPGA-IP Architecture: This architecture integrates eFPGA directly to specific IP blocks, allowing for fine-grained control and patching of individual vulnerabilities within the IP. While it offers lower overhead than eFPGA-REP, it may require multiple instances of eFPGA fabric for comprehensive coverage.

eFPGA-DMA Architecture: This approach integrates eFPGA to the DMA controller to monitor and control data accesses/transfers. It provides a balance between flexibility and overhead, allowing for targeted patching of data access-related vulnerabilities with moderate resource usage. The overhead costs of eFPGA-DMA are comparable to eFPGA-IP but the patching methodology differs.

eFPGA-REP Architecture: This architecture focuses on replacing entire IP blocks with eFPGA-based alternatives. It offers high flexibility and can address a wide range of vulnerabilities but comes with significant area and delay overhead due to the complexity of implementing the entire IPs in eFPGA fabric.

The experiments presented in Chapter 4 demonstrated the practical feasibility of using eFPGA to address specific hardware vulnerabilities.

6.1.3 Answer to RQ3: eFPGA Characterization

RQ3: How can a designer select the "best" eFPGA configuration given a set of hardware CWEs that need to be patched?

Answer: Chapter 5 presented the characterization of eFPGA-based patches for various CWEs. The experimental results highlighted the importance of selecting the appropriate eFPGA configuration based on the specific needs of the patching mechanisms and the nature of the vulnerabilities.

The characterization also underscored the impact of eFPGA fabric size and I/O width on the overall performance and area overhead. Smaller eFPGA fabrics were found to be practical for implementing targeted patches with reasonable area and delay penalties, suggesting their suitability for real-world applications where hardware designs need to be reconfigured in the field.

6.1.4 Key Insights

Flexibility vs. Overhead: The choice of patching architecture significantly affects the balance between flexibility and resource overhead. While eFPGA-REP offers the highest flexibility, it also incurs the highest cost in terms of area and delay. Conversely, eFPGA-IP provides lower overhead but may lack the flexibility

needed for comprehensive patching.

Granularity of patching: Effective hardware security requires fine-grained control over access and data flow mechanisms. The case studies showed that choosing appropriate signals for better observability and controllability can help in fine-grained patching with minimal overhead costs, emphasizing the need for detailed and precise patching strategies. Further optimization and study of eFPGA-IP architecture can aid in fine-grained patching.

Integration and Compatibility: Successful implementation of eFPGA-based patches depends on close integration with existing SoC architectures. The compatibility of eFPGA with SoC bus protocols and the ability to monitor and control critical signal paths are crucial for effective patching.

6.2 Conclusions and Future work

The increasing complexity of SoC architectures poses challenges in detecting hardware bugs, requiring advanced techniques and methodologies to mitigate these issues and ensure the reliability and correctness of SoC designs. In this thesis, we analyzed hardware CWEs to understand about hardware bugs and their implications if left unaddressed. To tackle hardware bugs, we presented three SoC + eFPGA design architectures to investigate the feasibility of eFPGA-based hardware patching. Our characterization results depict that the close coupling of the eFPGA fabric with the SoC offers better patchability with minimal impact on area and performance. Using the eFPGA to replace errant IPs would significantly affect the performance. Designers can use the contributions and findings of this thesis, such as our patch characterization and synthesized fabric size database for the area and critical path delay overhead estimation, to help them decide whether eFPGA can be used for their specific design. They can also gain insights from our patches using concurrent FSMs to mitigate interleaved exploits and improve fabric utilization.

eFPGA-based patching architectures can address and enable patching in response to hardware vulnerabilities detected in-field in SoC designs. By offering flexible and dynamic patching solutions, eFPGA can significantly enhance the security and resilience of modern computing systems, ultimately contributing to more secure and trustworthy hardware platforms. This thesis hypothesized that it is feasible to patch hardware bugs using eFPGAs, and our results suggest that this could be a useful approach.

While the research presented in my thesis suggests that smaller fabrics can implement hardware patching with minimal overheads, we identified several opportunities for future work:

Optimization of eFPGA Fabrics: eFPGA fabrics can be optimized reduce area and delay overheads. This could involve exploring new architectures and configurations that enhance the efficiency of eFPGA-based patching.

Automated Patching Frameworks: Develope automated frameworks for designing and deploying eFPGA-based patches could streamline the patching process, making it easy to integrate and broaden the use in other SoCs. These frameworks should incorporate comprehensive databases of CWEs and corresponding patches to facilitate in-field reconfiguration in response to newly discovered vulnerabilities.

Security Evaluation and Testing: Rigorous security evaluation and testing of eFPGA-based patches are essential to ensure their effectiveness and reliability. Our future work would include extensive validation of patching architectures under various attack scenarios to assess their robustness and resilience. Hardware used for bitstream programming needs special attention as any related vulnerability in the programming infrastructure would become an attack vector and subvert the use of eFPGA for patching. We will study and characterize the potential security impacts of the patch support architecture (i.e., analyze the implications of the misuse of hardware patch capability). We also plan to develop a CAD framework to insert the eFPGA in new SoC architectures and perform security verification of the eFPGA + SoC implementation to ensure that it does not introduce novel security weaknesses.

Integration with other SoC frameworks: Exploring the integration of eFPGA-based patching with other open-source frameworks, such as OpenTitan [42], could demonstrate the effectiveness of eFPGA in enhancing the overall security of future SoC designs.

Bibliography

[1] D. Canavese, L. Mannella, L. Regano, and C. Basile, "Security at the edge for resource-limited iot devices," *Sensors*, vol. 24, no. 2, p. 590, 2024.

[2] J. Karande and S. Joshi, "Real-time detection of cyber attacks on the iot devices," in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2020, pp. 1–6.

[3] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang, "The impact of quantum computing on present cryptography," *arXiv preprint arXiv:1804.00200*, 2018.

[4] L. Torres, P. Benoit, G. Sassatelli, M. Robert, F. Clermidy, and D. Puschini, *An Introduction to Multi-Core System on Chip – Trends and Challenges*. New York, NY: Springer New York, 2011, pp. 1–21. [Online]. Available: https://doi.org/10.1007/978-1-4419-6460-1_1

[5] M. M. N. Biasizzo, M. Mali, and F. Novak, "Hardware implementation of aes algorithm," *Journal of Electrical Engineering*, vol. 56, no. 9-10, pp. 265–269, 2005.

[6] Y. Chen and S. Li, "A high-throughput hardware implementation of sha-256 algorithm," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–4.

[7] E. Khan, M. El-Kharashi, F. Gebali, and M. Abd-El-Barr, "A reconfigurable hardware unit for the hmac algorithm," in *2005 International Conference on Information and Communication Technology*, 2005, pp. 861–874.

[8] S. Ray, A. Basak, and S. Bhunia, *SoC Security Policies: The State of the Practice*. Cham: Springer International Publishing, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1007/978-3-319-93464-8_1

[9] V. Dorsey and C. Morhardt, "Intel Security Development Lifecycle," Intel Corporation, Tech. Rep., 2020. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2020/10/sdl-2020-whitepaper.pdf

[10] C.-Y. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang, "Soc hw/sw verification and validation," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, 2011, pp. 297–300.

[11] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable hardware bugs," in *28th USENIX Secur. Symp. (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230.

[12] M. Amrani, L. Lucio, and A. Bibal, "Ml + fv = ♡? a survey on the application of machine learning to formal verification," *arXiv: Software Engineering*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49648152

[13] S. El Jaouhari and E. Bouvet, "Secure firmware over-the-air updates for iot: Survey, challenges, and discussions," *Internet of Things*, vol. 18, p. 100508, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660522000142

[14] W. Diehl, F. Farahmand, P. Yalla, J.-P. Kaps, and K. Gaj, "Comparison of hardware and software implementations of selected lightweight block ciphers," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[15] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/ high-performance embedded systems: The safes perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, 2008.

[16] J. Villasenor and B. Hutchings, "The flexibility of configurable computing," *IEEE Signal Processing Magazine*, vol. 15, no. 5, pp. 67–84, 1998.

[17] A. Duncan, F. Rahman, A. Lukefahr, F. Farahmandi, and M. Tehranipoor, "Fpga bitstream security: A day in the life," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[18] F. Renzini, C. Mucci, D. Rossi, E. F. Scarselli, and R. Canegallo, "A fully programmable efpga-augmented soc for smart power applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 2, pp. 489–501, 2020.

[19] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro*, vol. 27, no. 1, pp. 12–25, 2007.

[20] I. Wagner and V. Bertacco, "Caspar: Hardware patching for multicore processors," in *2009 Design, Automation and Test in Europe Conference and Exhibition*, 2009, pp. 658–663.

[21] A. Basak, S. Bhunia, and S. Ray, "A flexible architecture for systematic implementation of soc security policies," in *2015 IEEE/ACM Int. Conf. on Computer-Aided Des. (ICCAD)*, 2015, pp. 536–543.

[22] Z. Basnight, J. Butts, J. Lopez, and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76–84, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1874548213000231

[23] P. D. Schiavone, D. Rossi, A. Di Mauro, F. K. Gürkaynak, T. Saxe, M. Wang, K. C. Yap, and L. Benini, "Arnold: An efpga-augmented risc-v soc for flexible and low-power iot end nodes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 677–690, 2021.

[24] A. P. Deb Nath, S. Ray, A. Basak, and S. Bhunia, "SoC security architecture and cad framework for hardware patch," in *2018 23rd Asia and South Pacific Des. Automation Conf. (ASP-DAC)*, 2018, pp. 733–738.

[25] B. Tan, R. Elnaggar, J. M. Fung, R. Karri, and K. Chakrabarty, "Toward hardware-based IP vulnerability detection and post-deployment patching in SoC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1158–1171, 2021.

[26] T. Austin, "Diva: A dynamic approach to microprocessor verification," *Journal of Instruction-level Parallelism*, vol. 2, 07 2003.

[27] A. Dharavathu and B. Tan, "Investigating the feasibility of efpga-based hardware patching," in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2024, pp. 1–7.

[28] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2023. [Online]. Available: https://cwe.mitre.org/scoring/lists/2021_CWE_MIHW.html

[29] M. Corporation, "Mitre corporation," https://www.mitre.org/, 2024, accessed: 2024-08-07.

[30] X. Tang, E. Giacomin, B. Chauviere, A. Alacchi, and P.-E. Gaillardon, "Openfpga: An open-source framework for agile prototyping customizable fpgas," *IEEE Micro*, vol. 40, no. 4, pp. 41–48, 2020.

[31] "HACK@DAC21 – Hack@EVENT HW CTF." [Online]. Available: https://github.com/HACK-EVENT/hackatdac21

[32] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 26–37.

[33] J. T. Vesey, "Time-to-market: Put speed in product development," *Industrial Marketing Management*, vol. 21, no. 2, pp. 151–158, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/001985019290010Q

[34] R. Raducu, R. J. Rodríguez, and P. Álvarez, "Defense and attack techniques against file-based toctou vulnerabilities: A systematic review," *IEEE Access*, vol. 10, pp. 21 742–21 758, 2022.

[35] S. Trimberger, "Trusted design in fpgas," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 5–8. [Online]. Available: https://doi.org/10.1145/1278480.1278483

[36] C. Sturton, M. Hicks, S. T. King, and J. M. Smith, "Finalfilter: Asserting security properties of a processor at runtime," *IEEE Micro*, vol. 39, no. 4, pp. 35–42, 2019.

[37] I. Wagner, V. Bertacco, and T. Austin, "Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 2, pp. 380–393, Feb. 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4358502/

[38] W.-K. Liu, B. Tan, J. M. Fung, R. Karri, and K. Chakrabarty, "Hardware-supported patching of security bugs in hardware IP blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, pp. 54–67, 2023.

[39] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 159–172.

[40] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, ser. LEET'08. USA: USENIX Association, 2008.

[41] F. Restuccia, A. Meza, and R. Kastner, "Aker: A design and verification framework for safe and secure soc access control," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[42] "Opentitan: The first open source silicon root of trust," https://opentitan.org, accessed: 2024-06-07.

[43] P. Prinetto, G. Roascio *et al.*, "Hardware security, vulnerabilities, and attacks: A comprehensive taxonomy." in *ITASEC*, 2020, pp. 177–189.

[44] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[45] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "Magic: Malicious aging in circuits/cores," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, apr 2015. [Online]. Available: https://doi.org/10.1145/2724718

[46] M. Guri, B. Zadov, and Y. Elovici, "Odini: Escaping sensitive data from faraday-caged, air-gapped computers via magnetic fields," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1190–1203, 2020.

[47] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proc. 21st Int. Conf. Architectural Support for Programming Languages and Operating Syst.* ACM, 2016, p. 217–232.

[48] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1240.html

[49] MITRE Corporation, "CWE - Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1262.html

[50] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1231.html

[51] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1233.html

[52] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1260.html

[53] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1191.html

[54] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1244.html

[55] MITRE Corporation, "CWE - CWE Most Important Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1189.html

[56] "Icarus Verilog — Icarus Verilog documentation." [Online]. Available: https://steveicarus.github.io/iverilog/

[57] F. Zaruba and L. Benini, "Energy and performance analysis of a 64-bit risc-v core in 22-nm fdsoi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[58] MITRE Corporation, "CWE - CWE Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1220.html

[59] MITRE Corporation, "CWE - CWE Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1222.html

[60] MITRE Corporation, "CWE - CWE Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1280.html

[61] MITRE Corporation, "CWE - CWE Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1299.html

[62] MITRE Corporation, "CWE - CWE Common Hardware Weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1243.html

[63] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009.