Towards Multipronged On-chip Memory and Data Protection From Verification to Design and Test

Senwen Kan  
*Southern Methodist University*, skantx2@gmail.com

Jennifer Dworak  
*Southern Methodist University*

Follow this and additional works at: [https://scholar.smu.edu/engineering_compsci_etds](https://scholar.smu.edu/engineering_compsci_etds)

Part of the Computer and Systems Architecture Commons, Digital Circuits Commons, Hardware Systems Commons, Other Computer Engineering Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

**Recommended Citation**

Kan, Senwen and Dworak, Jennifer, "Towards Multipronged On-chip Memory and Data Protection From Verification to Design and Test" (2022). *Computer Science and Engineering Theses and Dissertations*. 27. [https://scholar.smu.edu/engineering_compsci_etds/27](https://scholar.smu.edu/engineering_compsci_etds/27)

This Dissertation is brought to you for free and open access by the Computer Science and Engineering at SMU Scholar. It has been accepted for inclusion in Computer Science and Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit [http://digitalrepository.smu.edu](http://digitalrepository.smu.edu).
TOWARDS MULTIPRONGED ON-CHIP MEMORY AND DATA PROTECTION
FROM VERIFICATION TO DESIGN AND TEST

Approved by:

Dr. Jennifer Dworak
Professor of Electrical and Computer Engineering

Dr. Ping Gui
Professor of Electrical and Computer Engineering

Dr. Sukumaran Nair
University Distinguished Professor of Electrical and Computer Engineering

Dr. Mitchell Thornton
Professor of Electrical and Computer Engineering

Dr. Eric Larson
Associate Professor of Computer Science
TOWARDS MULTIPRONGED ON–CHIP MEMORY AND DATA PROTECTION
FROM VERIFICATION TO DESIGN AND TEST

A Dissertation Presented to the Graduate Faculty of the
Lyle School of Engineering and Computer Science
Southern Methodist University
in
Partial Fulfillment of the Requirements
for the degree of
Doctor of Philosophy
with a
Major in Computer Engineering
by
Senwen Kan

Bachelor of Science, Electrical and Computer Engineering, Baylor University
Master of Science, Electrical Engineering, Virginia Polytechnic and State University

Fall, 2022
ACKNOWLEDGMENTS

First and foremost, I want to acknowledge my Lord and savior, Jesus Christ, for lighting in me the wisdom and interest in pursuing this work. Next, I want to thank my wife, Shuya, for her patience and support during my studies, without which none would be possible. My children’s laughter and giggling helped me as well.

Professor Jennifer Dworak’s guidance is instrumental in this work. Professor Dworak has always been kind, generous, thoughtful, and patient with me. She’s a great inspiration to me. I’m proud to call her my academic parent. I can’t thank her enough.

I also want to give tribute to my Dad, Xiaojun Kan, who fostered my interest in math and engineering and who also happened to be my best friend - I miss you dad. My mom, Meifang, provided steady support to me over the years as well — thanks mom!

To the rest of my family, a big thank you all your support! A nod especially to my uncle!

Finally, to my current and former co-workers that are too many to name, I want to thank you all for all your mentorship, friendship, and comradery.
Towards Multipronged On–Chip Memory and Data Protection
From Verification to Design and Test

Advisor: Dr. Jennifer Dworak
Doctor of Philosophy degree conferred Fall, 2022
Dissertation completed Fall, 2022

Modern System on Chips (SoCs) generally include embedded memories, and these memories may be vulnerable to malicious attacks such as hardware trojan horses (HTHs), test access port exploitation, and malicious software. This dissertation contributes verification as well as design obfuscation solutions aimed at design level detection of memory HTH circuits as well as obfuscation to prevent HTH triggering for embedded memory during functional operation. For malicious attack vectors stemming from test/debug interfaces, this dissertation presents novel solutions that enhance design verification and securitization of an IJTAG based test access interface. Such solutions can enhance SoC protection by preventing memory test instruments from adversarial access. To help with the test data collection of embedded memories that may improve SoC security and reliability, this dissertation contributes a memory test optimization unit (MTOU). To enhance functional memory protection, approaches to verify memory fault and security handling circuits are also presented.
TABLE OF CONTENTS

LIST OF FIGURES ................................................................. xii
LIST OF TABLES ............................................................... xiv

CHAPTER

1. INTRODUCTION ................................................................. 1
   1.1. Embedded Memories and Security Risks ................................. 2
       1.1.1. Supply Chain Risks .............................................. 3
            1.1.1.1. Mitigating Manufacturing Inserted HTH in Embedded
                    Memories ..................................................... 3
       1.1.2. IP Level Risks for Embedded Memories .......................... 3
            1.1.2.1. Memory IPs .............................................. 3
            1.1.2.2. 3PIP and IP Threats ..................................... 4
            1.1.2.3. HTH Presence and Detection in Embedded Memory IP ... 4
       1.1.3. Test and Debug Access Security .................................. 4
            1.1.3.1. Production Test Flow .................................... 5
            1.1.3.2. Vulnerability from Test and Debug Access Mechanisms ... 5
       1.1.4. Security Aspect of Design Verification ........................... 5
            1.1.4.1. Embedded Memory Protection Verification ................. 6
            1.1.4.2. Test Access Mechanism Verification ....................... 6
       1.2. Dissertation Contributions and Prior Work Discussions ............... 7
            1.2.1. Protecting Embedded Memories from HTH ..................... 7
            1.2.2. Solutions to Protect the TAM ................................ 7
            1.2.3. Solutions to Verify IC Memory Security and Reliability Features ... 9
       1.3. Dissertation Organization ............................................. 9

2. Protecting SRAMs Against Hardware Trojans Attacks at the Component
   Level With X-Propagation .................................................. 11
2.1. Overview ................................................................. 11
2.2. Hardware Trojan Introduction ...................................... 11
2.3. Background and Previous Work ...................................... 13
  2.3.1. SRAM Verification Approaches .................................. 13
  2.3.2. Trojan Detection in SRAMs ....................................... 14
  2.3.3. Novelty ............................................................. 16
2.4. Trojan Design and Insertion ......................................... 16
  2.4.1. Trojan 1 ............................................................ 17
  2.4.2. Trojan 2 ............................................................ 18
  2.4.3. Trojan 3 ............................................................ 19
  2.4.4. Trojan Insertion ................................................... 20
2.5. X-Triggering Of Trojan ................................................ 22
2.6. Experimental Results ................................................... 28
2.7. Conclusion ............................................................... 32

3. Protecting SRAMs Against Hardware Trojans Attacks at the Component Level With CRC based Design Obfuscation ........................................ 35
  3.1. Overview ............................................................... 35
  3.2. Introduction ........................................................... 35
  3.3. Hamming and CRC coding ............................................ 38
  3.4. Proposed Method ....................................................... 39
    3.4.1. SRAM Address Swizzler .................................... 39
    3.4.2. SRAM Data Obfuscation and Error Detection ............... 41
  3.5. Experimental Results ................................................. 42
    3.5.1. Wrapper Design Implementation ............................... 42
    3.5.2. Experimental Setup .......................................... 43
    3.5.3. Experimental Results ....................................... 43

vi
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.1.2. Protecting Control Access</td>
<td>77</td>
</tr>
<tr>
<td>5.4.2. Device-Specific Chain Stubbing</td>
<td>79</td>
</tr>
<tr>
<td>5.4.3. Re-configurable Chain Shift Checker</td>
<td>79</td>
</tr>
<tr>
<td>5.4.4. Final Architecture</td>
<td>80</td>
</tr>
<tr>
<td>5.4.5. Additional Considerations</td>
<td>82</td>
</tr>
<tr>
<td>5.5. Results</td>
<td>83</td>
</tr>
<tr>
<td>5.6. Conclusions and Future Work</td>
<td>85</td>
</tr>
<tr>
<td>6. IJTAG Integrity Checking with Chained Hashing</td>
<td>86</td>
</tr>
<tr>
<td>6.1. Overview</td>
<td>86</td>
</tr>
<tr>
<td>6.2. Introduction</td>
<td>86</td>
</tr>
<tr>
<td>6.3. Previous Work and Background</td>
<td>88</td>
</tr>
<tr>
<td>6.3.1. Previous Work in JTAG and IJTAG Security</td>
<td>88</td>
</tr>
<tr>
<td>6.3.2. SHA3 and Persistent Memory</td>
<td>91</td>
</tr>
<tr>
<td>6.4. Threat Model</td>
<td>92</td>
</tr>
<tr>
<td>6.4.0.1. Example 1 - Remote IJTAG Exploit</td>
<td>93</td>
</tr>
<tr>
<td>6.4.0.2. Example 2 - Insider Threat</td>
<td>93</td>
</tr>
<tr>
<td>6.5. Proposed Method</td>
<td>94</td>
</tr>
<tr>
<td>6.5.1. Hashing IJTAG Transactions</td>
<td>96</td>
</tr>
<tr>
<td>6.5.2. Chained Hashing</td>
<td>98</td>
</tr>
<tr>
<td>6.5.3. Incorporating Persistent Memory</td>
<td>100</td>
</tr>
<tr>
<td>6.5.4. Integrity Checking Operations</td>
<td>102</td>
</tr>
<tr>
<td>6.5.4.1. Check Operation</td>
<td>103</td>
</tr>
<tr>
<td>6.5.4.2. Load Operation</td>
<td>103</td>
</tr>
<tr>
<td>6.5.4.3. Update Operation</td>
<td>104</td>
</tr>
<tr>
<td>6.5.5. Control Finite State Machine Overview</td>
<td>104</td>
</tr>
<tr>
<td>6.5.6. Impact of Integrating Integrity Checking</td>
<td>106</td>
</tr>
</tbody>
</table>
9.7.1. Implementation ............................................................... 156
9.7.2. Design Simulation .......................................................... 157
9.7.3. Area Impact ................................................................. 158
9.7.4. Memory Test Flow Tester Implications ............................... 159
  9.7.4.1. Test Time Analysis .................................................. 160
  9.7.4.2. Observations about MTOU ....................................... 162
9.8. Summary ........................................................................... 163
10. Conclusion ........................................................................ 165
  10.1. Summary ................................................................. 165
  10.2. Research Impact .......................................................... 166
  10.3. Closing Remarks ......................................................... 167
BIBLIOGRAPHY ................................................................. 168
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simple SRAM Model</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Trojan_1 High Level Block Diagram</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Trojan_2 High Level Block Diagram</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>PseudoCode for Variable Assignment in SRAM without Trojan</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>PseudoCode for Variable Assignment in SRAM with Trojan_3</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>X-Triggering Example</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Algorithm 1</td>
<td>26</td>
</tr>
<tr>
<td>2.8</td>
<td>Algorithm 2</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>Algorithm 3</td>
<td>28</td>
</tr>
<tr>
<td>2.10</td>
<td>Walking 0 or 1 Access Sequence</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>BIST 6N Pseudo-code</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Final Circuit Design</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>A Simplified IJTAG Network</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>a) Block diagram for PCIE DFX-IP. b) ICL for DFX-IP in PCIE</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>iProc for various test modes</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Core-level ICL/PDL validation and ATPG flow</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>IJTAG chain structure</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>ICL MTAP description</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>SoC—level ICL extraction and validation flow</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>IJTAG based verification flow</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>IJTAG Abstract verification testing</td>
<td>61</td>
</tr>
</tbody>
</table>
4.10 SoC-level setup for a group of cores with low power compression mode .......... 64
5.1 IJTAG Reconfigurable TDR ................................................................. 74
5.2 IJTAG RTDR Chain Stubbing ................................................................. 79
5.3 Echeloned IJTAG Data Protection Architecture ......................................... 81
6.1 Hash-Based Secure IJTAG Access Protocol .............................................. 94
6.2 System Block Diagram ............................................................................. 95
6.3 Control FSM Diagram ............................................................................. 104
7.1 Test Generation Flow ............................................................................. 116
7.2 Guest Code Generation ........................................................................... 118
7.3 Test Program Example ............................................................................ 120
8.1 NoC Model for RAS .............................................................................. 130
8.2 NoC RAS Validation Flow ........................................................................ 132
9.1 MTOU Organization .............................................................................. 146
9.2 MTOU Flow FSM .................................................................................. 149
9.3 MTOU MBISD FSM .............................................................................. 153
9.4 MTOU Production Test Flow Chart ......................................................... 154
9.5 Design A: MBIST Based Test Flow Vs MTOU Based Test Flow ............... 164
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SRAM Simulation Results without Trojan</td>
</tr>
<tr>
<td>2.2</td>
<td>SRAM Simulation Results with Trojan</td>
</tr>
<tr>
<td>3.1</td>
<td>SRAM Simulation and Synthesis Results</td>
</tr>
<tr>
<td>4.1</td>
<td>IJTAG Bug Classification and Detectability</td>
</tr>
<tr>
<td>4.2</td>
<td>IJTAG Bug Verification Results</td>
</tr>
<tr>
<td>4.3</td>
<td>Test Setup Results impact of iMerge</td>
</tr>
<tr>
<td>5.1</td>
<td>Design Area and Complexity Matrix</td>
</tr>
<tr>
<td>6.1</td>
<td>Design Configuration Area Table</td>
</tr>
<tr>
<td>7.1</td>
<td>Test Generation Sweep</td>
</tr>
<tr>
<td>8.1</td>
<td>Validation Effort Summary</td>
</tr>
<tr>
<td>8.2</td>
<td>Simulation Time Comparison</td>
</tr>
<tr>
<td>9.1</td>
<td>MTOU Interface Pin Table</td>
</tr>
<tr>
<td>9.2</td>
<td>MTOU Chip Wide Area Impact</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

The interconnectedness in computing devices is rapidly increasing along with advancing manufacturing process technology. Computing devices are increasing in compute capability, integrating more functionality, being used in more critical applications, and all the while, being widely deployed. For the average American household, smart climate control devices [42] can be connected to the Internet, and now even smart light bulbs can be linked to the Internet [102]. These household smart devices can be controlled or monitored via the cloud with personal devices such as smartphones or even smart watches at locations around the world. Similar trends hold as well for automobiles - more cars are now connected to the Internet as well [68]. After examining semiconductor device use cases in the industrial manufacturing sectors, it is clear that these connected devices have taken hold there also [51]. Upstream from these devices, are network routing and switching and server processing semiconductor devices spanning across the globe. From the few aforementioned cases, computing devices are becoming vital in assisting in the daily functions of homes, individuals, and businesses. Embedded memories are commonly found in these semiconductor devices to allow program execution, sensitive information storage, and many other important functions. Therefore, establishing security and reliability for embedded memories found in computing devices at the integrated circuit (IC) level has a foundational impact on systems, devices, and services that are dependent on interconnected semiconductors.

This work seeks to propose a number of solutions that can enhance the reliability and security for embedded memories. To begin to explain the proposed solutions, the rest of this chapter will first examine embedded memories. Following that, it will delve into problem areas affecting embedded memory security and reliability.

Following these discussions, the solutions proposed by this work will be outlined.
1.1. Embedded Memories and Security Risks

Embedded memories are found in a variety of ICs. According to [74], an embedded memory is a non-standalone memory that supports on-chip blocks or cores and allows them to accomplish intended functionalities. For example, embedded memory can be used as cache by processor cores to facilitate program execution. In some use cases, they are used to either permanently or temporarily store critical programs, such as the operating system on chip.

One of the more common forms of volatile embedded memories is static random access memory (SRAM). They are frequently used by on-chip subsystems to buffer operational data such as communication or sensor data. In cryptographic cores, cryptographic computational data are often temporarily stored in SRAMs. In general purpose microcontrollers, SRAMs are often configured to function as main system memory. However, SRAM stored data will lose contents once the IC is powered off.

In order to provide more permanent on-chip storage, non-volatile memories (NVM) can be embedded on chip as well. A NVM implementation can be based on Erasable Programmable Read Only Memory (EPROM), Flash technology [74], MRAM technology [119], RRAM technology [18], and various other technologies.

In addition to SRAM and NVM, read only memories (ROM) can also be considered as a type of embedded memory. There are other types of embedded memories as well, such as One-Time programmable memories (OTP) and embedded Dynamic random access memories (EDRAM). However, SRAMs, NVMs, and ROMs are the more frequently implemented embedded memories.

Compromise, intrusion, or interruption in embedded memories and their operations may lead to IC malfunction or expose IC vulnerabilities to malicious adversaries. Therefore, securing embedded memory is critical in ensuring the security of the IC. This dissertation focuses on the following four areas:

- Supply Chain Risks
- Third Party IP Risks
Test and Debug Access Security

Security Aspects of Design Verification

1.1.1. Supply Chain Risks

Chip companies relying on third party semiconductor manufacturing are called “fabless” companies. The fabless chip companies may contract third party fabs located in other countries for device production. Third party fabs may be influenced by national security interests or personnel infiltration to insert malicious circuits into embedded memories in IC products unbeknownst to the fabless chip companies. Therefore, it is conceivable that malicious circuits - Hardware Trojan Horses (HTHs) can be inserted during the manufacturing process.

1.1.1.1. Mitigating Manufacturing Inserted HTH in Embedded Memories

There are many region on a die where HTH can be inserted through the manufacturing process. Embedded memory production technology are tightly coupled with foundry process technology. Thus, this association could allow foundry or personnel to have more readily available capability to insert HTH into embedded memories on ICs.

Therefore, this dissertation seeks to provide useful mitigation techniques in the face of the risk of HTH insertion during the manufacturing process. More specifically, the concepts explored here seek to provides a mechanism to obfuscate the HTH triggering mechanism and therefore preventing the HTH from being deterministically triggered.

1.1.2. IP Level Risks for Embedded Memories

1.1.2.1. Memory IPs

Embedded Memory IPs can be designed by IC providers internally. However, the integration of third party intellectual properties (IPs) among IC design houses is fairly common for cost mitigation — as is noted in [108]. Thus, the integration and adoption of third party
memory IP is not uncommon. Embedded Memory IPs come in the form of fully laid out physical implementations or memory compilers. The creation of these memory IPs may not be fully controlled by the chip company. The lack of control means the chip company may not be able to fully assess the vulnerability of such IPs.

1.1.2.2. 3PIP and IP Threats

Memory IPs are not trivial - they may incorporate design techniques such as data interleaving, advanced sense amplifier implementation, various repair mechanisms, low power techniques, and etc. It’s conceivable that a third party IP (3PIP) design house or some insiders could insert well hidden HTHs into the memory IPs they provide.

While third party IPs pose a significant threat to chip companies, it is imperative to recognize that the chip company’s internally developed IPs may not be free from risks as well. It’s also possible that design errors were not exposed in the design verification closure, which could lead to exploitable threats at the device level.

1.1.2.3. HTH Presence and Detection in Embedded Memory IP

While in section 1.1.1.1 HTH insertion due to potentially malicious foundry activity is discussed, it is also possible for HTH to be inserted into Embedded Memory IP by 3PIP providers or malicious internal infiltration. An HTH attack from on chip memory elements can corrupt or disrupt chip operations. For example, in some cases, denial of service (DOS) attack can be implemented with low effort. In potentially more complex cases, program counter locations can be manipulated due to well planted HTHs in memories.

The memory IP level risks mentioned in this subsection further indicate that an investigation into obfuscation techniques for HTH memory may not be sufficient to deter HTH risks in memories. Therefore, portions of this dissertation investigate the insertion and detection of HTH in on-chip SRAMs.
1.1.3. Test and Debug Access Security

1.1.3.1. Production Test Flow

Once the silicon wafer is produced by a Fab, each die on the wafer must undergo testing to ensure each die meets final product performance, reliability, and quality requirements. Production testing can be costly. Just as mentioned in 1.1.1, to lessen the cost, many device vendors again rely on third party test houses for device production testing in order to amortize production cost.

1.1.3.2. Vulnerability from Test and Debug Access Mechanisms

The manufacturing test program for each IC may access proprietary contents stored in on-chip test instruments. The test program typically accesses such instruments via a test access mechanism (TAM) such as a JTAG port or alternative test ports. These instruments may implement on-chip memory tests. Hence, if such TAMs and protocols are not properly protected, adversaries may exploit the TAM to allow instruments to be exploited to inject undesirable data into embedded memories, which would lead to unintended device malfunctions. In some cases, the TAM, if unprotected could allow instruments to be configured in such way as to allow confidential end user contents to be extracted.

Therefore, in order for on-chip embedded memories to be protected from adversarial attacks due to potential device TAM vulnerabilities, security measures for an on-chip TAM may be needed. To partially address security concerns for the on-chip TAM, this dissertation investigates access protection as well data integrity protection measures with respect to a well-established industry TAM standard.

1.1.4. Security Aspect of Design Verification

Even if the risks arising from 1.1.1, 1.1.2, and 1.1.3 are adequately addressed, increasingly complex silicon architecture and implementation could have functional defects, or bugs, that could be exploited by attackers to gain unauthorized device access into embedded
memories. Therefore, device features must be thoroughly verified to prevent functional bugs from manifesting into exploitable gaps for attackers.

1.1.4.1. Embedded Memory Protection Verification

Device manufacturers have had a legacy of affording functional reliability and security measures for embedded memories. On chip mechanisms for error detection and correction for embedded memories are exemplified by implementations on [53] and [112]. On chip security protections for embedded memories such as caches are also started to be found in products [4]. Chip designers have focused on error detection and correction methods to account for soft errors on chip. Such methods may involve physical error detection circuitry as well as error handling software routines. To afford security for embedded memories, embedded memories for virtualized computing resources must be isolated from one another via a combination of chip circuitry as well as system software. Hence, these efforts amount to complex on-chip systems that should be afforded adequate verification.

It’s conceivable that chip design bugs due to either reliability measures or security measures for embedded memories can lead to security vulnerabilities. Therefore, in addition to investigating adversarial HTH attacks from foundry as well as IP providers, this dissertation investigates verification methods to help ensure functional embedded memory protection mechanisms may be free from design bugs as well as HTH insertions.

1.1.4.2. Test Access Mechanism Verification

Observations from 1.1.3.2, indicates that TAM access without adequate security protection can lead to exploitable vulnerabilities. It can also be argued that inadequate verification for an on-chip TAM can lead to unexpected but exploitable gaps in TAM security access control. In fact, [39] demonstrated one such potentially exploitable gap due to insufficient TAM verification.

Therefore, a thorough verification methodology may be needed to ensure the on-chip TAM is free of design defects or bugs. In this dissertation, an effort to potentially improve
1.2. Dissertation Contributions and Prior Work Discussions

In response to observations noted in sections 1.1.1, 1.1.2, 1.1.3, and 1.1.4, this dissertation provides an echeloned three pronged approach to enhance the overall security for on-chip memories and on-chip test and debug accessible storage elements.

1.2.1. Protecting Embedded Memories from HTH

This dissertation provides two solutions aimed at the protection of embedded memories from a potential adversary seeking to insert HTH circuits in embedded memories. In the literature with respect to HTH insertion in embedded memories, one of the prior approaches closest to the work in this dissertation primarily focused on detecting HTHs in 3PIP cores instead of embedded memories [120]. The proposed solution in this dissertation, described in chapter 2, is among the first to investigate HTH models and potential insertion in embedded memories [54]. Furthermore, this work provides a method to detect HTHs during the register transfer level (RTL) chip design development stage along with experimental results. However, in the event that an HTH evaded detection at the RTL stage or is inserted during the manufacturing process, chapter 3 provides a novel solution that adds another layer of protection that may prevent an HTH from being triggered deterministically [59]. This proposed solution swizzles or remaps SRAM addresses and encodes SRAM data with CRC based obfuscation to prevent HTH triggering. With CRC based data obfuscation, SRAM reliability is enhanced, and furthermore, irradiation based attack mechanisms maybe mitigated.

1.2.2. Solutions to Protect the TAM

While HTHs are an avenue to compromise ICs with embedded memories directly, other means of attack should not be overlooked. As TAM and access network standards like IJTAG [49] are becoming more widely adopted, the lack of security on such TAMs may expose ICs from adversaries utilizing standardized TAMs as an attack surface. Such saboteours may
use the TAM as an attack surface to compromise IC internal storage elements inclusive of embedded memories.

In section 4.6, a verification method is proposed to address the lack of use of an advanced verification environment such as Universal Verification Methodology (UVM) [113] for the verification of on-chip IJTAG networks. With the proposed method, access to various nodes or test instrument registers in the IJTAG network and relevant operational sequences can now be randomized with verification stimuli. Thus, with more robust verification, potentially less exploitable design errors may be present in an IC.

However, robust verification alone may not be sufficient if an IC does not contain security countermeasures against unauthorized access attempts on the TAM. A variety of security measures exist in the literature addressing JTAG based authentication [28], IJTAG network protection by generating network reconfiguration complexity [34, 121], and various forms of IJTAG data obfuscation and ciphering schemes [25, 26, 30, 32, 33, 98]. Prior works have not leveraged the IJTAG topology in conjunction with cryptographic protection. The work proposed in Chapter 5 is one of the first to introduce a cryptographic IJTAG protection scheme involving three defensive echelons operating in unison. The three echelons imposed on the IJTAG network utilize dual independently operating ciphers, IJTAG chain stubbing, and IJTAG access intent checking [57].

Deploying effective IJTAG authentication [28, 98] and encryption [57] still may not prevent a man-in-the-middle attack. For instance, an adversary can record a device authentication sequence and replay it to the device under attack to trick it to grant access permission. Alternatively, a device transmitting data to and back from a host may have their communication intercepted and manipulated by an attacker. The attacker can then exploit this channel to potentially compromise the host as well as the device. Even an encrypted authentication sequence may still be intercepted and replayed when there are no adequate countermeasures. Prior to the work proposed in Chapter 6, such man-in-the-middle threat scenarios may not be sufficiently addressed in the literature as a data integrity checking mechanism guarding against man-in-the-middle scenario was not available [10, 22, 34, 44, 69, 89, 90, 98, 121].
work in Chapter 6 enhances IJTAG data integrity checking using a secure hash algorithm to compel anyone accessing the protected IC to demonstrate knowledge of all prior legal communication to the IC. This means a man-in-the-middle replay attack could not generate the checksum that the device would authenticate. However, when the server, or authority, containing knowledge of all prior legal communication to the device is compromised, then the device access may be compromised. Hence, the device communication knowledge should be stored in a distributed manner, which would then allow for multiple servers to self-check against each other prior to accessing the device. The proposed solution in this work is one of the first to demonstrate a chained integrity checking mechanism for IJTAG.

1.2.3. Solutions to Verify IC Memory Security and Reliability Features

At the processor level, this work proposes one approach to verify on-chip virtualization security and another approach aimed at the verification of reliability handling capability. The on-chip virtualization security examined here include processor features that aim to isolate virtual machines, or caches, pages, and/or memory from unauthorized host or virtual machine access. The features targeted by this dissertation include reliability, availability, and serviceability (RAS) features implemented for server system processors affecting on-chip memories.

In Chapter 7, an automated systematic assembly test generation scheme aimed at stressing processor core secure virtualization features is proposed. Prior to the proposed solution, a labor intensive manual coding approach had been deployed.

In Chapter 8, a chip level RAS verification methodology targeting server processors with an integrated network-on-chip (NoC) is proposed. Existing literature primarily focuses on error injection and handling for design verification closure [111, 112]. However, given the presence of a NoC on chip, these existing approaches may not be sufficient. The work in Chapter 8 offers a novel verification approach examining NoC error propagation along with SRAM level RAS error injection to stress the RAS features implemented on-chip.
1.3. **Dissertation Organization**

The rest of this dissertation is organized into nine chapters. Chapter 2 proposes a novel method of securing one of the common building blocks for modern System-On-Chips (SoCs), static random access memory (SRAM), from Hardware Trojan attacks at the IP level with a design verification based technique. Contrasting with Chapter 2, Chapter 3 proposes and examines a novel circuit level technique to simultaneously enhance the reliability and security of embedded on-chip SRAMs. Starting from Chapter 4, this work begins to focus on SoC or full chip level security with an examination of practices for SoC level IJTAG design, architecture, verification, and test generation in an industrial case. Chapters 5 and 6 then propose two novel methods that can be utilized to protect on-chip Test and Debug access instruments from adversarial or malicious attempts that may lead to a breach of confidential device or system data. Approaches provided in these two chapters are certainly applicable to protecting embedded memories other than SRAMs. Chapters 7 and 8 then propose two novel methods to ensure SoCs are secure and reliable at the design verification level. Chapter 7 illustrates an instruction level method that enables verification testing of hardware based Virtual Machine security for multi-core and multi-threaded processor designs. In Chapter 8, a verification method is proposed to ensure that on-chip reliability circuitry works as intended for server processors. Chapter 9 introduces a novel memory manufacturing test optimization and data collection apparatus. Chapter 10 offers conclusions and identifies needs for further research.
Chapter 2

Protecting SRAMs Against Hardware Trojans Attacks at the Component Level With X-Propagation

2.1. Overview

The content of this Chapter is primarily sourced from a paper first published in 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) [54]. This chapter proposes novel HTH design and insertion methods targeting SRAMs — see section 2.4. Subsequently, the novel concept of X-triggering a HTH is proposed in section 2.5 as a countermeasure against SRAM core level HTHs.

2.2. Hardware Trojan Introduction

As the complexity of semiconductor designs has increased, and as the design and manufacturing process has become fractured across multiple companies and and multiple countries, semiconductor devices have become increasingly susceptible to attack. For example, the cost of manufacturing these devices at successively smaller technology nodes has led to the rise of third-party fabrication facilities, where devices are often manufactured out of the direct control of the chip designer—or even the chip designer’s home country [93]. This loss of control provides an opportunity for malicious modification of the masks and corresponding changes in device operation.

The outsourcing trend is not limited to device manufacturing. Chip design firms are reducing their overall design cost through Intellectual Property (IP) outsourcing as well. This has led to the rise of third-party IP providers. A common example is the licensing of processor cores or instruction sets [60]. Such IPs may be designed to contain malicious hidden functionality—especially if they are created by untrustworthy agents. Furthermore,
even when a design is created entirely “in-house,” the internal design staff could be infiltrated by adversaries who try to insert hidden functionality into a design.

Thus, there are at least three vulnerable avenues (third-party manufacturing, third-party IP providers, and staffing vulnerability) through which an adversary can maliciously alter the functionality of a device so that it deviates from its intended use. Such alterations are called Hardware Trojan Horses (or simply Trojans) and may lead to harmful effects such as Denial of Service or the leakage of sensitive information in mission critical systems. The development of methods to detect these Trojans is imperative for the design and manufacture of trusted electronic systems.

SRAM cores are used to implement key system components in a System on Chip (SoC), such as processor caches, buffers, queues, lookup tables, and many more. At the architectural level of the design, they generally are modeled by arrays. A Hardware Trojan Horse in one of these SRAM arrays or its control logic could significantly impact system performance and correctness in a multitude of ways. In addition, on-chip security units, such as Cryptographic Cores, may also utilize SRAMs. For example, SRAMs are used in open source Cryptographic Cores [85]. Thus, securing SRAMs from Trojan infection is mandatory.

This chapter explores some mechanisms whereby SRAMs can be compromised by Hardware Trojans and how such Trojans can be detected effectively by simulation-based verification during the RTL design phase. In particular, we will show that the amount of simulation-based verification traditionally used in industry to detect design errors in SRAMs may be inadequate for the detection of Hardware Trojans. We then show how the insertion of X’s into the simulation environment can promote the observation of Trojan behavior—leading to significant improvements in the overall detection of hardware Trojans with reasonable simulation times.

The rest of this chapter is organized in the following manner. We discuss current practice for the verification of SRAMs and define our problem in Section II. We then examine the design of several potential Denial of Service (DoS) Trojans in on-chip SRAM circuits in Section III. We describe the concept of X-Triggering for SRAM Trojan detections in Section...
IV. This is followed by three proposed X-Triggering-based Trojan detection algorithms. In section V, our experiments show that the X-Triggering-based techniques can be far more efficient than standard simulation-based Design Verification tests. We conclude in Section VI.

2.3. Background and Previous Work

2.3.1. SRAM Verification Approaches

For a modern System-On-Chip (SoC), there are generally two types of SRAMs: compiled SRAMs and custom designed SRAMs. Compiled SRAMs are tool-generated based on a core memory circuit [92,117]. In contrast, custom SRAMs are typically designed to meet specific demands in an SoC micro-architecture for which compiled SRAMs are inadequate. Custom SRAMs can be internally developed by a firm, provided by the external manufacturer, or supplied as third-party IPs. In any case, the SRAMs must be verified—especially when supplied as third-party IP.

Simulation-based SRAM verification plans are typically executed at the transistor level and Register Transfer Level (RTL). However, verification at the transistor level is generally restricted to SRAM cells due to the prohibitive amount of computing required to simulate a large design at the transistor level. Therefore, an SRAM transistor netlist needs to be converted to high level model, such as a Verilog model, to enable large scaled simulation-based verification.

Formal approaches have also been applied to SRAM verification [17]. Existing tools generally extract a logic level model of the SRAM from the transistor level model to enable Logic Equivalence Checking (LEC) to a golden reference. Alternatively, model checking can be applied to a high level model of the SRAM [23]. Bryant et al. reported a formal verification-based switch level approach where X’s are applied to address lines for address pruning and SRAM bit cells primarily for SRAM read operation checking [16].
Formal techniques are advantageous because they can mathematically prove that certain properties of a design are met. However, in some cases, obtaining a full logical representation of the SRAM model can be difficult. In addition, although simulation-based checkers can be easily ported to the full SoC Verification environment, formal verification cannot be ported from the IP level to the level of the full SoC.

Another advantage of simulation-based verification in the case of SRAMs is in the need for regression testing. Regression testing allows a design to be re-verified on an ongoing basis to determine if any changes to the design (e.g. bug fixes) have caused issues in a previously verified module. In the presence of a malicious insider, regression testing could help to detect any changes inserted after the initial verification stage. In addition, compiled SRAMs are often not generated only once. If the process technology changes, if updates are needed, or if the IP supplier issues an errata, then regeneration is necessary. In such a case, the SRAM will need to be re-verified. Thus, although formal techniques can be an important tool for SRAM verification, simulation-based approaches remain an important standard approach in industry.

2.3.2. Trojan Detection in SRAMs

To the best of our knowledge, the existing SRAM verification literature does not address detecting SRAM-based Trojans as part of the verification flow. However, there does exist a significant body of work on Trojan insertion and detection in general circuits (e.g. [52, 101, 118, 120]). Unfortunately, these methods do not necessarily map well to the detection of Trojans in SRAMs.

One seemingly obvious approach for the detection of Trojans in SRAMs is formal verification. However, formal verification is often impractical for SRAMs for many of the reasons stated earlier. Furthermore, there is an additional complication with respect to Trojan detection. Specifically, using LEC-Formal Verification to verify SRAMs would require a trustworthy reference SRAM model. However, if the supplier is not trusted, then clearly the reference SRAM model delivered with that SRAM also cannot be trusted. In [101],
the authors created a second *spec* version of the circuit being verified so that they could use equivalence checking to help detect Trojans in general logic. However, creating another SRAM model specifically to serve as the golden reference would counteract much of the time-saving advantages of using third-party IP, and thus is not an ideal solution.

Model Checking could be applied, but Model Checking has pitfalls. It does not scale to large SOC designs [12] and thus is less useful when the SRAM is incorporated as part of a large SoC during verification. It also requires a highly structural logic level SRAM model, but SRAM models are often modeled with behavioral code. Another drawback is that Model Checking requires substantial property writing for every single SRAM, and if the correct properties are not written to verify the security of the design, even a design that appears to have been “verified” may still contain Trojans. Finally, Model Checking may not converge.

ATPG-based techniques were used to detect Trojans in [118]. However, modern commercial ATPG tools require an ATPG internal gate level representation of the SRAMs to operate on them. This means an SRAM must be converted into an ATPG consumable model before ATPG-based Trojan detection techniques can be applied. This poses two important problems. First, we have not yet seen any credible mechanism for converting an SRAM model to an ATPG-based model in a trusted manner. Secondly, representing the SoC containing the SRAMs at the gate level for regression verification could require that the SoC RTL model be re-synthesized multiple times per day to keep the ATPG version up-to-date. This is highly impractical.

Existing simulation-based work utilizes simulation traces and statistics logged as Coverages to enable pruning of the Trojan search space [120] for formal verification or ATPG-based approaches. However, in a large SoC design verification environment, simulation traces are often out of date and difficult to mine. Furthermore, as discussed earlier, formal verification or ATPG techniques are not amenable for SRAM Trojan detections, curtailing the applicability of [120].
Finally, physical measurement approaches such as [52] are not applicable to Trojans inserted at the design stage. They are used to detect Trojans inserted at manufacturing and involve the extensive collection of physical data, such as delay, on real chips.

2.3.3. Novelty

Of the work we have surveyed so far, [120] and [16] are the closest to our approach. The approach in [120] considers Trojan detection, but it has not been shown to be effective on SRAM circuits; furthermore, it is not easily regressible. The authors of [16] utilize simulation to achieve formal verification results for the verification of SRAMs. This approach was not designed with Trojan detection in mind. In addition, the simulation target is switch level parallelized simulation, which means it simulates one read or write operation at a time. This means it will not account for sequential Trojans in SRAMs at all. In contrast, our work focuses specifically on SRAM verification, and specifically how slight changes to the simulation procedure can allow a particular class of SRAM Trojans to be discovered without needing to find the exact trigger condition. Instead, we harness the behavior of unknown values, X’s, in SRAMs to allow Trojan behavior to be manifested even when the exact trigger condition is not found.

2.4. Trojan Design and Insertion

For this research, three Trojan circuits were designed. The first circuit models a combinationally keyword triggered Trojan, Trojan_1, where on trigger, the trigger state is stored and the payload is always active post Trojan activation. The second Trojan circuit, Trojan_2, is triggered sequentially and is enabled by an attacker code word. The third Trojan circuit, Trojan_3 is a pure combinational circuit with no sequential elements. These three Trojans were designed to minimize the chances of detection by verification simulation tests.

For Trojan insertion, we consider the simple SRAM model shown in Fig. 2.1. The green region corresponds to the SRAM bit cells and their respective structural organization. The orange regions correspond to the interface, control, and wrapping logic around the SRAM’s
storage elements. The blue arrows represent control flows or data flows in the SRAM. The yellow portion highlights ports on the boundary of the SRAM. Note this is a simple view of an SRAM; modern SRAM circuits may contain scan chains, redundant cells, repair configuration circuitry, error correction code circuitry, power controlling circuitry, and clock controlling circuitry. We assume the SRAM to be clocked even though a clock pin is not explicitly shown in Fig. 2.1. The orange regions are targeted for Trojan infection in this work.

2.4.1. Trojan 1


trojan_1

and its attack mechanisms are represented in Fig. 2.2. The green portions of Fig. 2.2 are assumed to be the control and/or data vectors from non-Trojan inserted regions of the design. trojan_1 contains five Sub-Trojan circuits. Each Sub-Trojan has its own trigger, payload, and input bus. For example, Sub-Trojan 1 consists of Input 1, Trigger 1, and Payload 1. For each Sub-Trojan, the input vector will be compared against its respective Trojan key, and when the input matches the key, the Sub-Trojan will be triggered. On
triggering of each Sub-Trojan, its payload will be delivered.

For *Trojan_1* we have designed two types of payload: tri-state buffering disablement and input scrambling. *Sub-Trojan 1* will buffer a portion of *Input 1* and deliver it transparently when the Sub-Trojan is not triggered. However, when *Sub-Trojan 1* is triggered, it will pull-down the outputs, leading to design malfunction. *Sub-Trojan 3* is similar to *Sub-Trojan 1* except that its payload has pull-up behavior. Like *Sub-Trojan 1*, *Sub-Trojan 2* will buffer some or all of *Input 2* and will deliver it transparently for normal operations when the Sub-Trojan is not active. However, when triggered, it will scramble the data it buffers to disrupt functional behavior. Here, scrambling involves the logical XOR’ing of the buffered data with outputs of an LFSR. *Sub-Trojans 4 and 5* are similar to *Sub-Trojan 2*, but they vary in the size of the output data buffered, and hence the size of the scrambler. For all *Sub-Trojans 1-5*, once each one of them is triggered, the respective Sub-Trojan will remain triggered until a reset event. This means that the *Trojan_1* triggers are latched or registered. *Trojan_1* is implemented to model a Trojan that allows multiple DoS attack mechanisms. The scrambling aspect of the Trojan payload is to model faulty design operating conditions.

2.4.2. Trojan 2

*Trojan_2* is designed to obfuscate potential silicon attack debug. In contrast with *Trojan_1*, *Trojan_2* contains only one Trojan attack mechanism. An overview of the *Trojan_2* attack mechanism is encapsulated in Fig. 2.3. The green portion of Fig. 2.3 represents design inputs from uninfected regions. The red portion shows the Trojan attack scheme. *Trojan_2* buffers a portion of the Input Data and delivers that data transparently to the appropriate location when the Trojan is not active. When the Trojan is active, a corrupted copy of the data is delivered. To create obfuscation for silicon debug of the error, Trojan activation is divided into two stages. In the first stage, *Input Data* is compared to a *Enable Key* to enable an 8-bit LFSR, where its seed is a portion of the *Input Data*. It is when this 8-bit pseudo-random number matches a final *Trigger Key* that the Payload in *Trojan_2* is delivered. As the LFSR number changes, the payload will once again be disabled. However, once *Trojan_2*
is enabled with the *Enable Key*, the Trojan itself, including LFSR cycling, remains enabled until a power-on-reset event.

### 2.4.3. Trojan 3

*Trojan_3* is purely combinatorial and contains no sequential elements. The goal of this Trojan is to hide the triggering logic in a set of legal logic statements such that no trigger signal is present that will be flagged during simulation for a lack of toggle coverage. Figs. 2.4 and 2.5 provide some example pseudo-code. Fig. 2.4 shows the original circuit, which contains
a 32-bit variable named “data” that is assigned (in this example) to the values of the inputs. In the Trojan circuit, shown in Fig. 2.5, a new value, “GoodData” is introduced, and the inputs are originally assigned to this new value. If the value of “GoodData” matches a predefined constant, TrojanKeyword, then the least significant bit of the “GoodData” variable is flipped before it is assigned to data. Because the triggering of the bit-flipping payload is encapsulated in a logical statement, there is no specific trigger variable that will have low toggle coverage. (Note that we assume that the TrojanKeyword is a constant and not a variable.) The exact details of this Trojan’s implementation, including which signals are affected or used for comparison, will vary from one SRAM to another.

//Data variable assignment without Trojan:
assign data = inputs;

Figure 2.4: PseudoCode for Variable Assignment in SRAM without Trojan

2.4.4. Trojan Insertion
These three Trojans are inserted into four SRAM RTL models with some variations. The first two SRAM models are from the publicly available OpenSPARC project [86], and these two SRAMs are custom designed SRAMs. The latter two are proprietary and used in state-of-the-art server processors, and these two SRAMs are compiled memories.

The first SRAM, $RAM_1$, is used in the Trap Logic Unit of the OpenSPARC or SPARC T2 processor as part of Trap Stack Array (TSA). The purpose of the TSA is to hold and buffer processor core exception reports. The state of TSA can indirectly affect how processor instructions are committed or processed. If a Trojan is activated in the TSA, the processor operating state will be corrupted and can lead to system failure. $RAM_1$ has 32 entries with 152-bit data width.

The second SRAM, $RAM_2$, is used in the Network Interface of the OpenSPARC processor. The purpose of this SRAM is to help translate an IP address to a MAC address. $RAM_2$ has 128 entries with 200-bit data width.

The third SRAM, $RAM_3$, is from the SRAM library of a proprietary processor. It is used as caches, building blocks of FIFOs, and buffering units across multiple major units of a production Server SoC. $RAM_3$ has 128 entries with 67-bit data width.

The fourth SRAM, $RAM_4$, is used as data caches for L2 and L3 components of a proprietary processor. When the data in such an SRAM is corrupted by a Trojan, that data could make its way into core pipelines and cause hard-to-detect errors that may not be easily handled by Operating System (OS) and processor exception handling logic. $RAM_4$ has 8192 entries with 72-bit data width.

---

**Figure 2.5:** PseudoCode for Variable Assignment in SRAM with $Trojan_3$

```c
//Data variable assignment with Trojan:
assign GoodData = inputs;
assign data[0] = GoodData[0] ^
    (GoodData[31:0] == TrojanKeyword);
assign data[31:1] = GoodData[31:1];
```
For \textit{RAM}_1, all three Trojan were inserted in the Data\_IN Buffer/Logic region. More specifically, \textit{Trojan}_1 and \textit{Trojan}_2 are placed in the middle of input data buffers. In effect, a portion of input data is rerouted to now traverse through \textit{Trojan}_1 and \textit{Trojan}_2. The payload affects the data being transacted into \textit{RAM}_1 SRAM cells. \textit{Trojan}_3, sources buffers of Data\_IN for Trojan Key comparison. When triggered, \textit{Trojan}_3 flips bits on the write data path into SRAM cells.

For \textit{RAM}_2, a portion of Data\_IN Buffer/Logic, Write Control Logic, and contents in the Address Decoder are used by \textit{Trojan}_1 and \textit{Trojan}_2 as the source for Trojan Key compares. For \textit{Trojan}_1 and \textit{Trojan}_2, the payload disables a global write enable signal in the Write Control Logic region. For \textit{Trojan}_3, buses in the Data\_IN Buffer/Logic, Write Control Logic, and Address Decoder are used for triggering, and when triggered, \textit{Trojan}_3 flips a bit in the Data\_IN bus to the SRAM cells.

For \textit{RAM}_3, buses in Data\_Out Buffer/Logic region are used for triggering \textit{Trojan}_1, \textit{Trojan}_2, and \textit{Trojan}_3. This means Trojans can only be triggered during SRAM read operations. For \textit{Trojan}_1 and \textit{Trojan}_2, the payload will corrupt up to half of SRAM read data. For \textit{Trojan}_3, the payload will flip the least significant bit (LSB) of the read data bus.

For \textit{RAM}_4, all three Trojans were inserted in a manner similar to to \textit{RAM}_1.

2.5. \textit{X}-Triggering Of Trojan

One of the problems with Trojan detection through simulation in general is the need to activate the Trojan by finding the correct trigger. In the results section, we will show that standard simulation-based verification techniques, that force the SRAM array to be initialized to known logic values at the beginning of the simulation, are often unable to find a well-hidden trigger. However, we will show that inserting and propagating X’s through the simulation environment can help detect the types of Trojans described in the previous section.

There has been a substantial body of work on the verification of SRAMs. To our knowledge, [16] is the most relevant to our work. The authors of [16] also use unknown’s, or X’s,
for search space pruning for SRAM verification. However, although our work is related, it is not identical. In [16], the verification targets are switch level SRAM circuits. Our work is concerned with logic level or RTL SRAM models. They also simulate one read or write operation at a time, and, as already stated, this will not account for sequential Trojans. Finally, in [16], the author is mostly concerned with address space pruning through the utilization of X’s, whereas in this chapter, the search space being pruned is the Trojan trigger code word space. In the rest of this section, the concept of Trojan X-Triggering is defined, the application of X-Triggering is explored, and the subsequent Trojan detection algorithms are defined and examined.

**Definition 2.1** For an SRAM $R_1$ with number of entries being $N$, the address index being $Index$, and the number of bits stored in each entry being $M$, we observe the following three properties.

Property 1. In Verilog, a combinational logical operator such as “==”, “XOR”, and “XNOR” will evaluate to X’s when either operand of the operator is set to X’s [48]. This means that when an input to an XOR/XNOR gate that performs a comparison with the pre-selected Trojan keyword—and thus triggers the Trojan—is equal to X, then the output of the XOR/XNOR (the trigger control line) is equal to X as well. We call this effect X-Triggering.

Property 2. We observe that it is legal to write an $M$-bit vector of X’s ($M\{X\}'s\}$ to each of the $N$ entries in $R_1$ and proceed to perform binary valued writes and reads to any arbitrary entry $Index$ with $Index>0$ and $Index<\leq N$.

Property 3. Read or write operations on any legal entry $R_1[Index]$ must not affect any $R_1$ entries $\neq Index$.

**Theorem 2.1** Writing $M\{1'bX\}$ to entries 1 to $N$, followed by read operations from entries 1 to $N$ in $R_1$, will X-Trigger the target Trojan when the trigger comparator is modeled with logical operators and depends on any data bit combinations as inputs. (Note that it could depend on address bits as well. It does not have to depend solely on data bits.) If the payload
of the corresponding Trojan is X-propagating within the simulation time to observable circuit locations, and if final target circuit responses are checked with the case operator (=== in verilog), where the evaluation results will be set to logical 0 or 1 even when either of the input operands is set to X, then these types of Trojans can be detected with mechanisms for X-Triggering.

Proof: As a result of Property 1, when the entire address space has been written to X, any Trojan trigger that taps into the write data path with any write data combination will be X-Triggered. (Note selected address bits could be part of the Trojan trigger as well. It does not have to depend solely on data bits because the entire address space is covered.) Similarly, if the target Trojan depends on any read data combinations, then it will be X-Triggered as well. Given that we can now X-Trigger such Trojans, if the payload is X-sensitive, then utilizing case operators, where X, Z, 1'b1, and 1'b0 can be distinguished for logical comparisons, the effect of the X-Triggering can then be propagated to the circuit outputs if it is not masked during the simulation time. This allows mis-comparisons to be generated when expected known values are compared with unknown X’s. Given property 2 and 3 showing us that such operations are legal, we have shown that such Trojans can be detected.

Lemma 2.2 X-Triggering does not provide detection of all Trojans. It is possible that a Trojan may be X-Triggered, but if the payload is not X-propagating, then there won’t be a detection. Similarly, we can attempt to X-Trigger a Trojan, but if the trigger circuit itself is not X-propagating, then detection is not possible with this method.

We show an example in Fig. 2.6 to illustrate the power of X-Triggering in SRAM Trojan detection. The XVector propagates to the trigger circuit first leading the trigger to evaluate to X. The objective is to cause the compare operation between the Trojan Key and the input vector of X’s to evaluate to X. This value then propagates to the payload control logic, and in turn causes the payload to evaluate to X’s. Due to property 2 and Theorem 1, we can
now reinitialize the entire SRAM and no $X$’s should appear on correct SRAM readouts, but if the SRAM initialization cannot clear away the inserted $X$-Triggered Trojan data (such as when the Trojan remains active once enabled), then the Trojan can be detected. This is not necessarily true in a logic circuit such as a finite state machine (FSM). In a FSM, once the inputs are set to $X$’s, the FSM states will most likely transition to $X$’s. Depending on the machine, the only way to clear the $X$’s in FSM may be to reset the FSM. This means that an $X$-Triggered Trojan payload may not be distinguishable from correct circuit behavior with any guarantee in an FSM. For SRAM circuits, the desired behavior is inherent.

![Figure 2.6: X-Triggering Example](image)

With a given bounding on $X$-Triggering, we now propose three checking algorithms exploiting $X$-Triggering for Trojan detection in SRAM circuits. The first algorithm is based on SRAM initialization. The second algorithm aims to $X$-Trigger the Trojan during random SRAM accesses. The final algorithm aims to detect purely combinational SRAM Trojans.

Algorithm 1’s pseudo-code is in Fig. 2.7. The goal is to $X$-Trigger Trojans in the design that have write/read data dependencies. Thus, the first step is to sanity-check the SRAM with basic initializations (lines 1 to 3). The second and the third steps (lines 4 to 6 and
Algorithm 1

1: for all Words in SRAM 1 do
2: initialize each word to either all 1’s or all 0’s and qualify initialization with valid READ operation on each entry
3: end for
4: for all Words in SRAM 1 do
5: WRITE X’s to all bits for each word
6: end for
7: for all Words in SRAM 1 do
8: Execute unqualified READ Operation on each word
9: end for
10: for i = 1 to N (where N is a user-defined # of iterations) do
11: Execute a write operation on a random legal address with random data followed by a qualified read operation on the same address with the case compare operator
12: end for

Figure 2.7: Algorithm 1

7 to 9) attempt Trojan X-Triggering on all the words in the SRAM during write and read operations. Given that a potential Trojan may now be X-Triggered, the fourth step (lines 10 to 12) aims to detect the potential Trojan.

This algorithm has some limitations. In step 4, when referring to qualification, we refer to a checking mechanism whereby the design under test (DUT) is considered to pass a given verification test or not. Unfortunately, it may not be able to detect a combinational Trojan such as Trojan_3. Secondly, it’s possible that a sequential Trojan may only be Triggered for a short period of time and will revert back to an un-triggered state. In this case then Algorithm 1 may miss the detection time window for such Trojan since it needs to iterate through every address at least twice during X-Triggering. (Note that a Trojan that does not become X-propagating until after a long delay could also fail to be detected.) Therefore,
Algorithm 2 and Algorithm 3 are designed to remedy Algorithm 1.

(Note that if we remove lines 4 to 9 from Fig. 2.7, we arrive at a model for randomized verification testing that is standard practice in design verification. We refer to this randomized model in Section V.)

Algorithm 2’s pseudo-code is shown in Fig. 2.8. The key idea is to shorten the X-Triggering time window from the scale of entire address space to one address at a time. In a practical verification environment, we expect this X-Triggering window to be randomized.

**Algorithm 2**

1: for Each word in SRAM 1 do
2:   initialize each word to either all 1’s or all 0’s with WRITE operations
3: end for
4: for Each word in SRAM 1 do
5:   Execute a qualified READ operation
6: end for
7: for i = 1 to N (where N is a user-defined # of iterations) do
8:   Execute Randomized Access in the following manner for random entry A in SRAM 1:
9:   WRITE vector of X’s to entry A
10:  Perform unqualified READ on entry A
11:  WRITE randomized data consisting of 0’s and 1’s to entry A
12:  Perform qualified READ operation on entry A
13: end for

Figure 2.8: Algorithm 2

Algorithm 3’s pseudo-code is shown in Figs. 2.9 and 2.10. It compensates for the lack of combinational Trojan detection because it allows the writing/reading of each bit to be qualified immediately. The idea is to X-Trigger a combinational Trojan and X-propagate its effects to its respective payload for checking. Algorithm 3 is also deterministic, whereas the
computational complexity for Algorithm 1 and Algorithm 2 are largely user bound as the user determines the number SRAM access iterations. Algorithm 3’s cost can be expressed as $N \times (4M + 1)$ SRAM read/write operations.

*Algorithm 3*

1: for Each word in SRAM 1 do
2: initialize the word to either all 1’s or all 0’s with WRITE operations
3: end for
4: for Each word in SRAM 1 do
5: Apply a walking-0 access sequence and a walking-1 access sequence
6: end for

Figure 2.9: Algorithm 3

*Walking 0(1) Access Sequence*

For each bit in the Word Vector

*Set the bit to a logical 0(1) while setting the rest of the bits to X’s.*

*Write the Word Vector to the SRAM word being targeted*

*Read out the word and qualify the read out data*

Figure 2.10: Walking 0 or 1 Access Sequence

2.6. Experimental Results

For this work, we implemented an RTL SystemVerilog based TestBench. In this TestBench, SRAM models are instantiated and bound to SystemVerilog checkers via SystemVerilog interfaces. In short, this implementation is easily portable to a modern SoC verification
environment because the constructs employed are consistent with industry standard practice and methodology [113]. We use the number of simulation cycles as a metric for performance evaluation. We select this metric because modern verification environments typically employ a key metric, cycles per second (CPS), to measure the efficiency of simulation or emulation performance. Therefore, simulation cycles are independent of machine performance and along with the CPS of a given simulation environment indicate the total runtime.

We implemented all three verification algorithms described in the previous section in a parameterized SystemVerilog based SRAM checker. We implemented a built-in-self-test (BIST) algorithm and SRAM initialization algorithm in this checker as well. This checker employs the SystemVerilog concept of ScoreBoards to enable qualification of SRAM read operations.

As comparison for our proposed algorithms, we model existing simulation based approaches for SRAM verification with randomized testing and simplified BIST testing. There are three main types of simulation-based verification testing: directed functional testing, randomized testing, and BIST based testing. With respect to SRAMs, BIST testing and directed testing are similar. From our experience, these three approaches are likely to be applied to verify SRAMs in an industrial design setting. To this end, we chose Random and BIST testing as our reference point. We consider BIST to be sufficient replacement for directed testing. We used five million simulation cycles as bounds for random testing. For Random SRAM verification tests, our implementation is similar to Fig. 2.7 except for the exclusion of SRAM initialization of X’s, which is contained from lines 4 to 9 in Fig. 2.7. For BIST testing to model directed tests, we implemented a BIST 6N algorithm for stimulating and checking SRAMs [114]. Pseudo-code of the 6N algorithm is given in Fig. 2.11. This algorithm consists of three march elements and executes six read or write operations for each address in the SRAM. Typically for a single BIST algorithm execution, a BIST Engine utilizes expansion and variation of a limited set of data background patterns.

We first show that our Trojan detection schemes will not cause false positives. In other words, a good Trojan detection scheme, especially one that utilizes X’s, should not flag
**BIST 6N**

1: for Each word in SRAM 1 do
2: initialize the word to either all 1’s or all 0’s with WRITE operations
3: end for
4: Let N be the total number of addresses in SRAM 1
5: Let P be a data background pattern
6: for int i=1; i <= N; i++ do
7: WRITE complement of P to SRAM 1[i]
8: end for
9: for int i=1; i <= N; i++ do
10: READ (SRAM 1[i]) === (complement of P)
11: WRITE P to SRAM 1[i]
12: end for
13: for int i=N; i > 0; i– do
14: READ(SRAM 1[i]) === P
15: WRITE complement of P to SRAM 1[i]
16: READ (SRAM 1[i]) === (complement of P)
17: end for

Figure 2.11: BIST 6N Pseudo-code

failures in an SRAM that does not contain Trojans. The results are shown in Table 2.1. The first row of Table 2.1 shows column labels. RAND indicates randomized verification tests. BIST 6N indicates a 6N BIST algorithm based SRAM Test. Columns four to six indicate Algorithms 1–3 based verification tests. The first column, SRAMs, labels the SRAM instance under test. For each SRAM instance, we document two data points: 1) whether a particular verification test passed or failed, and 2) the number of simulation cycles. Note that these results in Table 2.1 also present the baseline results for uninfected SRAM models.

In these experiments, we establish an upper bound of five million cycles for any random-
ized simulation run. Assuming a design environment having an average CPS of one hundred, five million cycles would cost roughly fourteen hours of compute time. Such a CPS number is optimistic; for complex modern designs, this number will often range from the single digits to triple digits. The BIST 6N algorithm and Algorithm 3 are deterministic, and therefore a pre-selected simulation limitation does not apply. For Algorithms 1 and 2, we set a simulation cycle limit of 10000 for SRAMs 1, 2, and 3, and 100000 for SRAM 4. The rationale is to allow a simulation window where every address in the SRAM has an opportunity to be accessed. SRAM 4 is the largest of the four SRAMs.

As shown in Table 2.1, all of the Trojan-free circuits successfully passed verification for all verification approaches. Thus, there were no false positives.

The verification results for SRAMs containing Trojans are shown in Table 2.2. The formatting for Table 2.2 is similar to that of Table 2.1. We consider an SRAM instance with exactly one type of Trojan inserted as being distinct. Therefore, we present results for every distinct SRAM instance accordingly.

Table 2.2 clearly shows that random testing and BIST 6N testing failed to identify any inserted Trojan. Algorithm 1 and Algorithm 2 were able to detect Trojan_1 and Trojan_2 infected SRAM instances. However, they failed to detect Trojan_3 infected SRAM instances. Algorithm 1 aims to X-Trigger the Trojan during initialization, and if the Trojan remains
triggered post initialization, then detection can be achieved. This is not the case with \textit{Trojan}_3. Algorithm 2 aims to introduce Trojan \textit{X}-Triggering dynamically during random testing. Algorithm 1 and Algorithm 2 achieve \textit{X}-Triggering by writing the entire word-line to \textit{X}'s. However, Algorithm 3 achieves combinational \textit{X}-Triggering by driving all bits in a word-line to \textit{X} except for one bit. Note that without iterating this bit, detection can not be guaranteed. This explains why Algorithm 3 was effective against \textit{Trojan}_3.

Performance-wise, we note that Algorithm 3 is consistently the most costly of the three proposed algorithms. Nonetheless, Algorithm 3 is also shown to be effective in detecting all three proposed Trojans. This is because even though it was designed to detect combinational Trojans, application of the Algorithm itself is sequential as well. It’s not clear which one of the three should be the prevailing Trojan detection mechanism. In a practical verification environment, all three algorithms can be utilized. To control the compute cost of Algorithm 3, regression with it can be executed on a limited set of random addresses. Assuming such regressions are run frequently and in parallel, functional coverages can be collected with each address defined as a coverage point. Thus, we can leverage functional coverage to determine test sufficiency.

2.7. Conclusion

In this chapter, we examined the motivation for understanding the threat of Trojan circuits in SRAM’s. We proposed three types of DoS Trojan for SRAMs. We then proposed a framework for Trojan detection based on \textit{X}-Triggering. We developed three algorithms based on \textit{X}-Triggering. We then showed their effectiveness in detecting the three proposed Trojans against standard simulation based verification methods. It’s important to note that these three algorithms are meant to serve as examples for more Trojan detection schemes. We analyzed experimental results and provided recommendations for developing verification tests utilizing \textit{X}-Triggering more specifically targeting DoS Trojans.

Our results clearly show that detection schemes based on \textit{X}-Triggering for the types of SRAM DoS Trojans studied are effective. Where regular simulation techniques failed, the
proposed algorithms succeeded. Our approach of using $X$-Triggering to produce simulation based verification tests and checkers yields easy to implement verification structures and test cases with bounded runtime.

Looking forward, we see SRAM leakage Trojan as another area for $X$-Triggering based algorithms. Other areas of future research involve detecting Trojans that are triggered on combinations of signal lines that do not include data or for which the $X$-propagation is significantly delayed. Also, we will investigate the application of $X$-Triggering to non-SRAM circuits. Finally, $X$-Triggering can fail to detect Trojans in designs with $X$-blocking constructs. In future work, we will aim to modify our algorithms to better detect Trojans hidden by such constructs.
Table 2.2: SRAM Simulation Results with Trojan

<table>
<thead>
<tr>
<th>RAMs</th>
<th>Pass/Fail</th>
<th>RAND</th>
<th>BIST6N</th>
<th>Algo1</th>
<th>Algo2</th>
<th>Algo3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM_1 Trojan 1</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>635</td>
<td>144</td>
<td>276</td>
<td>280</td>
<td></td>
</tr>
<tr>
<td>RAM_2 Trojan 1</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>3830</td>
<td>785</td>
<td>1583</td>
<td>130019</td>
<td></td>
</tr>
<tr>
<td>RAM_3 Trojan 1</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>2554</td>
<td>527</td>
<td>1043</td>
<td>1307</td>
<td></td>
</tr>
<tr>
<td>RAM_4 Trojan 1</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>163834</td>
<td>32772</td>
<td>65544</td>
<td>65839</td>
<td></td>
</tr>
<tr>
<td>RAM_1 Trojan 2</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>635</td>
<td>144</td>
<td>276</td>
<td>284</td>
<td></td>
</tr>
<tr>
<td>RAM_2 Trojan 2</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>3830</td>
<td>785</td>
<td>1583</td>
<td>130025</td>
<td></td>
</tr>
<tr>
<td>RAM_3 Trojan 2</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>2554</td>
<td>527</td>
<td>1043</td>
<td>1307</td>
<td></td>
</tr>
<tr>
<td>RAM_4 Trojan 2</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>163834</td>
<td>32772</td>
<td>65544</td>
<td>65839</td>
<td></td>
</tr>
<tr>
<td>RAM_1 Trojan 3</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>635</td>
<td>10001</td>
<td>10001</td>
<td>16268</td>
<td></td>
</tr>
<tr>
<td>RAM_2 Trojan 3</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>3830</td>
<td>10002</td>
<td>10008</td>
<td>130235</td>
<td></td>
</tr>
<tr>
<td>RAM_3 Trojan 3</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>2554</td>
<td>10000</td>
<td>10000</td>
<td>35343</td>
<td></td>
</tr>
<tr>
<td>RAM_4 Trojan 3</td>
<td>PASS/Fail</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>FAIL</td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td>5 Mil</td>
<td>163834</td>
<td>100000</td>
<td>100000</td>
<td>2424847</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3
Protecting SRAMs Against Hardware Trojans Attacks at the Component Level With CRC based Design Obfuscation

3.1. Overview

Chapter 2 is focused on detecting HTHs in SRAM cores during the design development stage. However, it’s possible that malicious HTHs got inserted into embedded memories during the manufacturing process. This chapter proposes a novel approach that leverages CRC data scrambling and self checking along with address scrambling to prevent such HTHs from being triggered deterministically while potentially enhancing device SRAM error checking with CRC — see section 3.4. The content in this chapter was first published at the 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) [59].

3.2. Introduction

One way to lower product development cost is the integration of third party intellectual properties into the design. This applies not only to logic cores, but to embedded memories as well. In particular, the usage of memories is so widespread that it has become common to leverage third party memory circuit designs into Systems on Chip (SoCs), processors, and other devices. Unfortunately, when companies do not control the design of third party SRAMs and/or SRAM IC manufacturing, security holes may be present in the final productized device.

For example, Hardware Trojan Horses (HTHs) consist of malicious circuitry inserted into a design at any stage in the design or manufacturing process, from the specification to final manufacturing and packaging [2,108]. HTHs may be designed to steal information,
insert errors, or otherwise prevent the affected circuit from operating correctly, and in some cases, they can destroy the affected chip. HTHs have become especially worrisome as the design and manufacturing process has splintered across multiple companies and countries. Detection and mitigation approaches depend on how and when the Trojan is inserted. For example, detection of Trojans inserted into RTL (Register Transfer Level) code at the design stage was studied in [9,54,120].

Many HTHs consist of a trigger and a payload. When a triggering event is seen, the payload (e.g. insertion of an error) is launched. The prevalence and importance of memories throughout modern processors and systems make memory Trojans especially important to handle. For example, on an HTH Trigger, data content modification may occur and cause errors. An HTH attack may also cause user software to enter into a secure memory region. Memory-based attacks are not uncommon in the software domain [15]. In fact, [15] documents that more than ten percent of overall computer system attacks are memory corruption attacks. Unfortunately, if the underlying hardware memory also lacks trustworthiness, many software security mechanisms may not provide adequate protection. SRAM HTH insertion and detection at the verification stage was explored in [54], but those techniques may miss some Trojans, and they cannot protect against Trojans that are already present in off-the-shelf standalone memory chips.

The overall reduction in memory cost on a “per bit” basis is also driven in part by technology advances that allow memories to be manufactured at smaller process nodes. However, the interaction of circuitry with charged particles (such as alpha particles, ionized radiation, and even neutrons because of the rise in secondary interactions) increases at smaller features sizes [72,100]. This can cause the occurrence of so-called single event effects (SEE), i.e. the unwanted change in voltage levels on sensitive nodes. The occurrence of single event effects has a particularly negative impact when the temporary change in voltage of a node (i.e. a glitch) affects and/or gets latched into a memory element such as a DRAM or an SRAM. Such a case is referred to as a Single Event Upset (SEU), or soft error.
In addition to the interaction with radiation, a concurrent trend that is giving rise to an increase in soft error rates in memories is the constant push towards ultra low power because of the ubiquitous presence of portable devices. Reduction in the refresh rates of DRAMs [70] and dynamic voltage scaling (DVS) to near threshold voltage in SRAM-based caches, such as with the drowsy cache paradigm [40], can cause the loss of information stored in the memory cells. These errors can also be modeled as SEUs [8] [29]. Although SEUs are not permanent hardware faults, the errors induced by them can cause disruptive effects and system crashes (imagine, just as an example, a SEU affecting the instruction scheduling queue). Therefore, it is very important to design systems able to correct or at least detect their occurrence.

Error correcting codes (ECC) are often incorporated in memories as standard practice to detect and hopefully correct these kinds of errors. Unfortunately, ECC may miss (or mis-correct) some errors. Furthermore, when the ECC is already included in memory provided by an untrusted third party, the ECC itself may be suspect. A saboteur could design Trojans that circumvent such protection or even disable the ECC protection when the Trojan is triggered. This makes it even more difficult to ensure the integrity of memory contents.

Thus, this chapter addresses two potential sources of errors for IC memories: security risks in the form of Hardware Trojan Horses (HTHs) and soft errors due to voltage scaling, device aging, and radiation-induced upsets. In particular, we propose a novel memory obfuscation and error detection solution that tackles both problems simultaneously:

- Cyclic Redundancy Codes (CRC) are used for scrambling and obfuscation of data written to the memory as well as the detection of errors that may have been missed by internal ECC checkers. The obfuscation ability of this method can help prevent deterministic triggering of Trojans while the error detection can help mitigate the effect of any Trojans that allow errors to get through the ECC layer. It can also protect against simple soft errors that are missed by ECC.

- A “swizzling” approach is used to obfuscate the address actually being accessed in the memory. This makes it harder to deterministically trigger a Trojan at a particular memory location.
The CRC and address swizzling are incorporated as a “wrapper” around and external to the memory IP. This is important because it reduces the chance that a malicious actor who provides compromised IP will be able to successfully remove all error checking and countermeasures from the design. He may be able to defeat the ECC within the compiled memory, but he should not have access to the wrapper design or implementation. This also means that our approach will not only work for embedded memories within a processor or SoC, but may also be used to protect memory chips on a board. Ideally the wrapper will be designed only by a trusted party.

3.3. Hamming and CRC coding

In this chapter, we assume that the memory array may implement a standard single error correction/double error detection (SEC-DED) coding method, such as Extended Hamming Codes. Extended Hamming Codes allow one to correct one error and detect two errors in the codeword [63].

We then add another layer so that words written to the memory are themselves CRC encoded. CRC is a very effective way to detect the occurrence of errors in data storage and communication [63]. CRC is an error-detecting cyclic linear code in the Galois Field GF(2) (i.e. each codeword can be obtained by a cyclic shift of another codeword). Starting from a k-bit message \( m \) represented in polynomial form by a degree \( (k-1) \) polynomial, \( m(x) \), CRC codewords are generated by the multiplication of \( m(x) \) with a generator polynomial \( g(x) \) of degree \( (n-k) \), which is the same amount of the added redundancy. Thus, \( c(x) = m(x)g(x) \) where \( c(x) \) is the representation of the codeword in polynomial form of degree \( (n-1) \).

A CRC encoded codeword has the ability to detect burst errors of length up to \( (n-k) \) adjacent bits. Decoding implies that, with the division of the encoded codeword by the generator polynomial, the resulting quotient is the decoded codeword when the remainder is zero. A non-zero remainder indicates the presence of an error. CRC codes can also be constructed in a systematic form so that the original k bits can be easily isolated in the codeword—thus simplifying the decoding operation. We decided to use non-systematic CRC
encoding to have codewords that don’t include the same patterns as the original messages—thus scrambling the data written into memory to avoid deterministic activation of possible Trojans.

3.4. Proposed Method

Our wrapper design approach’s first goal is obfuscation to make it difficult for a Trojan writer to have the Trojan trigger at a time of his choosing. To achieve this goal, we first employ a simple memory input address scrambler to shield the memory from address-driven HTH triggers. However, address obfuscation alone is insufficient because the memory’s data access bus may also serve to provide HTH trigger points. Furthermore, address and data obfuscation do not offer protection against any errors (whether arising from natural or intentional sources) that do manifest. Therefore, we also utilize CRC coding to scramble and encode the input data bus as well as function as a data integrity checker by detecting errors.

Figure 3.1 shows an overview of the wrapper design. It consists of the address swizzler, which scrambles addresses, and CRC encoders and decoders (denoted as GF2 Encoder and GF2Decoder) that allow the data stored in the memory to be stored in an obfuscated form. They will also allow errors missed by the internal memory ECC to be detected. The Configuration Register Bank determines the mapping and coding applied and allows the encoding details to change over time—providing additional obfuscation.

3.4.1. SRAM Address Swizzler

If security vulnerabilities such as HTHs are inserted into the SRAM, it is likely that they may have some triggers based on either combinational or sequential constructs [54]. Such triggers may have logical dependencies on the address selection. We note that embedded SRAMs typically have an exhaustible address space. Thus, we anticipate that data written to the SRAM and/or a history of previously accessed addresses may also be used as triggers or parts of triggers by potential HTHs to make detection during verification harder. To counter
this and make deterministic triggering of sequential HTHs by an attacker more difficult, we scramble addresses so that potential attackers may now need to exhaust the total address space multiple times in different orders to trigger malicious HTH behavior in the field. In addition, even a combinational HTH that was designed to trigger on a specific address or corner case may no longer do so.

We consider an address bus width $AWIDTH$. We denote the address bus as $addr_{bus}$. We refer to $addr_{bus}[AWIDTH-1:0]$ as a bundle of all wires in $addr_{bus}$. To refer to a particular wire in $addr_{bus}$, we use an integer variable $wire_{sel}$ where $0 \leq wire_{sel} \leq (AWIDTH-1)$ and $\geq 0$.

Our solution is to intercept the $addr_{bus}$ inputs to the memory on a write or read operation and rearrange the incoming $addr_{bus}$ signals such that a new order is imposed on the signals indexed in $addr_{bus}$. We call this kind of remapping “swizzling”.

We can design an $addr_{bus}$ swizzler by utilizing an $AWIDTH$-to-1 multiplexer on all $addr_{bus}$ bits we intercept to yield a single output $addr_{bus}$ bit. We can create an $AWIDTH$ number of such multiplexers to allow complete remapping of all signals in $addr_{bus}$. Note that each multiplexer will map to a bit position in the swizzled version of $addr_{bus}$. Some multiplexer outputs could also be inverted to increase the level of obfuscation. Now all select
line inputs to the \( \text{AWIDTH} \) copies of the \( \text{AWIDTH} \)-to-1 multiplexer need to be controlled. For this, we populate an array of size \( \text{AWIDTH} \) with all the multiplexer select line values. We call this array as \( \text{swizz\_map} \). It is contained within the the Config Reg Bank block in Figure 3.1. We ensure that all members of \( \text{swizz\_map} \) are non-repeating and are within the \( \text{addr\_bus} \) range. The output of the swizzled \( \text{addr\_bus} \) is sent to the memory we intend to secure during write and read operations. Note that \( \text{swizz\_map} \) generation could be done with either securely executed software or dedicated hardware.

3.4.2. SRAM Data Obfuscation and Error Detection

We also want to transform the data we send into the SRAM so that an HTH trigger condition does not get sent when the attacker intends. This helps protect against data-dependent triggers. Further, we want to self check the integrity of SRAM contents and protect against errors that may not be correctly handled by internal ECC. For example, an internal ECC checker in the memory may miscorrect when it attempts to correct multi-bit errors. An external ECC checker could miss such errors because they would appear to the checker to have been corrected already. Thus an external ECC checker is not a good choice for our wrapper design. ECC also computes a set of parity bits to be stored alongside SRAM data. This means HTH trigger inputs that use the SRAM data lines have a one-to-one mapping. Hence, ECC does not allow us to scramble data into the SRAM. However, we can utilize CRC principles as proposed in [63] to both encode SRAM data words and enable better self checking.

Because CRC operations are on GF(2), we can encode SRAM data words with GF(2) multiplication. Given an SRAM data bus \( DB \) with width of \( \text{DWIDTH} \), we can encode it with a CRC polynomial, labeled as \( \text{CRC\_Poly} \) with width of \( \text{PWIDTH} \). The operation was described earlier in Section 3.3 and namely: \( DB[\text{DWIDTH}−1 : 0] \times \text{CRC\_Poly}[\text{PWIDTH}−1 : 0] \) on GF(2) leading to resulting vector of width \( \text{DWIDTH}+\text{PWIDTH}−1 \). For example, given a 3-bit \( \text{CRC\_Poly} \) of 101, we can encode a 4 bit word of 1111 with the logic operations \( (001111 \ll 0) \oplus (001111 \ll 2) \) to get a codeword of 110011. We chose a combinational
logic-based approach to perform this operation in at most a few clock cycles rather than
time-consuming LFSR based logic.

GF(2) multiplication requires an outcome with \((PWIDTH-1)\) more bits than the original
data, and we must store these \((PWIDTH-1)\) bits along with data the size of the original
SRAM write content to enable successful CRC data integrity check and SRAM data recovery.
Because we are now concerned about not just the CRC remainders, we need to implement a
full GF(2) divider to recover the encoded data words, and when the remainder is non-zero,
a CRC error has been identified.

3.5. Experimental Results

To evaluate our proposed method, we simulated our circuit against a mature industry
SRAM design and examined the physical implementation aspects of our circuit.

3.5.1. Wrapper Design Implementation

A block diagram of our design and its relation to the target SRAM was shown in Fig.
3.1. We implemented three main RTL blocks in synthesizable SystemVerilog. The first block
is the address swizzler, shown in Fig. 3.1 as Addr Swizzler. The second block is the GF(2)
data encoder and CRC check vector generator, shown in Fig. 3.1 as GF(2) Encoder. The
third block is the CRC data decoder and data integrity checker, which is shown as GF(2)
Decoder in Fig. 3.1.

These wrapper logic blocks were then connected to a 28nm compiled SRAM model. We
validated our circuit by stimulating the design model with over one million read and write
operations, where each write operation contains a randomized data packet. On each read
operation, the expected contents of the SRAM were checked, thus completing validation
loop.

Three different versions of our design were created with increasingly expensive CRC en-
coders/decoders. These are called CRC4, CRC8, and CRC12, where the number corresponds
to the degree of the generator polynomial.
3.5.2. Experimental Setup

We setup our design simulation testbench to allow each version of our design plus the SRAM model to be stimulated with uniformly randomized address and data on each write operation. The SRAM model contains 8192 entries, where each entry is 72 bits wide. In our testbench, however, we restricted the testbench to treat the SRAM logically to have a data width of only 60, where the remaining 12 bits are reserved for CRC operations. In the case where a circuit cannot fully utilize all 72 bits of the SRAM, we zero pad the write data bus.

To account for the fact our SRAM address swizzling circuit is randomized only once per simulation run, we iterated our randomized SRAM access simulations ten times, where we stimulated the design with a tenth of the total accesses shown in row two of Table 3.1. We then proceeded to generate our data points based on all ten iterations. (Row two corresponds to the number of write/read pairs simulated.)

3.5.3. Experimental Results

3.5.3.1. Data and Address Scrambling

To evaluate the effectiveness of our design in terms of SRAM address and data obfuscation, we simulated over one million vectors for each circuit configuration and measured the Hamming distance between the obfuscated address and data content for each write operation when compared to the original SRAM address and data. This provides us an estimate of the scrambling achievable with our approach.

Ideally, the Hamming distance will correspond to approximately 50% of the bits, and the scrambling of different bits will be difficult for the Trojan writer to predict so that knowing the value of a bit in the original address/data is likely to provide little information regarding data in the scrambled code word.

Specifically, for each write operation we simulate, we take the Hamming distance of the testbench generated (address,data) pair and compare it to their obfuscated version at the
boundary of the SRAM excluding the (CRC degree-1) least significant bits—allowing same size vector comparison. The corresponding Hamming Distances and percentages over all testbench generated write operations are shown on rows 3 and 4 of Table 3.1. As one can see, the Hamming Distances correspond to between 46% and 48% of all address/data bits. This is encouraging and indicates that the average scrambling factor is good for this type of input data.

3.5.3.2. Error Detection

To examine the detection effectiveness of our designs against unintended SRAM errors, we inject random three-bit errors on the SRAM’s internal write data bus (i.e. after the wrapper) for every single write operation. We then collect ECC error correction and detection statistics over one million vectors. Note that because 3-bit errors are injected, true correction is generally impossible with ECC. Also note that ECC was implemented in the memory as an Extended Hamming Code capable of SEC-DED.

Row 5 shows the percentage of errors detected by ECC when error correction is turned off. Not surprisingly, because every write has a 3-bit error, the parity calculation over all bits flags an error every time—leading to a detection rate of 100%.

The problem arises when we try to correct. In that case, the syndrome will generally be non-zero, the parity over all bits will be non-zero, and the hardware will indicate that a bit should be flipped—introducing another error. This is a “mis-correction.” The mis-correction rate is shown on Row 6 of Table 3.1 and is approximately 98.55%. Cases where a mis-correction did not occur correspond to cases where the syndrome generated by the three bits happened to equal 0. The data is still wrong, but an additional bit was not flipped.

Rows 7 and 8 provide information regarding the effectiveness of the CRC codings for error detection. Because the ECC in our design is incapable of correcting three-bit errors, all read data sent to the CRC checker is erroneous. The rate at which the CRC detects these errors varies from 77.35% for CRC4 up to 99.96% for CRC12, as shown on row 7. Row 8 shows the corresponding silent errors that range from 22.65% for CRC4 to 0.04% for CRC12.
Note that each of these detections is important. Whether the error arose due to a Trojan writer disabling the ECC, a Trojan writer inserting 3-bit errors, or 3-bit errors occurring due to natural phenomena, the implemented ECC is unable to handle them. It is only the CRC wrapper that is preventing more silent data corruption. Also note that although we only collected data on 3-bit errors, larger numbers and other configurations of errors may also be detected by CRC. The number of errors that can be detected in a burst is a function of the degree of the generator polynomial.

3.5.3.3. Overhead

We use a 28nm synthesis flow on each circuit with target frequency of 2GHz. We then examine the circuit area and circuit performance impact on the SRAM design as well the impact on a potential SoC. We report circuit areas in microns on row 9 of Table 3.1. The area for each of our circuits as a percentage of a particular instance of the target SRAM design is reported in row 10. We then derive the area required for a production server SoC, and we report the ratio of this area versus the existing process die size in row 11. We then proceed to examine the timing achieved by the three circuits in terms of MHz, and this is reported in row 12. In row 13, we report the 2GHz read and write latency in terms of clock cycles caused by our circuits as a percentage over the SRAM timing closure report. In row 14, we report the increase in the overall processor core SRAM read pipeline latency our design insertion would incur.

We proposed a novel packet obfuscation and data check solution for potentially untrustworthy SRAMs while achieving better than ECC soft error coverage. The obfuscation is capable of randomizing the data address mapping to make triggering by a HTH less deterministic. Combinational, non-systematic CRC is also used to scramble the data vector of an access packet. On read out of the obfuscated packet, our solution can de-scramble and self-check the read data packet. This helps guard against errors as well as makes data-based SRAM triggers less deterministic. Our experiments have also shown our solution to be capable of catching most 3-bit errors without substantially impacting design area. Higher bit
Table 3.1: SRAM Simulation and Synthesis Results

<table>
<thead>
<tr>
<th>Design Metrics Vs Circuit</th>
<th>Design on CRC4</th>
<th>Design on CRC8</th>
<th>Design on CRC12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Accesses</td>
<td>1249855</td>
<td>1249851</td>
<td>1249845</td>
</tr>
<tr>
<td>Average Hamming Dist</td>
<td>34.01</td>
<td>34.01</td>
<td>34.50</td>
</tr>
<tr>
<td>Average Scrambling Factor</td>
<td>46.58%</td>
<td>46.59%</td>
<td>47.26%</td>
</tr>
<tr>
<td>ECC Error Detections (%)</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>ECC Mis-Corrections (%)</td>
<td>98.55%</td>
<td>98.54%</td>
<td>98.55%</td>
</tr>
<tr>
<td>CRC Error Detections (%)</td>
<td>77.35%</td>
<td>98.67%</td>
<td>99.96%</td>
</tr>
<tr>
<td>Silent Errors (%)</td>
<td>22.65%</td>
<td>1.33%</td>
<td>0.04%</td>
</tr>
<tr>
<td>Circuit Area (microns)</td>
<td>2180</td>
<td>4712</td>
<td>6497</td>
</tr>
<tr>
<td>Per SRAM Area Overhead (%)</td>
<td>0.92%</td>
<td>1.99%</td>
<td>2.75%</td>
</tr>
<tr>
<td>Per Die Area Overhead (%)</td>
<td>0.065%</td>
<td>0.141%</td>
<td>0.195%</td>
</tr>
<tr>
<td>Circuit Performance (MHz)</td>
<td>564MHz</td>
<td>497MHz</td>
<td>445MHz</td>
</tr>
<tr>
<td>Access Latency Overhead - SRAM (%)</td>
<td>125%</td>
<td>150%</td>
<td>150%</td>
</tr>
<tr>
<td>Access Latency Overhead - Pipeline (%)</td>
<td>24%</td>
<td>28%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Error rates may also be detectable. Some limitations include the timing penalty required and the fact that lower address bits may need to remain unscrambled in the case of caches when multiple words in a block must remain contiguous.
4.1. Overview

While chapters 2 and 3 are focused on protecting embedded memories from HTHs, this chapter and the subsequent two chapters pivot into the topic of IJTAG usage and IJTAG securitization. This chapter contains a detailed introduction to IJTAG and its usage with an industrial example. The reach of IJTAG network access on-chip shown in this chapter illustrates the potential risk of adversarial intrusion via IJTAG. If the IJTAG network is successfully attacked, it’s clear that embedded memory test instruments on the IJTAG network can be compromised, which can lead to further attacks on embedded memories. The proposed work to robustly verify IJTAG can be found in section 4.6. The contents in this chapter were first published in the proceeding of the 2015 IEEE International Test Conference (ITC) [88].

4.2. Introduction

IEEE 1149.1 [50] based top level access to IEEE 1500 [47] compliant intellectual property (IP) cores is commonly used in industrial designs, including in AMD designs, as the underlying infrastructure to facilitate various test, power, and functional control and debug access features. In this chapter, IP core-level IEEE 1500 compliant structures are called design-for-excellence IPs (DFX-IPs). Validating the DFX-IPs during the early design stages at the core level as well as in the full system on chip (SoC) context, including their access infrastructure, is critical for project success.

The production project discussed in this chapter is AMD’s first 64-bit ARM-based server processor [3]. The challenges are not only to satisfy all design requirements with an efficient
and versatile architecture, but also to optimize verification and automate the automatic test pattern generation (ATPG) test-setup. This includes automating ATPG compression mode with low scan shift power, memory scan chain bypass mode, chip-wide single chain mode, etc. to achieve fast time to setup and verify the chip in various test modes, and to enable the back-end silicon debug and platform analysis. Both verification and test-setup proved challenging to existing in-house design environment and methodology, which needed to adapt to various architectural decisions made during project design cycles, including silicon revisions.

An IEEE 1687 (IJTAG) [49] based method was initially explored with a focus on simply automating test-setup vector generation for hierarchical ATPG test modes, but it was quickly realized that the IJTAG-based methodology and tool can be adopted to also improve design verification flow and test generation for functional IPs. The existing AMD internal methodology used for register transfer level (RTL) verification was particularly labour-intensive because every micro-architected IJTAG configuration must be manually modeled from the specification. Similarly, the internal automatic test equipment (ATE) level ATPG test-setup vectors were script-generated to extract the test-setup procedure TCK cycle-wise for a top-level SoC. As such, the IP-level test-setup procedures were completely non-reusable at the SoC level. So the AMD internal flow only complied with the IJTAG standard in terms of RTL implementation, but not with the possible automation enabled by the standard. Hence, the streamlining and automation of all aspects of the flow, from IP-level test wrapper insertion to SoC integration and verification, is of primary interest.

In this chapter, the automated flow deployed for IJTAG test insertion, integration, and verification are described, and with this flow, it can be shown that significant reductions in time and effort that are possible.

IJTAG—method, referring to a Tessent—developed IJTAG tool [49]. It was learned that the IJTAG method can be used as a general access method to embedded IP from any level of hierarchy. It adapts well to architectural changes, accelerates ATPG setup and network validation, and allowed front-end DFX IP verification optimization, design—for—testability (DFT) mode setup, and back-end silicon debug.
The following aspects of the chapter in discussed in detail:

- Briefly describe key IJTAG concepts
- ICL and PDL modeling of test structures in our IP cores and their respective validation
- SoC—level reuse of IP core—level PDL, ICL network extraction, and SoC—level test setup
- An efficient design verification methodology utilizing IJTAG
- Application of IJTAG to automate SoC—level test setup vector generation

4.3. Overview of IJTAG (IEEE 1687)

The IEEE 1687 standard describes a methodology for accessing instrumentation embedded within a semiconductor device via the IEEE 1149.1 test access port (TAP) and/or other signals. A good overview of the IJTAG standard and its capability is available in [62]. The elements of the methodology include hardware architecture for the on-chip network connecting the instruments to the chip pins (i.e., an IP that is 1687 compliant), a hardware description language (instrument connectivity language - ICL) to describe this network, and a software language and protocol (procedural description language - PDL) for communicating with the instruments via this network. PDL describes the operation of the instrument through commands writing to or reading from the instrument’s ports, scan chains, or data registers. These operations are described with respect to the boundaries of the instrument. PDL is compatible with the popular Tcl scripting language.

ICL provides the abstract, hierarchical definition of input/output ports (I/Os), registers, and scan chains that are necessary to control the instrument through the instrument’s IJTAG standard compliant test interface. The ICL does not include the details of the inner workings of instruments, but only I/Os, registers, bit fields, and enumerations of data elements that are necessary to carry out the operation declared in the PDL. ICL also describes the network connecting the different instruments, although the standard only intends to describe the
access to the instruments, and not the details of the instruments themselves. Therefore, ICL does not necessarily need to fully describe the design implementation. With the combination of the ICL description and PDL operation for an instrument, the instruments can now be packaged for reuse. The standard ensures that a PDL sequence of operations written and verified at the instrument module-level can be used without modification after that instrument has been inserted inside a design. However, it is the process of retargeting that translates this PDL sequence from the module’s instance level to the top level of the design (or any other hierarchy level in between).

4.4. IJTAG Modeling and Validation of IP Cores

![Figure 4.1: A Simplified IJTAG Network](image)

We now discuss the ICL and PDL modeling of our IP cores inserted with IEEE 1500 compliant test structures. Each core may have at least one set of IEEE 1500 wrapper serial port (WSP), wrapper instruction register (WIR), wrapper bypass register (WBY) and data registers for various functional controls, clock/reset controls, power control, and/or debug control. The size of the WIR and the number of data registers varies depending on the test features implemented per core. We modeled each DFX-IP interface and TDR selection scheme with an ICL description. Each DFX-IP’s respective usage is modeled with a PDL.
description. We aimed for the network connecting DFX-IPs to utilize powerful IEEE 1687 features. Namely, we augmented this network with extensive usage of segment insertion bits (SIBs).

A simplified structure of an IJTAG network is shown in Fig.1. It is composed by daisy-chaining DFX-IPs via their IEEE 1500 compliant wrapper pins. These chains are selected for inclusion with multiplexers controlled by SIBs. When opened, a SIB provides access to additional scan segments and IP (such as an IEEE 1500 wrapper). When closed, it bypasses that portion of the scan path.

Beginning with one example DFX-IP, a PCIE IP core, the ease with which IPs can be modeled with the IJTAG-based method at the IP core-level can be shown. The PCIE IP core is IEEE 1500 compliant and has a 3-bit WIR. This WIR selects the following registers: BYPASS, EDT˙CONTROL, DBUS˙REG, and a few other registers. The PCIE DFX-IP module level description is shown in Fig. 2. A simplified block diagram is shown in Fig. 2(a). The corresponding ICL description is shown in Fig. 2(b). As can be observed, each register only needs to be described with two sets of attributes: a WIR selection vector and bit field description labels. For example, in Fig. 2(b), the EDT˙CONTROL register is specified to be an eleven bit register selected with the WIR selection vector 3’b001. Bit field labels are then listed in the file starting with edt˙bypass and ending with test˙mode. As shown in Fig. 2(b), it’s straightforward to compose an ICL abstraction for this DFX-IP. In addition, any IJTAG port can be declared with ICL keywords. ICL can also express register multiplexing for an instruction register (IR) and a data register (DR). How they are connected with respect to the rest of the SoC can be abstracted, and the logical connectivity does not need to be described explicitly.

Given an ICL for a DFX-IP, PDL describes the usage of that DFX-IP. The PDL keyword iProc is used in conjunction with an IP identification label to identify and express the DFX-IP usage. A PDL snippet for our PCIE IP is shown in Fig. 3 above. For each DFX-IP, the respective setups for each ATPG mode can be expressed as corresponding iProcs in PDL that can be reused in the SoC-level or in any higher-level hierarchy. For example,
Figure 4.2: a) Block diagram for PCIE DFX-IP. b) ICL for DFX-IP in PCIE
those modes could be compression mode with or without low power scan test shift control, compression bypass mode, single chain mode, etc. iWrite is the PDL keyword that specifies the test control vector to be written to a register. For example, the first iProc in Figure 3, edt˙intest˙lowpower˙setup, specifies how to turn on compression mode with low power shift. Note that in the iProc, the user only needs to iWrite to bits that are different from their default value. For example, edt˙low˙power˙shift˙en needs to be 1. The iWrites must be followed with an iApply statement, which instructs the IJTAG tool to compute sequencing to load the specified vector into the registers.

Both ICL and PDL files are human readable and can be self-documenting. Once they are described, the tasks of validating the syntax, semantic checks, connections rules, simulation testbench and ATPG test-setup are all automated with the IJTAG tool. This flow is shown below in Fig. 4.

The first step is to create IP-level ICL and PDL descriptions, as shown in Fig. 4. For
validation, the IJTAG method loads the ICL and PDL and checks for syntax and semantic errors in the ICL and PDL codes. It then applies a suite of Design Rule Checks (DRC), which includes static connectivity checks, by evaluating the ICL/PDL against the DFX-IP RTL. After the IJTAG analysis, the user can invoke the tool to write resulting test sequences in different formats. For example, the simulation testbench or tester vectors can be written in commonly used formats, such as Waveform Generation Language (WGL).

With validated IJTAG structures at the DFX-IP-level, ATPG can incorporate IJTAG control sequencing for localized structural test generation; this is shown in the final step of Fig. 4. This is as simple as invoking an iCall keyword embedded in the ATPG control scripts. This degree of automation enables the engineers to sign-off on the DFX-IP-level ATPG without any explicit forces or constraints on design internal signals, thereby allowing comprehensive problem detection in the early stages of design. This is especially true for DFX-IPs with highly complex reset/set and test clock control, such as layered test clock generation.

4.5. SoC-level Reuse of the IP-level IJTAG Model

At the SoC-level, the Master Test Access Port (MTAP) needs to be described in ICL first. Given the ICL at the DFX-IP-level, the IJTAG method can be used to extract a complete
ICL network for the entire design. Note that the master TAP may have more features than are relevant to the IJTAG network connections. For use by the IJTAG tool, the user can choose to either describe it completely in ICL or to include only the minimum feature sets that are needed for communicating and controlling IJTAG structures. For example, in an abstracted view of the structure in shown in Fig. 1, the user may only describe the IJTAG chain structure shown in Fig. 5 with ICL. The ICL of Fig. 5 is shown in Fig. 6.

![IJTAG chain structure](image1.png)

**Figure 4.5: IJTAG chain structure**

![ICL MTAP description](image2.png)

**Figure 4.6: ICL MTAP description**

The flow in Fig. 7 depicts SoC-Level ICL extraction and ATPG test access generation.
by the IJTAG tool. At the SoC-level, we start with modeling of the MTAP block with ICL. From there, ICL network extraction will check, validate, and extract the connectivity between all DFX-IPs. At the end of the network extraction, the entire SoC’s ICL network inclusive of point-to-point IP connections is automatically generated. The iProc for any SoC-level DFX-IP, such as ATPG setup, can reuse core-level iProc. By now the IJTAG tool can be utilized to generate simulation testbenches.

4.6. Verification Flow Leveraging IJTAG

The increase in SoC test complexity and scale induces tremendous verification complexity. More test structures require more verification tests to cover the test features. Further, with the advent of reconfigurable scan chains, such as the insertion of SIB controls as allowed in IJTAG, the number of possible IJTAG chain configurations increases exponentially in relation to the number of SIB bits. The design goal is to be able to verify DFX-IP for such a SoC in a time efficient and comprehensive manner.
To the design team, comprehensive verification means DFX-IP operations must be ensured not only function for DFX-IP purposes, but also that they do not negatively impact chip functional operations. A prime example of such issues is manifested in security, where test control circuits may be used to adversely gain control of secret system keys. In addition, verification stimuli sequencing needs to be consistent with ATE pattern application in order to minimize silicon bringup effort.

To address the verification requirements, a novel solution leveraging the power of the IJTAG standard with the IJTAG method in conjunction with highly abstracted and robust SystemVerilog-based verification testing [62] was developed. As discussed in section IV, the IJTAG method offers the capability to extract a fully-specified IJTAG-compliant ICL in tandem with a suite of static design checks at the RTL during ICL extraction. ICL extraction yields two advantages: initial RTL DRC and automated construction of a fully specified ICL. RTL DRC allows us to potentially achieve faster time to first bug because a basic connectivity DRC check can be applied. Secondly, a fully specified ICL gives us the flexibility to build abstracted verification tests while leveraging advances in verification technology such as SystemVerilog with ease. The end result is a much more efficient verification flow that enabled the detection of a set of common bugs much faster than previous practice. Furthermore, design changes now no longer will cause rework to verification test suites.

Table 4.1: IJTAG Bug Classification and Detectability

<table>
<thead>
<tr>
<th>Bug Types</th>
<th>Detailed Description</th>
<th>Detectability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Current Method</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>Specification not matching correct RTL</td>
<td>Yes, weeks of time</td>
</tr>
<tr>
<td><strong>IJTAG Connectivity</strong></td>
<td>Connection at SoC for DFX-IP’s</td>
<td>Yes, hours of simulation</td>
</tr>
<tr>
<td><strong>IJTAG Network Ordering</strong></td>
<td>Ordering of SIB bits</td>
<td>Yes, hours of simulation</td>
</tr>
<tr>
<td><strong>IJTAG Compliance</strong></td>
<td>Register update and capture time</td>
<td>Yes, manual effort and hours of simulation</td>
</tr>
<tr>
<td><strong>DFX-IP</strong></td>
<td>Functionality</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The design currently detected in this design are summarized in Table 1. Bug types are labeled in the first column. Generalized bug descriptions are provided in the second column. Detectability and time required per method are described in columns three and four for the existing methodology and IJTAG-based static checks, respectively.

Specification bugs occur due to documentation not properly reflecting the intent of the design. For example, fields of a write once only register may be marked as state retaining. It is also possible that as architectural updates occur, obsolete registers are not removed from the specification.

Connectivity bugs are found typically along register bits on any reconfigurable TDI to TDO IJTAG data shift path. IJTAG ordering primarily refers to the ordering of IPs along the IJTAG data shift path. When more than a handful of SIBs are stitched together, verifying each SIB to IP and IP to SIB connectivity and ordering of the SIB bits can become extremely time consuming, as the stitching is influenced by power partitioning, demand for test functionality, and security control among a myriad of other factors.

Achieving IJTAG compliance means that logical and physical implementation must ensure correct timing relations in each SIB-enabled DFX-IP so that IP-level CaptureDR and UpdateDR operations can be performed in full compliance with the IEEE 1687, IEEE 1149.1, and IEEE 1500 timing relations governing every TAP controller regulated signal across the entire IJTAG network. At the IP core-level, every IJTAG instrument must also be verified for proper functionality.

The current internal validation method requires a complete manual specification of an entire IJTAG network inclusive of the traditional TAP controller specification as well as full IJTAG point-to-point connections. This must be done in a format that allows metadata on the IJTAG network to be extracted automatically. In short, every feature, every IP, and every connection must be specified in a rigid format for computerized processing. Such metadata are then fed into design simulator embedded IJTAG solver, and design verification tests can call on the IJTAG solver to generate serial IJTAG access stimuli during logic simulation. In addition, the internal flow allows verification flow to apply IJTAG test vectors along with a
suite of SoC functional checkers and sequences concurrently.

This kind of unification, from AMD internal verification solution, of SoC functional and DFT verification allows for better verification for design features having functional as well as DFT applications. One example of such an application would be memory repair distribution during the SoC boot process. Cores need to boot after repair distribution completes, but repair distribution relies on a portion of the IJTAG controllable instruments. Thus, the benefit of such an approach is that the DFT test application model is now aligned with functional operations. Finally, it allowed the design team to reduce the number of testbenches that needed to be maintained.

From Table 1, it is clear that the internal verification flow can be improved by incorporating IJTAG tool usage along with better abstraction of IJTAG manipulation. In the proposed flow, as shown in Fig. 8, rather than utilizing manual and error prone specification to map out metadata for dynamic IJTAG pattern generation, fully specified ICL generated with the IJTAG method to automatically extract metadata that can be exploited to control access our IJTAG network at the SoC-level. This extraction is automated via TCL scripting and is shown above the ICL extraction step of the pyramid in Fig. 8.

![Figure 4.8: IJTAG based verification flow](image)
The design team has an IJTAG aware solver that can read the resulting metadata and resolve every IJTAG controllable register access into vector sequences that map to the five JTAG pins. To take advantage of this, a set of abstracted routines to wrap the metadata solver and IJTAG controlled operations was built in SystemVerilog. For example, a single function call can be utilized to invoke access to any IJTAG controlled TDR on the SoC. Multiple IJTAG access sequences can also be wrapped into a single function call. Once this is done, SystemVerilog-based verification tests can be written and executed with great ease, as shown at the top of the pyramid in Fig. 8.

The AMD internal IJTAG solver maybe unique, but such metadata can be easily mapped into SystemVerilog constraints to achieve similar solving capability.

To illustrate the power of the proposed method, a simple example is shown in Fig. 9. Lines 1 - 6 in Fig. 9 to stimulate the IJTAG network with ten accesses to ten random TDR’s in random order with random legal values. The abstraction unified write and read sequences into a single routine, IJTAG\textasciitilde TDR\textasciitilde Access. On line 2, an abstracted querying function to find every single register in the IJTAG network can be implemented. With SystemVerilog’s support of random variables, and constraints on random variables, valid and randomized TDR control sequencing and values can be generated with ease. Complex protocols and programming sequences can be further abstracted into easy-to-use routines. This is shown on lines 4, 5, 9, and 11.

For example, as shown on lines 4 and 5 in Fig. 9, an abstracted IJTAG test access mechanism is developed to enable SystemVerilog test calls to write and read each JTAG Register by name and write and read values respectively.

Given that abstracted IJTAG calls can be made with ease, SystemVerilog built-in randomization can also be leveraged to randomize the order and value in which we access the JTAG test structures, as shown in lines 1-6.

IJTAG sequences required to write and read a memory mapped register can also be encapsulated in the IJTAG\textasciitilde MemMapped\textasciitilde tdr\textasciitilde access routines shown in lines 9 and 11. This is important because memory mapped registers can be accessed by a processor core. In other
words, functional software may have access to it. As such, it is updated with functional clocks rather than TCK. To issue write or read operations, multiple TDR programming sequences are required to generate a glitch free update to a memory mapped register. Further, this register is used to issue BIST control sequences, which require another layer of protocol. Again such complex operations can be abstracted into a single routine. The entire test essentially can exercise multiple IJTAG controlled instruments concurrently.

The power of the proposed approach comes from the fact that with IJTAG and the Tessent IJTAG tool, IJTAG model construction and metadata generation are automated—no longer need to rely on labour-intensive manual IJTAG specification; rather, automated SoC ICL extraction is relied upon. As a result, verification can now start before the design is even simulated via IJTAG static DRC checks. Furthermore, with better abstraction at SystemVerilog, verification search for corner cases can be done with ease, and the reach of the verification search is expanded with randomization. In addition, in the proposed framework, functional operations are allowed to simulate in tandem with IJTAG instrument operations as the verification testbench for functional verification and test verification can be merged.

```
1. Loop 10 time{
2. Pick TDR_A randomly inside [set of all TDRs];
3. Generate random legal TDR_A_Value for TDR_A;
4. IJTAG_tdr_access(TDR_A, TDR_A_Value, Write);
5. IJTAG_tdr_access(TDR_A, TDR_A_Value, Read);
6. } End Loop;
7. ...
8. Pick random legal Memory BIST Control Vector MBIST_cnt_vec;
9. IJTAG_MemMapped_tdr_access(BIST_Control_Register,
   MBIST_cnt_vec, Write);
10. Wait for expected BIST run times;
11. IJTAG_MemMapped_tdr_access(BIST_Status_Register,
   MBIST_result_vec, Read);
12. Assert [MBIST_result_vec[Fail Bit] & MBIST_result_vec[Done bit]]
13. Else $ISSUE_ERROR("MBIST FAILED!");
```

Figure 4.9: IJTAG Abstract verification testing

Although prior work discussed utilizing formal approaches to verify the IJTAG network
[6], [7], ensuring mission mode operation correctness was not discussed. Even though this work did not perform formal analysis, the silicon results did not reveal any logical connectivity problem in the process of silicon bring up and ATE test program development. Further, the processor silicon sample does boot successfully even with complex interaction between functional logic and IJTAG instruments. In contrast to prior work, the proposed solution is more comprehensive as no prior work addressed validating functional operations in tandem and against pure test operations. To the best of the author’s knowledge, the proposed verification flow as presented here is the first to harness the automation enabled by IJTAG to accomplish efficient constrained random verification on IJTAG instruments and structure of a state-of-the-art Server Processor.

4.7. Applying IJTAG for SoC-Level ATPG and IP Test Patterns

After ICL network extraction for the SoC, the manual effort for test setup or configuration at the SoC-level is fairly simplified because the PDL operations at the core-level can be reused at the SoC-level without modifications. Further, the IJTAG tool offers the capability to further optimize test access for test time reduction. These two features are best illustrated with examples. Data are then provided to showcase the degree of automation.

Fig.10 shows an SoC-level iProc procedure to set up a group of IP cores in compression mode with low power shift enabled. Note each identifier after each iCall invocation is a test setup procedure defined at the IP-level. The entire procedure includes iCalls for setting up the device into scan mode, selecting a group of cores for scan test, setting up processor core resets, phased locked loop (PLL) control settings, and compression mode enable for each core in the selected group. Observe the iMerge PDL command, which enables optimization of IP-level setup sequences from iCall invocations. Without iMerge, only one core can be setup at a time, resulting in an unnecessary increase in cycle time for test setup. In the IJTAG standard, selecting test access optimization is as simple as adding the iMerge command to the procedures. It is then up to the IJTAG tool to figure out how to best make use of the parallel hardware that was implemented in the design. On the other hand, for debugging
test access issues, iMerge can be removed and reorder the iCalls for better debug granularity. The iRead command can also be used to check if the setting by iCall commands are valid. Furthermore, with the different scan modes our design has, the previously mentioned iP roc can be reused without modifications, resulting in a meaningful test cycle reduction. For SoC-level ATPG, Tessent IJTAG also offers a command to append iCalls to an existing test setup, for example that executes power up and resets the chip, then runs the setup of the chip in the desired test mode.

The example in Fig. 10 targets logic-based tests. However, benefits to IP functional test pattern development can also be seen. Creating traditional functional IP tests often involves hand-writing the testbench, saving the VCD dump from simulation, and then using scripts to convert the VCD dump to cycle-by-cycle based STIL vectors. Unfortunately, STIL vectors generated this way lose the annotations that are much needed for silicon debug on the ATE. However, IJTAG allows for a framework to develop tool optimizable functional test patterns.

Tessent IJTAG also associates IJTAG register reads with meaningful identifiers. This translates into better test vector readability. The STIL vectors written by the IJTAG method not only annotate the status of the JTAG state machine but also annotate what each vector is trying to do.

At this point the power of IEEE 1687 IJTAG can clearly be seen with the abstraction of ICL and PDL for the IP cores and the reuse at the SoC-level.

4.8. Results and Observations

The target design is AMD Opteron™ A1100 Processor [3]. This design contains well over 2000 SRAMs, 4 million scan flops, 60 million gates in total, and has around 200 ICL ScanRegisters. The data of two facets of the work on this processor is examined: verification and production test.
4.8.1. Impact of IJTAG Automation on Verification Time

To examine improvements on the verification front, one particular metric is used, potential time to first bug after RTL is delivered. However this is not the only valid metric, but from a practical standpoint, in consideration of ever tightening design schedules, the earlier bugs are found, the more likely that design milestones can be achieved.

Verification results are tabulated in Table 2. Column 1 contains bug classification, column 2 records the approximate time it takes to find the first bug in the respective bug category with the existing verification method. Column 3 tabulates the amount of effort for the proposed verification method. For the first three bug types, the current method requires at least a week of manual effort to generate manual IJTAG network metadata, and then engineers need to run laboriously composed directed verification tests to begin finding bugs. In comparison, the IJTAG method usually take only 5 to 10 minutes to perform ICL network extraction, and another minute or less to generate metadata for logic simulation. The simulation test on average takes about an hour to run-just for the IJTAG network IP-level access. For IJTAG compliance testing, Essential timing-driven assertions needed to be developed manually for every DFX-IP on chip. It is deemed at least four weeks are necessary
to enable existing verification method to find the first bug. In contrast, the IJTAG tool can
generate IJTAG compliance aware sequences to help congruent checks to catch problems
almost immediately. These sequences can be leveraged in the verification test suite. This
can be safely achieved within a day. However, DFX-IP feature specific verification requires
domain specific knowledge to develop valid verification tests. Despite having a high degree
of automation in ICL generation, it still takes at least three weeks in the design team’s
experience to code a valid suite of tests. Even in these cases, the IJTAG-based verification
method is still faster as IJTAG metadata generation can be automated. This reduced effort
by at least a week.

Table 4.2: IJTAG Bug Verification Results

<table>
<thead>
<tr>
<th>Bug Types</th>
<th>Time to 1st Bug From Date of RTL and Hierarchical ICL Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Current Approach</td>
</tr>
<tr>
<td>Documentation</td>
<td>1 Week</td>
</tr>
<tr>
<td>IJTAG Connectivity</td>
<td>1 Week</td>
</tr>
<tr>
<td>IJTAG Network Ordering</td>
<td>1 Week</td>
</tr>
<tr>
<td>IJTAG Compliance</td>
<td>4 Weeks</td>
</tr>
<tr>
<td>DFX-IP</td>
<td>4 Weeks</td>
</tr>
</tbody>
</table>

4.8.2. Impact of IJTAG Automation During Test Setup

The production test use of IJTAG is concentrated on test setup automation and op-
timization. The runtime for ICL and PDL validation (automatically solving an iCall or
iWrite/iRead to any registers) for any core takes a few minutes. Runtime for SoC-level
complex iProc validation may require a few minutes. The ICL network extraction with the
entire gate-level design netlist takes about 30 minutes. This can be improved by using a shell
model for the cores because most non-test logic inside the cores can be ignored. If this is
done, the ICL network extraction is down to a few minutes. Core-level ATPG DRC runtime
is almost unchanged compared to traditional cycle-by-cycle test setup. SoC-level ATPG has added a few minutes runtime due to TestKompess natively solving the iCall iProc in test setup.

For ATPG setup, it was found that the IJTAG method offers great value. Due to the design size and the pin limitation, the design has to be configured in different scan modes for testing; each scan mode involves writing to different registers in different cores. Tessent IJTAG, with the use of iProc, which invokes iCall and iMerge commands, is very effective in generating the test setup for different test modes. For example, consider the data points from the ATPG and debug flow shown in Table 3.

Table 4.3: Test Setup Results impact of iMerge

<table>
<thead>
<tr>
<th>Test Mode Setup</th>
<th>Description</th>
<th>Cycles Without iMerge</th>
<th>Cycles With iMerge</th>
<th>Reduction</th>
<th>iMerge Time</th>
<th>In-house Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan 1</td>
<td>Partial chip AC Scan</td>
<td>3173</td>
<td>2391</td>
<td>24.66%</td>
<td>Less than a minute</td>
<td>Days</td>
</tr>
<tr>
<td>Scan 2</td>
<td>Partial chip AC Scan, complex clocking</td>
<td>3312</td>
<td>2440</td>
<td>26.33%</td>
<td>Less than a minute</td>
<td>Days</td>
</tr>
<tr>
<td>Scan 3</td>
<td>Partial AC Scan, more complex clocking</td>
<td>7650</td>
<td>3757</td>
<td>50.89%</td>
<td>Less than a minute</td>
<td>Days</td>
</tr>
<tr>
<td>Scan 4</td>
<td>Full Chip Complex AC Scan</td>
<td>8661</td>
<td>3528</td>
<td>59.27%</td>
<td>Less than a minute</td>
<td>Days</td>
</tr>
<tr>
<td>Scan Dump</td>
<td>Configure full chip ScanDump</td>
<td>9199</td>
<td>3819</td>
<td>58.48%</td>
<td>Less than a minute</td>
<td>Days</td>
</tr>
</tbody>
</table>

In Table 3, columns 1 and 2 briefly describe each test mode. Columns 3 and 4 tabulate the test setup cycle time with or without using iMerge. Column 5 shows the reduction in test setup cycles achievable with the IJTAG tool’s iMerge. Column 6 documents tool runtime from iMerge use. In contrast, Column 7 provides the estimated effort required when using the original internal tool instead of IJTAG tool.

As shown in Table 3, using iMerge, the number of test setup cycles was reduced up to 59.27% with little compute cost. This is due to the IJTAG tool being able to optimally resolve IJTAG SIB and control states to yield test configuration sequences. Otherwise, there are two alternatives. The design team can invest in labor-intensive manual optimization to derive an optimized test setup, or the design team can simply ignore the optimization and execute each configuration step in series without any optimization, inclusive of manual
optimization, and the results are shown in column 3. Compared to the internal tool, which is largely manual and error prone, engineering hours saved can be measured in days, as the IJTAG-based method automates manual effort.

The actual savings in tester time due to the test setup time reduction shown in Table 3 may be small or large, depending on the complexity of the test setup. Table 3 simply provides a quantitative indicator of the effectiveness of IJTAG. However, the primary takeaway is the minimal effort required to modify the test setup with the IJTAG method.

The effectiveness of the IJTAG method was also proven in debugging scan issues during silicon bring-up. The iProc that is used for scan mode setup can be easily modified to vary the setup to help reveal scan issues. The debugging time is reduced approximately from weeks to days.

The learning curve of using Tessent IJTAG is also less than our in-house method, as the tool inputs are human readable. Automation is easier to achieve because Tessent IJTAG works seamlessly with Tessent TestKompress for scan pattern generation.

4.9. Conclusion

The IJTAG standard allows reconfigurable scan chaining of embedded instruments, provides ICL to hierarchically model a set of IJTAG networked IPs, and simplifies IJTAG access with PDL. These features provided us as SoC integrators with substantial automation at specification, design modeling, design verification, and production test pattern generation and validation. Being able to reuse the core-level test at the SoC-level for embedded IPs as well as adapting well to SoC-level architectural changes have been some of the critical reasons for the design team to pursue the IJTAG standard and IJTAG tool based methodology. Human readable ICL and PDL iProcs can be processed by the IJTAG tool. Furthermore, they provide efficient team communication and automation among DFX IP design and verification teams, the ATPG team, and the backend silicon debug team. Specifically the documentation provided by IJTAG ICL and PDL makes the actual test intent very clear, which significantly improves understanding in application and debug.
Once the ICL and PDL are described, the rest of the validation, including syntax, semantic and connection rules checking, simulation testbench, and test vector generation are all automated by the IJTAG tool. This increase of automation significantly improves the turn-around time to get to a testable solution, and overall reduces the chance for human error and identifies mistakes earlier. In particular, this allows very fast responses to any change requests that typically happen during the course of a project, especially during the early phases. Effective functional testing as well as testing of the IJTAG network were implemented, in comparison to prior work.

The IJTAG method also allowed for faster time to generate production test patterns. This resulted from re-use of core-level ICL and PDL at the SoC-level. Taking advantage of the iMerge capability of the IJTAG tool also saved test time, without costing additional engineering time.

The production project experience shows that IJTAG is a general method that can be used for and improves embedded IP test reuse for designs with complex IEEE 1500 infrastructure and an IEEE 1149.1-based top-level control architecture.
Chapter 5
Echeloned IJTAG Data Protection

5.1. Overview

The previous chapter, Chapter 4, showcased the application and verification of an IJTAG TAM in a server processor. In this chapter, three echelons of protection working in tandem for IJTAG is proposed — see section 5.4. This approach was first published in the proceeding of 2016 Asian Hardware Oriented Security and Trust Symposium (AsianHOST) [57].

5.2. INTRODUCTION

With advances in semiconductor device process geometry and processor architecture, increasing numbers of features are being integrated onto dies, all of which must be tested to ensure reliability. In addition, when in-the-field failures occur, debug and diagnostic mechanisms may need to be used to isolate failures. System developers may also need debug and diagnostic access to enable better optimization and system software development. As a result, semiconductor device manufacturers have developed a wide variety of on-chip instruments for test, on-chip debug and diagnostic access mechanisms, and device-compatible debug systems and software as means to address these challenges.

The corresponding on-chip instruments, testers, buffers, sensors, and configuration circuitry are often accessed and enabled via an on-chip test access port, such as the port specified in the IEEE 1149.1 Joint Test Action Group (JTAG) Standard [46]. Securing this test port from unauthorized access is important to maintain the integrity and reliability of the chip as well as to protect on-chip data. For example if access to on-chip instruments, especially debug features, is not restricted, a malicious agent may gain protected and sensitive

This work was supported in part by NSF grant CCF-1061164.
information, such as passwords or financial data, during the device’s functional operation. Unprotected debug access may also inadvertently allow illegal system modification, such as illegal modification of device firmware. Furthermore, on-chip design elements often require process-specific tuning, and this information is the equivalent of intellectual property (IP). Without adequate protection, this information may be leaked to unintended parties. Device features that were disabled during the manufacturing flow may also be illegally enabled if debug and test access ports are accessible. In some cases, devices failing manufacturing test may be falsified to be good working parts with illegal test and debug port access. Therefore, for all of these reasons, protected test and debug access is essential.

However, protected test and debug access is non-trivial in today’s design, manufacturing, and supply chain environment. In particular, silicon devices from design to manufacturing and from manufacturing to final system integration and to device debug are often part of an increasingly outsourced global supply chain. As a result, multiple facets of silicon device manufacturing, bring-up, validation, and debug may not be centralized in a single company. For example, devices may be designed by one company and manufactured and tested by one or more additional companies. Device integration and system bring-up may be accomplished by a third company, and device debug support maybe provided by yet another company.

The presence of different players in the supply chain and their exposure to various sets of device test and debug features may allow unauthorized access to information even if the players themselves are authenticated for the instruments they need to access. For example, one firm with physical access to a device may need to be given access to on-chip instruments and data to perform tests or extract a diagnostic stream that can be given to an off-site debug team at the design house. However, if the diagnostic stream itself contains sensitive IP or data, it will not be protected from the firm physically running the diagnostics unless some type of cryptographic or data obfuscation is provided. Thus, methods beyond merely authentication-based protection are needed.

The recent ratification of the IEEE 1687 standard has led to increased adoption of instrument JTAG (IJTAG). This new standard is an extension of traditional JTAG and was
designed to provide efficient access to on-chip instruments without requiring encoding new instructions for each combination of instruments that may need to be accessed. Instead, instruments are accessed through reconfigurable scan chains, which are controlled by data shifted through the chains, and operated with retargeted vectors [49]. JTAG access may often be protected with password based authentication protocols [82]. This chapter proposes a method to orchestrate useful design principles and techniques to attain IJTAG data security and use case isolation apart from authenticated access protection. In particular, this chapter proposes to protect IJTAG-based on-chip instrument control and access by adding three defensive echelons. The term ‘echelon’ is used to capture the interlocking effect of the independent defensive layers of the proposed scheme.

These three echelons afford crytographic protection, allow use case isolation of IJTAG access, and provide detection of unintended IJTAG access. These three echelons are:

- A dual cipher scheme wherein IJTAG control sequences and data accesses may be obfuscated.

- Insertion of IJTAG stub chains into the reconfigurable test data register to complicate IJTAG topology exploration by an adversary.

- Implementation of an access legality checker to deter adversarial attempts at crypto-analysis or power analysis of the implemented ciphers.

An IJTAG access authentication scheme can be incorporated to work in concert with the three proposed echelons.

In Section 5.3, background and prior art are discussed. In Section 5.4, the proposed method is detailed. Section 5.5 explores the overhead involved and cryptographic randomness. Conclusions are discussed in Section 5.6.
5.3. Background

5.3.1. IJTAG

IJTAG is a term associated with the IEEE 1687 Standard [49]. IEEE 1687 is an extension of the IEEE 1149.1 Standard, which is commonly known as JTAG [46]. IJTAG and JTAG contain protocol, language, control, and access specifications. Here, we briefly describe three relevant design structures from the standards: a test access port (TAP) controller, a test data register (TDR), and a reconfigurable TDR (RTDR).

A JTAG-compliant Design under Test (DUT) should have a test access port (TAP) consisting of at least 3 input pins: test mode select (TMS), test clock (TCK), and test data input (TDI), and one output pin: test data out (TDO). The test reset (TRST) pin is optional. TMS in conjunction with TCK controls the finite state machine in the TAP controller to select either a test data register or the instruction register and allow the corresponding register to go through Capture, Shift, and Update modes. During shift, either test data or a test instruction can be shifted into and out from the TAP via the TDI/TDO port pair. JTAG instructions shifted into the instruction register (IR) can select a TDR, such as the boundary scan register or a TDR corresponding to part of the IJTAG network, and allow test data stimulus and control to be sent to the corresponding portion of the DUT. In cases where DUT responses need to be observed, the same mechanism allows TDR access to sample DUT responses to test stimuli.

One of the disadvantages of traditional IEEE 1149.1 JTAG in the presence of large numbers of on-Chip instruments is the fact that it is instruction-based. As a result, each combination of instruments that might need to be accessed together requires the encoding of a new instruction for the TAP controller’s instruction register. When a chip may contain hundreds or thousands of instruments, encoding new instructions for even a fraction of all combinations is a potentially daunting task.

To address this issue, IEEE 1687 was designed specifically to allow the scan network to be reconfigured using the data shifted through the network instead of requiring a new instruction.
to be installed in the instruction register. This reconfiguration can be accomplished by changing the value in a segment insertion bit (SIB). A SIB is simply a bit on the scan chain that allows a new TDR segment to be added to/subtracted from the currently accessible chain depending upon the logic value that has been clocked into the SIB’s “update cell” on the assertion of UpdateDR in the JTAG state machine. The added segment may consist of an instrument interface, simple scan cells, additional SIBs, or any combination thereof. We call a TDR to which segments may be added or subtracted a reconfigurable TDR (RTDR).

Thus, a single RTDR allows the formation of a SIB-switched serial access network into multiple regions of the DUT.

The design structures of concern to this chapter are shown in Fig. 5.1. In Fig. 5.1, the master TAP consists of both the input/output pins of the test access port as well as the TAP controller. Although it is not shown in the figure, the TAP controller contains an instruction register that allows one of several different TDRs to be connected to the test port between TDI and TDO, depending upon the values placed in the instruction register. The figure also contains a reconfigurable TDR (RTDR) that consists of SIBs and embedded instrument interfaces that may be added to/removed from the accessible portion of that TDR. In this figure, the accessibility of the instrument interface is controlled by the SIB located directly to the left of the instrument. (Note that in reality SIBs do not need to be directly adjacent to the segments they control, although they often are.) The figure shows all of the instrument interfaces (e.g. Instrument_0, etc.) as currently being included in the scan path. If the value in a SIB (such as SIB_0) were inverted, then the corresponding instrument interface (e.g. Instrument_0) would be bypassed during scan shift operations.

5.3.2. Stream Ciphers

Stream ciphers encode the plaintext one bit at a time by XOR’ing the current bit of the plaintext with a cipher-generated keystream bit. The same keystream is subsequently XOR’ed with ciphertext to allow plaintext extraction. A wealth of information on stream cipher advances can be found in [35]. Recent advances in stream ciphers, such as the Trivium cipher [21], incorporate block cipher techniques [20, 87]. A block cipher employs linear
diffusion to maximize nonlinear code substitution as an iteration in ciphertext generation. For stream ciphers, a linear feedback shift register (LFSR), which acts as a linear diffusion, along with small serialized bit substitution can be viewed as a layer in key stream generation. The Trivium cipher is constructed using three such layers connected in a ringed manner. The generated key streams are then fed back into the cipher. Such ciphers require deterministic vectors to populate the cipher initially and may require an initialization sequence. Note that a lightweight stream cipher can also be constructed from block ciphers such as Simon and Speck [11].

5.3.3. Existing Work

There has been a substantial body of existing works addressing JTAG or IJTAG authentication, access restriction, and illegal access detection. Examples of JTAG-based authentication can be found in [10,28]. These papers demonstrated the possibility of incorporating a user-device authentication protocol into JTAG access. The authors of [34,121] demonstrate that IJTAG instrument access can be restricted on a fine grain level by controlling IJTAG SIB configuration through the use of key bits and other conditions that must be satisfied for SIBs to open or close. In [34,95], it was shown it’s possible to detect malicious behavior a device may be subjected to and impose countermeasures. For example, [34] introduces trap bits that prevent SIBs from opening until the circuit is reset if incorrect values are placed in the trap. Other research has considered the methods for obfuscating data trans-
fer in a board environment, including hiding information needed to open or close SIBs is a longer bitstream [32] and the obfuscation of data actually sent to an instrument though the LFSR-based “decryption” and “re-encryption” of data as it traverses an instrument interface [33]. Encryption of JTAG TDR shift out data with pseudo random bit streams is proposed in [30,98]. In [25,26], encrypting communication data between an external debugger and a device with a single stream cipher with an external cipher key is proposed. The proposed approach in this chapter aims to provide cryptographic IJTAG access protection, IJTAG use case isolation—where user specific IJTAG control and content access for RTDR configurations can be protected from each other, and a defense mechanism in the event of adversarial IJTAG access attempts. To our knowledge, the proposed approach is one of the first attempt to combine Trivium-based dual ciphering, IJTAG, and associated checking to achieve synergistic defensive echelons. Therefore, it seeks to build on, complement, and enhance existing IJTAG based security measures.

5.4. Proposed Architecture

The proposed architecture in this chapter is based on three primary design principles that form three echelons of protection in addition to the protection formed by access authentication. First, the data sent to a device as a control sequence or extracted from a device as a read access sequence must be protected to minimize the exposure to active attacks. Secondly, we utilize IJTAG TDR re-configurability to effectively increase the number of ways to encode control as well as readout sequencing. Thirdly, legal IJTAG data shift latency can be utilized to impose checks to further reduce active threats.

Because this architecture is imposed at the IJTAG shift-in and/or shift-out paths, it does not conflict with other encryption techniques such as block cipher or key exchange protocols. Hence, the proposed approach can serve to further enhance other types of encryption and data protection techniques.

In the proposed architecture, the target device is assumed to contain an authentication mechanism, but it’s not an absolute necessity. We denote this as IJTAG password lock-
unlock, which we use interchangeably with IJTAG authentication. We expect this lock-
unlock mechanism to also disable the IJTAG re-configurable TDRs’ SIB control bits on lock,
which would prevent access to protected instruments, and on unlock, the opposite to be true.

5.4.1. Protecting Scan Shift Data through Dual Ciphers

There are two distinct types of operations for IJTAG: control and access. Control op-
erations involve a stream of bits being serially shifted into the DUT to control, configure,
program, signal external to DUT communication requests, or execute DUT features and
functions. We consider this type of operation an IJTAG DUT update. Subsequently, the
results and responses due to an IJTAG DUT update need to be captured and shifted out for
user sampling. We consider this type of operation an IJTAG DUT Data capture. Clearly,
the captured data from the DUT must be protected, so we first examine utilizing a stream
cipher to protect the shifting out of captured data. We then look at the rationale for adding
protection on IJTAG DUT update.

5.4.1.1. Protecting Captured IJTAG DUT Data

When authentication is required before a user is allowed access to at least some instru-
ments in the IJTAG network, it clearly indicates a design intention to protect or at least
restrict content access for data generated by those instruments. To maximize the data pro-
tection and reduce complexity, we therefore consider the entire accessible IJTAG network to
require data protection whenever at least one IJTAG instrument requiring authentication
is accessed as a result of the user successfully authenticating himself. In this regard, the
proposed data ciphering is gross in that every data shift out bit must be ciphered during
authenticated access.

To enforce ciphering of the entire IJTAG network, we specify that the IJTAG shift out
pin, \( TDO \), be \( XOR'ed \) with the output from a cipher key stream. To enable a stream
cipher to operate to its designed cryptographic operation, an on-off indicator from IJTAG
authentication is insufficient. More specifically, stream ciphers may require an initialization
sequence. Therefore, the IJTAG-based authentication sequence must allow time for the initialization of the stream cipher to occur once the user has successfully carried out the authentication protocol. Thus, the IJTAG-based authentication sequence must be padded with cycles to allow cipher initialization. After cipher core initialization, the cipher core key stream is available for encoding plaintext. After this initialization sequence, IJTAG access throughput should remain unchanged until another RTDR is accessed because cipher key stream generation is synchronous with every TCK pulse, and no additional TCK cycles are required to allow data to shift out correctly once ciphering has begun.

Once the cipher is initialized, key stream generation and hence encryption should only occur during IJTAG shift operations on the protected RTDR. The cipher controller controls initialization and subsequent key stream generation. Once the cipher core is activated, only a power-on-reset (POR) event, which restores the DUT into its unaccessed state, would disable the cipher controller. When an authenticated session expires or invalidates due to events such as RTDR selection changes or checker triggered alerts, then the cipher controller would pause or stop keystream generation. When another authenticated RTDR access session is requested, if the cipher core’s initial state needs to be changed, then the cipher core should once again be initialized with additional clock cycles.

Another implication of operating a cipher is the need to store a secret key and the cipher initialization vector. It’s also possible to specify at least one key per instrument. Keys could be fused during the DUT production flow, ideally with each device consuming a unique set of keys and initialization vectors. Physically unclonable functions (PUFs) could also be used. These keys and cipher initialization vectors then need to be accessed on authentication of particular instruments to allow the cipher to be initialized correctly. In the cases where multiple protected instruments need to be authenticated, the DUT designer needs to account for key arbitration and potentially limit the number of protected instruments that can be accessed in protected IJTAG access sessions.
5.4.1.2. Protecting Control Access

Ciphering shift out data provides only read data protection, but it still leaves open several vulnerabilities if there is no protection on the IJTAG control access stream. This would allow a potential adversary to have unrestricted access over the entire control vector space and may potentially allow him to explore the IJTAG SIB configuration without any restriction depending on what other protection is provided in the system. Furthermore, given some knowledge about the IJTAG network configuration, a potential adversary may be able to extract correlation between input data and the corresponding ciphered shift out data, and this may allow crypto-analysis to be performed to extract the secret key used for cipher key stream generation. For example, the adversary could shift a constant stream of zeroes into TDI in shift mode without going through capture or update. The ciphered data stream arriving at TDO would then correspond exactly to the keystream itself.

To guard against the aforementioned threats, the DUT can use another stream cipher with an independent key and initialization vector apart from the IJTAG DUT capture data cipher to decrypt the incoming control bitstream. We refer to this cipher as the input cipher. Thus, after IJTAG based authentication, the DUT expects ciphered data to be shifted in at the TDI path of the IJTAG network. The key stream from this control access cipher is used to decode the incoming data to allow propagation to the rest of the IJTAG network. This input cipher then compels the potential adversary to discover both the input key stream generation as well as output key stream generation to control the DUT and extract the DUT’s protected data.

The input cipher requires separate key storage as well as its own initialization vector. Clearly, additional die area is required to accommodate this cipher. Both ciphers can be initialized simultaneously after IJTAG authentication and can operate concurrently. Note that the dual-cipher architecture does require DUT access software to be aware of output data deciphering and input data ciphering to allow a legal user to write and read from the DUT via IJTAG. We henceforth refer to this architecture as a Dual-Cipher architecture in this chapter.
5.4.2. Device-Specific Chain Stubbing

The Dual-Cipher scheme helps protect the DUT from unauthorized examination of IJTAG data and control. However, the basic IJTAG network design is often fixed across device instances. We can further complicate the efforts of an adversary by making the basic IJTAG network differ from one copy of the device to another. In particular, IJTAG network reconfiguration via SIB registers allows us to add stub chains of varying lengths to the shift-out path of each SIB register. For example, in one implementation, the stub chain for a given SIB may be composed of four different chains (1 bit, 2 bits, 3 bits, or 4 bits long) where a multiplexer is used to select which of the 4 different chains is actually inserted into the IJTAG scan network. The select line of the multiplexer could be set at manufacturing (e.g. with fuses) or could be read from the response of a physically unclonable function (PUF). (It could also change over time a function of SIB status, user authentication status, keystream values, etc., while still maintaining some device-specific attributes.) In this way, each device instance would have its own configuration that would uniquely determine the position of SIBs and instrument interfaces on the chain. The configuration could be kept in a trusted database associated with the serial number of the device and could be used to configure software for the correct and authorized IJTAG-based access of the device. It could also serve a dual-purpose as an anti-counterfeiting measure. A counterfeit device with the same serial number would need to copy the specific internal JTAG network configuration. Note that the stub chain should only be included in the RTDR during authenticated access. This can be accomplished with a second multiplexer. An example of the IJTAG network containing a single stub chain after SIB_0 is shown in Fig. 5.2.

Figure 5.2: IJTAG RTDR Chain Stubbing
5.4.3. Re-configurable Chain Shift Checker

With chain stubbing, we have increased the complexity of the effort to decode the protected user data. However, the DUT may still be attacked by adversaries employing various active attack methods, such as trial and error repetition, to analyze the DUT’s cipher system. A more serious threat emerges from power correlation analysis, where the attacker utilizes power signatures from the DUT cipher operations to solve for the key via statistical correlation [109].

To deter these threats, we propose to check the intention of the DUT access agent based on how the IJTAG network is being accessed. More specifically, for an IJTAG network with a given SIB configuration, the total valid shift cycles can be determined. We assume a legal user must be aware of this information (or more likely has access to official software that is capable of keeping track of the appropriate chain length). Therefore, any deviation of shift operation from this requirement constitutes a violation of the IJTAG network’s use policy. Upon IJTAG authentication, the DUT can pre-compute the IJTAG chain shift cycles for the current SIB configuration, and a checker circuit can be implemented to count the number of shift cycles during an IJTAG access operation. During an IJTAG access operation, if the user applied excessive or inadequate shifts, the checker signals to the DUT a need to lock the entire system until the next POR event. This checker may introduce additional routing when SIBs are not physically centralized.

5.4.4. Final Architecture

The proposed architecture can thus be realized with three echelons of data protection based on stream ciphers and IJTAG. We assume an authentication instrument is available in the proposed scheme. In the event it is not used, mechanisms from sections 5.4.1, 5.4.2, and 5.4.3 would apply immediately upon the selection of the target RTDR.

A simplified schematic of the proposed approach is shown in Fig. 5.3. The authentication instrument is intended to take data from the TAP and/or chain and determine whether the current user should be given access to the instruments behind SIBs on the secured
IJTAG network. Multiple approaches to authentication of JTAG access have previously been proposed and could be used for this purpose.

Prior to authentication, the RTDR would appear only with the authentication instrument accessible, and all SIBs on the TDR should be closed. If authentication is unsuccessful, then no access at all should be given to the secure TDRs/instrument interfaces controlled by the SIBs. The orange lines in the figure show that these SIBs can be disabled by the authentication instrument. If authentication is successful, then the SIBs can be enabled so that they can be opened, if desired, on an UpdateDR. In addition, the STUB Chains may be added to the network. Cipher in and cipher out are initialized simultaneously to generate key streams for ciphering or deciphering at \( TDO_i \) and \( TDI_i \). We use the blue arrow to indicate inputs to the Chain checker, which includes the SIB states and the values of the JTAG TAP pins. The chain checker flags suspicious use of the IJTAG network, and when suspicious behavior is detected, the SIBs can be permanently closed until reset.

![Figure 5.3: Echeloned IJTAG Data Protection Architecture](image)

We now examine the complexity of breaking through each echelon. For example, let \( P \) be the number of bits required for the authentication password, assuming a simple scheme where unlocking is achieved when the programmed password matches the expected password. Thus, the password space is \( 2^P \). The expected time for successful authentication would depend at a minimum on the ability of the attacker to successfully submit a password to the authentication instrument, his attack strategy, and the password’s size.

The ability to extract the cipher keys depends on the type of cipher, physical imple-
mentation, and attack methods. However, the concurrent use of two ciphers significantly increases the complexity.

To evaluate the number of potential IJTAG network configurations as a function of stub chain settings, we assume that a stub chain for a given SIB can be of any length from 0 bits (no extra bits inserted) to a maximum length denoted as $S$. Thus potential chain lengths are $0, 1, 2, ..., S$. If there are $N$ SIBs on the RTDR that each have their own stub-chain configuration, then the number of possible stub-chain configurations for the RTDR is: $(S + 1)^N$, and an attacker will need to determine which configuration corresponds to the current DUT if he wants to reverse engineer the location of the instrument interfaces and SIB bits in the data stream.

The use of the chain checker further requires a valid RTDR access stream at all times. This can minimize cipher activity for an adversary without knowledge of the protected RTDR configuration.

5.4.5. Additional Considerations

A semiconductor device’s lifetime has at least two stages—in manufacturing and when deployed in the field. For some companies, it may be desired that protection schemes be disabled during manufacturing and test to ease test access and potentially minimize test time. When the device reaches the end of production, then protection features such as the proposed features can be irreversibly enabled. This can be achieved with electronic fuse-based techniques [64]. This will allow diagnosis and debug to proceed on field returns while still reducing the chances that on-chip IP will be compromised by unauthorized users in the field. To reduce the chances that successfully compromising one device will allow other devices to be easily compromised, unique cipher keys or authentication passwords for each device may need to be implemented. For in production uses, it is critical a trusted site is utilized to fuse in cipher keys as well as stub chain configurations.

The proposed architecture also requires that the IJTAG access software comprehend the mechanisms to access protected IJTAG instruments. Such software should only be licensed
to authorized users, who should also be authenticated.

We can also adapt the dual cipher to impose access restriction on other DUT test access mechanisms. For example, processor test structures requiring more than one input/output pin pair for parallel test program loading can be protected by the proposed dual ciphers.

5.5. Results

To evaluate the overhead added to the design, multiple components were implemented at the register transfer level (RTL). In particular, a Trivium cipher core (instantiated twice) was implemented along with a cipher controller. Because the control sequencing for identical ciphers should be similar, the cipher controller was designed to be scalable to an arbitrary number of stream cipher cores. An authentication block and a chain shift checker were also implemented in RTL. These blocks were then stitched into a top level design block. Note that the integration of the RTDR and stub chains was not implemented due to the effort needed on a server processor design. However, the need to calculate the effect of the stub chains on the chain length was included in the chain shift checker logic.

The RTL model of the proposed system is parameterizable. This allows different design configurations to be explored. Each selected configuration was synthesized with a synthesis tool using a server grade 28nm process technology. The area results were then evaluated against an in-production server processor deployed in the micro-server area, which has a small die size when compared to other server parts. The results are tabulated in Table 5.1.

The first column of the table denotes the number of SIBs assumed to be present in the RTDR. The second column denotes the maximum stub chain size, $S$, for all stub chains. The third column denotes how many keystream bits each cipher core is allowed to generate in a single clock cycle. Multiple keystream bits per cipher core may be needed in cases where an instrument can be configured to consume data from more than one external pad. For example, it may consume data from pins that are not associated with the JTAG port. Such pins may also be used to drive data to multiple instruments. The Trivium cipher can securely support up to 64 keystreams.
The physical implementation area for the cipher cores, cipher controller, authentication block, and the chain shift checker is shown in column four in square microns. In the fifth column, the area of the cipher, control, authentication, and checker logic is compared to the total die area of the server processor. All the configurations target a clock speed of 40MHz with evenly split duty cycles. With a fixed number of SIBs, our data shows that the stub chain size and keystream width affect the overall area of the proposed cipher and control circuitry to some extent, but compared to the area of the processor, the impact is minimal.

Table 5.1: Design Area and Complexity Matrix

<table>
<thead>
<tr>
<th>SIBS</th>
<th>Stub Chain Size</th>
<th>Keystream Size</th>
<th>Area</th>
<th>Area Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>8</td>
<td>1</td>
<td>4207</td>
<td>0.00197%</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1</td>
<td>5100</td>
<td>0.00239%</td>
</tr>
<tr>
<td>200</td>
<td>40</td>
<td>1</td>
<td>5372</td>
<td>0.00251%</td>
</tr>
<tr>
<td>200</td>
<td>80</td>
<td>1</td>
<td>5647</td>
<td>0.00264%</td>
</tr>
<tr>
<td>200</td>
<td>160</td>
<td>1</td>
<td>5919</td>
<td>0.00277%</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>2</td>
<td>4211</td>
<td>0.00197%</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>16</td>
<td>4298</td>
<td>0.00201%</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>32</td>
<td>4396</td>
<td>0.00206%</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>64</td>
<td>4594</td>
<td>0.00215%</td>
</tr>
<tr>
<td>200</td>
<td>40</td>
<td>64</td>
<td>5761</td>
<td>0.00270%</td>
</tr>
</tbody>
</table>

To account for a case where the IJTAG shift-in pattern may be XOR'ed twice with two independently generated keystreams (e.g. if control data shifted in is not overwritten in the CaptureDR state of the JTAG FSM before being shifted out), one must assess the effectiveness of dual ciphering. One way to do so is to examine the randomness of the keystreams and the corresponding ciphertexts. The National Institute of Standards and Technology (NIST) provides a random number generator testing tool based on empirical analysis [99]. We utilize this tool to evaluate keystreams as well as ciphertext generation.

We considered the following combinations: a single cipher keystream, a bitstream obtained by XOR'ing two independent cipher keystreams, and the ciphertext generated from XOR'ing two independent cipher keystreams with plaintext represented by repeating pat-
terns, such as $8'hAA$. For each combination, we generated fifty-five sequences, where each sequence consists of one million bits generated from simulating the RTL of the experimental design. For each combination, the corresponding fifty five sequences were fed into the randomness testing tool as described in [99]. We were then able to collect a significant amount statistical data on each combination of bit streams. Our results indicate no apparent loss of cryptogram randomness from the use of dual ciphers.

5.6. Conclusions and Future Work

The methodology described in this chapter adds three echelons of protection to an IJTAG RTDR and can be deployed with a base authentication layer. The potential area cost of the cipher cores, cipher controller, simple authentication instrument, and chain shift checker is insignificant. The method also enables the device manufacturer to apply RTDR-specific and device-specific IJTAG data encryption and IJTAG network configuration for a low die cost. However, we do anticipate IJTAG access throughput to be impacted by the size of stub chains inserted. Furthermore, we believe it’s possible to significantly enhance IJTAG data protection with additional security echelons.
6.1. Overview

In chapter 5, it can be seen that the IJTAG TAM can be cryptographically protected with dual-stream cipher along with chain-stubbing and access intent checking. This level of protection may not be sufficient against a replay attack — see section 6.4. To counter such threats, a novel secure hash chain based scheme is proposed in section 6.5. The work in this chapter was first published in the proceeding of 2018 International Test Conference (ITC) [56].

6.2. Introduction

Modern Integrated Circuit (ICs) often have multiple cores, including processors, memories, I/O blocks, and interconnects. To aid in the manufacturing test, debug, and configuration of these systems, a large number of embedded instruments are included as well. Such embedded instruments may include Built-In-Self-Test (BIST) circuitry for logic or memory, on-chip sensors, bit-error-rate testers, configuration for SerDes channels, hardware assertion checkers, trace buffers, etc.

These instruments have historically been accessed through serial test data registers (TDRs), i.e. scan chains, that are connected to the IEEE 1149.1 JTAG TAP (Test Access Port) [46]. Unfortunately, placing what may be hundreds of instruments on a single scan chain leads to very long access times, and thus ways of placing different instruments on different TDRs and/or bypassing chain segments have been developed. Among others, this includes the new IEEE 1687 Standard (IJTAG) [49].
Unfortunately, while these embedded instruments are absolutely necessary for many legitimate test, debug, and configuration tasks, they may also open a back-door for an adversary to obtain protected and confidential information or redirect device control to malicious agents. This is especially problematic if the test network can be accessed remotely (e.g. over the internet) or through functional ports.

Multiple approaches have previously been proposed for adding protection to the test and manufacturing subsystems of silicon devices. Some of these methods attempt to prevent access by requiring authentication through the use of challenge/response pairs (CRPs) [10, 98], passwords [82], or keys [34, 121] used to lock/unlock some or all of the test access network. Other approaches have attempted to obfuscate the data entering or exiting the network through information hiding [22] or encryption/decryption schemes [28]. However, while such approaches are useful, they do not necessarily guard against replay attacks (where communication data is intercepted and used for later unauthorized access) unless the key is changed and/or time-dependent. Furthermore, such approaches do not provide “tamper evidence” or “integrity checking.” Thus, they do not indicate that previous unauthorized access has occurred.

In this chapter, we propose another layer of protection that uses the history of device access to both restrict unauthorized access and to provide evidence of previous unauthorized access attempts. This is accomplished through a hash of the data scanned into and out of a device. The hash is stored in a persistent memory and is “chained” such that the hash of each subsequent access is dependent on the previous access history. New accesses to the device require the user to provide data that can be checked against the currently stored hash on chip. This provides additional authentication value. Furthermore, from the user’s perspective, a stored checkable hash may be read from the device on request and checked against an expected value. If the values do not match, the user may infer unauthorized access attempts may have occurred. Note that in the context of this chapter, integrity checking refers to hardware and software methods that detect unintended or unauthorized modifications on messages via computed signatures or digests. This definition is consistent
with NIST’s definitions on integrity checking [81].

Thus, this chapter makes the following contributions:

• A scalable chained hash generation scheme is proposed for IJTAG transactions.

• A set of operations allowing both the device as well as the IJTAG access agent to mutually accomplish integrity checking is explored.

• The ability of enhanced integrity checking to mitigate replay attacks and provide evidence of previous unauthorized access attempts is investigated.

6.3. Previous Work and Background

6.3.1. Previous Work in JTAG and IJTAG Security

Multiple approaches have been proposed for securing test hardware from unauthorized use. In the simplest case, companies have considered fusing off the JTAG port once test has been completed to prevent future access. However, this prevents the JTAG port from being used for future test (e.g. at the board level) or for debug of field returns. Thus, to provide alternatives, multiple researchers have suggested protocols that restrict access for unauthorized users while still providing access to those with the correct “credentials.”

For example, a secret password-based access control locking mechanism was proposed in [82] to restrict control access to the JTAG port. Challenge Response Pair (CRP)-based authentication schemes for JTAG and IJTAG have also been proposed in [10, 22, 89, 90, 98]. Such schemes require both the DUT and the access agent to authenticate based on shared secrets. A CRP scheme can be enhanced with challenges based on true random number generator outcomes. Such an approach has seen industrial adoption [83].

Other researchers have attempted to modify the chains to obfuscate the interaction of the test data and the internal circuitry. For example, in [66] the scan chain was divided into segments that could be randomly reordered if the initial bits shifted into the device did not match a predetermined key.
Additional approaches have focused specifically on the protection of embedded instruments in an IJTAG network. For example, [34, 44, 69, 121] have investigated ways of making it harder to open SIBs by requiring that additional key bits be set to predetermined values. This allows fine-grained protection of the IJTAG network because different instruments or groups of instruments may have their own keys.

While all of these methods may help prevent an unauthorized user without knowledge of the correct keys, passwords, or responses from accessing protected portions of the test network, they do not provide full protection. For example, they don’t prevent an attacker from snooping the communication between the chip and the tester, and they do not protect against replay attacks. (Replay attacks occur when an attacker “watches” the data sent to/from a DUT during successful accesses and then uses the same data sequences to subsequently access the device.)

To address the problem of an attacker obtaining useful information from observing the test data and control sequences, various JTAG and IJTAG encryption schemes that encrypt test access control streams with cryptographic ciphers have been proposed. [25, 26, 30, 57]. In [28], a public key exchange cryptography scheme was proposed to ensure the test access agent and the DUT can achieve mutual authentication as well.

However, while encryption schemes can make it difficult to interpret data sent to/received from the DUT, even encrypted data can be used in replay attacks if the key is not changed. In [32], a method involving hiding information in a pseudo-random bit stream was used to exchange changing key information and restrict the software running on an IJTAG access agent from operating on chips with invalid Chip IDs. The secure portion of the network was also hidden behind Locking SIBs with keys that varied from chip-to-chip and possibly over time based upon the value of a real-time clock. This would help prevent replay attacks, but portions of the approach may be susceptible to side-channel attacks, such as power analysis. Replay prevention is considered in [28] as well, however, it requires the reseeding of a large internal linear-feedback shift register after every authenticated session. These approaches also do not ensure that the data sent to/received from the chip has not been tampered with.
In [98], an approach that uses a message authentication code (MAC) was proposed to be applied on every JTAG data shift-in operation after a JTAG access session has been established with challenge-response pairs. However, once the session is established, the key utilized in the MAC does not change. This potentially exposes each JTAG data shift-in operation to replay-based adversarial manipulation. Furthermore, there is no checking of the integrity of the content shifted out from the DUT. It also does not provide information about previous unauthorized access attempts.

This chapter adds to this previous work by proposing a unified method of identifying when data sent to/received from a device has been modified using a secure hash function. It simultaneously provides evidence of previous access attempts. It also contains the exchange of shared, changing “secret” information that can be used to provide a measure of authentication as well as additional protection against replay attacks. Such qualities are likely to be especially important when communication between the DUT and tester is done remotely over a wired network or through a wireless network, such as may occur during test and debug of a device in the field.

The Internet can be well protected from man-in-the-middle attack against the communication channel when communication are secured with standards and protocols based on HTTPS [1] secured with TLS 1.3 [96]. However, individual computer systems sitting between the DUT and the access agent may still be compromised by an adversary through attack vectors such as malware or phishing attacks that can bypass the HTTPS secured communication channels. Furthermore, HTTPS can secure Internet communication, but after the networked data is localized and decoded into an IJTAG data stream by a computer system, the transmitted data may no longer be secure. In such examples, without additional protection on the IJTAG port, IJTAG communication between the access agent and the DUT may still be susceptible to adversarial interception. The intercepted communication trace can then be used by a replay attack to gain unauthorized access to the DUT. The proposed approach may prevent such an attack since the attacker must know all DUT IJTAG transactions across all DUT life cycle stages, which is less likely. Finally, the approach can easily
work as an additional layer of protection even when other measures, such as encryption and CRP, are being utilized.

6.3.2. SHA3 and Persistent Memory

Secure hash algorithms (SHA) compute a small digest or signature from a digital message. When a strong SHA function is used to extract a message signature, it is computationally infeasible to find a way to alter the message and yet extract the same signature. These algorithms are vital cryptographic tools used to create digital signatures. They are standardized by the National Institute of Standards and Technology (NIST). In this chapter, the SHA3 algorithm is used to generate signatures for IJTAG transactions for integrity checking [79]. A brief overview of SHA3 as relevant is given.

At a high level, SHA3 is constructed with a Sponge Function. The Sponge Function iterates the Keccak-p function to generate a final digest that is half the length of the SHA3 capacity parameter \(SHA3\_CAP\) [79]. To do this, the Sponge Function iterates the Keccak-p function over blocks of \(SHA3\_R\) bits received from the message. As standardized in SHA3, the Input String and the Output String should each be 1600 bits long, referred to as \(SHA3\_L\) [79]. \(SHA3\_CAP\) and \(SHA3\_R\) are parameters specified by the SHA3 standard. \(SHA3\_L\) is the sum of \(SHA3\_CAP\) and \(SHA3\_R\). Each Keccak-p iteration consists of evaluating a Round function 24 times, where the output of one round function is driven to the next.

The SHA3 standard specifies a family of six hash functions with varying signature sizes. These are SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. The sizes of the signatures supported are directly reflected by each function name. For example, SHA3-224 has a signature size of 224 bits, which means its \(SHA3\_CAP\) is 448 bits. The SHAKE labeled functions are extendable output functions where the digest size can be modified; however, their capacity and hence their hash rate, are still derived from their labels—meaning SHAKE128 still has a \(SHA3\_CAP\) of 256.

The proposed method seeks to utilize persistent memory, where memory content is retained even with the device powered off. One such feasible technology is ferroelectric random
access memory (FRAM) [27]. Current works are showing that sub 130nm, FRAM can become feasible [78]. For the purpose of this chapter, a small FRAM with a size of 4KB (kilobytes) can be implemented and made accessible to IJTAG control structures. Because the SHA3 hash digest sizes are multiples of a byte, the FRAM integrated can be byte addressable or organized into entries where each entry consists of multiple bytes to form a hash digest.

6.4. Threat Model

In this chapter, it is assumed that the person running the IJTAG access agent, the DUT, and the communication medium between the two may all be subject to potential compromise. Furthermore, this chapter assumes the IJTAG access agent and the DUT will not be compromised simultaneously.

The IJTAG access agent may correspond to a tester, a benchtop setup with a computer running licensed JTAG/IJTAG software, a remotely accessible IJTAG-based debug system, etc. Compromise of the access agent could occur when the test setup is being run by an unauthorized user or a user who should no longer have access. For example, someone with access to licensed software may want to record the communication sent between the access agent and the DUT so that unauthorized access may be obtained later without the licensed setup. A static authentication approach would allow this later access to occur. In contrast, the proposed chained hashing approach uses the entire access history of a device to repeatedly modify the information that must be exchanged during authentication—making simple replay attacks ineffective. This authentication is also not a one-time event but occurs continuously with each JTAG/IJTAG transaction.

The DUT itself could be considered compromised if it is a counterfeit device or a clone or if it has experienced previous unauthorized access attempts by a malicious agent. Once again, the chained hashing approach we propose that encompasses the entire test access history of a device can help address these issues. Cloned devices would need to replicate not only external characteristics, ChipIDs, keys, etc. of a legitimate device. They would also
need to replicate the hash signatures generated based on all previous test access from the legitimate part. Evidence that someone unauthorized has tried to get into the part (whether or not they were successful) would also appear in the returned signature—signaling that tampering may have occurred. Furthermore, if these tampered parts were later recycled and sold as counterfeits, such tampering would show up during test as well.

Finally, the communication medium itself could be compromised. This is particularly an issue when test and debug occur remotely—for example, when the DUT is in the field. In this case, an adversary could attempt to perform replay attacks and/or maliciously modify the data sent/to from the DUT in a “man-in-the-middle” approach. Note that in some cases even encrypted IJTAG sequences can be manipulated by savvy replay attacks. The attacker can also feign to be the DUT and extract useful IJTAG sequences from the IJTAG access agent. The use of a constantly changing authentication and integrity checking protocol can help to mitigate such attacks.

The following examples further illustrate some of these risks when one or more avenues of compromise are exploited.

6.4.0.1. Example 1 - Remote IJTAG Exploit

Remote DUT field silicon can be now diagnosed via the Internet. In such cases, the debug equipment can be accessed remotely by trusted diagnostic engineers. However, this does not mean the debug equipment, which can physically access the DUT, is safe from tampering. An attacker could snoop on the sensitive information transmitted, record critical control sequences, and then manipulate these sequences to illegally extract information from the DUT or reconfigure the DUT illegally—such as through unauthorized firmware or fuse changes. Furthermore, such an attacker can also send misleading diagnostic data to diagnostic engineers and leave the DUT in a compromised state.

6.4.0.2. Example 2 - Insider Threat

Insiders maybe considered trustworthy, but their privileged access can be exploited. Fur-
thermore, they may exploit their inside knowledge and privilege to devise methods to mali-
ciously gain access to the DUT. In fact, according to [24], more than 74% of data breaches
originate from insider compromises. Work in [94] provides an example of individuals stealing
integrated circuit intellectual properties for their own gain. In these scenarios, challenge-
response approaches can be completely compromised if the CRP information is stolen. En-
cryption keys can be stolen as well. Even if encryption keys frequently do update, insiders
may leverage their inside knowledge to bypass that.

6.5. Proposed Method

To counter the threat model in Section 6.4, this chapter introduces integrity checking
mechanisms protecting IJTAG transactions. This scheme proposes the integration of cryp-
tographically robust hash functions such as the NIST SHA family to generate checkable
signatures for each IJTAG transaction. Each signature can then be used by the DUT as
well as by the IJTAG access agent to check the integrity of completed IJTAG transactions—
ensuring that both entities are trustworthy with respect to each other. From an architectural
viewpoint, the proposed method conforms to the notion of a defensive echelon as discussed
in [57]. Hence, this scheme can form an echelon to be integrated into a layered IJTAG secu-
rity architecture. The proposed initiation protocol is captured in Fig. 6.1. First, the IJTAG

![Diagram of Hash-Based Secure IJTAG Access Protocol]

Figure 6.1: Hash-Based Secure IJTAG Access Protocol

access agent attempts to access the DUT with a mutually synchronized hash signature. If
the signature matches what the DUT expects, the DUT accepts the IJTAG access sequence. Otherwise, the DUT prevents any further IJTAG access until a power-on-reset event.

As the DUT completes the IJTAG access sequence, a new hash signature is generated and stored by the DUT based on the data received by the DUT and returned to the IJTAG access agent. Because the IJTAG access agent is also aware of the content being sent into the DUT and received from the DUT, it can compute a matching hash signature. Thus, as each message is sent, both the DUT and the IJTAG access agent can calculate the new value of the hash and can send the current hash value to the other party to verify that no modifications to the bitstreams have been made. On the next pending IJTAG transaction, the IJTAG access agent would shift in the hash signature to be matched against the signature the DUT computed. The proposed method also provisions a checking operation that would allow the IJTAG access agent to receive the DUT-computed hash signature to ensure communication to the DUT and/or from the DUT itself has not been compromised.

Therefore, the resulting access scheme would have two components, an IJTAG access agent software component as well as a DUT signature generation scheme. Given that software can imitate hardware behavior, the rest of this section focuses on describing the required on-chip design elements for the DUT. The DUT’s block diagram for the proposed integrity
checking scheme is shown in Fig. 6.2. Subsequent sections will elaborate on the specific functionality shown in Fig. 6.2. The secure hash function specified by the SHA3 standard [79] has been chosen for the implementation of the proposed method. SHA3 is one of the latest secure hash standards published by NIST. It offers the benefits of unlimited message size and protection against length extension attacks [13,61,79].

6.5.1. Hashing IJTAG Transactions

The first step in generating a checkable signature is to forward data to be checked into the hash function. The data to be secured must also be organized. One convenient way to do so is to consider the notion of an IJTAG transaction. In this chapter, an IJTAG transaction is triggered by either an IJTAG update event occurring during the JTAG UpdateDR state or during an IJTAG capture event occurring during the JTAG CaptureDR state.

Thus, each transaction corresponds to the bitstream transferred into the chip through TDI and TDO. Note, SHA3 can generate a hash signature on a message of any arbitrary length—which applies to each IJTAG transaction as well.

Given the serial nature of the JTAG and hence the IJTAG data update and capture scheme, the protected messages consist of bits from both TDI and TDO paths. Hence, in Fig. 6.2, an Input String Packer is used to pack serial data streams into SHA3 consumable segments. The Keccak-p function consumes a data segment exactly the length of SHA3_R. This means the Input String Packer needs to fill at least $SHA3_R$ number of bits before the Control FSM invokes the Keccak-p function. At the end of an IJTAG Capture/Shift/Update sequence, it’s possible that a segment is not of length $SHA3_R$. The Control FSM as shown in Fig. 6.2 would then provide padding as specified in [79] to complete the final segment of the hash stream.

The Input String Packer as shown in Fig. 6.2 is implemented as a multi-input shift register with a counter. This shift register latches its inputs and shifts by the number of its data inputs. Each shift is coupled to the counter incrementing by the number of its data inputs. The shift register width is set to $SHA3_R$. The counter increments until $SHA3_R$
number of bits have been filled, and at this point the shift register content would be sent to Keccak-p function. The counter would then be reset. For example, the shift data input and shift data output would be driven into the Input String Packer on each TCK cycle. They would be latched by the shift register. On the next TCK cycle, the 2 latched bits would shift left by exactly 2 bit positions while two new bits would be latched in the shift register. Once the Input String Packer is filled with SHA3_R number of bits, the Control_FSM would invoke the Keccak-p function. Therefore, in a single IJTAG shift-in/shift-out sequence, both transactions can be hashed simultaneously.

Therefore, when IJTAG shifting occurs, the Input String Packer would begin to pack serial IJTAG shift data streams being shifted out and shifted in. When an IJTAG update event is signaled via the IJTAG Interface as shown in Fig. 6.2, the hash signature would be generated incorporating both shift-in and shift-out data (the captured data).

This formulation can be expressed via Eq. 6.1 below, noting that TL refers to the length of the transaction and SI and SO refer to the shift-in and shift-out streams, such that SI[0] refers to the first bit shifted in and SO[0] refers to the first bit shifted out. In addition, SigNxt refers to the signature that would be computed.

$$\text{SigNxt} = SHA(\{SI[0], SO[0], SI[1], SO[1], \ldots, SI[TL - 1], SO[TL - 1]\})$$  \hspace{1cm} (6.1)

IJTAG access can also be coupled with direct access to other test channels. One example would be concurrent scan test data access along with IJTAG access. These test channels would not be driven via TDI or drive TDO, but they can be be connected to instruments controlled by the IJTAG accessed RTDR or other TDRs. In such cases, it’s entirely possible to feed all test access input and output data streams into the Input String Packer via Additional SHA Inputs — up to SHA3_R number of bits, as shown in Fig. 6.2. Now the signature computation in terms of the SHA can be expressed as in Eq. 6.2. Note SI_i denotes the i^{th} input bit and SO_i denotes the i^{th} output bit to Input String Packer. NCh is
the number of channels.

\[
\text{Sig\textit{Nxt}} = \text{SHA}(\{\text{SI}_1[0], \text{SO}_1[0], \text{SI}_2[1], \text{SO}_2[1], \ldots, \\
\text{SI}_{\text{NCh}}[\text{TL} - 1], \text{SO}_{\text{NCh}}[\text{TL} - 1]\})
\]

(6.2)

In fact, when additional test channels are driven to the \textit{Input String Packer}, a signature incorporating test results of the DUT other than results captured in the IJTAG RTDR can be generated. Such channels can be either input test channels or output test channels. This envisioned capability provides additional protection for other inputs and outputs leading to the notion of multilateral integrity checking in this chapter.

6.5.2. Chained Hashing

Thus, the proposed method can generate a signature for each IJTAG update/capture transaction pair, where hashing occurs during shifting with the final hash invocation and signature collection at the end of shift and prior to the UpdateDR state. Next, this signature needs to be stored and used for checking. In the context of this chapter, this signature is the outcome of hashing both the IJTAG shift-in bitstream as well as the shift-out bitstream. To ensure the output channel as well as the input channel are not modified from the DUT’s point of view, the DUT would capture this signature to be used as a reference in the \textit{Reference Signature Register}, as shown in Fig. 6.2. The comparison for this reference is sent in via the IJTAG access agent. The access agent is expected to generate a hash signature that should be exactly the same as the signature computed by the DUT’s hash generation.

The IJTAG access agent needs to demonstrate to the DUT that it is accessing the DUT with unmodified input and output channels. To do so, it would shift its own computed signature into the device’s \textit{Input Signature Register}. This register is an always inserted segment on the IJTAG RTDR when hash checking is enabled. The DUT’s \textit{Control FSM} would compare the value against the value from the \textit{Reference Signature Register} after the \textit{ShiftDr} state. If an exact match is found at the end of IJTAG shifting, the IJTAG operation would be allowed to continue—the \textit{Control FSM} would cause the \textit{Gating Control Logic}, as
shown in Fig. 6.2, to allow the IJTAG UpdateDR and CaptureDR signals to be ungated. In particular, the Gating Control Logic logically ANDs the signal that indicates a successful comparison between the values in the Input Signature Register and Reference Signature Register with the UpdateDR and CaptureDR signals that fanout to the IJTAG test network. However, when the comparison fails, the IJTAG UpdateDR and CaptureDR signals are gated off to prevent reading from or writing to logic and instruments attached to the IJTAG network.

In the process of signature checking, it can be observed that the signature state driven by the access agent is naturally shifted into the input stream of the hash function as well. Therefore, an IJTAG hash chain is formed. That is, to derive a valid hash signature from only the data content of an IJTAG transaction during an IJTAG access session, all preceding signatures must be derived successively from the very first IJTAG transaction in the respective IJTAG access session. This derivation is shown in Eq. 6.3, where the next signature, $SigNxt$, is now formed by hash chaining the current signature, $SigCurr$, along with the content of the current transaction.

$$ SigNxt = SHA(SigCurr, \{ SI_1[0], SO_1[0], SI_2[1], SO_2[1], ..., SI_{NCh}[TL-1], SO_{NCh}[TL-1] \}) $$

The consequence of chained hashing means that an adversary attempting to gain access to the DUT’s protected IJTAG RTDR without recording or snooping on any prior IJTAG transactions needs to know all previous signatures computed in the DUT’s IJTAG access session — this potentially requires the adversary to be aware of all previous IJTAG access data and access sequencing in order to recreate the right signature to allow the adversary to successfully complete a current IJTAG transaction.

Assuming the Input Signature Register is shifted in from its most significant bit (or MSB) and is inserted as a segment immediately before the shift out path, the IJTAG bit stream can be formed as follows: \{IJTAGData, MatchingSignature\}. Hence the signature bit
stream is shifted in first. However, in a final design implementation, there does not need to be restrictions on where the Input Signature Register is inserted in the IJTAG RTDR. The chained hashing would take effect regardless of where the Input Signature Register is positioned along the RTDR chain.

6.5.3. Incorporating Persistent Memory

Chained hashing can significantly increase the difficulty an adversary has to overcome to access the protected IJTAG RTDR during or after an IJTAG access session when the device remains in a powered-on-state. However, if the IJTAG session terminates with the device powered off and powered on again, the effects of chained hashing are reduced or entirely eliminated. Furthermore, if the adversary is able to record both the shifted in shifted out data on the DUT’s latest IJTAG transaction, then the adversary can compute the hash signature and gain access on the next transaction. Hence, to mitigate such attacks, the hash signature should be made persistent.

Hash signatures can be stored in persistent memory. By making chained hash signature persistent, the session dependence of the security of chained hashing is eliminated. FRAM is one potential candidate for persistent memory [27]. Further, the proposed method assumes that FRAM IPs can be integrated in a system-on-chip (SoC). Given the hash signature sizes in SHA3 are multiples of bytes, the integrated FRAM can be organized into chunks of bytes, where each chunk should be the size of a hash signature in bytes. Therefore, the adversary can no longer utilize session termination conditions such as power-on-reset or session termination protocols to restart the hashing on the DUT. However, to avoid an adversary that would compute the integrity hash signature and hence pass the device level integrity check, the integrity signature needs to be more secure and robust.

\[
\text{SigNxt} = \text{SHA}(\{\text{PerSigInit}, \text{Padding}\}, \text{PerSigCurr}, \{SI_{1}[0], SO_{1}[0], SI_{2}[1], SO_{2}[1], ..., SI_{NCh}[TL - 1], SO_{NCh}[TL - 1]\})
\]
To increase the difficulty of an adversary being able to compute the SHA3 signature just as an IJTAG accessing agent would do with recorded IJTAG transactions, the IJTAG transactions can be encrypted. This can be accomplished by encrypting the IJTAG data stream [25,26,30,57]. However, the proposed method aims to orchestrate a security defensive scheme with limited dependence on alternative protections, such as encryption techniques. Thus, when persistent memory is available for integrity checking purposes, another unique hash signature computed during one of the past IJTAG transactions can be inserted into the signature generation of the current IJTAG transaction. Alternatively, a factory-generated device-unique signature can be stored. This would force the adversary to not only have knowledge of current IJTAG transactions, but previous transactions as well. The signature expression is shown in Eq. 6.4, where $\text{PerSigInit}$ is the additional signature utilized from past IJTAG access sessions or a device unique identification number accessible via the persistent storage. In Eq. 6.4, the current signature in the DUT IJTAG is now persistent hence labeled $\text{PerSigCurr}$. At the end of the transaction, $\text{PerSigCurr}$ would be updated to $\text{SigNxt}$.

So far, at least two hash signatures should be stored in persistent memory: the signature that is used for chaining IJTAG transactions, and other one being either a recorded signature from past IJTAG accesses or a device unique one. To increase the difficulty of reverse engineering hash signature generation, more than one past recorded signature should be stored. Further, the authorized IJTAG access agent (able to compute the hash matching that of the device) should be allowed to modify the hash chain by replacing $\text{PerSignInit}$ with the XORing of it with a signature from previous transactions. Thus, an adversary now has to overcome not a fixed set of signatures, but the hash chaining of a set of potentially modified signatures. Therefore, the expression for the $\text{SigNxt}$ can be modified as shown in Eq. 6.5, where $\text{PerSigInit}$ is substituted to $\text{PerSigSel} \oplus \text{PerSigInit}$. $\text{PerSigSel}$ is selectable by the IJTAG access agent. This signature as computed according to Eq. 6.5 is considered a Distributed Signature. The design intent for signature distribution is to allow the access agent to store a set of signatures in different physical storage media (such as various cloud storage elements), where each medium would require unique authentication. In this way, the
compromise of one storage medium by an insider may not allow an adversary to compute a valid hash signature, which can be applied to help address example 2 in Section 6.5.3. By XORing \(PerSigInit\) and \(PerSigSel\), even when all prior IJTAG transactions are captured by an adversary from an unprotected channel, the adversary is still compelled to brute force extract \(PerSigInit\) in order to compute the matching hash digest provided \(PerSigInit\) is not exposed. Alternatively, if the adversary attempts to alter the \(PerSigSel\) update operations during an IJTAG transaction, such efforts must still resort to brute force extraction of \(PerSigInit\). Noting that \(PerSigInit\) must be made unwritable by any \(PerSigSel\) update operations.

\[
SigNxt = SHA\{\{PerSigSel \oplus PerSigInit, Padding\}, PerSigCurr, \{SI_1[0], SO_1[0], SI_2[1], SO_2[1], ..., SI_{NCh}[TL - 1], SO_{NCh}[TL - 1]\}\}
\]

6.5.4. Integrity Checking Operations

Prior Sections 6.5.2 and 6.5.3 show that in order to fully leverage the proposed capabilities, a trusted IJTAG access agent has to be able to successfully perform at least three integrity checking operations. First, the IJTAG accessing agent should be able to enable the DUT to check the integrity of IJTAG transactions with the selected signatures stored in the DUT’s permanent memory. Second, the IJTAG access agent should be able to modify the hash chaining. Finally, the IJTAG access agent should be able to check whether the device is maintaining integrity.

These three operations would specify opcodes which would be shifted in during the IJTAG transaction. Thus, the Input Signature Register can be extended to accommodate two more fields: 1) a two bit wide opcode used to indicate one of three operations (Load, Update, or Check), and 2) \(PerMemAddr\), which is used to point to a location in the persistent memory
where the Distributed Signature is stored. Any unspecified operation decoded from the opcode would be aliased to a Load operation. If the number of entries allocated for hash signatures in the persistent memory array is PerMemSz, then the field PerMemAddr is \( \log_2(\text{PerMemSz}) \) bits long.

6.5.4.1. Check Operation

When a trustworthy IJTAG access agent is initiating an IJTAG access session or check on whether the DUT may be compromised, the access agent should be able to check the device to ensure the hash signature is matching expectations. Thus, the Check operation is introduced. Referring to Fig. 6.2, the Reference Signature Register can be captured and shifted out via IJTAG to the trusted IJTAG access agent. The content of the Reference Signature Register can be evaluated by the DUT’s hash operation just as the Load operation, except for the Check operation, the Reference Signature Register’s content is latched in position of the Input Signature Register, which in the two other operations would be captured with content of all zeros. This means when the access agent reads the captured data from an IJTAG transaction with a check operation, expected signature data would be shifted out. In the other two operations, the IJTAG segment for the Input Signature Register would be zeros.

6.5.4.2. Load Operation

The goal of the Load Operation is to specify the address, PerMemAddr, of the Distributed Signature, PerSigSel, to be concatenated during the evaluation of SigNxt as well as allow the DUT to perform continuous checking on the signature shifted in by the access agent. Subsequently, the Control FSM would decode the operation, and when the DUT verifies that the access agent has shifted in the correct signature, the Control FSM would update the hash function with the next UpdateDR. If matching failed, no updates would occur to PerSigSel. If a match is found, SigCurr would be updated with the hash digest of both the IJTAG shift-in stream as well as the IJTAG shift-out stream generated in a particular load operation.
6.5.4.3. Update Operation

The Update operation would occur when the IJTAG access agent wants to update the Distributed Signature stored in the PerMem. This would occur when the Control FSM verifies that the input signature is expected, and on the UpdateDR state of the current IJTAG transaction when SigNxt evaluation is complete and updated to SigCurr, SigCurr would then be written into PerMem at the address specified by PerMemAddr. Note, this write operation does not affect the content of PerSigSel - which is updated on a load operation. Furthermore, Update operation cannot update PerSigInit.

6.5.5. Control Finite State Machine Overview

![Control FSM Diagram](image)

Figure 6.3: Control FSM Diagram

A simplified diagram of the Control FSM is shown in Fig. 6.3. At a reset event, such as power-on-reset or a JTAG-triggered reset, the Control FSM would be set to the Reset state. When reset becomes inactive, the Control FSM transitions to the Idle state. In the Idle state, the SHA3 function (Keccak-p) is initialized. In addition, the Control FSM would load the Reference Signature Register with a hash signature from PerMem read from a default
When the JTAG state machine transitions to the ShiftDR state, the Control FSM transitions to running the hash function as indicated in the RHash state. In the RHash state, the Control FSM allows Input String Packer to collect the incoming bits from IJTAG RTDR shift-in and shift-out path. When SHA_CAP bits are collected, the Control FSM captures content of the Input String Packer into another register while Input String Packer can keep on shifting, and the capture data would feed the SHA3 Keccak-p function and run its hash operations.

When the IJTAG shift operation stops, the Control FSM would now transition to the FHash (for final hash) state to finalize the SHA3 computation—and the Control FSM would add the necessary padding as per [79] up to SHA_CAP bits with the bit stream accumulated in the Input String Packer. The data in the Input String Packer will be sent to the SHA3 Keccak-p function. The Control FSM would also request the SHA3 circuitry to generate the digest or signature once hash computation completes. This computation can be accomplished in one test clock cycle or multiple cycles. In the case where the SHA3 digest is generated in one test clock cycle, the Control FSM would transition to the Update state if the shifted in signature in the Input Signature Register matches the content in the Reference Signature Register, which was generated from the previous transaction.

In the Update state, the hash digest would be written into PerMem, the persistent storage element. If the hash computation is pipelined and requires more clock cycles, the Control FSM would transition to the EHash, or extended hash state, where the hash function is allowed finish computation. After this, the Control FSM would transition back to the Idle state to wait for the next IJTAG transaction.

The Control FSM accounts for failing conditions by detecting abnormal operations and input signals. When exception signals are sent in via the Exceptions Interface, as shown in Fig. 6.2, during the Update, EHash, or Idle states, the Control FSM will transition into the Fail state. When the signature comparison in the FHash state registers a mismatch, the Control FSM will transition into the Fail state. Furthermore, when the the Control FSM enters into any undefined states, it will transition into the Fail state. For the Control FSM,
the Fail state is a self looping state where IJTAG capture and update signals are gated off, hash operation stops, and the integrity checking failure flag to the rest of the DUT would be set.

The proposed integrity checking system assumes the overall on chip security system governing the DUT would be responsible for persistently tracking and updating unsuccessful access attempts inclusive of hash signature mis-comparisons. Therefore, upon a certain number of unsuccessful access attempts, regardless of power-on-reset events or system reset events, the DUT should deny all access as the IJTAG access agent is no longer deemed trustworthy. However, a trustworthy IJTAG access agent can attempt to re-establish trust by initiating a CRP protocol, where the DUT would issue a challenge based on Distributed Signatures and alternative on-chip secrets.

Note that for the first integrity checking enabled access, the current signature is synchronized between the DUT and IJTAG access agent during a manufacturing step. It is assumed the manufacturer would store the initial set of signatures where each signature access may require unique authentication. The authorized IJTAG access agent is assumed to be able to obtain initial secure access to each signature.

6.5.6. Impact of Integrating Integrity Checking

Design integrity checking in the DUT can be simplified if the DUT contains persistent memory such as FRAM and SHA3. A small section of FRAM can be reserved for the IJTAG integrity checking purpose. An integrity checking control block akin to Control_FSM can be devised to control the FRAM segments and the hash function for the IJTAG integrity checking. In addition to the die area used for integrity checking, each IJTAG Capture/Shift/Update operation will now require $\text{length}(\text{PerSigCurr}) + 2 + \log_2(\text{PerMemSz}) + \text{cycles}(EHash)$ more cycles to complete, where $\text{length}(\text{PerSigCurr})$ refers to the length of the hash signature, the opcode costs 2 shift cycles, and cycles for shifting in PerMem address needs to be accounted for as well as the number cycles for hashing to complete.
6.6. Experimental Results

To examine and explore the feasibility of the proposed method, a number of design configurations matching the block diagram shown in Fig. 6.2 were explored, contrasting area and configuration trade-offs. A custom SHA3 Keccak-p function was designed to meet the SHA3 Standard [79]. The area of each configuration is also compared against a multi-core 28nm server processor. The timing impact is discussed as well. Due to the unavailability of 28nm FRAM area and performance data, the area impact of a small block of 28nm FRAM is not included in the experimental data. However, assuming that 4KB of FRAM is used for Distributed Signature storage and an FRAM bit cell area is approximately 2X of a SRAM bit cell, the FRAM area impact should amount to the addition of approximately 8KB of SRAM.

The proposed method with the exception of the FRAM model is implemented with synthesizeable SystemVerilog at the Register-Transfer-Level (RTL). The design has two main tunable parameters. The first parameter is the SHA3 signature size. These correspond to 224, 256, 384, or 512 bits. The other parameter is the number of stages the SHA3 Keccak-p function would be organized into.

The Keccak-p function consists of 24 daisy chained round functions. This entire function can be implemented combinatorially. The combinational implementation of the Keccak-p function can simplify the Control_FSM design by reducing one state as noted in Section 6.5. Each round function can also be implemented as a sequential stage where its contents are registered to implement a sequential Keccak-p circuit.

The RTL is designed to support both of these parameters: the number of sequential stages and the hash signature size. For each signature size, the Keccak-p function can be divided into up to 24 pipelined sequential stages, where each stage should implement a round function. The experiments explore 1, 3, 6, 12, and 24 sequential stages, where a stage of 1 indicates a combinational implementation of Keccak-p.

The aforementioned combination of design parameters led to 20 main configurations. The latest synthesis tool targeting a 28nm process design library is deployed to examine potential
Table 6.1: Design Configuration Area Table

<table>
<thead>
<tr>
<th>Config</th>
<th>SigSz</th>
<th>Stg</th>
<th>TArea</th>
<th>SArea</th>
<th>CbArea</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>sha3_224_p1</td>
<td>224</td>
<td>1</td>
<td>25465</td>
<td>3294</td>
<td>22171</td>
<td>0.0119%</td>
</tr>
<tr>
<td>sha3_224_p3</td>
<td>224</td>
<td>3</td>
<td>23810</td>
<td>1950</td>
<td>21859</td>
<td>0.0111%</td>
</tr>
<tr>
<td>sha3_224_p6</td>
<td>224</td>
<td>6</td>
<td>12719.8</td>
<td>1797.8</td>
<td>10922</td>
<td>0.006%</td>
</tr>
<tr>
<td>sha3_224_p12</td>
<td>224</td>
<td>12</td>
<td>5176.9</td>
<td>1798.8</td>
<td>3378.1</td>
<td>0.0024%</td>
</tr>
<tr>
<td>sha3_224_p24</td>
<td>224</td>
<td>24</td>
<td>3748.4</td>
<td>1799.8</td>
<td>1948.6</td>
<td>0.0018%</td>
</tr>
<tr>
<td>sha3_256_p1</td>
<td>256</td>
<td>1</td>
<td>18751.3</td>
<td>239.8</td>
<td>18511.5</td>
<td>0.0088%</td>
</tr>
<tr>
<td>sha3_256_p3</td>
<td>256</td>
<td>3</td>
<td>23932.56</td>
<td>2150.52</td>
<td>21782.03</td>
<td>0.0112%</td>
</tr>
<tr>
<td>sha3_256_p6</td>
<td>256</td>
<td>6</td>
<td>12754.97</td>
<td>1825.16</td>
<td>10929.81</td>
<td>0.006%</td>
</tr>
<tr>
<td>sha3_256_p12</td>
<td>256</td>
<td>12</td>
<td>5166.81</td>
<td>1826.14</td>
<td>3340.67</td>
<td>0.0024%</td>
</tr>
<tr>
<td>sha3_256_p24</td>
<td>256</td>
<td>24</td>
<td>3782.94</td>
<td>1827.13</td>
<td>1955.81</td>
<td>0.0018%</td>
</tr>
<tr>
<td>sha3_384_p1</td>
<td>384</td>
<td>1</td>
<td>19463.84</td>
<td>348.24</td>
<td>19115.61</td>
<td>0.0091%</td>
</tr>
<tr>
<td>sha3_384_p3</td>
<td>384</td>
<td>3</td>
<td>24254.34</td>
<td>1932.58</td>
<td>22321.76</td>
<td>0.0113%</td>
</tr>
<tr>
<td>sha3_384_p6</td>
<td>384</td>
<td>6</td>
<td>12904.58</td>
<td>1933.57</td>
<td>10971.01</td>
<td>0.0060%</td>
</tr>
<tr>
<td>sha3_384_p12</td>
<td>384</td>
<td>12</td>
<td>5359.19</td>
<td>3424.64</td>
<td>1934.55</td>
<td>0.0025%</td>
</tr>
<tr>
<td>sha3_384_p24</td>
<td>384</td>
<td>24</td>
<td>3938.84</td>
<td>1935.54</td>
<td>2003.3</td>
<td>0.0018%</td>
</tr>
<tr>
<td>sha3_512_p1</td>
<td>512</td>
<td>1</td>
<td>27428.54</td>
<td>3496.34</td>
<td>23932.2</td>
<td>0.0128%</td>
</tr>
<tr>
<td>sha3_512_p3</td>
<td>512</td>
<td>3</td>
<td>24399.5</td>
<td>2041.84</td>
<td>22357.66</td>
<td>0.0114%</td>
</tr>
<tr>
<td>sha3_512_p6</td>
<td>512</td>
<td>6</td>
<td>13051.09</td>
<td>11008.26</td>
<td>2042.83</td>
<td>0.0061%</td>
</tr>
<tr>
<td>sha3_512_p12</td>
<td>512</td>
<td>12</td>
<td>5508.13</td>
<td>2043.82</td>
<td>3464.31</td>
<td>0.0026%</td>
</tr>
<tr>
<td>sha3_512_p24</td>
<td>512</td>
<td>24</td>
<td>4078.84</td>
<td>2044.8</td>
<td>2033.58</td>
<td>0.0019%</td>
</tr>
</tbody>
</table>

The effects of implementing the proposed method. The results are tabulated in Table 6.1. Row 1 contains entries for data labels. Column 1 specifies the design configuration name. Column 2 refers to the signature size. Column 3 indicates pipeline staging. Column 4 refers to the total area in square microns of the design. Column 4 refers to the total area in square microns of the design. Column 5 points to non-combinational design area. Column 6 shows combinational design area. These two area data points show the level of optimization achieved for the Keccak-p function. Finally, column 7 tabulates the area overhead as percentages over a 28nm in-production server processor. In terms of timing, all design configurations met the synthesis speed target of 500MHz. A 463mW (milliwatt) upper bound is observed for all configurations for power load.
The SHA3 signature size is proportional to the security of the signature. However, it may also correlate to an area increase as well. Thus, the design configuration with the strongest signature, the sha3_512 variations, require more area when compared to the corresponding configurations with smaller signatures for all pipeline stages.

For the version of the design where the Keccak-p function is implemented in combinational logic, sha3_256_p1 and sha3_384_p1 instances appear to require the least amount of area. Combinational and non-combinational area consumption for sha3_224_p1 and sha3_512_p1 is high due to higher hash rate for sha3_224 and high hash capacity for sha3_512_p1, as both conditions can require the synthesis tool to utilize more sequential as well as combinational cells. According the NIST SHA3 standard [79], the sha3_384 based implementation should have security strength of 384 bits for first and second preimage attacks and 192 bits for collision attacks. Therefore, the sha3_384_p1 design configuration can be a good candidate for combinational implementation of the proposed method as it achieves high security with small area impact.

Pipelining the Keccak-p function requires the Control FSM to contain the EHash state and IJTAG access agent software to wait for hashing to complete. From Table 6.1, pipelining with 24 stages provides the greatest area reduction. Sha3_224_p24 has the smallest area. However, sha3_512_p24 only requires marginally more area (4078.38 microns versus 3748.4 microns) than sha3_224_p24 but provides more than 2X increase in security: 256 security bits in collision attacks and 512 security bits in first and second preimage attacks vs 112 security bits in collision attacks and 224 security bits in first and second preimage attacks. The sha3_512_p24 configuration should provide a good tradeoff between security and area for implementing the proposed method with pipelined Keccak-p.

6.7. Conclusions

In this chapter, an IJTAG integrity checking method incorporating a hash chain and persistent memory was proposed. This method allows every IJTAG transaction to be checked by the silicon device or by its accessing systems or agents. It may also be used to provide
tamper evidence and can make it harder for an adversary to execute replay attacks.

This approach can be built alongside existing IJTAG-based security features, and can work with them to provide extra layers of protection (e.g. second factor authentication). The silicon area footprint of the proposed method does not appear costly even when high-quality hashes are used.
Chapter 7
Systematic Test Generation for Secure Hardware Supported Virtualization

7.1. Overview

The previous three chapters, chapters 4, 5, and 6, this dissertation delved into the verification and echeloned protection of IJTAG to enhance IJTAG security thereby enhancing the security of embedded memories. But once the IC device is deployed in functional operation, the device may be exposed to multi-faceted attacks. One potential attack that may occur in a virtual machine hosting server system would be adversarial virtualize machine attempting to access confidential data of another virtual machine. Fortunately, chip designers do offer functions that can offer protection for such type of attacks. This does mean that these functionalities need to be verified. This chapter proposes a novel method to systematically verify processor virtual machine protection functions, see section 7.4. These virtualization protection functions impact embedded memories such as the core cache subsystem. Therefore, sound verification for these features can directly provide assurance of the security of the embedded memory in these processors. The work in this chapter was first published in The 15th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC) [55].

7.2. Introduction

The drive to increase data center density, efficiency, utilization, and security is converging into new design criteria for semiconductor devices developed for data center solutions [31, 36]. Thus, the central processing unit (CPU) must be architected to meet these emerging requirements. More specifically, processors need to be architected to accommodate more cores and more threads [53, 65, 71]. More efficient on-chip as well as off-chip communication
systems need to be developed for processors as well [36]. Virtualization and hardware-assisted virtualization methods have also been developed to increase processor utilization [104].

Unfortunately, without adequate protection, virtual machines (VMs) and their respective host or hypervisor can be subject to intrusions and malicious execution changes. In addition, VMs and hosts need protection from each other. For example, a malicious guest (where a guest is defined as the program executed by the VM) can attempt to snoop into or modify its host’s memory image. Conversely, an inquisitive or malicious host may attempt to access a guest’s instruction stream and data from the guest’s memory space. Therefore, extra security hardware should be considered to afford greater security for virtualization-supporting processors. A large number of hardware techniques have been proposed to address security for virtualization systems. For example, memory encryption is commonly used [31] to protect system memory. An instruction and data encryption based execution system is demonstrated in [91]. Processor based memory encryption systems allowing for memory tamper resistance via counter-mode encryption is described in [43, 97, 106]. In such cases, the plaintext is XOR’ed with write counts and a one-time pad before being encrypted for blocks or segments of the memory. Additionally, hashes can be computed for memory writes and checked on reads to ensure data integrity [43, 67, 106]. Further, work in [67] proposed to apply integrity checks on speculative fetches.

Hardware resource-partitioning-based approaches for isolating secure execution from an unsecure execution environment have emerged as well. In [116], it was shown that via zoning and partitioning of on-chip memory, storage, and peripheral access and control, two virtual CPUs can be derived—one of which is secure while the other is left insecure. This is true even though both virtual CPUs would share the same physical CPU with the technology provided in [7].

Virtual machines can be described by a data structure storing processor internal states, such as general purpose register states, a stack pointer, processor control register states, privilege levels, the program counter state, the guest program pointer, guest page table location, etc. A single threaded processor can potentially execute an arbitrary number of
VMs [4, 6]. Therefore, if the aforementioned hardware schemes are utilized to impose VM-to-host and VM-to-VM isolation, significantly more control structures need to be added. For example, works such as [43, 97] do not distinguish host from guest or guest from guest unless additional hardware is added. Architectural innovations are emerging to utilize VM-aware memory encryption to enforce VM-to-host and VM-to-VM isolation [4]. Clearly, architectural support and the underlying design structures for VM-aware memory encryption or VM-aware isolation schemes would require adequate design verification effort to ensure production processors are delivered without security relevant design errors that could induce exploitable design vulnerabilities.

A substantial amount of work on verifying hardware security features is available. In [45], security property language-driven formal approaches are proposed. However, it is not clear that VM-to-VM cryptographic isolation can be modeled adequately with [45]. In [105], formal verification tools were applied to verify security properties at the system-on-chip (SoC) level. For large processor designs containing multiple cores each supporting multiple threads, such an approach may not scale. Novel approaches utilizing game theory to expose design flaws have been proposed as well [103]. Unfortunately, validation of data center critical multi-core and multi-threaded processors supporting a high number of encrypted VMs or hardware isolated VMs appears to be lacking. To address this issue, this chapter provides a practical systematic approach to automatically generate tests that can exercise the CPU design structures that manage and maintain VM-to-host and VM-to-VM isolation, thereby enabling sufficient validation of the CPU security features.

This chapter provides the following contributions:

- A systematic test generation scheme scaling to an arbitrary number of encryption enabled VMs and threads.
- An evaluation of the effectiveness of the proposed system.
- Strategies to deploy the proposed system.
7.3. Background

The proposed solution aims to provide a foundation for multi-threaded VM test generation, which can then be leveraged to test the various micro-architectural conditions. The following discussion aims to further clarify the specific requirements that need to be met for this system.

7.3.1. Scalable Virtualization Support

Modern multi-threaded CPUs must be able to support a high number of securitized VMs of various security levels, and memory types and sizes. Therefore, the proposed system must be able to generate an arbitrary number of virtual machines to more fully exercise the scalability of the design. Further, to ensure the processor is capable of executing VM programs, the system should also be capable of generating guest programs. Such guest programs may also execute encrypted code and access encrypted data when VMs utilize encryption to protect their respective content [4]. This implies the system must account for VM-specific memory allocation with various sets of attributes.

The need to allow the CPU and its subsystems to be loaded with a high number of hardware supported encrypted VMs can be important. The execution of each encrypted VM would cause supporting CPU hardware resources, such as the cache subsystem, to be stressed. If a large number of VMs have been executed by the CPU, and if only a subset of the VMs are still active, the CPU may need to allocate resources only for the active set of VMs by purging out data cached by the older VMs. For unencrypted VMs, the CPU’s resource allocation policy may inherently handle cache usage. However, for encrypted VMs, the VM isolation property must be enforced. This may require critical specialized hardware involvement.

7.3.2. Virtual Machine Execution Scheduling

In a processor, hundreds or thousands of securitized VMs may be processed concurrently. Therefore, the processor design model under design verification must be exercised to ensure
it can tolerate and execute the maximum number of allowed VMs concurrently. This means the generated test program must be able to execute a large number of VMs. Because the number of VMs is typically greater than the number of threads, each thread can only execute a set of VMs sequentially. Therefore, executions of VMs need to be scheduled for each thread.

7.3.3. Virtual Machine and Host Interaction

The host must also be able to exercise VM controls in addition to executing the VM. For example, the host may need to manage the VM cache presence in the memory hierarchy. The host must also ensure VMs are executed coherently. For example, the same VM control structure cannot be executed concurrently on two threads because that would cause the two threads to potentially corrupt each other’s states—which would lead to unintended test failure.

7.3.4. The Problem

The aforementioned requirements for verifying such a processor design require stimuli that can span a large number of VMs and a high number of threads. The existing manual test methodology typically requires the test author to understand a specific set of microarchitectural properties to compose tests to exercise and check such properties. For a CPU design supporting a small number of encrypted VMs, this approach may still be effective. However, when a large number of encrypted VMs need to be supported by hardware concurrently, the design search space to explore increases dramatically.

For example, if a processor supports up to one hundred encrypted VMs, each one of those VMs can be in various stages of execution, with their corresponding state information stored in the main system memory. This implies that the processor must be able to distinguish each encrypted VM from another that is also in an executed state, or else potential VM leakage could occur. To help verify that this is done correctly, a labor intensive and highly time consuming (in the number of weeks) manual assembly test can be written to stimulate a particular set of ordered micro-architectural events to test the processor’s management of
encrypted guests. Unfortunately, the manual generation of many such tests is prohibitively expensive. Thus, while manual test composition may be useful for testing a limited set of targeted processor security properties (such as VM data structure validation checks), it would not be sufficient, nor would it fully explore the design space in a timely manner. Therefore, a highly automated verification test generation system was needed.

### 7.4. Proposed Solution

#### 7.4.1. Test Generation Flow Description

![Test Generation Flow Diagram](image)

The proposed automatic test generation scheme is shown in Fig. 7.1. The first step of the test generation flow aims to allocate memory for all the VMs. This would include program memory as well as guest data memory segments. In this step, each VM is allocated its memory in the form of pages. Hence, corresponding page tables are set up as well. Note
that this step of the test generation flow also allows for randomization of memory types. For example, some pages could be encrypted, some pages could require uncacheability, etc. [4]. Randomization of memory types can be important for high quality verification because the types used may influence the memory access transactions being sent out from the CPU core to the memory controller. Further, encrypted memory regions are vital to the exercising of encrypted VM supporting hardware.

After memory allocation, the system next targets guest code creation, as shown in Fig. 7.1. Every VM would need its own guest code—in other words, each VM must contain its own instruction sequences to be executed by a given VM’s program. There are two key concerns here: 1) guest code execution correctness, and 2) VM exit points. The guest code generation flow is shown in Fig. 7.2.

The system generates guest code in iterations, where the number of iterations corresponds to the number of times a particular VM is expected to be executed, as shown in Fig. 7.2. For each iteration, the system can randomly choose to insert User Code Segments or Default Code. Alternatively, the user of the system can enforce either User Code Segments or Default Code insertion for every guest.

For Default Code in Fig. 7.2, the test generation system generates a main memory access code that randomizes access addresses across all guest-addressable pages. The corresponding memory accesses of guest-addressable memory may arise through the fetching of guest program instructions in standard program order, the execution of test program branches (e.g. used for loops), guest program data memory fetches, and guest program data stores.

Alternatively, highly directed or targeted manual code sequences, shown in Fig. 7.2 as User Code Segments, can be inserted as well—e.g. this would allow the targeting of critical micro-architectural logic. At the end of the guest code, an exit point is introduced. This exit means that the guest will relinquish control of the thread back to the hypervisor. The test generation system at this point would examine the intended number of iterations this particular VM is expected to execute. If more iterations are needed, it would continue to generate guest code in a looped manner until the user-controlled iteration is reached.
The system treats *User Code Segments* and *Default Code* as functions supporting randomization. For example, memory access is expected to be coded with address randomization and data randomization - hence allowing different types of memories to be addressed and different data to be sent across the processor. For *User Code Segments*, it is expected such a function would consume randomized inputs to allow guest behavioral variations. This allows each iteration of the guest code execution to explore a different portion of the design space.

![Diagram of Guest Code Generation](image)

The mere availability of guest code does not allow VM execution. The test generation system also has to populate a VM control data structure that defines the VM’s architectural states as well as the attributes of the VM. Therefore, each VM must have its own VM control block. This step is shown in Fig. 7.1 as *VM Control Block Generation*. This control block essentially defines how the VM would be run, and more importantly, its security attributes. The test generation system is aware of the security protection support each VM is assigned and assigns the appropriate guest code pointer and guest page table directory to the respective VM control block. For example, encrypted memory blocks or encrypted guest codes would only be assigned to encryption-enabled VM control blocks.

Once a control block is available for each VM, an address space identifier (ASID) must be assigned to each VM control block to allow each VM control block to become executable. The CPU may support a large number of ASIDs, but it may not be practical to exercise all ASIDs in a single design simulation test. Therefore, in such cases, the system test generator would randomly choose a subset of ASIDs and assign them randomly to corresponding VM control blocks. At this point, the VMs are ready to be executed, as shown in Fig. 7.1. It is also possible to assign one ASID to multiple VMs. In such cases, the test generator allows...
Given that the VMs are ready for execution, they may need to be executed by the hypervisor on multiple threads. The number of threads is generally significantly smaller than the maximum number of supported ASIDs. Therefore, to properly test and stress the scalability of the processor, all VM execution must be scheduled onto a thread-aware parallel hypervisor.

The test generator’s current scheduling scheme is basic: attempt to evenly distribute the VMs into each hypervisor’s execution schedule in a random manner. If additional hypervisor level test codes or test conditions need to be set for a specific set of VMs, such as host-initiated VM memory/cache management, these operations are scheduled into each thread’s hypervisor program. Additionally, as noted earlier, each guest program may have more than one VM exit point inserted, which requires the hypervisor to execute the guest’s VM multiple times.

This particular flow is indicated in the Generate VM to Thread Schedule and Execute portion of Fig. 7.1. This approach allows a particular VM to potentially be executed by more than one thread. This is achieved because VM execution is rescheduled across all threads at each iteration. This would allow the generated test set to exercise VM management design elements more completely. The generated VM guest codes as well as the hypervisor code do contain self-check conditions. These conditions are checked during test and at the end of the test. The final passing or failing conditions are aggregated across all test checks on all threads and VMs.

7.4.2. Example

To illustrate the output of the test generator, an example is shown in Fig. 7.3. In this example, a CPU with two threads supporting up to 16 ASIDs is configured to be tested with a test program deploying four encryption-enabled VMs with two iterations of testing. The generated test spawns two threads, as indicated with the green boxes. It then generates four executable VMs with randomly chosen ASIDs. In Fig. 7.3, the ASID is appended to the
label VM in each blue box. The VMs are then randomly assigned to be executed sequentially on the two respective threads for the first iteration. In Fig. 7.3, the orange boxes labeled Iteration2 indicate the beginning of the second iteration, where the four VMs (with the same ASIDs) are randomly rescheduled on the two threads. After the last two VMs have been executed, the test program completes.

Figure 7.3: Test Program Example

7.4.3. Test Generation Complexity Analysis

The proposed systematic test generation scheme can potentially address significant portions of the CPU design space of interest with a combination of automatically generated as well as user-created test codes. The effectiveness of this approach lies in the fact that, with the automation of much of the code, the system can scale to a high number of threads, and the test generation is designed to handle potentially encrypted VMs.

Theoretically, a test could target a processor supporting up to $M$ ASIDs while employing $T$ threads and $N$ VMs over $I$ test iterations. The test generation procedure can randomly select from $M^N$ VM ASID assignment combinations, if we assume that the same ASID can be assigned to an arbitrary number of VMs even if some ASIDs remain unused. On the other hand, if the ASID selection for each VM needs to be unique, the number of VM ASID assignment combinations is the binomial coefficient: $\binom{M}{N} = \frac{M!}{N!(M-N)!}$.

For implementation simplicity, the test generation system can be constrained to assign each VM exactly once to a particular thread in a uniform manner during a test iteration, and every thread will execute the same number of VMs (assuming that the number of VMs is evenly divisible by the number of threads—otherwise some threads may have one extra VM assigned to them). Therefore, in each $I$’th iteration, for a particular order of threads, up to $N!$ number of VM-to-thread schedules can be made. (We assume that assignments of
a single as yet unassigned VM can be made to each thread in a round-robin fashion until all VMs are assigned to their respective threads.)

Note that the threads are not necessarily interchangeable. For example, in a processor, some threads may share the same cache subsystem, but some may not. In such cases the CPU needs to manage encrypted VM access correctly across the cache hierarchy if an encrypted VM transitions from one cache subsystem to another cache subsystem. Therefore, the “Thread ID” to which a particular schedule of VMs is assigned may impact the design’s behavior. (Thus, the number of possible cases to consider may increase when the thread-VM assignment process leads to some threads having an extra VM because the ordering of the threads in the round-robin assignment will determine which threads have extra VMs.)

Finally, each VM can transition up to \( I \) times among available threads. Note that memory types and paging structures are subject to randomization as well during test generation.

The ability to automatically generate different test suites with large values of \( M, N, I \), etc. allows the creation of tests that fully exercise the hardware structures designed to keep track of VMs and their respective cache and memory management. This is especially important for encryption enabled VMs. Their encrypted memory hierarchy requires isolation and encrypted key control, and the additional hardware support needed must be verified for correct operation.

Allowing VMs to transition among various CPU threads also mimics actual data center VM usage as the VM itself is abstracted from the physical hardware. For an encryption enabled VM, transitioning from one thread to another thread can test whether the CPU would still operate correctly when the preceding VM has a different encryption key or is simply not encryption enabled. Therefore, VM transitioning among threads is a critical element for encryption enabled VMs.

7.4.4. Implementation Notes

The test program generator is implemented with a suite of tools and scripting. The top-level algorithm corresponding to Fig. 7.1 is implemented with scripting language and
Netwide Assembler (NASM) directives [80]. The output is a set of assembly code and NASM directives. This output is consumed with NASM to generate the final assembled binary, which is executable by the CPU simulation model under verification.

The test generation system does provide self-checking code for the guest code as well as in the hypervisor, but it is designed to be used in conjunction with design verification checks such as property assertions and runtime comparison against golden processor architectural as well as subsystem verification models to allow maximized design checking. The self-checking code can be ensuring stored encrypted memory contents on addresses match the loads from the same addresses for the same VM. Each VM on exit back to host would log exit information on their respective virtual machine control block, these can be checked against the expected exit information. For example, if a VM exits due to illegal memory access, its exit code would be different it had taken an expected VM exit point.

7.5. Experimental Results

To examine the effectiveness, scalability, and potential usage trade-offs of the proposed test generation system, a suite of tests of varying configurations were generated via the proposed system and examined further. The outcomes are compared against the manually written tests that were used prior to the development of the proposed system. The results for the experimental analysis are detailed in Table 7.1.

In Table 7.1, column 1 contains labels for each subsequent test entry. Columns 2 to 5 contain the relevant test parameter values. Specifically, Column 2 contains the number of encrypted VMs the test is targeting, Column 3 specifies the number of threads the test aims to utilize on the processor under verification, and Column 4 specifies the number of ASIDs the test generator can assign to VMs. (Note that the number of threads in column 3 does not refer to the number of threads utilized by the physical computer that simulates the verification tests, but instead it refers to the number of threads the test program simulated on the processor architectural model. This is used to model multi-threading.) Column 5 specifies the number of iterations the test will run through. Column 6 tabulates the size
Table 7.1: Test Generation Sweep

<table>
<thead>
<tr>
<th>Label</th>
<th>VMs</th>
<th>Threads</th>
<th>NASID</th>
<th>Iteration</th>
<th>Assembly Size</th>
<th>Expanded Assembly Size</th>
<th>Binary Size</th>
<th>Total Instructions Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManTest</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>26KB</td>
<td>4.3MB</td>
<td>3MB</td>
<td>752</td>
</tr>
<tr>
<td>2T16V1I</td>
<td>16</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>159KB</td>
<td>16MB</td>
<td>102MB</td>
<td>32154</td>
</tr>
<tr>
<td>2T24V1I</td>
<td>24</td>
<td>2</td>
<td>24</td>
<td>1</td>
<td>204KB</td>
<td>22MB</td>
<td>159MB</td>
<td>47894</td>
</tr>
<tr>
<td>2T32V1I</td>
<td>32</td>
<td>2</td>
<td>32</td>
<td>1</td>
<td>249KB</td>
<td>28MB</td>
<td>215MB</td>
<td>63361</td>
</tr>
<tr>
<td>2T40V1I</td>
<td>40</td>
<td>2</td>
<td>40</td>
<td>1</td>
<td>294KB</td>
<td>36MB</td>
<td>290MB</td>
<td>79233</td>
</tr>
<tr>
<td>4T16V1I</td>
<td>16</td>
<td>4</td>
<td>16</td>
<td>1</td>
<td>160KB</td>
<td>20MB</td>
<td>147MB</td>
<td>34994</td>
</tr>
<tr>
<td>4T24V1I</td>
<td>24</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>205KB</td>
<td>29MB</td>
<td>234MB</td>
<td>49769</td>
</tr>
<tr>
<td>4T32V1I</td>
<td>32</td>
<td>4</td>
<td>32</td>
<td>1</td>
<td>250KB</td>
<td>37MB</td>
<td>304MB</td>
<td>65667</td>
</tr>
<tr>
<td>4T40V1I</td>
<td>40</td>
<td>4</td>
<td>40</td>
<td>1</td>
<td>295KB</td>
<td>44MB</td>
<td>368MB</td>
<td>81537</td>
</tr>
<tr>
<td>8T16V1I</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>1</td>
<td>162KB</td>
<td>28MB</td>
<td>223MB</td>
<td>37970</td>
</tr>
<tr>
<td>8T24V1I</td>
<td>24</td>
<td>8</td>
<td>24</td>
<td>1</td>
<td>207KB</td>
<td>39MB</td>
<td>327MB</td>
<td>53476</td>
</tr>
<tr>
<td>8T32V1I</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>1</td>
<td>252KB</td>
<td>50MB</td>
<td>437MB</td>
<td>69626</td>
</tr>
<tr>
<td>8T40V1I</td>
<td>40</td>
<td>8</td>
<td>40</td>
<td>1</td>
<td>297KB</td>
<td>68MB</td>
<td>620MB</td>
<td>85978</td>
</tr>
<tr>
<td>2T16V2I</td>
<td>16</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td>177KB</td>
<td>16MB</td>
<td>98MB</td>
<td>43890</td>
</tr>
<tr>
<td>2T24V2I</td>
<td>24</td>
<td>2</td>
<td>24</td>
<td>2</td>
<td>231KB</td>
<td>22MB</td>
<td>154MB</td>
<td>64329</td>
</tr>
<tr>
<td>2T32V2I</td>
<td>32</td>
<td>2</td>
<td>32</td>
<td>2</td>
<td>285KB</td>
<td>29MB</td>
<td>213MB</td>
<td>85308</td>
</tr>
<tr>
<td>2T40V2I</td>
<td>40</td>
<td>2</td>
<td>40</td>
<td>2</td>
<td>339KB</td>
<td>35MB</td>
<td>276MB</td>
<td>107441</td>
</tr>
<tr>
<td>4T16V2I</td>
<td>16</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>178KB</td>
<td>21MB</td>
<td>149MB</td>
<td>47546</td>
</tr>
<tr>
<td>4T24V2I</td>
<td>24</td>
<td>4</td>
<td>24</td>
<td>2</td>
<td>232KB</td>
<td>28MB</td>
<td>215MB</td>
<td>69863</td>
</tr>
<tr>
<td>4T32V2I</td>
<td>32</td>
<td>4</td>
<td>32</td>
<td>2</td>
<td>286KB</td>
<td>38MB</td>
<td>308MB</td>
<td>89830</td>
</tr>
<tr>
<td>4T40V2I</td>
<td>40</td>
<td>4</td>
<td>40</td>
<td>2</td>
<td>340KB</td>
<td>49MB</td>
<td>421MB</td>
<td>115301</td>
</tr>
<tr>
<td>8T16V2I</td>
<td>16</td>
<td>8</td>
<td>16</td>
<td>2</td>
<td>180KB</td>
<td>28MB</td>
<td>215MB</td>
<td>50890</td>
</tr>
<tr>
<td>8T24V2I</td>
<td>24</td>
<td>8</td>
<td>24</td>
<td>2</td>
<td>234KB</td>
<td>40MB</td>
<td>335MB</td>
<td>73496</td>
</tr>
<tr>
<td>8T32V2I</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>2</td>
<td>288KB</td>
<td>56MB</td>
<td>501MB</td>
<td>96413</td>
</tr>
<tr>
<td>8T40V2I</td>
<td>40</td>
<td>8</td>
<td>40</td>
<td>2</td>
<td>342KB</td>
<td>66MB</td>
<td>580MB</td>
<td>119510</td>
</tr>
</tbody>
</table>
of the generated assembly language file of the given test in kilobytes. Column 7 contains
the fully expanded assembly file size post NASM processing in megabytes. Column 8 has
the final processor executable binary file size in megabytes. Column 9 describes the total
number of instructions executed by all threads the test utilized in aggregate when the tests
are simulated against the golden CPU architectural model.

The manually written test under examination consists of two encrypted VMs executed
sequentially on a single thread. This might be sufficient when the number of supported
encrypted VMs for a CPU design is small, but as the number of encrypted VMs sup-
ported increases, manual creation of assembly test code for various sets of VMs-to-thread-
configurations can be error prone, labor intensive, and lacking in sufficient randomization to
adequately explore a sufficient portion of the space of VM-to-thread scheduling.

The increase in multi-threaded and multi-VM test program complexity and scale that
can be attained with automatic test generation can be seen as ManTest in Table 7.1 is
compared against test 2T16V1I on the next row. The automatic test generation system
easily generated a test that schedules 16 VMs to be executed in a random schedule across
two CPU threads. The system automatically generated 6.1X more assembly code than
ManTest. Accounting for NASM driven automation, 11.7 megabytes of additional code was
generated. The generated binary is 34X larger than the manually created test. The system
also generates controlled hypervisor and guest test code that led to a more than 42X increase
in the number of instructions executed. In test cases such as 4T40V2I, more than a 153X
increase in total instructions can be achieved when 40 VMs were exercised by the test over
four threads in two iterations. It is clear the proposed system achieved significant gains
in test sizes and the number of accompanying scenarios that could be covered with this
automation.

Of course, the increase in instruction count that naturally arises with the generated tests
that have high values of $M$, $N$, and $I$ brings about higher design simulation time. For
example, test 2T40V1I would execute more than 2.4X the number of instructions with
an expected corresponding increase in the simulation time compared to the 2T16V1I test.
The difference is due to the increase in the number of virtual machines, while the number of threads and iterations is kept constant. Similarly, $8T40V1I$ would cost approximately $2.7X$ more instructions than $2T16V1I$. Furthermore, because this test utilizes more threads, the final design simulation time cost would be even greater as more actual simulation time would now be spent in RTL thread synchronization—which can cost significantly more simulation cycles than simulation on the CPU architectural model as effects of the testbench randomization and inter-thread communication are now fully modeled.

Therefore, intelligently selecting test generation parameters (such as thread count, encrypted/unencrypted VM support size, and the number of test iterations) to meet the critical CPU micro-architectural usage targets is needed to allow for the crafting of a more optimal and practical design verification regression suite. For example, if the target design is sensitive to VM transitions among threads, in order to hit potential corner cases, the number of test iterations can be increased while the VM count can be reduced. If the target design has critical logic managing the operation of a large number of encrypted VMs operating on a reasonably large pool of threads while being less sensitive to VM transitions among threads, then the number of test iterations can be reduced. For instance, $8T40V1I$ can be chosen instead of $8T40V2I$—thereby reducing the instruction count by 28% from $8T40V2I$.

Despite the fact that the test configurations top out at 40 VMs, 8 threads, and 2 test iterations in Table 7.1, the proposed system can continue to scale beyond these configurations. Furthermore, it allows the user to insert critical guest test code segments if necessary. These advantages allow for the crafting of custom-fit test suites that can scale to the design.

In practice, on a state-of-the-art CPU, the use of the proposed system has proven to be essential. The design under verification supports a high number of encrypted VMs with extensive design elements. The proposed system held final design correctness in check and enabled critical verification coverage metrics to be reached.

### 7.6. Conclusion and future work

A scalable systematic test generation flow has been proposed to verify a potentially
thread-dense processor design that has been enhanced for virtualization and security. Experimental results have shown that this approach can be used to successfully generate complex and randomized simulation test cases that can verify the processor as the number of VMs, threads, and VM transitions among threads is increased. The new approach automated a previously utilized manual and effort-intensive assembly test composition approach that would have been prohibitively time consuming to use to obtain the desired simulation test cases on this state-of-the-art design. The application of the automated approach described in this chapter has proven to be effective in a production processor design verification environment.

The current proposed flow could be enhanced in a number of ways. First, processor scheduling can be made dynamic as opposed to being scheduled at test generation time—which may lead to an unbalanced VM-to-thread ratio across all threads. Second, VM guest code can be improved to account for processor run time statistics, such as on-chip traffic quality. Finally, either guest or host adversarial code segments can be automatically generated and exercised during test as well to validate the effectiveness of implemented design security features.
Chapter 8

A Case Study: Pre-Silicon SoC RAS Validation for NoC Server Processor

8.1. Overview

In the previous chapter, chapter 7, verifying server virtual machine security to ensure the security of embedded memory was described. A system-on-chip (SoC) nowadays can contain multiple networks on-chip (NoC) for control and data communication purposes. Due to radiation, soft errors can occur in embedded memories and main system memory. In some cases, these radiations may be adversarial, i.e. laser attacks. Therefore, verifying correct memory soft error detection and reporting in SoCs provide additional assurance in the reliable operation of embedded memories and system memory. This chapter proposes a novel approach to verify memory soft error detection and propagation capability in SoCs constructed with more than one on-chip network in section 8.4. The work in this chapter was first published in 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV) [58].

8.2. Introduction

Intellectual Property (IP) components have been traditionally connected by on-chip buses or interconnects for system-on-chip (SoC) silicon devices. With each succeeding generation of new SoC architectures or process geometries, the complexity of these SoCs has been growing significantly. Bus architectures that used to connect a few subsystems are often sub-optimal for increasingly large SoCs. This is particularly relevant for SoCs slated for use in Data Centers. More IPs, from communication to external storage control to on-chip memory controllers, are being integrated into a single SoC die or package. Existing bus architectures may require untenable routing, timing, and area overhead. As such, network theory based
approaches are now being implemented to enable communication between IPs on SoCs [5].

A network-on-chip (NoC) based server SoC must also implement reliability, availability, and serviceability (RAS) capabilities to minimize server downtime, failures, and errors [77]. Therefore, RAS must be extended and scaled efficiently to NoCs. Furthermore, the corresponding verification methodology must also adapt to the NoC RAS architecture. This chapter proposes a novel verification methodology to enable effective pre-silicon validation of RAS capabilities for NoC Server processors.

8.2.1. Background and Prior Work

Modern processor RAS architecture and implementation are detailed in [77]. Traditionally, server processor RAS verification methodology relied extensively on various cache or design error injection techniques [111, 112]. The advent of the NoC methodology enabled the SoC to be interconnected with on-chip networks [76]. There is a significant amount of literature available in the NoC domain. For example, NoC overviews are available in [5, 76]. In general, the NoC allows for packet switched on-chip communication among IPs. These packets are formatted and partitioned into a set of fixed-length units, or flits. These flits are transmitted via on-chip routers to their destinations.

The NoC also allows the need for side band signals and buses that traditionally carry specialized information, such as point-to-point interrupts or system configuration data, to be absorbed. This information can now be packetized and transmitted across on-chip networks, which eliminates the overhead required to place and route dedicated signals. Because pre-NoC designs relied heavily on these side band signals for RAS verification, a new approach is needed to ensure that RAS relevant information can still be transported across IPs.

Because sideband signals are no longer routed across the chip, the information they carried must be packed and unpacked to and from the NoC interfaces. Fortunately, the NoC’s backbone components, such as routers and interfaces, are readily-verified IPs and are reusable. Each IP hosting the NoC interface must provide glue logic to allow packing and unpacking of on-chip communications. Hence, RAS relevant handling is no exception.
For example, core RAS interrupts must be signaled and arbitrated (potentially) through a glue logic layer to the target destination. Similarly, incoming RAS packets to the core must funnel through the NoC glue logic before reaching the appropriate core RAS handling logic.

Existing work such as [111, 112] heavily focus on using error injection, and subsequent handling schemes to achieve design verification closure without explicit NoC consideration. Benefits of NoC with respect to design verification had been briefly discussed in [41], but that paper did not specifically target RAS design features in their verification analysis.

In this chapter, a NoC aware SoC RAS verification methodology is presented. This methodology aims to achieve SoC level RAS verification closure utilizing NoC awareness. It provides a SoC level NoC RAS verification model for RAS task simplification. It outlines key tasks required to achieve SoC wide verification closure. It also demonstrates the paring down of error generation scenarios by focusing on RAS packet generation. These elements constitute the proposed validation flow.

8.2.2. Chapter Organization

First, a RAS-specific abstract model to represent key information required for RAS validation is presented. Then, a novel validation methodology based on the model is described. Finally, we detail our implementation and practical experience with our methodology on a state-of-the-art production server SoC design.

8.3. NoC SoC Model for RAS Validation

The goal of modeling an NoC for RAS is to primarily expose NoC elements of concern to RAS operations. To this end, the proposed simplified NoC model is shown below in Fig. 8.1. In this work, we associate RAS errors with soft errors, data transmission errors, and specific sets of access violations that may potentially arise in an SoC.

In Fig. 8.1, the CPU complex is expected to be the target computation resources exposed to the user operating system, and it would contain processor cores and potentially cache hierarchy such as L1, L2, and L3.
The SoC Control Complex is expected to generate reset, clocking, and SoC-wide initialization. It is also expected to interact with other IPs via on-chip networks and hence can be exposed to RAS errors generated by other IPs. In general, the SoC Control Complex can be viewed as system control processors operated by firmware.

An NoC may contain more than one on-chip network due to the needs of different networks. For example, a cache coherent on-chip network would require network packets to contain explicit commands to maintain the coherency protocol. For non-cache coherent traffic, such protocols may not be needed. In this abstract model, three NoC elements relevant to SoC-wide RAS register access and RAS error reporting can be exposed: Fixed Routing, Programmable Routing Points, and Interrupt Configuration, as shown in Fig. 8.1.

![Figure 8.1: NoC Model for RAS](image)

*Fixed Routing* refers to a hard coded hardware implemented routing configuration to enable NoC access targets, such as registers or IPs exposed to respective on-chip networks. This is relevant as we expect a portion of RAS relevant registers and control structures to be accessible via on-chip networks implemented with *Fixed Routing*. (Note that *Fixed Routing* in this case does not refer to the physical route taken by the packets through the NoC.)

Although some registers may be accessed through fixed routing, other registers may be accessed through programmable routing. In particular, the routing table from CPU complex or SoC Control Complex to SoC IPs may be re-configurable. We expect some RAS relevant registers mapping from CPU Complex or SoC Control Complex to be programmable to allow better scalability. For example, when IPs are de-featured (fused off), their respective routing table entries need no longer be programmed. Such re-configurable routing tables are shown
as Programmable Routing Points in Fig. 8.1.

With the NoC methodology, interrupts would need to be delivered via the on-chip network as well. Hence, we encapsulate interrupt type, interrupt destination, interrupt severity, and the interrupt generation scheme in the Interrupt Configuration block shown in Fig. 8.1.

Note that Fig. 8.1 is not meant to be topologically, physically, or hierarchically accurate, but rather, it’s meant to be an abstract model to allow an NoC RAS validation methodology to be developed. We also note that each IP is connected and accessed via on-chip networks. In this model, we do not expect any RAS-relevant side band signals.

8.4. Proposed Methodology

RAS features may be controlled and accessed with a set of feature-mapped registers or register-controlled structures across the NoC. Because IPs on the NoC communicate with each other via on-chip networks, a register access from an IP previously completed via on-chip ringed buses or cross-bar switches now needs to be formatted, packetized, and transmitted across a channel composed of a set of on-chip routers and IP level NoC glue logic to and from the target IP where the target register resides.

Further, IPs that are protected with RAS circuits, such as error detection, must log and report potential errors such that the system firmware or operating system can be made aware of errors. In the case of critical errors, the end user system must be interrupted to log and handle reported errors. As already stated, such interrupts in an NoC should be transmitted over the on-chip networks as well. Hence verifying processor wide RAS protection and handling logic must include validating all RAS specific packets transmission generated from all possible sources against the processor on-chip networks.

The proposed flow as illustrated in Fig. 8.2 is centered around three key elements: access validation of NoC RAS elements, error packet generation and handling, and NoC traffic generation in the presence of errors.
8.4.1. NoC RAS Access Validation

To ensure that software can properly configure RAS features and handle RAS errors on an NoC, the routing between control and processing cores to RAS feature-mapped registers must be validated with respect to the on-chip networks. As noted earlier, sometimes such routing is fixed—meaning that a register destination invoked in the transmitted packet will be uniquely mapped to particular physical or hierarchical locales. In other cases, the routing can be dynamic—meaning the registers can be rerouted. For example, one way to do so is to use pointer registers to reference NoC address mapped registers. By changing the pointer addresses, NoC register access can be rerouted. Hence, in the flow, Routing Initialization is considered the first step. Interrupt Configuration Initialization is the next step. Each type of RAS error may have unique error attributes tagged to its corresponding interrupt. For example, interrupt packet destination, severity, error type information, etc. These attributes must be programmed for each main IP under test on the NoC. Both of the steps can be implemented and validated with directed test cases. These tests need to show that each RAS feature-mapped register or interrupt configuration can be addressable via fixed or dynamic routing. Further, the tests need to ensure all valid data bits in each RAS relevant registers can be read and written by relevant processor cores.
8.4.2. Error Packet Generation

Once RAS-relevant register control access and interrupt configuration are validated, interrupt generation and propagation must also be validated over the on-chip networks. The NoC architecture therefore provides a natural boundary where errors in IP are only visible when an error packet or interrupt packet is being sent over the NoC to the system software. Note that such system software may correspond to the operating system or firmware and can operate in CPU Complex and/or in the SoC Control Complex.

Hence, to validate error reporting and propagation, we need to only be concerned with the types of RAS error carrying packets emanating from each RAS protected IP. It is not necessary to exhaustively verify SoC interaction with all possible errors inside a particular IP. IP level verification is assumed to be able to validate all RAS error detection and RAS error logging. Furthermore, RAS error reporting to the IP level NoC interface glue logic is assumed to be validated by IP level verification. However, each IP level glue logic must make assumptions with regard to the rest of the networked IPs responses when a particular RAS error packet is generated. Therefore, it may not be possible for an IP level testbench to model RAS specific IP to IP interaction across the SoC completely. Thus, the IP level validation of RAS error packet generation and reception to and from the SoC can not be considered complete.

For example, an IP with a single SRAM can have errors on any combination of its bit cells. From an NoC/SoC perspective though, single bit or double bit error marked packets may be the main concerns, as opposed to enumerating every possible error. It is expected that IP level verification provides coverage for the detection and packet generation of those errors already. Furthermore, it is not uncommon to expect IP level error logging logic to categorize RAS errors based on fault types, such as single bit error, double bit error, correctable error, etc. Each error category can be assigned a severity attribute based on operational significance. For example, an instruction cache error can be more severe than a correctable data fetch operation. It is from error categorization and corresponding attribution that the corresponding error packets can be formed. Therefore, a way to generate
types of packets of concern rather than performing comprehensive error injection testing is needed. These RAS error packets allow testing whether their corresponding target IPs made the correct assumptions on RAS packet interpretation and handling at the level of the glue-logic, NoC, and IP to IP communication. The proposed validation based on these observations is shown in Fig. 8.2.

As RAS errors occur during mission mode operation, each error may lead to corresponding network packets being generated. For each IP in the SoC, the potential set of error containing packets are identified first during the Error Packet Generation phase of the flow with most of them being interrupt packets in our experience. These packets may be generated with error injectors located within the IP itself (e.g. an error injector may directly alter the contents a bit in the cache). Alternatively, they could alter a hardware support register directly (e.g by setting error status bits in an error status register). For error or interrupt packets from IPs where either explicit or implicit hardware controlled packet generation is not available, verification code needs to be devised and inserted into the testbench to allow Interrupt Packets Generation. Legal RAS interrupts per each IP can thus be generated. The possible destinations of these interrupts are specified per system architecture, and typically, they tend to be CPU Complex or SoC Control Complex. This allows for Interrupt and Error Handler validation code at such locales to verify that RAS interrupts reached their destination, and that destination logic can interpret RAS interrupts and can redirect processing cores to handle each type of legal error packet. Typically in RAS interrupt handling operations, further NoC traffic is generated to allow processing cores or software to ensure that errors are being generated from the expected IP and that the error signatures are correct.

8.4.3. NoC Traffic Generation

With an NoC, it’s natural to envision scenarios where RAS error or interrupt packet transmission may be lost or excessively delayed due to unexpected NoC traffic congestion or poor IP level NoC integration. Therefore, we devised automatic network traffic generation drivers and enable them alongside RAS interrupt or error generation verification tests to
ensure no interrupt packets are missed by either the CPU Complex or SoC Control Complex and will arrive in a timely manner for correct RAS error handling. We expect on-chip network components to be well verified in the absence of RAS errors with respect to network quality of service (QoS). We intend for the addition of the validation of this particular scenario to ensure RAS error or interrupt packets won’t be incorrectly de-prioritized.

8.5. Experiences on NoC Server Pre-Silicon Validation

8.5.1. Target Design

The proposed flow is developed from practical experience working on a state-of-the-art server NoC design. This processor has eight cores, large amounts of caches, and a high number of IPs integrated and networked on-chip. This design is labeled as NoC1. Generally, an NoC processor contains at least one network protocol that links all the key sub-components or IPs into the respective network. The design under test (DUT) has two such mission mode active on-chip networks. One network is a processor data access network (PDAN). The other network is called an IP control network (ICN). The PDAN links CPU Complex with on-chip memory controllers as well as external input and output devices. Protocols of both networks allow interrupt packets. All elements of the flow in Section 8.4 are utilized, and subsequently the application of this flow on key RAS validation problems for this project will be described to demonstrate its applicability.

8.5.2. NoC Initialization

NoC1’s CPU Complex accesses IP level RAS control/status registers via programmable network routes. As a result, a routing table particular to all SoC IP RAS registers is initialized during a power on reset (POR) sequence. Further, we need to program a set of control switches to allow all IP-level RAS circuits to generate interrupts across their respective networks properly.
8.5.3. NoC Access Validation

*NoC Access Validation* was achieved with tests derived from Section 8.4. The key element unique to *NoC1* is that two categories of cores are architected. One set of processing cores are used by the operating system. The other set of cores are used for system services such as system initialization, system resets, and power on self test. *NoC1* allows both types of cores to be interrupted, and it supports both types of cores to have access to most of the RAS feature-mapped registers. NoC RAS access therefore needs to be validated with on-chip network transactions extending from both types of cores. Stimuli were developed to ensure all RAS feature-mapped registers can be written to and read from.

8.5.4. CPU Complex Error Packet Generation

In analyzing CPU generated interrupt packets (*CPU Complex* contains OS visible cores), we identified only a few types of outbound interrupt packets from the *CPU Complex* to the rest of the SoC. Given that at the *CPU Complex* IP level, all RAS error detection and handling had been verified with IP level verification effort and any *CPU Complex* communication occurs only on two network ports, we only need to generate these relevant packets outbound of *CPU Complex*. To this end, rather than using error injection with verification structures, we utilized privileged hardware control to enable an interrupt packet to be generated without actually causing an error on any actual caches. For example, we were able to create an SoC fatal error event by writing the interrupt packet directly with a hardware-enabled backdoor mechanism. In this manner, we achieved the desired verification coverage with only a few tests.

8.5.5. DRAM Data Poisoning

Detecting, reporting, and isolating Dynamic Random-Access Memory (DRAM) data errors is critical in modern servers. In the context of *NoC1*, we are primarily referring to a data poison error as the propagation of uncorrectable errors detected by the memory controller during a DRAM read operation on a particular cache line to the consuming *CPU*
Complex [77, 84]. On data poison consumption, the core should be interrupted and execute its error handler. Fortunately, a set of hardware based error injection techniques requiring configuration of disparate registers are available to us. Further, we focused our effort on generating only a few unique interrupt packets across the on-chip network.

8.5.6. Multi-Die Data Poisoning

Because the target design can be coherently linked to an identical chip via die stacking techniques, poisoned data can also propagate from one die’s memory channel to another die’s CPU complex [14]. This requires a testbench modeling two stacked dies. This means the entire chip must be fully instantiated twice when incorporating die stacking modeling constructs and leads to more than doubling of design simulation time. Fortunately, efforts in sections 8.5.2 and 8.5.3 allow rapid test coding of two dies’ NoC initialization and access configuration. One die is used to setup hardware-based error injection, which allows its controller to inject an error when a DRAM fetch occurs on addresses that have been chosen to contain errors. We then use the second die to trigger a DRAM fetch on the faulting address, which will allow the memory controller to forward a poisoned packet to the consuming die. This allows us to verify data poison logging and detection at the second die’s CPU complex. This is consistent with the proposed methodology and drastically minimizes the amount of verification effort at the multi-chip testbench level.

8.5.7. SoC IP Error Injection

Most of the SRAM-containing IPs on the NoC necessitate RAS protection. Similar to Sections 8.5.4 and 8.5.5, the idea is to isolate unique interrupt or error packets and generate them to send to on-chip networks. It differs from Sections 8.5.4 and 8.5.5 in the lack of hardware controls to generate interrupt or error packets from the respective IPs. Hence, a suite of error injectors were developed to inject errors such that interrupt and error packets of concern would be generated.
8.6. Results

For NoC1, testbench elements and stimuli with assembly, C/C++, and SystemVerilog were developed to implement the proposed methodology. The ability to generate randomized on-chip network traffic via a testbench was developed. In addition, scenarios where multiple legal RAS interrupts are generated in a short span of simulation time were developed to validate the ability of NoC1 to handle multiple interrupt packets arriving in a short order. With the NoC-centric validation methodology, verification closure with an aggressive design schedule was achieved.

8.6.1. Design Effort Characterization

Data in Table 8.1 are used to provide a sense of development effort and problem areas. The tests are organized into six groups. For each group, three metrics are provided. Under the Stimuli column, the percentage of verification tests developed for each group over all verification tests used to verify NoC RAS properties is shown. Similarly, under the Effort column, engineering effort is shown for each test group as an approximated percentage of a few verification engineers’ overall assigned engineering hours. Typically an engineer would be assigned a set of tasks. The time he or she spent in terms of engineering man hours on each of those tasks was approximated and categorized. The total engineering man hours were summed. The percentages of effort were then derived. Under Issues, we report specification problems, bugs, or non-trivial testbench problems each group of tests found in our effort as a percentage of all such issues identified during NoC RAS verification. Labeled rows in Table 8.1 explain the functionality and purpose of each group of tests. Note that data points in column four of Table 8.1 refer to problems identified exclusive of any problem found in IP level verification.

8.6.2. The Need to Minimize SoC level Verification Effort

SoC integration typically happens when most of the key architected IPs have reached maturity, which means SoC integration is phased at the tail end of an SoC project. Yet,
Table 8.1: Validation Effort Summary

<table>
<thead>
<tr>
<th>Test Grouping</th>
<th>Stimuli (%)</th>
<th>Effort (%)</th>
<th>Issues (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoC Initialization</td>
<td>3%</td>
<td>14%</td>
<td>16%</td>
</tr>
<tr>
<td>CPU to SoC IP Routing</td>
<td>39%</td>
<td>18%</td>
<td>8%</td>
</tr>
<tr>
<td>SoC Control to SoC IP Routing</td>
<td>11%</td>
<td>14%</td>
<td>23%</td>
</tr>
<tr>
<td>CPU Error &amp; Handling</td>
<td>8%</td>
<td>9%</td>
<td>15%</td>
</tr>
<tr>
<td>SoC Controller Error &amp; Handling</td>
<td>5%</td>
<td>6%</td>
<td>0%</td>
</tr>
<tr>
<td>SoC IP Error &amp; Handling</td>
<td>34%</td>
<td>41%</td>
<td>38%</td>
</tr>
</tbody>
</table>

RTL SoC level pre-silicon validation for a complex SoC, such as a server processor, may require more compute resources for design compilation, elaboration, and simulation than at the IP level. This is because the computational resource requirements at the SoC level are a function of the compute cost of IP size and complexity in addition to the compute cost of the SoC interconnects. The magnitude of the difference can be observed from the data in Table 8.2.

Table 8.2: Simulation Time Comparison

<table>
<thead>
<tr>
<th>Target Design</th>
<th>CPS Gain</th>
<th>Test Cycles Reduction</th>
<th>Test Time Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoC</td>
<td>1X</td>
<td>1X</td>
<td>1X</td>
</tr>
<tr>
<td>IP1</td>
<td>6.8X</td>
<td>2.2X</td>
<td>14.7X</td>
</tr>
<tr>
<td>IP2</td>
<td>4.2X</td>
<td>3.7X</td>
<td>15.7X</td>
</tr>
<tr>
<td>IP3</td>
<td>5.2X</td>
<td>1.3X</td>
<td>6.9X</td>
</tr>
</tbody>
</table>

For Table 8.2, data were collected on representative batches of tests executed on selected IPs and the final SoC design model. Data were then gathered for a batch of tests for each selected IP or SoC. Then the average data points for each test were derived for SoC or selected IPs. These average per test data are used in tabulating Table 8.2.

The first column specifies the respective designs. IP1, IP2, and IP3, which are major large IPs such as processor cores or memory controllers incorporated into the final server SoC. The second column tabulates the ratio of the number of clock cycles that can be simulated per second for an SoC level verification test vs. an RTL verification test for an individual IP.
(Note that the comparison is not between the same tests at two levels of abstraction. We are simply trying to show the cost of verifying SoC level properties with SoC level tests vs. the costs of verifying other properties within an individual IP.) The higher the CPS, the faster a particular simulation would be executed. The third column shows the reduction of IP level simulation versus SoC level simulation—it’s a ratio of average total SoC level simulation cycles versus average total IP level simulation cycles. The fourth column tabulates the average time per test reduction an IP level verification simulation would have over an SoC level simulation. For example, it’s shown that the average SoC simulation can be as much as 15.7X longer than IP2’s simulation due to the size of the SoC and the further need for SoC level initialization to get IPs into a functional state. Of course, simulation failure debug incurs even more computation resources as simulation and verification tests are reiterated.

SoC RAS validation work mandates largely full SoC level simulation. Thus, any ability to reduce the SoC level simulation time is very valuable. To this end, the proposed methodology has been extremely successful as we managed to close SoC RAS verification in roughly half a year. With tests in Table 8.1 rows two to four, it was found that on-chip network routing along specific routing points or end points may exhibit packet decoding problems. Further, when routing transactions are across two on-chip networks, the corresponding interconnect logic requires careful design consideration. With tests in Table 8.1 rows five to seven, it is observed that RAS interrupt packet handling at the CPU Complex or IP boundary can constitute problems. We have yet to find issues of on-chip network transmission error of RAS error or interrupt packets—though this largely depends on the maturity of the underlying NoC IP.

Verification IPs developed from this methodology have also proven to be easily portable to projects with significantly different SoC architectures. Verification work developed for this server part was also applied on an accelerated processing unit (APU), that utilized similar NoC components. For this particular APU, verification IP reconfiguration were minimal because the server part essentially yielded mostly reusable SoC level RAS verification components.
8.7. Conclusion and Future Work

This novel methodology and practical experience shows it’s possible to exploit the NoC system architecture to reduce and simplify SoC Server RAS validation at pre-silicon. This methodology exploits the NoC’s inherent IP isolation to steer verification effort towards RAS specification NoC routing and packet generation along with NoC traffic generation to enable efficient design space exploration to achieve verification closure. We believe future Server NoC RAS validation effort will be evolving alongside NoC design methodology as NoC design techniques mature.
Chapter 9
Scalable Memory Test Data Capturing

9.1. Introduction

From Chapter 7 and Chapter 8, it can be seen that memory errors or faults on-chip can potentially lead to functional failures or exceptions. One avenue to gain insights into memory failure mechanisms in the field is to understand how they fail in the stressful production environment. The testers in the production environment apply embedded memory tests to the production lot of the target IC device in volume over process, temperature, voltage, and frequency corners. The resulting failure statistic of memory failures can yield valuable information such as the distribution of memory fault mechanisms. To gather such data, efficient memory test failure capturing mechanisms that are capable of scaling over large production volume is needed.

Testing of on-chip memory is generally accomplished by driving test patterns to access memories to generate responses that can be checked against reference results. Sometimes, this process can be accomplished with tester-driven patterns that can be multiplexed directly to the memories from tester-driven pins. However, for embedded memories, the common practice is the integration of on-chip memory built-in-self-test (MBIST) instruments to test for memory defects. MBIST operates by testing memories running various memory test algorithms. MBIST test algorithms can indicate the memory failing mechanism. Hence, a type of test data can be constructed by associating failures for an SRAM or a group of SRAMs with their respective failing algorithms. Other useful data can consist of granular fail data indicating the SRAM’s failing wordline and bitline positions. When these data points are collected from volume device production tests across temperature and supply voltage corners, it may be possible to derive models of SRAM failures, such as the distribution of
failing SRAMs across wafer lots, process variations, supply voltage corners, and temperature corners. Furthermore, such models may also be tuned to characterize SRAM failures that may not match expected failure patterns due to process variations.

Production memory test flows consist of multiple test algorithms. Therefore, there can be multiple data points to collect. However, dedicating tester resources and time could lead to undesirable consequences such as increases in test time or reduction in test parallelization. Hence, rather than utilizing tester resources and procedures to diagnose and log memory test failures, it may be possible to implement an on-chip memory test data collection instrument to streamline memory test data collection.

In this work, an on-chip instrument that can collect sufficient failing memory test data for each die with embedded memory is proposed. This on-chip instrument is designed to occupy little area overhead. It enables high volume production test memory test data logging without any production test interruptions. It accomplishes this by collecting memory test data across all MBIST instruments on-chip. Such MBIST instruments can be third party IP or custom developed. Without this instrument, collecting memory test data with third party MBIST IPs may require a separate test flow that works with additional third party memory test diagnostic software. Similarly, in case of custom or in-house designed MBIST IPs that do not utilize the proposed instrument, additional custom software maybe needed. Furthermore, the proposed instrument incorporates additional memory test time optimization features.

9.2. Prior Work

With respect to MBIST test data collection, the current state of the art MBIST design IP and insertion tools from electronic design automation tool developers such as [107, 110] offer ways to analyze memory test failures. These methods are typically based on a stop-on-$n^{th}$-error scheme, where for one test algorithm, such as a march test, the individual MBIST engine must stop on the first error it encounters to log the respective failing address and bit position to report to the tester. It would then repeat the test algorithm while ignoring the first error and log the failing address and bit position of the second error. It would
repeat this process of logging the failing address and bit position of each error while ignoring
the previous error until a hardware error limit is reached or all errors less than the error
limit are logged. This iterative process requires tester monitoring servers running additional
analysis software. In a volume production environment, this process will add unpractical
test time overhead to production test time. Therefore, these approaches are practical only
on a few wafers or limited volume of devices with embedded memories. For manufacturing
memory test flows that execute multiple test algorithms back to back, the existing error
logging methods may not provide a straightforward way to log memory test failures due to
specific memory test algorithms.

In the literature, there are existing works that aim to advance MBIST micro-architecture
to more efficiently test and diagnose memory test failures. A good example of such an
effort can be found in [73], where memory test operations are encoded to optimize MBIST
engine area to achieve engine level memory fail diagnostics. In [37,38], bus directly accessible
memories are exploited to achieve lower MBIST engine area and power use. The work in [115]
illustrates some diagnostic features can be implemented in an MBIST engine testing multiple
types of memories on-chip.

Existing works and the state-of-the-art MBIST tools mostly aim to provide or enhance
MBIST engine level diagnosis, which may require separate test flows apart from production
test to achieve memory failure diagnosis. Thus, to the best of our knowledge, existing works
and MBIST software tools may not provide an on-chip method that can optimize memory
test and repair time for test flows incorporating multiple test algorithms across multiple
MBIST engines.

9.3. Memory Test Optimization Unit

To fulfill the data collection aspect of production embedded memory test, this work now
advances an on-chip control instrument that incorporates on-chip SRAM test optimization
as well as test data collection by exercising control over a subset of available on-chip MBIST
engines. To the best of our knowledge, There is no similar publicly available IC test instru-
ment. This instrument is named Memory Test Optimization Unit (MTOU).

This unit largely integrates tester operated memory BIST (built-in self test), BIRA (built-in repair analysis), and BISR (built-in self repair) operational sequences for all or a subset of memories on-chip. Thus, it can reduce the majority of the tester driven sequences for embedded memory test. From the tester’s perspective, the MTOU’s control of the MBIST(s) can then reduce otherwise needed MBIST test vectors/sequences. The reduction of such tester sequences can lead to reductions in overall embedded memory test vectors and test time. When implemented over large volumes at scale, this can enable considerable test time savings.

In addition to the control flow of BIST, BIRA, or BISR operations, the MTOU is designed to select a target algorithm for execution with BIST, BIRA, and BISR resources on a selected set of memories. It monitors the completion of memory test operations as well as the corresponding pass/fail status. When failures are encountered, then the MTOU can log and format the failing results into storage elements that can later be read by the tester.

Given the potential benefit of production device test data collection for embedded memories, the MTOU can serve as a framework to further develop on-chip test data collection capabilities for embedded memory testing from large scale volume production.

9.3.1. MTOU Block Overview

The MTOU consists of three small blocks, mtou_mbisd, mtou_testflow_fsm, and mtou_ijtag_if. The mtou_testflow_fsm block is the main design execution component, where FSM stands for finite state machine. The mtou_mbisd is a high level BIST test result collection unit, mbisd here stands for memory built-in self diagnostics. The mtou_ijtag_if allows the mtou_testflow_fsm and mtou_testflow blocks to interface with the test clock (TCK) domain and test access mechanism (TAM) logic, and the _if stands for interface.

A high level depiction of this instrument is shown in Fig. 9.1. The primary input control sources are from IJTAG Test data registers (TDRs). The user inputs test flow sequencing requirements (BIST, BIRA, BIRA+BISR, etc.) via the IJTAG accessible TDR. The user
then enables the MTOU test flow execution. The mtou_testflow_fsm block would then start its own internal FSM and orchestrate BIST execution sequences via the bist_if interface, where it sends BIST execution conditions and samples the BIST status signals for errors, failures, and algorithm completions. The mtou_testflow_fsm would communicate its operational status with mtou_bisd so that mtou_bisd would be able to correctly collect the error or fail status of the BIST execution flow. The mtou_bisd would then send the collected and aggregated BIST flow results via the JTAG interface to the tester. The tester can then store the results in tester memory which can later be uploaded into test data repositories.

![Figure 9.1: MTOU Organization](image)

### 9.4. MTOU Microarchitecture

The MTOU consists of three internal sub-blocks, and it also will need a command TDR and result collecting TDR accessible via IJTAG. The mtou_flow_fsm, mtou_mbisd, and mtou_ijtag_if sub-block level specifications are detailed in this section.
9.4.1. MTOU_IJTAG_IF Sub-Block

This is a largely clock domain synchronization block where low speed TCK command and control signals from the JTAG/IJTAG controlled TDR can be synchronized to at-speed BIST clock signals and vice versa. This block does generate a rising edge detection on the TAP FSM Capture DR transition to generate the signal that transits the mtou_mbisd's FSM to the st_dr_cap_deq state.

A list of input/output signals on this interface are tabulated in Table 9.1. The first column provides the signal name. The second column shows the input/output direction. The third column shows the signal/bus width. The fourth column describes the functionality of the signal labeled in the same row. The io_cmd_in_* list of signals are signals originating in the IJTAG controlled command TDR. The io_cmd_resp_* list of signals are the MTOU’s responses to the commands sent to the MTOU via the command TDR, and these signals are captured by the IJTAG accessible result TDR. The io_mtoutap_if_* signals are IJTAG control signals controlling IJTAG data capturing signaling for the MTOU. The io_mbsld_out_* list of signals sends the MTOU-collected test result data to the IJTAG result TDR.

9.4.2. MTOU_Testflow_fsm Description

The mtou_testflow_fsm is the heart of the MTOU. It interprets commands sent in from the tester via the ITJAG network. It then controls MBIST engine(s) on-chip to test on-chip memories following a test flow configured from commands. It then collects a handshake to the mtou_mbisd to allow test results to be sampled and stored.

The mtou_testflow_fsm executes on commands and test flow configurations from the command TDR. It gets synchronized versions of io_cmd_in_if_* signals from the mtou_ijtag_if, as shown in Table 9.1. The command signals listed in Table 9.1 are checked by the mtou_testflow_fsm to launch MBIST test flows.

The mtou_testflow_fsm’s FSM state diagram is shown below in Fig. 9.2. On a power on reset reset (POR) or a test logic reset event, the FSM would stay in the st_reset state. When
Table 9.1: MTOU Interface Pin Table

<table>
<thead>
<tr>
<th>Signal Names</th>
<th>Direction</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>in</td>
<td>1</td>
<td>At-Speed Memory Test Clock</td>
</tr>
<tr>
<td>reset</td>
<td>in</td>
<td>1</td>
<td>Async Reset from the TAP</td>
</tr>
<tr>
<td>io_algo_mask</td>
<td>in</td>
<td>15</td>
<td>Indicates Algorithm selection</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_go_no_go</td>
<td>in</td>
<td>1</td>
<td>Starts test flow when set to 1</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_bist_sel</td>
<td>in</td>
<td>1</td>
<td>Selects BIST test flow</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_bira_sel</td>
<td>in</td>
<td>1</td>
<td>Selects BIRA test flow</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_FAIL_stop</td>
<td>in</td>
<td>1</td>
<td>When high, stop on 1st fail</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_enable</td>
<td>in</td>
<td>1</td>
<td>Allows MTOU_Testflow_fsm to operate</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_start_bist_id</td>
<td>in</td>
<td>4</td>
<td>Selects which MBIST to run first</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_end_bist_id</td>
<td>in</td>
<td>4</td>
<td>Selects which MBIST to stop at</td>
</tr>
<tr>
<td>io_cmd_in_if_inst_err_limit</td>
<td>in</td>
<td>4</td>
<td>Error limit threshold</td>
</tr>
<tr>
<td>io_mto_tap_if_inst_tck</td>
<td>in</td>
<td>1</td>
<td>Test clock</td>
</tr>
<tr>
<td>io_mto_tap_if_inst_dr_cap</td>
<td>in</td>
<td>1</td>
<td>Tap Data Capture signal</td>
</tr>
<tr>
<td>io_mto_tap_if_inst_sel_cap</td>
<td>in</td>
<td>1</td>
<td>Tap Capture Select</td>
</tr>
<tr>
<td>io_mto_tap_if_inst_deq_en</td>
<td>in</td>
<td>1</td>
<td>Dequeue Enable</td>
</tr>
<tr>
<td>io_cmd_resp_if_inst_go_no_go_ack</td>
<td>out</td>
<td>1</td>
<td>Go No Go Ack Signal</td>
</tr>
<tr>
<td>io_cmd_resp_if_inst_command_valid</td>
<td>out</td>
<td>1</td>
<td>Command Validity Check Output</td>
</tr>
<tr>
<td>io_nbisd_out_inst_syndrome_out</td>
<td>out</td>
<td>32</td>
<td>Collected Data to IJTAG Result TDR</td>
</tr>
<tr>
<td>io_nbisd_out_inst_bist_fail</td>
<td>out</td>
<td>1</td>
<td>BIST Fail to IJTAG Result TDR</td>
</tr>
<tr>
<td>io_nbisd_out_inst_bist_done</td>
<td>out</td>
<td>1</td>
<td>BIST Done Signal to IJTAG Result TDR</td>
</tr>
<tr>
<td>io_nbisd_out_inst_deq_valid</td>
<td>out</td>
<td>1</td>
<td>Dequeue Valid Flag to IJTAG Result TDR</td>
</tr>
<tr>
<td>io_nbisd_out_inst_bist_err_cum</td>
<td>out</td>
<td>1</td>
<td>Error During BIRA to IJTAG Result TDR</td>
</tr>
</tbody>
</table>
the FSM is in the st_reset state, if the io_cmd_in_if_inst_enable signal is not asserted, the FSM would stay in the st_reset state, which is the default reset state for the mtou_testflow_fsm block, in which none of the BIST engine control/execution signals can be active. When the io_cmd_in_if_inst_enable signal is asserted, the FSM transitions to the st_idle state. If the io_cmd_in_if_inst_go_no_go signal is not asserted during the st_idle state, although not shown, the FSM would remain in the st_idle state. When the io_cmd_in_if_inst_go_no_go signal asserts, the FSM transitions to the st_chk_cmd state. In this state, the illegal combinations of the signals, io_cmd_in_if_inst_bist_sel, io_cmd_in_if_inst_bira_sel, and io_cmd_in_if_inst_bisr_en,
are checked. More specifically, the `io_cmd_in_if_inst_bist_sel` signal and the `io_cmd_in_if_inst_bisr_en` signal both asserting are not compatible operations and the `io_cmd_in_if_inst_bist_sel` and the `io_cmd_in_if_inst_bira_sel` signals both asserting would not lead to a supported test flow. When the `st_chk_cmd` state validates a command combination, the FSM transitions to the `st_pdom_start` state. Note, the four-wire input signal bundle `io_cmd_in_if_inst_go_no_go`, `io_cmd_in_if_inst_bist_sel`, `io_cmd_in_if_inst_bira_sel`, and `io_cmd_in_if_inst_bisr_sel` to the MTOU is considered to be the MTOU command packet.

The FSM state `st_pdom_start` is the starting state for the BIST test flow execution over all controlled MBIST engines. Thus, each `pdom` iteration corresponds to the execution of exactly 1 MBIST engine. The “p” in `pdom` stands for power and the “dom” stands for domain. The MTOU design prefers that each MBIST engine is partitioned based on the power domain, hence the annotation. However, this preference is not a necessity. The state `st_pdom_start` kicks off 2 nested loops where the outer loop iterates over the MBIST engines and the inner loop iterates and executes all assigned MBIST algorithms. At the `st_pdom_start` state, if the `io_cmd_in_if_inst_go_no_go` signal is still set, the FSM transits to the `st_pdom_run` state. At the `st_pdom_run` state, the outer loop control flow is being regulated. If all the MBIST engines have executed, then the FSM transits to the `st_idle_ack` state, where it waits for the de-assertion of the `io_cmd_in_if_inst_go_no_go` signal to transition back to the `st_idle` state. This is to prevent unnecessary repeat MBIST executions; it’s a loop breaking safety mechanism needed in case the `io_cmd_in_if_inst_go_no_go` signal doesn’t de-assert—leading to an infinite loop. From the `st_pdom_run` state, if there is any MBIST execution to be performed, then the FSM will transit to the `st_start_algo` state, where the FSM’s algorithm gets initialized before transitioning to the `st_run_algo` state. At the `st_run_algo` state, the `io_algo_mask` vector is examined, where each logically high bit in the mask flag an MBIST engine hardcoded algorithm to be executed. Each logically low bit in the mask indicates an algorithm to skip. If an algorithm needs to be skipped, the FSM transits to the `st_stop_algo` state to advance the algorithm counter. Otherwise, the `st_start_bist` state is the next hop for the FSM. At the `st_start_bist`, the FSM asserts the MBIST control
signalling that would cause the MBIST engine to execute BIST, BIRA, or BISR operations—more notably, the signals io_bist_if_mbist_bist, io_bist_if_mbist_bira, io_bist_if_mbist_bisr, and io_bist_if_mbist_smart would be asserted or pulsed accordingly. For any given MBIST operation, its corresponding io_bist_if_mbist_bi* signal must be held high during the duration of the operation. However, the io_bist_if_mbist_smart signal has to be pulsed with a pulse width exceeding a TCK period. The FSM coarsely accomplishes this pulse with a counter-based mechanism. The MBIST operation is then kicked off, and the FSM moves to the st_run_bist state, where it would wait for the MBIST operation to complete. If it encounters MBIST error, which is possible during BIRA operations, the FSM would transition to the st_check_error state, where a hand shake is sent to the mtou_mbisd block to check and log the MBIST engine operational results.

If the io_cmd_in_if_inst_bisr_en signal is set on the command packet for BIRA operations, the FSM would proceed to execute the BISR operation with the BIST engine. The purpose of this is to dynamically perform engine level soft repair, where, as with the tester driven operations, this operation would be indiscriminately applied to all the parts under test rather than on a need-only basis. Furthermore, this feature allows the MTOU test flow to request post repair BIST test to verify the correctness of repair solution generated in the BIRA test. This means separate repair verification testing does not need to be applied later on. If the FSM encounters an MBIST failure or that MBIST still fails after BISR operation, it would then move to the st_check_fail state. In this state, the mtou_mbisd would be alerted to log BIST failure information. After which, the FSM would move to the st_end_bist state, which indicates the current MBIST test based on the selected algorithm is passing. The FSM can now transit back to the st_run_algo state to start the execution of the next selected algorithm. If all the algorithms were executed for an MBIST engine without a failure, then the FSM will transit back to the st_pdom_run state to start executing the next MBIST engine. If all the MBIST engines had been executed without any failure, the FSM will transit to the st_idle_ack state. However, if the io_cmd_in_if_inst_fail_stop signal asserts during the st_run_bist state, the test flow execution stops immediately.
While not explicitly shown in Fig. 9.2., the FSM would log the number of errors the test flow encountered at st_check_error state. If the number of errors is greater than the threshold set at the io_cmd_in_if_inst_err_limit vector, than the FSM would flag fail and return a failing result at the end of the test flow execution.

With the use of the mtou_testflow_fsm, a number of optimizations and improvements may now be accomplished. First, manual aggregation of algorithmic level tester vector sets for MBIST test flows may now be eliminated. Secondly, previously required explicit BISR operations may now be eliminated; furthermore, unnecessary BISR operations for parts that do not need repair are now eliminated. Finally, during the BIRA test flow, when multiple memory errors were counted during BIRA flows, parts can be rejected. This allows for the detection of unrepairable parts despite successful BIRA execution for an algorithm.

9.4.3. MTOU_MBISD_fsm Description

The mtou_mbisd interfaces with the io_cmd_in_if_inst_* signals from the command TDR, the io_mtou_tap_ifInst_* signals from the TAP, and the mtou_testflow_fsm. The io_* signals can be directly referenced from Table 9.1.

The mtou_mbisd’s operates a seven-state FSM, and its state diagram is shown in Fig. 9.3. On POR or a test logic reset event, the FSM stays in the st_reset state. Asserting the io_cmd_in_if_inst_go_no_go signal via the IJTAG command TDR would transition the FSM into the st_idle state. The FSM then waits for the mtou_testflow_fsm to indicate whether the BIST Engine is in execution or not. If not, it will continue to stay in st_idle mode. If mtou_testflow_fsm indicates that the MBIST Engine is executing a particular test algorithm, then it proceeds to the st_cap_bistmode state where it latches the BIST’s mode of operation. From there, it proceeds to the st_wait_2_sample state, where it waits for BIST execution completion—which occurs when 1 BIST algorithm completes or when the mtou_testflow_fsm explicitly informs it to collect BIST execution status. The FSM then moves to the st_sample_synd state to latch and accumulate the BIST execution status into its own local status registers. In the event where the MBIST experiences failure or error,
it will form a syndrome packet that consists of the BIST engine id, the algorithm which the MBIST failed on, the BIST mode of operation, the error or fail status, and a packet valid status bit into the mtou_mbisd’s internal syndrome queue, which defaults to 7 packets deep queue. After this, the FSM will move to the st_sample_end state where it will push the syndrome into its own internal queue. It will then transition to the st_idle state to wait for the next MBIST algorithm or operation to execute. From the st_idle state, when the IJTAG interface is ready to sample and capture information from the syndrome queue, the io_mtoutap_if_inst_dr_cap signal would assert. It will transition the FSM into the
9.5. Potential MTOU Production Usage Flow

When a MTOU is integrated in a chip, a potential embedded memory test flow can be seen in Fig. 9.4. The tester can send an MTOU test program via JTAG commands. The TDRs that control the MTOU can execute an effective embedded memory test and repair program involving multiple test algorithms. However, a subset of the test program can be executed by the MTOU as well by setting the bits in the io_algo_mask bus because each bit not set in the io_algo_mask bus from IJTAG will cause the corresponding test algorithm to be skipped. For example, only a few test algorithms can be selected instead of the full suite. The MTOU controlled test program can be executed in multiple steps: this is accomplished...
by selecting only a subset of the MBIST engines controlled by the MTOU in one complete execution step of the MTOU followed by another test flow execution with a different set of MBIST engines selected. If the test program does not enable repair, then the MTOU will bypass the repair portion of the flow. However, if repair needs to be implemented as part of the flow, then the MTOU would execute the BIRA test suite. When memory errors are reported as shown in Fig. 9.4, a request for repair distribution will happen. This would be followed up by a BIST test to ensure repair worked as intended. With the MTOU, the test engineer can decide to test and repair all embedded memories on-chip all at once or just portions at a time—which can be helpful when debugging is needed.

The results collected by the MTOU can be used for debugging purposes. When the part is in volume production, the tester only needs to log the tester results by popping syndromes in the syndrome queue. This means only tester logs are needed to collect well-structured MBIST test results; hence, expensive dedicated tester diagnostic sessions can be entirely avoided.

9.6. Limitations

While the MTOU can optimize memory test execution and collect memory test data results, it should be noted that there are memory testing schemes to which the MTOU does not apply. For memory testing requiring on-chip memory sensors, which can be found in [75], the MTOU is not applicable since the MTOU assumes memory testing is accomplished via march test based memory test algorithms. For nonvolatile memories (NVMs), MTOU applicability may be limited, as these memories may require specialized test circuitry or specialized test algorithms [119] that are distinct from bulk SRAM test algorithms. This would render the optimization aspect of the MTOU moot for NVMs.

In terms of test time optimization, the current MTOU design only supports one MBIST engine execution rather than multiple MBIST engines executing concurrently. However, it’s not difficult to extend the MTOU to support multiple MBIST engines executing test algorithms simultaneously.
9.7. Industrial Case Study

In order to test out the concept of the MTOU, the first step is to model the design with a hardware design language such as Verilog or SystemVerilog at the register transfer level (RTL). Once the RTL model is complete, the model needs to be verified. On verification completion, the RTL model can be transformed into actual standard cells; this is accomplished with a synthesis software tool. At this stage, the area and power of the block can then be estimated.

Fortunately, an industrial microcontroller allowed for the integration of the MTOU on an experimental basis. This microcontroller, referenced as Design A, targets a 40nm production process. Design A contains a substantial number of embedded SRAM instances designed for low power operations. This design configuration has ramifications for production test. Chips configuring SRAMs to operate at more than one power state would often adjust the voltage and frequencies of SRAMs in order to transition to another power state. Hence, the SRAM manufacturing test flow could involve testing at multiple device process variation over temperature (PVT) corners. Therefore, memory test flow optimization and test data collection over the respective PVT corners need to be efficient.

9.7.1. Implementation

Given that every chip’s memory configuration may be different, the MTOU’s RTL should be able to scale to, be reused on, or be adjusted to chips with various memory configurations. Here, the memory configuration refers to the types of memories used, the number of memories used, the clock domain assignment of various memories, etc. For instance, an MBIST engine should only test memories in one particular clock domain instead of being used to test memories in multiple clock domains. Hence, for a chip design containing memories in multiple clock domains, the number of MBIST engines may increase versus designs with memories assigned to fewer clock domains. So the MTOU design should be able to scale to such design configuration variations. Furthermore, if more features, such as new data collection formatting is needed, the MTOU should be allowed to accommodate such changes.
As a result of the aforementioned considerations, the CHISEL design framework is lever-aged to implement the RTL [19]. CHISEL is underpinned by the scala programming language. The hardware blocks designed with CHISEL are represented in fully parameterizable scala objects, or classes. Therefore, the MTOU code constructed with CHISEL cannot just be parameterized, but it can also serve as a base class to be extended by successive derivatives. Composing digital designs in CHISEL also occurs at a higher level of abstraction than Verilog or SystemVerilog since many design components such as interfaces, queues, and various encoders are packaged into the design environment, and hence, the coding can also be more compact. Once the MTOU is fully modeled with CHISEL, the CHISEL environment consumes the CHISEL model and can generate a corresponding synthesizable Verilog model of the design. Hence, developing the MTOU in CHISEL corresponds to effectively developing a Verilog design generator. The CHISEL generated MTOU Verilog model is then used for chip level integration.

At the chip level, the command TDR and the result TDR are generated with an EDA tool such as [110]. The EDA tool can be scripted to instantiate the MTOU and connect it to the command TDR, result TDR, and the MBIST engine(s) that the MTOU is expected to control. The MBIST engine/controller is 3PIP (3rd party intellectual property) generated from from the EDA based on design configurations. For the MBIST the MTOU controls, it is generated with ports allowing algorithms and test execution operations to be controllable at the MBIST engine ports. These ports are then connected to the MBIST control interfaces on the MTOU. The EDA tool based insertion of the MTOU also allows for the control and result collection of the MTOU to be mapped to the Chip’s IJTAG’s ICL (Instrument Connectivity Language) model.

9.7.2. Design Simulation

Once the MTOU is integrated on the chip, it becomes necessary to check the proper functionality of the MTOU. To this end, IJTAG based MTOU test vectors were developed and simulated against the full chip. Also, prior to full chip integration, basic verification
stimuli were applied against the MTOU at the block level as well.

To this end, the BIST test flow stimuli, the BIRA test flow stimuli, and the BIRA test flow + BISR enabled stimuli were applied to the chip design model. For Design A, it contains a single MBIST engine instance. Design A contains 30+ on-chip SRAMs. So in a typical MTOU/MBIST stimuli, the MTOU is configured to control the MBIST block to execute all or a subset of all the available memory test algorithms for some or all of the SRAMs. MTOU based stimuli can also be incorporated into chip level repair distribution stimuli, where repair analysis results from the MTOU-driven repair flows can be applied for chip level repair distribution. Typically, at the end of an MTOU based test stimuli, the queued results are sampled to check for expected behavior.

To finalize MTOU verification, functional verification test suites were utilized to verify that even with the MTOU integrated in the SoC, no functional operations were affected; in other words, chip functionality was not altered with the presence of MTOU. Hence, MTOU functionality and chip functionality were fully verified with the MTOU integrated.

9.7.3. Area Impact

Given that the MTOU is implemented on a production design, Design A, realistic physical implementation data can be gathered. This means the area utilization data presented here is based on a post layout, post routed, post power routed, and post scan-inserted design netlist. Table 9.2 tabulates the area impact of the MTOU and provides a few more useful data points for reference.

Table 9.2 contains 4 columns. Column 1 labels the design unit. Column 2 shows the area used by the corresponding design unit in roughly the number of NAND gates. Column 3 reports the area utilization as a percentage of the Full_chip area of Design A. Column 4 provides some useful description of the matching design unit. The design units with * are under the hierarchy of the design unit entry immediately above it without * . The first entry in the table shows the full_chip design unit, which represents the entire chip. From the eighth entry, the design units tabulated are primarily used for reference.
The design area data shown in Table 9.2 demonstrate that the MTOU occupies significantly less than 0.1% of the total chip area, noting that design units in entries 3, 4, and 5 are counted inside of mtou_top. Even comparing against other test circuitry such as the MBIST engine (mbist_central), MBIST repair, TAP (test access port) interface (mtou_server_top), and scan test data compression and decompression circuitry (high_compression_test), the MTOU’s area utilization is small. However, when compared against the test access port (TAP) block (jtag_tap), the MTOU does appear to be greater than the TAP’s area utilization.

The MTOU’s area use can be further broken down for more granularity. Inside of the MTOU, the error logging unit (mtou_mbisdl) occupies most of the area due to the register it uses. The mtou_ijtag_if contains multiple staged data synchronizers to synchronize data transfer between the JTAG TAP and the MTOU. The main MTOU FSM actually utilizes the least amount of area.

In terms of timing, the BIST clock domain timing is closed at 150MHz and the test clock domain logic is closed at 20MHz. No problems of note showed up during static timing analysis.

Table 9.2: MTOU Chip Wide Area Impact

<table>
<thead>
<tr>
<th>Design Unit</th>
<th>Unit Area (1 nand gate)</th>
<th>Area Utilization(%)</th>
<th>Unit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full_Chip</td>
<td>7341538</td>
<td>100%</td>
<td>The area of the entire chip</td>
</tr>
<tr>
<td>mtou_top</td>
<td>1465</td>
<td>0.020%</td>
<td>The top level mtou block</td>
</tr>
<tr>
<td>*mtou_ijtag_if</td>
<td>469</td>
<td>0.006%</td>
<td>Test clk to bist clk interface</td>
</tr>
<tr>
<td>*mtou_testflow_fsm</td>
<td>403</td>
<td>0.005%</td>
<td>mtou main finite state machine</td>
</tr>
<tr>
<td>*mtou_mbisdl</td>
<td>593</td>
<td>0.008%</td>
<td>The bisd unit</td>
</tr>
<tr>
<td>mtou_cmd_tdr</td>
<td>395</td>
<td>0.005%</td>
<td>The command TDR</td>
</tr>
<tr>
<td>mtou_status_tdr</td>
<td>232</td>
<td>0.003%</td>
<td>The status TDR</td>
</tr>
<tr>
<td>mbist_central</td>
<td>8399</td>
<td>0.114%</td>
<td>The main MBIST engine</td>
</tr>
<tr>
<td>mbist_server_top</td>
<td>16590</td>
<td>0.226%</td>
<td>Repair and Tap interface for MBIST</td>
</tr>
<tr>
<td>jtag_tap</td>
<td>407</td>
<td>0.006%</td>
<td>Main test access port</td>
</tr>
<tr>
<td>high_compression_test</td>
<td>8475</td>
<td>0.115%</td>
<td>scan test compression/decompression logic</td>
</tr>
</tbody>
</table>
9.7.4. Memory Test Flow Tester Implications

The goal of the MTOU is to enable low effort logging of memory test diagnostic results in a memory test flow for every single die or packaged device. Secondarily, the MTOU can also provide at least some test optimization. Thus for Design A, the benefits of the MTOU can be illustrated in Fig. 9.5. The side by side listings from Fig. 9.5 are derived from the production test programs for Design A.

On the left of Fig. 9.5, the memory test flow for memory test utilizing the main MBIST for Design A is listed. On the right of Fig. 9.5, the memory test flow for memory test controlled and executed by the MTOU is shown. Each Macro is a subsection of the entire test flow. So each Macro is composed of a series of IJTAG transactions that can provide MBIST test instructions or chip level system resource settings that enable MBIST testing. For instance, BIRA_ALGO_1 instructs the MBIST engine to test all the memories with algorithm 1 via a set of IJTAG transactions and then check the MBIST execution completion and pass/fail status. The Test_pll_Configure Macro programs the on-chip PLL (phase-lock loop) circuit to clock the memories and the MBIST engine at the correct test target frequency. At the beginning of the test flow, the PLL should be programmed, and at the end of the test flow, the PLL should be turned off—seen as the pll_auto_off Macro. The MRR2INBR Macro just loads the repair data from the memory repair registers into the MBIST repair analysis registers for initialization. Hence, to accumulate the repair data correctly, MRR2INBR has to be executed prior to MBIST testing. The MTOU_BIRA_BISR_FLOW Macro is the MTOU programming and results sampling sequence composed of IJTAG transactions as well.

9.7.4.1. Test Time Analysis

The test time use of the MBIST test flow with or without using the MTOU can be further analyzed. Referring to Fig. 9.5, each Macro’s tester use can be broken down into 3 sections: command transactions, wait/idle transactions, and result checking transactions. For a Macro $M$, let $CT_M$ represent the tester cycles spent on the command transactions, let $WT_M$ represent the tester cycles spent during the wait/idle transactions, and let $RCT_M$ be
the number of tester cycles used to perform result checking transactions. The tester cycles consumed by the Macro BISR can be labeled by $BISR_T$. $CT_M$ and $RCT_M$ may consume a few hundred tester cycles depending on the size of the IJTAG network. $WT_M$ on the other hand depends on the computational complexity of the selected MBIST test algorithm. For instance, a $6N$ test algorithm needs $WT_M = (6 \times Total\Num\Addr \times F_{\text{period}})/TCK_{\text{period}}$ cycles to execute, where $Total\Num\Addr$ is the total number of memory addresses the MBIST engine can access serially. $BISR_T$ is associated with the amount of time to transfer repair solutions stored in the MBIST engine into the repair registers located by the memories across the chip. Assuming this transfer process relies on serial shift access to the repair registers, then $BISR_T$ depends on the number of repairable memories on-chip and the length of the repair chain. The FLOW_ALL_MBIST pattern executes a repair priority test flow. This means the test flow does not execute BIST testing first, but rather, the test flow starts by running BIRA testing first — BIRA test executes BIST test with repair analysis function enabled. The reasoning here is test time reduction — accumulate repair first and then verify repair with BIST tests instead of BIST, BIRA, BISR, and BIST sequence. Hence, for the pattern FLOW_ALL_MBIST, each BIRA_ALGO_* Macro consumes the same number of tester cycles as a BIST_ALGO_* Macro. The pattern FLOW_ALL_MBIST then performs another MBIST suite after the BISR Macro to verify memory repair - given the pattern itself doesn’t execute conditionally based on the MBIST test results.

As shown on Fig. 9.5, pattern FLOW_ALL_MTOU calls the MTOU_BIRA_BISR_FLOW Macro, which then accesses the MTOU to directly control MBIST engine and on-chip repair logic to implement the functionalities of the Macros called by the FLOW_ALL_MBIST pattern. Therefore, FLOW_ALL_MTOU only has to access the MTOU block via the IJTAG network to kickoff MBIST test flow. Hence, $CT_{\text{MTOU}}$ approximates to $CT_M$ and $RCT_{\text{MTOU}}$ is roughly $RCT_M$. Since MTOU eliminates MBIST control sequencing with fast on-chip handshakes rather than IJTAG transactions, only one command transaction and one result checking transaction are needed. However, since the MTOU does not alter MBIST test execution time, it must wait for each memory test algorithm to end before starting another
memory algorithm. Note, the algorithms labeled in Fig. 9.5 do map to production memory test algorithms.

With the preceding analysis, the tester cost cycle-wise of an MBIST test flow without the MTOU can be expressed as follows:

\[ \text{NO\_MTOU\_CYCLES} = 2 \times \left( \sum_{i=1}^{N\text{ALGO}} (CT_i + WT_i + RCT_i) \right) + BISR_T. \]

The tester cost cycle-wise of an MBIST test flow with the MTOU can be expressed below:

\[ \text{MTOU\_CYCLES} = CT_{\text{MTOU}} + RCT_{\text{MTOU}} + \sum_{i=1}^{N\text{ALGO}} (WT_i) + (NR \times BISR_T) + ABT. \]

Please note that \( N\text{ALGO} \) refers to the total number of algorithms, \( NR \) in the second equation refers to the number of repairs, and \( ABT \) in the second equation refers to the additional MBIST test time needed when repairs are identified.

For a particular device, if no repair is needed, then a test flow using the MTOU can lead to a test time reduction close to a factor of 2 compared against a non-MTOU based flow; this is in part due to the fact that test algorithm wait time typically dominates command transactions and result checking transactions. Furthermore, the MTOU circuit can dynamically invoke repair or BISR operation during the test flow to conditionally execute BISR and MBIST once repair is found to be necessary. So when no repair is needed, \( NR \) and \( ABT \) drop to 0. However, when a device needs a high number of repairs, then the \( ABT \) and \( NR \) in the MTOU based test flow will increase. In practice, it’s unusual to see an IC with embedded memory needing more than 5 repairs. Therefore, in the worst case, \( NR = 5 \) and \( ABT = 5 \times \text{AvgAlgoCycles} \), which may still yield a \( \text{MTOU\_CYCLES} \) significantly less than \( \text{NO\_MTOU\_CYCLES} \), where \( \text{AvgAlgoCycles} \) represents the average MBIST algorithm test cycles. Hence, the MTOU circuit is trying to verify repair success on demand rather than do another repair verifying pass of MBIST testing. Therefore, the MTOU-based test flow may lead to test time reduction when compared against test flows without the MTOU.
9.7.4.2. Observations about MTOU

The lists in Fig. 9.5 may at least lead to two conclusions. First, in terms of memory test diagnostic results, the MTOU_BIRA_BISR_FLOW macro will generate the diagnostic data that can be directly logged by the tester, which means little additional effort is needed to diagnose the failure modes of memories in the event of test failures. Secondly, the number of IJTAG transactions sent through the JTAG TAP and the necessary test engineering development in the form of Macros may be reduced.

9.8. Summary

The MTOU on-chip instrument for embedded memory test diagnostic data and results collection is proposed in this chapter. The proposed method is scalable thanks in part to a generator styled implementation with CHISEL [19]. The generated MTOU instance can then be adapted to and integrated into a particular design. Here, the MTOU was instantiated in an actual industrial design. Silicon implementation results demonstrate that the MTOU leads to little design area overhead for Design A. The use of the MTOU can help collect useful diagnostic embedded memory test results for every die of Design A during volume manufacturing. The ease in data collection is further complemented by a potential reduction in test time. Future work can build on the MTOU’s concepts to further enhance the diagnostic data logging capability.
Pattern FLOW_ALL MBIST begin:
- Macro Test_pll_Configure;
- Macro MRR2INBR;
- Macro BIRA_ALGO_1 main_bist;
- Macro BIRA_ALGO_2 main_bist;
- Macro BIRA_ALGO_3 main_bist;
- Macro BIRA_ALGO_4 main_bist;
- Macro BIRA_ALGO_5 main_bist;
- Macro BIRA_ALGO_6 main_bist;
- Macro BIRA_ALGO_7 main_bist;
- Macro BIRA_ALGO_8 main_bist;
- Macro BIRA_ALGO_9 main_bist;
- Macro BIRA_ALGO_10 main_bist;
- Macro BIRA_ALGO_11 main_bist;
- Macro BIRA_ALGO_12 main_bist;
- Macro BIRA_ALGO_13 main_bist;
- Macro BIRA_ALGO_14 main_bist;
- Macro BIRA_ALGO_15 main_bist;
- Macro BISR;
- Macro BIST_ALGO_1 main_bist;
- Macro BIST_ALGO_2 main_bist;
- Macro BIST_ALGO_3 main_bist;
- Macro BIST_ALGO_4 main_bist;
- Macro BIST_ALGO_5 main_bist;
- Macro BIST_ALGO_6 main_bist;
- Macro BIST_ALGO_7 main_bist;
- Macro BIST_ALGO_8 main_bist;
- Macro BIST_ALGO_9 main_bist;
- Macro BIST_ALGO_10 main_bist;
- Macro BIST_ALGO_11 main_bist;
- Macro BIST_ALGO_12 main_bist;
- Macro BIST_ALGO_13 main_bist;
- Macro BIST_ALGO_14 main_bist;
- Macro BIST_ALGO_15 main_bist;
- Macro pll auto off;
end

Pattern FLOW_ALL MTOU begin:
- Macro Test_pll_Configure;
- Macro MRR2INBR;
- Macro MTOU_BIRA_BISR_FLOW;
- Macro pll auto off;
end

Figure 9.5: Design A: MBIST Based Test Flow Vs MTOU Based Test Flow
10.1. Summary

In this dissertation, a number of IC verification and and design methods were proposed to address the concerns found in sections 1.1.1, 1.1.2, 1.1.3, and 1.1.4 on IC security and reliability. Novel ways to compromise or gain malicious access to modern ICs are being invented constantly. The similar can be said of attempts to contain and counter such threats. The set of solutions found in this dissertation are meant to be tools that IC design engineers can leverage as building blocks in a IC design flow to improve the security and reliability of the final IC product. Therefore, the effectiveness of the proposed approaches can be discussed further.

In order to begin the aforementioned discussion, a brief summary of the work in this dissertation is necessary. In Chapter 2, a technique called X-Propagation was shown to be effective at detecting SRAM IP level HTHs at the RTL—which can be effective if an accurate RTL SRAM model is extracted and HTH are not inserted in analog components such as sense amplifiers. In Chapter 3, it was shown that embedded SRAM IPs can be protected with CRC encoding of memory logical words along with redundancy check information and address swizzling, which allows for error checking as well as obfuscated encoded SRAM data. In Chapter 4, IJTAG was discussed along with a silicon case study, where IJTAG design verification through random IJTAG location access based test was proposed and found to be effective for the actual target chip. To further protect the IJTAG access protocol, Chapter 5 proposed the application of a stream cipher, password protection, chain stubbing, and IJTAG chain checking in an echeloned manner to yield strong protection against adversarial attempts to gain access to the IJTAG network. In Chapter 6, a method to further protect
IJTAG access from a man-in-the-middle attack with a hash chain scheme was proposed. In Chapter 7, an assembly level verification test program generation scheme was proposed to stress potentially worst case hardware based Virtual Machine encryption protection schemes when executed on multiple threads. In Chapter 8, a verification flow to verify RAS features of a q processor SoC with a network-on-chip fabric was proposed. Finally, the MTOU work from Chapter 9 demonstrated that memory test time optimization and embedded memory test diagnostic data logging can be accomplished in synergy.

10.2. Research Impact

The findings in this dissertation can be further interpreted and abstracted into realizable parts of IC design flows. One potential three-pronged approach can be developed in part based on this work. First, IP level verification and protection measures can be utilized to help ensure IPs are free of or protected against malicious circuits such as HTHs to form trusted IPs. Then, at the system level, the SoC test and debug access ports can be protected with hardware measures relying on obfuscation, ciphering, and integrity checking. Furthermore, these protections along with the debug and test ports can be verified utilizing the proposed work on IJTAG verification. Finally, for the SoC mission mode functional hardware operations, security verification techniques can be deployed to ensure IC functional security and error handling operations can function correctly—which can help prevent design errors from being exploited by attackers.

In fact, parts of this work are deployed in actual IC design flows already. The methodology derived from Chapter 7 is being deployed to verify on-chip Virtual Machine features [55]. The IJTAG development flow found in Chapter 4 had pivotal impact on the Design-For-Testability feature developments for a production server processor [88].

The aim for the work proposed in Chapter 9 is to optimize test time and enable meaningful memory test diagnostic data to be logged in volume. MTOU was integrated into a production design. The area impact of implementing MTOU on-chip was shown to be low. MTOU based testflow implementation also demonstrates that embedded memory production test time may
be reduced while enabling diagnostic data logging for every single die of a semiconductor product with integrated embedded memory. The data from such statistics may be used to help identify dice with unusual failure characteristics.

10.3. Closing Remarks

A number of novel approaches aimed at protecting on-chip memories were proposed in this dissertation. The approaches proposed offer useful solutions ranging from design verification to design methodology and test/debug aspects of the IC design to production flow. The proposed approaches may certainly be adopted to enhance existing industrial design or production flows. Furthermore, the knowledge contribution in this dissertation can be used to further advance research in protecting embedded memories.
BIBLIOGRAPHY


