SYSTEM-ON-CHIP VULNERABILITY ANALYSIS AND MITIGATION TECHNIQUES

By

YUANWEN HUANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2017

© 2017 Yuanwen Huang

To my family

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Prabhat Mishra for the continuous support of my Ph.D. study, for his patience, motivation and immense knowledge. His guidance helped me in all the time of research and writing this dissertation. He is the person who made this dissertation come true.

Besides my advisor, I would like to thank the rest of my dissertation committee: Prof. Sartaj Sahni, Prof. Shigang Chen, Prof. Sanjay Ranka, and Prof. Ann Gordon-Ross for their insightful comments and encouragement, but also for the hard questions which incented me to view my research from various perspectives. I also appreciate the help from Prof. Swarup Bhunia and Prof. Anupam Chattopadhyay.

I thank my fellow labmates: Farimah Farahmandi, Terek Arce, Yangdi Lyu, Subodha Charles, Alif Ahmed, Anirudh Canumalla, Emre Ozgener, and Jonathan Cruz. It was my great pleasure to work with you. The friendship with you has made my Ph.D. life so delightful, especially when I was stressed with experiments and deadlines.

Last but not least, I sincerely thank my parents and my sister for their love, support and encouragement. They gave me the courage to take my own path, the strength to go through the difficult times, and the perseverance to try again and again until I find the way. This dissertation would not be possible without their unconditional love. I dedicate this dissertation to them.

This work was partially supported by grants from National Science Foundation (CCF-1218629, CNS-1526687 and CNS-1441667), Semiconductor Research Corporation (2014-TS-2554) and Cisco Systems (F020375). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS . . . 4

LIST OF TABLES . . . 8

LIST OF FIGURES . . . 9

ABSTRACT . . . 12

CHAPTER

1 INTRODUCTION . . . 13

1.1 Vulnerabilities in System-on-Chip . . . 15
1.2 Challenges . . . 16
1.2.1 Vulnerability Due to Soft Errors . . . 16
1.2.2 Vulnerability Due to Hardware Debug Infrastructure . . . 17
1.2.3 Vulnerability Due to Malicious Implantation . . . 19
1.3 Research Contributions . . . 22

2 BACKGROUND AND RELATED WORK . . . 25

2.1 Dynamic Cache Reconfiguration for Vulnerability Reduction . . . 25
2.1.1 Cache Vulnerability . . . 25
2.1.2 Dynamic Cache Reconfiguration . . . 28
2.1.3 Partially Protected Caches for Vulnerability Reduction . . . 31
2.2 Trace Buffer Attack on AES . . . 33
2.2.1 AES Specification . . . 34
2.2.2 AES Attacks . . . 35
2.3 Hardware Trojan Attacks and Detection Techniques . . . 36
2.3.1 Hardware Trojan Attacks . . . 36
2.3.2 Trojan Detection Approaches . . . 37

3 VULNERABILITY AND ENERGY-AWARE CACHE RECONFIGURATION . . . 41

3.1 Motivation: Illustrative Example . . . 42
3.2 Inter-task Cache Reconfiguration . . . 44
3.2.1 System Model . . . 44
3.2.2 Vulnerability-Aware Energy Optimization (VAEO) . . . 45
3.2.3 Heuristic Approach for VAEO Problem . . . 46
3.3 Intra-task Cache Reconfiguration . . . 50
3.3.1 Phase Extraction . . . 52
3.3.2 Cache Assignment for Phases . . . 54
3.3.3 Inter+Intra Cache Reconfiguration . . . 56
3.4 Experiments . . . 59
3.4.1 Experimental Setup . . . 59

3.4.2 VAEO and Intra-task VAEO Configurations 62
3.4.3 Results for Inter-task VAEO and (Inter+Intra)-task VAEO 63
3.4.4 Hardware Overhead 65
3.5 Summary 67

4 VULNERABILITY-AWARE RECONFIGURATION FOR PARTIALLY PROTECTED CACHES 68

4.1 Energy Models 69
4.1.1 Energy Model for Unprotected Cache 69
4.1.2 Energy Model for Protected Cache 69
4.2 Cache Reconfiguration of PPC 70
4.2.1 Data Partitioning 72
4.2.2 Cache Exploration 73
4.2.3 Fast Exploration 76
4.2.3.1 Fast exploration of caches (FEC) 78
4.2.3.2 Fast exploration of caches and data (FECD) 79
4.3 Experiments 79
4.3.1 Synergistic Exploration 80
4.3.2 Comparison with Previous Works 81
4.3.2.1 Improvement to DCR 81
4.3.2.2 Improvement to PPC 82
4.3.3 Fast Exploration 88
4.4 Summary 89

5 VULNERABILITY-AWARE CACHE TUNING FOR MULTICORE SYSTEMS 91

5.1 Modeling Systems with Reconfigurable Caches 92
5.1.1 Multicore Architecture Model 92
5.1.2 Energy and Vulnerability Models 93
5.1.3 Illustrative Example 94
5.1.4 Problem Formulation 95
5.2 Vulnerability-Aware DCR+CP 97
5.2.1 Task Profiling 97
5.2.2 Optimization on Each Core 99
5.2.3 Optimization Across All Cores 100
5.3 Experiments 101
5.3.1 Deadline and Vulnerability Threshold 102
5.3.2 Vulnerability-Aware Energy Reduction 104
5.4 Summary 106

6 TRACE BUFFER ATTACK ON AES CIPHER 109

6.1 Trace Buffer Attack with RTL Knowledge 110
6.1.1 Attack Model 110
6.1.2 Determine Trace Buffer Signals 111
6.1.3 Signal Restoration 113

6.2 Trace Buffer Attack without RTL Knowledge 114
6.2.1 Mapping Signals to Algorithm Variables 114
6.2.2 Attack by Taking Advantage of Rijndael's Key Expansion 117
6.3 Experimental Results 122
6.3.1 Case Study 1: Iterative AES-128 124
6.3.2 Case Study 2: Pipelined AES Ciphers 128
6.4 Proposed Countermeasures 130
6.5 Summary 133

7 STATISTICAL TEST GENERATION FOR TROJAN DETECTION 134

7.1 MERS: Increasing the Trojan Detection Sensitivity 136
7.1.1 Multiple Excitation of Rare Switching 138
7.1.2 Hamming Distance Based Reordering 140
7.1.3 Simulation Based Reordering 141
7.2 Experiments 143
7.2.1 Experimental Setup 143
7.2.2 Evaluation Criteria 144
7.2.3 Exploration of N 146
7.2.4 Side-effect of MERS: Increased TotalSwitch 146
7.2.5 Reordering and Exploration of C 147
7.2.6 Effectiveness of MERS in Creating Trojan Activity 150
7.2.7 Side Channel Sensitivity Improvement 151
7.2.8 Calibration and Multiple-Parameter Side-Channel Analysis 152
7.2.9 Scalability to Large Designs 153
7.3 Summary 154

8 CONCLUSIONS AND FUTURE WORK 155

8.1 Conclusions 155
8.2 Future Research Directions 157

APPENDIX: LIST OF PUBLICATIONS 159

REFERENCES 161

BIOGRAPHICAL SKETCH 170

LIST OF TABLES

Table page

3-1 Four task sets with twelve benchmarks 61

3-2 Overhead of profile table (180nm technology) 66

3-3 Overhead of profile table (65nm technology) 66

4-1 Vulnerability reduction compared to [5] 82

4-2 *DCR+PPC(VulMin)*: vulnerability minimization under energy constraints . . . 87

4-3 *DCR+PPC(EnergyMin)*: energy minimization under vulnerability constraints . 87

4-4 Simulation time for three exploration strategies: *Exhaustive*, *FEC* and *FECD* . 87

5-1 TASK SETS FROM MIBENCH [46] AND CPU2000 [57] BENCHMARKS 108

5-2 TASK SET 1: CACHE CONFIG ($[c_I, c_D, w_k]$) 108

5-3 TASK SET 9: CACHE CONFIG ($[c_I, c_D, w_k]$) 108

6-1 Recover when the fourth word of the round key register are known. 120

6-2 Restore the primary key from the available trace buffer content. 125

6-3 Signal restoration for iterative AES-128. 127

6-4 Pipelined AES-128, AES-192 and AES-256. 130

6-5 Comparison of LFSR-based and PUF-based countermeasures 133

7-1 Runtime for MERS test generation and test reordering. 144

7-2 Comparison of MERS (N=1000) with Random (10K) for average MaxDeltaSwitch and average AvgDeltaSwitch. 150

7-3 Comparison of MERS (N=1000) with Random (10K) for average *MaxRelativeSwitch* (Side Channel Sensitivity) and average *AvgRelativeSwitch*. . 151

7-4 Comparison of average *Side Channel Sensitivity* between Random (10K), MERO, and MERS testsets, N=1000, C=5 for MERS-s, over 1000 random samples of 4-trigger Trojans. 152

7-5 Comparison of average *Side Channel Sensitivity* between Random (10K), MERO, and MERS testsets, N=1000, C=5 for MERS-s, over 1000 random samples of 8-trigger Trojans. 152

LIST OF FIGURES

Figure		page
1-1	The Internet of Things (IoT) is pervasive in every aspect of our daily life.	13
1-2	IoT devices typically contain a System-on-Chip (SoC) computing platform with reusable hardware IP components.	14
1-3	System-on-Chip vulnerabilities.	15
1-4	Overview of trace buffer in system validation and debug	18
1-5	Example of a combinational and a sequential Trojan with triggers from two rare internal nodes A and B.	21
1-6	Dissertation Outline	24
2-1	Vulnerable intervals of two data elements in cache (where W=Write Access, R=Read Access, E=Evict). (a) data with both *write* and *read* accesses; (b) data with only *read* accesses.	27
2-2	DCR in a system with three tasks.	30
2-3	AES Encryption Flow	34
3-1	Energy and vulnerability values of *pegwit* benchmark using different cache configurations. (a) Energy and miss rate, (b) Vulnerability and miss rate.	42
3-2	VAEO and PO configurations of two benchmarks: (a) *epic* and (b) *qsort*.	48
3-3	Data cache misses for benchmark *qsort*. (a) Without intra-task reconfiguration, using one cache for the whole task; (b) With intra-task reconfiguration, using a different cache for each of the three phases.	51
3-4	Phases identified for different benchmarks.	53
3-5	Comparison of DL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations. (a) Data cache energy consumption, (b) Data cache vulnerability.	59
3-6	Comparison of IL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations. (a) Instruction cache energy consumption, (b) Instruction cache vulnerability.	60
3-7	(a) Data cache energy and (b) Data cache vulnerability.	63
3-8	(a) Instruction cache energy and (b) Instruction cache vulnerability.	64
4-1	A reconfigurable PPC-base architecture with one protected cache and the other unprotected cache at the same level of hierarchy.	68

4-2 Our exploration methodology consists of two design decisions: data partitioning and cache reconfiguration. 71

4-3 Trade-off between vulnerability, runtime and energy, for benchmark *cjpeg*. (a) Vulnerability and runtime trade-off, (b) Energy consumption. 71

4-4 Configurations covered by (a) Exhaustive Exploration (108), (b) FEC Exploration (41), for benchmark *cjpeg*. 76

4-5 Configurations covered by (a) Exhaustive Exploration (108), (b) FEC Exploration (29), for benchmark *pegwit*. 77

4-6 Page exploration during data paritioning for six different benchmarks 77

4-7 Exploration of all cache configurations for benchmark *cjpeg*, with $rThresh = 5\%$. 84

4-8 Exploration of all cache configurations for benchmark *pegwit*, with $rThresh = 5\%$. 85

4-9 Comparison of *DCR+PPC(VulMin)* and *DCR+PPC(EnergyMin)* with PPC[39]. (a) Vulnerability improvement, (b) Energy improvement. 86

4-10 Comparison of three exploration strategies: *Exhaustive*, *FEC* and *FECD*. (a) DCR+PPC(VulMin), (b) DCR+PPC(EnergyMin) 90

5-1 A multicore system with reconfigurable L1 caches and a partition-enabled shared L2 cache. 92

5-2 Inter-dependence of L1 DCR and L2 CP on (a) L2 Misses, (b) IPC, (c) Runtime, (d) Energy and (e) Vulnerability. 93

5-3 Three-step optimization: the first step statically profiles each task, the second step optimizes for each partition factor on each core to find the best L1 cache configurations, the third step combines the optimal solution on all cores to find the best L2 partition scheme. 97

5-4 Recursive formula for dynamic programming 100

5-5 Effects of Deadline and Vulnerability Threshold. 103

5-6 Comparison of vulnerability and energy consumption for the cache hierarchy. (a) Vulnerability, (b) Energy consumption. 104

6-1 Illustration of signal restoration for an AND gate 113

6-2 C Code Snippet for AES-128 116

6-3 An example showing the recovery of missing bits in $RK_{(4,4)}$ by using *sbox* lookup table (Rule 1). 121

6-4 AES *sbox* lookup table (the numbers are in hexadecimal format) 122

6-5 Security and observability trade-off using different buffer widths and buffer depths. 127

6-6 Pipelined AES-128, AES-192, and AES-256 ciphers: security and observability trade-off. 129

6-7 LFSR-based Countermeasure 131

6-8 PUF-based Countermeasure 131

7-1 Trojans with rare nodes as trigger conditions. The 4-trigger Trojan will only be activated by the rare combination 1011 and the 8-trigger Trojan will only be activated by the rare combination 10110011. 136

7-2 Test generation framework for side-channel analysis based Trojan detection. . . 143

7-3 Impact of N (number of times that a rare node have rare switching) on MaxDeltaSwitch for benchmarks (a) c2670 and (b) c3540. 145

7-4 MaxDeltaSwitch versus TotalSwitch for different N for benchmarks (a) c2670 and (b) c3540. 147

7-5 Side Channel Sensitivity versus Total Switch for Random, the original MERS, MERS-h and MERS-s (with different C) for benchmarks (a) c2670 and (b) c3540. 148

7-6 Distribution of Side Channel Sensitivity for Random, the original MERS, MERS-h and MERS-s (with different C) for benchmarks (a) c2670 and (b) c3540. 149

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SYSTEM-ON-CHIP VULNERABILITY ANALYSIS AND MITIGATION TECHNIQUES

By

Yuanwen Huang

August 2017

Chair: Prabhat Mishra
Major: Computer Engineering

As Internet of Things (IoT) stretches into every aspect of our life, the IoT devices provide us with efficiency, convenience and economic benefits. The IoT devices typically contain a System-on-Chip (SoC) computing platform. However, more and more vulnerabilities are discovered in SoCs, which suggests the urgent need of research to analyze and mitigate these vulnerabilities. This dissertation focuses on the hardware vulnerabilities of SoCs, which includes the memory vulnerability due to soft errors, the vulnerability of debug infrastructure, and malicious hardware implantation in integrated circuits. My research has made five major contributions: (i) it analyzes vulnerability of cache due to soft errors, and proposes scheduling-aware cache reconfiguration algorithms to reduce vulnerability for single-core systems; (ii) it proposes a reconfiguration approach for partially protected caches to mitigate cache vulnerability and reduce energy overhead; (iii) it proposes a cache reconfiguration strategy for multicore systems to reduce cache vulnerability; (iv) it analyzes the vulnerability of trace buffers, demonstrates that an attack can be mounted to steal the encryption key, and proposes countermeasures against this type of attack; (v) it analyzes the vulnerability due to hardware Trojans, and proposes statistical test generation algorithms for side-channel based Trojan detection. Experimental results suggest that the proposed approaches can significantly reduce hardware vulnerability to enable design of secure and reliable SoCs.

CHAPTER 1
INTRODUCTION

The Internet of Things (IoT), defined as the infrastructure of the information society, is the network of physical devices embedded with electronics, software and network connectivity. As shown in Figure 1-1, IoT devices are pervasive in every aspect of our daily life. They enable us to have smart home, smart transport, smart finance, smart health, and so on. It is estimated that the IoT will consist of 50 billion objects by 2020 [1]. Safety, security and privacy concerns of IoTs have received significant attention from both academia and industry in recent years. Safety is a natural concern, since IoT devices and systems inherently have humans in the loop. People's lives depend on safety-critical devices, such as pacemakers and self-driving cars. The report by HP [2] indicates that 70% of the IoT devices have serious vulnerabilities. Many of these IoT devices are becoming easy targets for attackers. Privacy also becomes a concern, as many of the IoT devices have sensors to collect data from humans. It raises great concern regarding privacy of confidential personal information and health-related sensitive data.

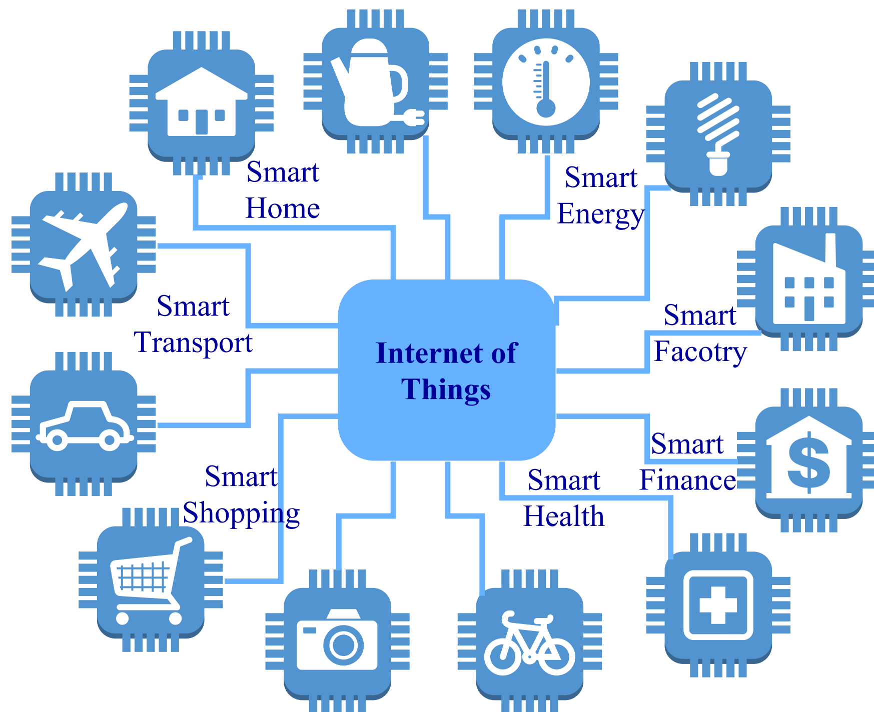


Figure 1-1. The Internet of Things (IoT) is pervasive in every aspect of our daily life.

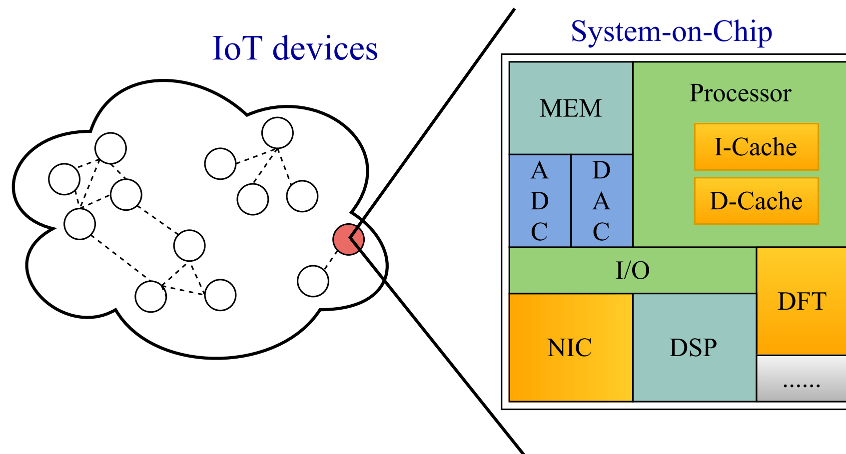


Figure 1-2. IoT devices typically contain a System-on-Chip (SoC) computing platform with reusable hardware IP components.

Majority of these IoT devices are designed using System-on-Chip (SoC) components and software applications [3]. In order to reduce design complexity as well as the time to market, SoCs use third-party Intellectual Property (IP) components. For example, Figure 1-2 shows a typical SoC with a processor core, a Digital Signal Processor (DSP), memory (MEM), analog-to-digital (ADC) and digital-to-analog (DAC) converters, network interface card (NIC), peripherals (I/O), design-for-test facility (DFT), and so on. In many cases, these IP components are designed by different companies across the globe. Each functional component may travel through long supply chain involving multiple vendors before they are integrated into a SoC. This design methodology of employing reusable hardware IPs severely affects the security and trustworthiness of SoC computing platforms, primarily due to three factors. (1) The third-party components might not be fully tested/verified under all conditions (such as corner-case inputs, extreme environment conditions). The reliability and availability of the SoC might be compromised when such a corner-case condition happens. (2) These SoCs may have undocumented test or debug interfaces that could be used as a backdoor for information leakage. (3) An SoC may come with malicious implants to incorporate undesired functionality (e.g. hardware Trojans).

1.1 Vulnerabilities in System-on-Chip

The top ten IoT vulnerabilities identified by the Open Web Application Security Project (OWASP) include [4]: insecure web interface, insufficient authentication or authorization, insecure network services, lack of transport encryption or integrity verification, privacy concerns, insecure cloud interface, insecure mobile interface, insufficient security configurability, insecure software or firmware, and poor physical security. These reported vulnerabilities mostly focus on analysis of the network/cloud and software/firmware holes in IoT devices. There are cases involving poor physical security, which is mostly gaining access to the software system through unprotected physical interfaces. There has not been much analysis of vulnerabilities of the underlying hardware itself. In this dissertation, we zoom into the underlying SoCs and investigate the hardware components for vulnerability analysis.

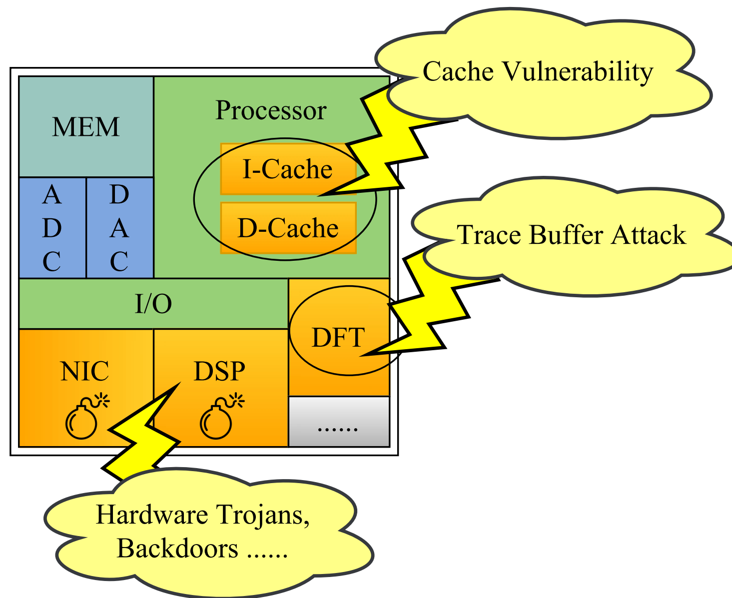


Figure 1-3. System-on-Chip vulnerabilities.

Vulnerabilities of a SoC come from its hardware IP components. According to the weakest-link theory, a SoC system is as vulnerable as the most vulnerable component. It is therefore important to verify each hardware component thoroughly before integrating them into a SoC. As shown in Figure 1-3, we have illustrated three possible vulnerabilities

that pose threats to the reliability and security of the SoC. Assume the caches in the processor have no protection against corrupted data (for example, a flipped bit). This error in caches may propagate and cause the whole system to crash. As for the Design-for-Test (DFT) module, the debugging information stored in the trace buffer could be used by attackers as a source of information leakage. We have shown a successful attack mounted to the trace buffer to steal the AES encryption key. Assume that the Network Interface Card (NIC) module and Digital Signal Processing (DSP) module are from untrusted third-party vendors, malicious implantations (such as hardware Trojans and backdoors) might be inserted for tampering the SoC or leaking information. In this dissertation, we focus on these three types of hardware vulnerabilities in SoCs. Analysis and mitigation techniques are proposed to improve the reliability and security of SoCs.

1.2 Challenges

In this dissertation, three aspects of hardware vulnerability analysis are covered: (1) vulnerability of cache cells due to soft errors; (2) vulnerability of trace buffer used in hardware debugging; (3) vulnerability introduced by malicious implantations in integrated circuits. The following section describes the challenges concerning these vulnerabilities.

1.2.1 Vulnerability Due to Soft Errors

Soft errors are transient faults in CMOS circuits, which are caused by energy carrying particles (cosmic rays or substrate alpha particles). These transient faults flip bits in storage cells or change the logic values in functional units. Soft error rate per chip is expected to grow due to the growing density of transistors on chip [6].

Previous studies have concluded that unprotected memory elements are the most vulnerable components to soft errors [7]. The cache in embedded microprocessors is most susceptible to soft errors for several reasons: (i) cache occupies the majority of chip area, (ii) cache has an extremely high density of transistors, and (iii) cache cell size scales down, which reduces the critical charge needed to flip a bit in stored data. Due to widespread

use of embedded systems in safety-critical devices, it is necessary to protect embedded caches from soft errors.

Detection and correction of soft errors in logic circuits usually involve the techniques of fault tolerant design. These include the use of redundant circuitry, such as triple modular redundancy (TMR). In this technique, three identical copies of a circuit are employed to compute on the same data in parallel and the result would be the value occurred in at least two of the three cases. In this way, the failure of one circuit due to soft error is discarded if the other two copies of the circuit work correctly. However, it will come at 200% overhead in circuit area and power, which is unacceptable in many cases. Another concept of temporal (or time) redundancy uses one circuit on the same data and checks consistency between several repetitive executions. This approach is more efficient than the modular redundancy approach, although it still incurs performance and power overhead. In this dissertation, we propose several cache reconfiguration techniques to mitigate vulnerability due to soft errors, with negligible impact on power and performance.

1.2.2 Vulnerability Due to Hardware Debug Infrastructure

One of the major challenges in post-silicon validation and debug is the limited controllability and observability of the fabricated integrated circuit. Trace buffer is widely used to improve the observability of circuit and thus assist post-silicon debug and analysis. It is a buffer that traces (records) some of the internal signals in a silicon chip during runtime. If an error is encountered, the content of trace buffer would be dumped out through JTAG interface for off-line debug and error analysis. Due to design overhead constraints, the number of trace signals is only a small fraction of all internal signals in the design. The size of the trace buffer directly affects the observability that we can get from the trace buffer.

Figure 1-4 illustrates how the trace buffer is used during post-silicon validation and debug. Signal selection is done during the design time (pre-silicon phase). Let us assume that S_1, S_2, ..., S_n are the selected trace signals. Figure 1-4 shows a trace buffer with a

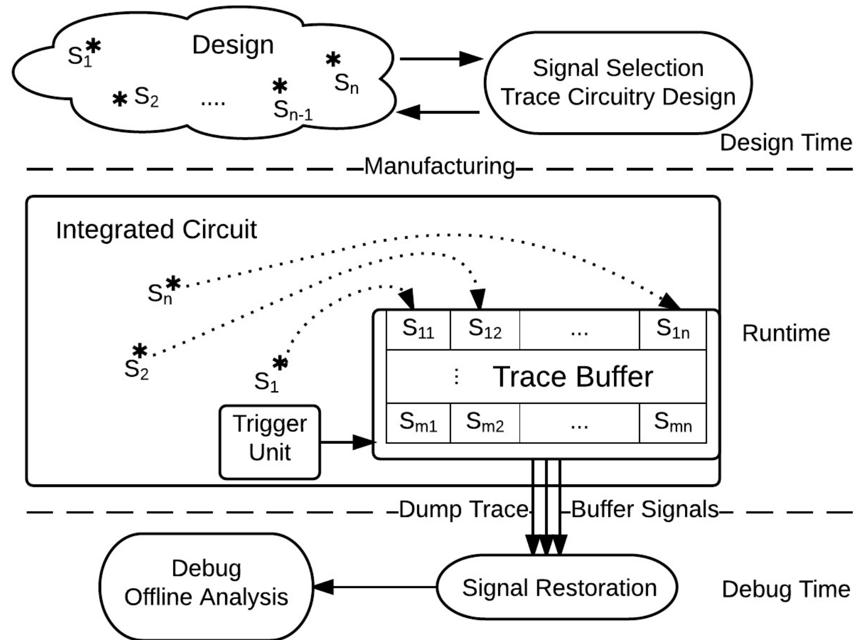


Figure 1-4. Overview of trace buffer in system validation and debug

total size of $n \times m$ bits, which traces n signals (buffer width) for m cycles (buffer depth). For example, the ARM ETB [62] trace buffer provides buffer sizes ranging from 16Kb to 4Mb. In this case, a 16Kb buffer can trace 32 signals for 512 cycles (i.e., n=32 and m =512). Once the trace signals are selected, they need to be routed to the trace buffer. A trigger unit is also needed that decides when to start and stop recording the trace signals based on specific (error) events. The trace buffer records the states of the traced signals during runtime. During debug time, the states of traced signals are dumped out through the standard JTAG interface. Signal restoration is performed to restore as many states as possible, which tries to maximize the observability of the internal signals in the chip. The off-line debug and analysis are based on the traced signals and the restored signals.

System-on-Chip (SoC) designs have in-built trace buffer that traces a small set of internal signals during execution, and the traced signal values are used during post-silicon (off-line) debug. There is an inherent conflict between security and observability. While debug engineers would like to have better observability, the security experts would like to enforce limited or no visibility with respect to the security modules in a SoC design.

A trade-off is typically made where trace signals are carefully selected to maintain security while providing reasonable debug capability. To the best of our knowledge, the vulnerability of trace buffers in cryptographic implementation has not been studied in the literature.

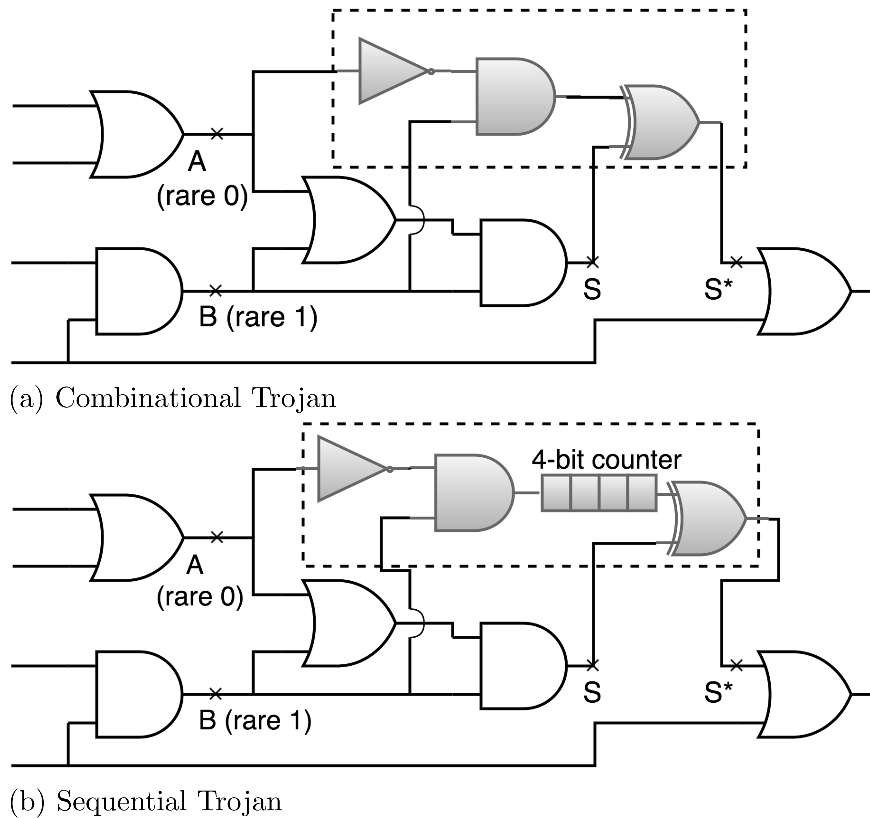
In practice, one routinely faces a situation where the cryptographic schemes are deployed in different adversarial setting, where keys are compromised, and the internal memory is not fully opaque. This situation leads to a set of physical cryptanalysis techniques, commonly known as *side channel attacks*. Side channel attacks exploit the physical implementation of cryptographic algorithms. The physical implementation might enable *leakage*, i.e., observations and measurements on the implementation details, as well as tampering with them. Trace buffer in post-silicon can provide observability into the hardware implementation, which implies that it can be employed as a source of information leakage by attackers.

1.2.3 Vulnerability Due to Malicious Implantation

Hardware Trojan attacks relate to malicious modifications in the design of integrated circuits (ICs) at different stages of the design or fabrication process [100][103][105]. An adversary can introduce these modifications in a design in order to cause disruption in normal functional behavior and/or to leak secret information from a chip during operation in field. Increased globalization of IC design and fabrication process coupled with reduced control on these steps by a trusted manufacturer makes the ICs highly vulnerable to these attacks. Since the threat of hardware Trojan in the form of a malicious implant in a design came into light about a decade ago through an US Department of Defense announcement [97], it has triggered wide array of research activities in threat analysis as well as design/validation solutions to evaluate this threat and protect against it. Hardware Trojan attacks are also being increasingly recognized in the semiconductor industry as a serious security concern.

A Trojan is expected to be covert and difficult to detect, i.e. an intelligent adversary will likely insert a Trojan circuit in a way that evades detection during post-manufacturing functional/parametric testing, but manifests itself during long hour of in-field operation. This can be achieved by externally triggering its operation or by making it dependent on rare circuit conditions inside an IC. The condition of Trojan activation as commonly referred to as *trigger condition*, which can be purely combinational or sequential, related to the clock or a sequence of rare events in the state elements (e.g. flip-flops of registers). The internal circuit nodes affected by a Trojan activation are referred to as *payload* of a Trojan. Fig. 1-5 shows some example Trojan circuits including a combinational and a sequential Trojan. For example, a Trojan circuit could be triggered only when a data bus attains a unique rare value or when the number of times it attains the rare value equals a particular count. The malicious effects of Trojan payloads can range from passive, such as leakage of secret information to altering the original functionality of the chip in a critical or destructive fashion.

Protection against hardware Trojan attacks can be accomplished in two broad ways: (1) design-for-security techniques that make Trojan insertion difficult or make a Trojan easily detectable through post-silicon testing; and (2) manufacturing test approaches that aims at detecting an arbitrary Trojan by observing its effect into a circuit's operational behavior. The first class of techniques, primarily relies on different types of hardening approaches - e.g. insertion of dummy cells into empty spaces in a circuit layout; or key-based obfuscation of a design that make malicious alteration by an adversary provably hard. Design techniques, however, come at the cost of additional design, verification, and test time, as well as additional design overhead. For example, key-based obfuscation, even though is capable of providing high level of robustness against Trojan attacks, come at a cost of 10% or more area overhead [98]. More importantly, design solutions, however, only work for new designs and not legacy designs, and hence has limited applicability. Hence, efficient test/validation approaches that can provide high level of confidence regarding IC



(a) Combinational Trojan

(b) Sequential Trojan

Figure 1-5. Example of a combinational and a sequential Trojan with triggers from two rare internal nodes A and B.

trustworthiness in the presence of Trojan threat provides an attractive solution to the IC manufacturers.

Existing test solutions for hardware Trojan detection can be broadly classified into: 1) logic testing and 2) side-channel analysis approaches. In logic testing approach, directed structural or functional tests are generated to activate rare events in the circuit and propagate the malicious effect in logic values to primary outputs. Such approaches are known to be more effective in detecting ultra-small Trojans (typically a few gates in size) reliably under large process variations. The main challenge with logic testing approaches, however, is the difficulty to trigger a Trojan and observe its effect, particularly the complex sequential Trojans, and the inordinately large number of possible Trojan instances an adversary can exploit. On the other hand, side-channel analysis approaches, which depend on measurement of physical "side-channel" parameters like power signature

of an IC in order to identify a structural change in the design. Such approaches have the advantage that they do not require triggering a malicious change and observing its impact at the primary output. Side-channel analysis (SCA) - primarily based on supply current has been extensively investigated by large number of research groups and various solutions to increase the signal-to-noise (SNR) has been proposed. A disadvantage of SCA is in terms of large process variations which can potentially mask the minute effect of a Trojan in the measured side-channel parameter e.g. 20X leakage power and 30% delay variations in 180nm technology [99].

A solution to the sensitivity problem can be achieved by judicious test generation approach that aims at maximizing the sensitivity for an arbitrary Trojan in unknown circuit location. To maximize sensitivity of a given Trojan, one needs to amplify activity inside the Trojan circuit and simultaneously minimize the background activity (i.e. activity in the original circuit). However, since the number of possible Trojan instances in a design can be inordinately large, a deterministic test generation method similar to conventional stuck-at fault test generation, cannot work. To address this issue, in this dissertation, we present a novel test generation framework that can maximize the detection sensitivity for an arbitrary Trojan.

1.3 Research Contributions

My research proposes novel techniques to address vulnerability challenges described in Section 1.2. The objective of my research is to identify/analyze vulnerabilities and design efficient tools and techniques to mitigate these vulnerabilities in SoCs. The proposed research focuses on three major directions of hardware vulnerabilities: (1) the vulnerability due to soft errors, induced by high-energy particles hitting the microchip; (2) the vulnerability due to the exploitation of hardware debug infrastructure; (3) the vulnerability due to malicious implantations (hardware Trojans).

Figure 1-6 outlines the major research contributions of the dissertation that are summarized as follows.

- **Vulnerability-aware Cache Reconfiguration:** This dissertation examines the vulnerability issue of cache cells due to soft errors. The proposed approach exploits dynamic reconfiguration of embedded caches in multitasking soft real-time systems. By profiling each task with its optimal cache configurations statically, and dynamically selecting configurations at runtime, our approach can reduce vulnerability as well as energy consumption.

- **Vulnerability-aware Reconfiguration for Partially Protected Caches:** This dissertation proposes a reconfigurable cache architecture with the partially protected caches. The research combines the advantage of cache reconfiguration, which is favourable for energy and performance, and the advantage of partially protected caches, which is favourable for vulnerability reduction. The proposed approach synergistically explores data partitioning schemes and cache configurations to achieve both vulnerability reduction and energy improvement with minor impact on performance.

- **Vulnerability Reduction for Multicore Systems:** This dissertation proposes a cache reconfiguration approach for multicore systems to reduce vulnerability. The proposed approach uses dynamic reconfiguration of the private L1 caches and static partitioning of the shared L2 cache. Each task for each core is statically profiled. The optimal configurations for L1 caches and partition factors for L2 can be determined by a dynamic programming algorithm.

- **Debugging Infrastructure Vulnerability:** This dissertation analyzes the vulnerability of debugging infrastructure, specifically the trace buffer used in post-silicon debug. We investigate the trace buffer as a source for information leakage. Our proposed approach is able to mount an attack on different implementations of the AES ciphers through the help of trace buffer. Unless proper countermeasure is taken, the proposed attack can steal part of the key from the AES chip.

- **Vulnerability due to Malicious Implantation:** This dissertation surveys the pros and cons of test generation and side-channel analysis in detecting malicious implantations (hardware Trojans). I propose a novel side-channel-aware test generation approach, based on a concept of Multiple Excitation of Rare Switching (MERS). The tests generated by the proposed approach can significantly increase the Trojans sensitivity, thereby making Trojan detection effective using side-channel analysis.

The rest of this dissertation is organized as follows. Chapter 2 describes relevant background and related work. Chapter 3 presents the vulnerability-aware cache reconfiguration technique in single-core multitasking systems. Chapter 4 describes the synergistic exploration of data partitioning and cache reconfiguration for partially

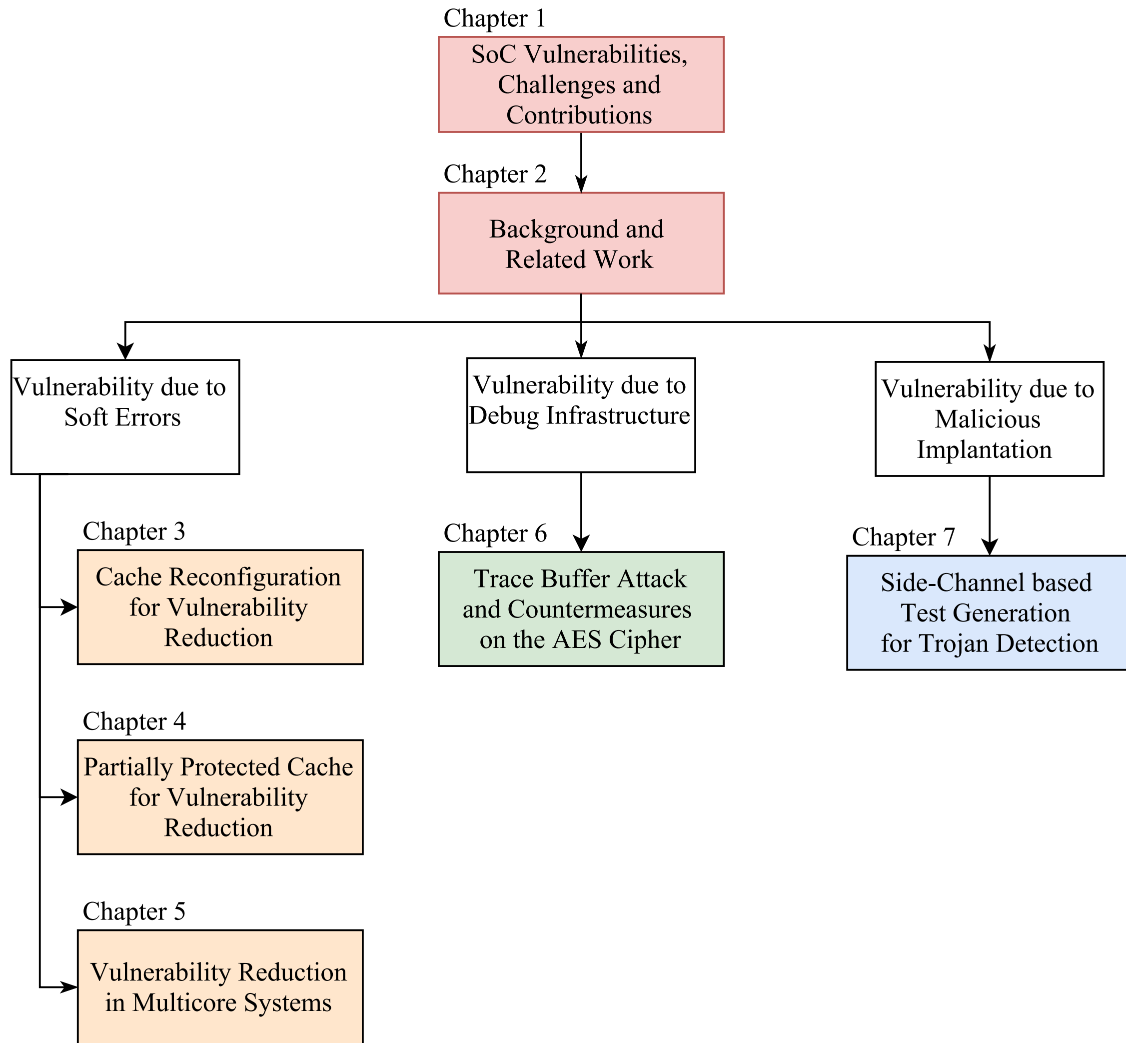


Figure 1-6. Dissertation Outline

protected caches. Chapter 5 presents the cache reconfiguration approach for multicore systems to reduce cache vulnerability. Chapter 6 analyzes the vulnerability of trace buffers and presents an attack on the Advanced Encryption Standard (AES) cipher. Chapter 7 presents a statistical test generation approach for side-channel based Trojan detection. Chapter 8 concludes the dissertation.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter surveys existing System-on-Chip vulnerability analysis and mitigation techniques. For ease of presentation, we have divided the existing approaches into three categories. First, we describe the existing techniques for cache vulnerability reduction. Next, we discuss existing research on security vulnerabilities in post-silicon debug. Finally, we present vulnerability due to hardware Trojans and associated detection techniques.

2.1 Dynamic Cache Reconfiguration for Vulnerability Reduction

This section is organized as follows. First, we describe how to compute vulnerability of caches. Next, we present existing cache reconfiguration techniques. Finally, we highlight the importance of partially protected caches for vulnerability reduction.

2.1.1 Cache Vulnerability

Soft errors are transient faults in CMOS circuits, which are caused by energy carrying particles (cosmic rays or substrate alpha particles). These transient faults flip bits in storage cells or change the logic values in functional units. Soft error rate per chip is expected to grow due to the growing density of transistors on chip [6]. Previous studies have concluded that unprotected memory elements are the most vulnerable components to soft errors [7]. According to [29], DRAMs have soft error rate (SER) of 1000 FIT, while SRAMs have SER of 100000 FIT (1 FIT is 1 failure in one billion device hours). Soft error rate of SRAM is significantly higher than that of DRAM [29, 31, 32]. The cache in embedded microprocessors is most susceptible to soft errors for several reasons: (i) cache occupies the majority of chip area, (ii) cache has an extremely high density of transistors, and (iii) cache cell size scales down, which reduces the critical charge needed to flip a bit in stored data. Due to widespread use of embedded systems in safety-critical devices, it is necessary to protect embedded caches from soft errors.

Major reliability improvement techniques include error correction and error prevention [6], [16], [19]. Error correction techniques, such as parity caching and error-correcting

codes (ECC), use spatial redundancy to detect errors. If an error is detected in a cache block and this block is not dirty (i.e. memory has a correct copy of this block), it is possible to recover by reloading from memory. But if an error is detected in a dirty block, there is no way to recover the corrupted data. An important idea in protecting cache data from soft errors is to ensure that there is an updated copy of all cached data in memory (so data can be reloaded if soft error corrupts data). Even when caches have error detection or correction techniques, the detection/correction process comes at a cost. It consumes multiple clock cycles to correct the error and the CPU might get stalled to re-fetch the data if the error cannot be successfully corrected by ECC. To combat the data vulnerability due to soft errors, error correction codes (ECC) are used in lower levels of the memory hierarchy. However, ECC might not be suitable for caches because of short access time constraints [43]. Error prevention techniques [20], such as periodic flushing and early write-back, are introduced. However, too many memory-writes will keep the data-bus busy, which results in longer cache-miss latency and decreased overall performance. Particularly, write-through caches will always write data all the way to memory, but may not be a good idea for embedded systems. Moreover, too many data accesses will also consume a lot more energy than write-back caches. These hardware techniques need extra hardware support in cache, and are not sensitive to the data access pattern of the applications. In this dissertation, we assume no error prevention, as we aim to reduce vulnerability with the given reconfigurable cache architecture using Dynamic Cache Reconfiguration (DCR). Our approach for vulnerability reduction can work on top of any error correction or error prevention techniques. Our goal is to take advantage of the reconfigurable cache and the data access pattern of applications to reduce vulnerability and improve energy efficiency while meeting task deadlines.

In order to facilitate reliability analysis of cache, a measurement method is needed for the quantification of cache vulnerability due to soft errors [15]. Mukherjee et al. [17] introduced the concept of Architectural Vulnerability Factor (AVF). Vulnerability analysis

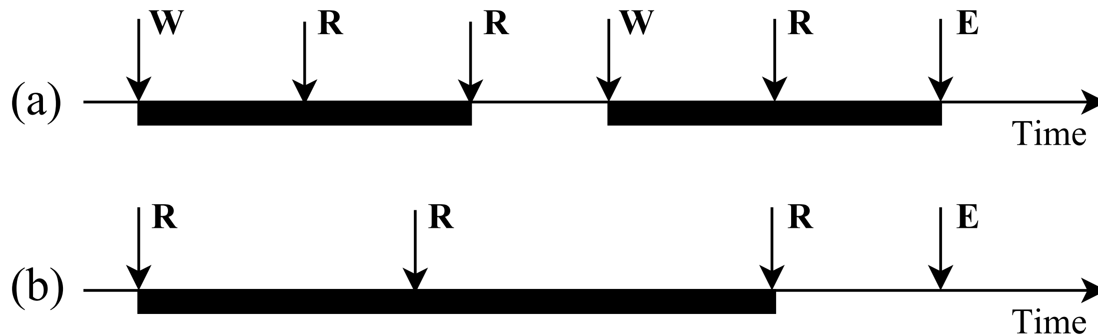


Figure 2-1. Vulnerable intervals of two data elements in cache (where W=Write Access, R=Read Access, E=Evict). (a) data with both *write* and *read* accesses; (b) data with only *read* accesses.

divides a bit's lifetime into vulnerable and un-vulnerable intervals [17, 18]. A bit is vulnerable for an interval, if soft errors that happen in this interval cause the program to get contaminated data. Similar to [17] and [21], we measure the vulnerability of cache on a per-byte basis. Activities during the lifetime of a byte includes "*idle*", "*read*", "*write*" and "*eviction*". Figure 2-1(a) shows a data with both *read* and *write* accesses, and the vulnerable intervals are marked by two black rectangles: the data is vulnerable between the first *write* and the second *read* as well as between the second *write* and the *evict*. During these two intervals, the data is read for reuse, while a flipped bit can corrupt the data, causing the program to use the corrupted cache data. The interval between the second *read* and the second *write* is un-vulnerable, since the data will be updated by the *write* operation even if soft errors corrupt it. The duration between the last (third) read and evict is also vulnerable, since this is dirty (modified) data and needs to be written back to memory. Therefore, any bit flip in this duration will result in corrupted data being written back to memory. Figure 2-1(b) shows a data with only *read* accesses, and the intervals between *read* accesses are vulnerable. However, the interval between the last *read* and the *evict* is un-vulnerable, since data will not be reused or written back to memory. *Byte Cycles* is an widely used term for measuring cache vulnerability [7, 34]. We measure the vulnerability of cache as the summation of vulnerable intervals of all bytes. It can be

defined as follows:

$$\mathit{Vulnerability} = \sum_{\mathit{all\ bytes}} \mathit{vulnerable\ time\ of\ byte}_i$$

2.1.2 Dynamic Cache Reconfiguration

Dynamic Cache Reconfiguration (**DCR**) is a widely studied method for optimizing energy and performance in embedded systems [8]. The basic idea of cache reconfiguration is that different programs have varying data and instruction access characteristics during execution (runtime) and DCR tries to find the optimal cache configuration for a given application (program). For example, we can improve performance by increasing cache size when a program needs a lot of data accesses. Similarly, we can save energy by shutting down a part of the cache if the program is not data-intensive. However, cache reconfiguration will also affect the vulnerability due to soft errors. A large cache size for a data-intensive program might have fewer cache misses and thus improve energy and performance efficiency, but it is also likely to increase the vulnerability of cache data because of longer data retention in the cache.

Applications have varied instruction and data access patterns, which means that they require different cache requirements in terms of cache size, line size, and associativity. If the cache configuration is tuned according to the need of the application, we can gain performance improvement and energy savings. Figure 2-2 illustrates that inter-task and intra-task DCR can improve overall performance by tuning cache size for a system with three tasks. We assume that cache size is the only tunable parameter of cache for the ease of illustration (line size and associativity remain the same). Figure 2-2(a) shows a traditional system using a fixed *base cache*[1] , whereas in Figure 2-2(b) each task uses its

[1] ***Base cache*** refers to the cache used in typical real-time systems, which is chosen to ensure durable task schedules. Typically, *base cache* is the globally optimal cache configuration determined during design time for a set of tasks.

favorable cache configuration and the overall execution time is improved. In the traditional system with a fixed *base cache*, Task 1 starts execution at time t_0, Task 2 and Task 3 start at t_1 and t_2, respectively. The fixed *base cache* has a 4096-byte cache size for all tasks. In inter-task (application-based) cache reconfiguration, DCR tunes the cache when a new task starts its execution. Assuming Task 1 is computation-intensive, we choose a smaller (2048-byte) cache to save energy, while the execution time will increase. Assuming Task 2 is data-intensive, we choose a larger (8192-byte) cache and its runtime is greatly improved. Figure 2-2(c) shows the effect of combining inter- and intra-task cache reconfiguration. By introducing intra-task reconfiguration, Task 1 can improve its performance if suitable configurations are applied to the four phases during execution. Assuming Task 2 has three phases, we set the cache to be large (8192-byte) for the first and third phase for performance consideration, and set the second phase to 4096-byte to reduce energy consumption. For Task 3, inter-task reconfiguration is not able to find a better cache than 4096-byte, while intra-task reconfiguration can find three phases and improve the performance and/or energy.

It is a major challenge to improve the reliability of real-time embedded systems with special design considerations of real-time constrains. Hard real-time systems require that all tasks must complete execution before their deadlines to ensure correct execution. A task set is considered schedulable if there exists a schedule that satisfies all the timing constraints. Due to stringent timing constraints, scheduling for hard real-time systems must perform task schedulability analysis based on task attributes (such as deadlines, priorities, and periods) [9]. For soft real-time systems, minor deadline misses may result in temporary service degradation, but will not lead to incorrect behavior. An efficient cache reconfiguration framework is proposed for energy optimization in soft real-time systems in [8]. They exploit the flexibility of soft real-time systems and manage to achieve considerable energy savings with minor impact on user experiences. However their method does not consider the vulnerability of cache due to soft errors.

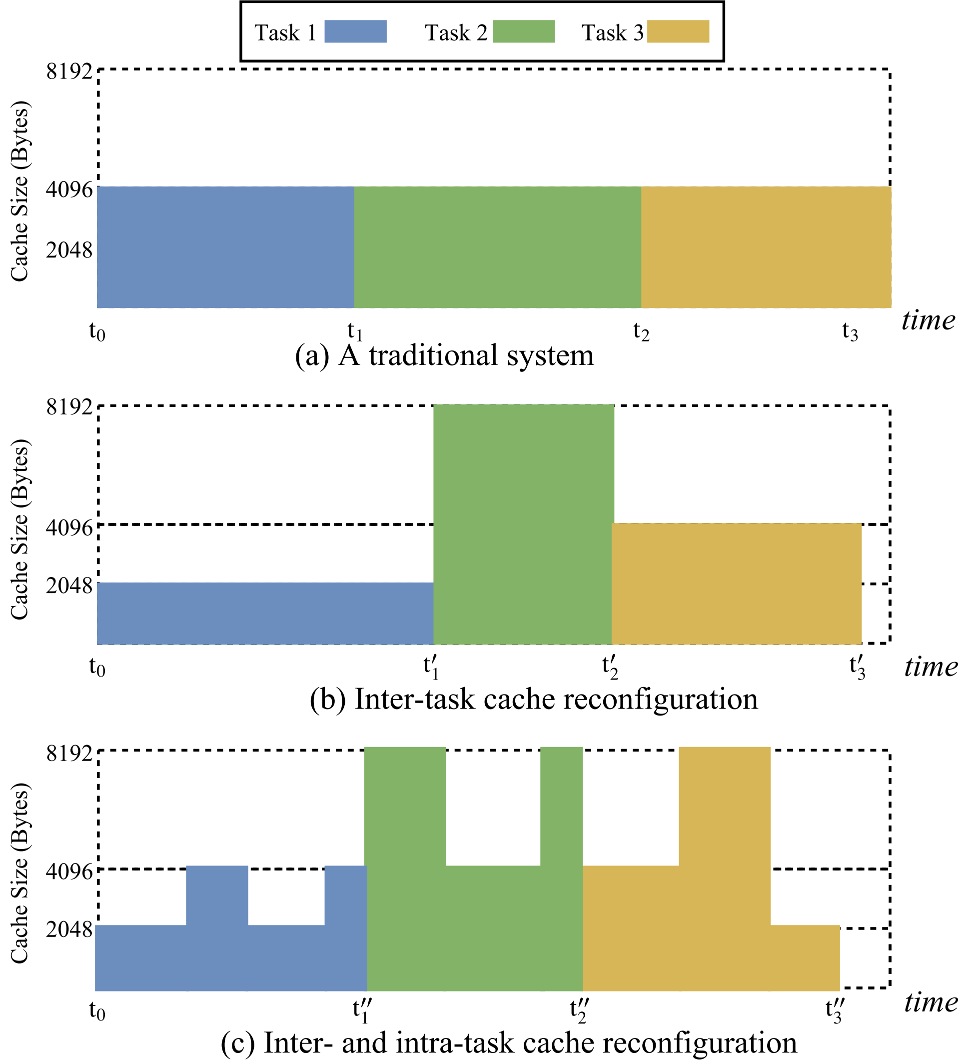


Figure 2-2. DCR in a system with three tasks.

DCR has been extensively studied by previous works [8], [10], [11], [12], [13]. Reconfigurable cache architectures are extensively studied in [49]. Gordon-Ross et al. [50] utilizes DCR to improve performance by online feedback and dynamic self-tuning of the cache. An energy-efficient approach using DCR is proposed in [11] for soft real-time systems using static profiling and dynamic reconfiguration. DCR in two-level cache hierarchy in uniprocessor has been studied by [47]. DCR for multicore systems has been studied by [13] for thread-fairness and performance improvement. Wang et al. proposed an energy-efficient approach for multicore systems in [12] by using DCR on private L1

caches and cache partitioning (CP) on the shared L2 cache. CP is a special case of reconfiguration on the shared cache among multiple cores [51, 52]. Initial works of CP aim at improving the performance of multicore systems [52, 53]. Reddy et al. investigates energy-efficient CP for multitasking embedded systems in [54]. However, none of the above approaches takes vulnerability into consideration. Cai et al. [14] is the first attempt to consider the effect of cache configurations for energy and vulnerability in time-constrained systems. Their approach only considers simple exploration of cache sizes.

In Chapter 3, we propose a vulnerability-aware DCR approach to explore performance, energy and vulnerability trade-offs in uniprocessor-based systems. In Chapter 5, we extend vulnerability-aware cache reconfiguration for multicore systems.

2.1.3 Partially Protected Caches for Vulnerability Reduction

Several microarchitectural techniques have been proposed to reduce the vulnerability of memory data due to soft errors. Error Correction Codes (ECC) is a popular technique used to detect the transient faults in memory and correct the corrupted data. The most common ECC uses Hamming codes that provide single bit error correction and double bit error detection (SEC-DED) [43]. Previous research shows that SEC-DED codes implementation can increase the cache access time by up to 95% [44] and power consumption by up to 22% [45]. While it might be possible to hide the performance overhead through hardware optimization, it is not possible to hide power/energy overhead.

The idea of Partially Protected Caches (PPC) initially comes from horizontally partitioned caches [40], where a processor has two or more caches at the same level of the memory hierarchy. Horizontally partitioned caches can help reduce cache pollution and thereby improve performance and/or energy consumption. Lee et al. [38, 39, 41] extends the idea of horizontally partitioned caches into the PPC architecture, by assuming that one of the two caches is protected from soft errors. The protected cache has redundancy logic like SEC-DED [43] for error protection, which has overhead in access time, area and power consumption [44, 45]. To align the access latency with the unprotected cache,

the protected cache is typically smaller than the unprotected cache. PPC is expensive in terms of energy consumption and performance, compared to the original architecture with only an unprotected cache. PPC works very well for multimedia applications where the partition between vulnerable data and multimedia data (not so vulnerable) is very clear. It is a challenge to make PPC reduce vulnerability for general applications, while incurring acceptable performance and energy penalty. In other words, designers would like to use PPC to reduce vulnerability but cannot afford significant increase in both execution time and energy consumption.

Figure 4-1 shows that the PPC architecture protects one cache from soft errors, while keeping the other cache unprotected. Each page from the memory is exclusively mapped into one of the two caches. Pages have a mapping attribute set by the compiler, which indicates the cache where a specific page should be mapped. When a new page is requested by the processor, the Translation Look-aside Buffer (TLB) will be checked to figure out which cache has the data. Therefore, only one cache will be accessed for each data access. Every time data is written into the protected cache, the data needs to be encoded, and it needs to be decoded when data is read from the cache. The challenge of using PPC is to properly partition data into two caches to ensure that it would not introduce too much penalty in performance and energy consumption. Lee et al. [38] used PPC to mitigate soft error failures for multimedia applications by selective data protection. They partition the data into failure non-critical (multimedia data) and failure critical, and map them into the unprotected and protected caches, respectively. Their approach works well for multimedia applications since the separation of multimedia data and other data is relatively easy. In [41], Lee et al. presented a data partitioning method which can work for general applications. It exhaustively searches for data pages to map them into the protected cache to reduce vulnerability, while not violating the 5% performance penalty constraint. In [39], another data partitioning method is proposed to search possible mapping schemes with a greedy approach. However, the existing

approaches use PPC only for the purpose of reducing vulnerability, and these approaches can introduce unacceptable energy overhead. In Chapter 4, we propose a DCR approach using PPC architecture to enable reduction in both energy consumption and cache vulnerability.

2.2 Trace Buffer Attack on AES

Trace buffer provides observability into the circuit so as to assist post-silicon debug and test. The quality of selected trace signals directly affects the observability that we can get from the circuit. The goal of trace signal selection is to obtain a set of signals, which can restore the maximum number of internal states in the chip. Basu et al. [76] proposed a metric based algorithm that employs total restorability for selecting the most profitable signals. Chatterjee et al. [77] proposed a simulation based algorithm which is shown to be more promising than metric based approaches. Li and Davoodi [78] proposed a hybrid approach which combines the advantages of metric and simulation based approaches. A simulation based approach using augmentation and ILP techniques by Rahmani et al. [79] demonstrated very high restoration capability and thus high observability of the internal signals. It is accepted in the research community that there is a strong link between observability/testability and security [83] for Design for Testability (DfT) facilities. Scan chain based DfT has been studied for attacks on block ciphers, including Data Encryption Standard (DES) [84] and Advanced Encryption Standard (AES) [85], and stream ciphers [86]. However, it is surprising that the vulnerability of trace buffers in cryptographic implementation has not been studied so far. This forms the core motivation of our work.

2.2.1 AES Specification

AES works on a block size of 128 bits and a key size of 128, 192 or 256 bits, which are referred to as AES-128, AES-192 and AES-256, respectively[2] . We briefly review AES-128 here, for further details readers can refer to [61].

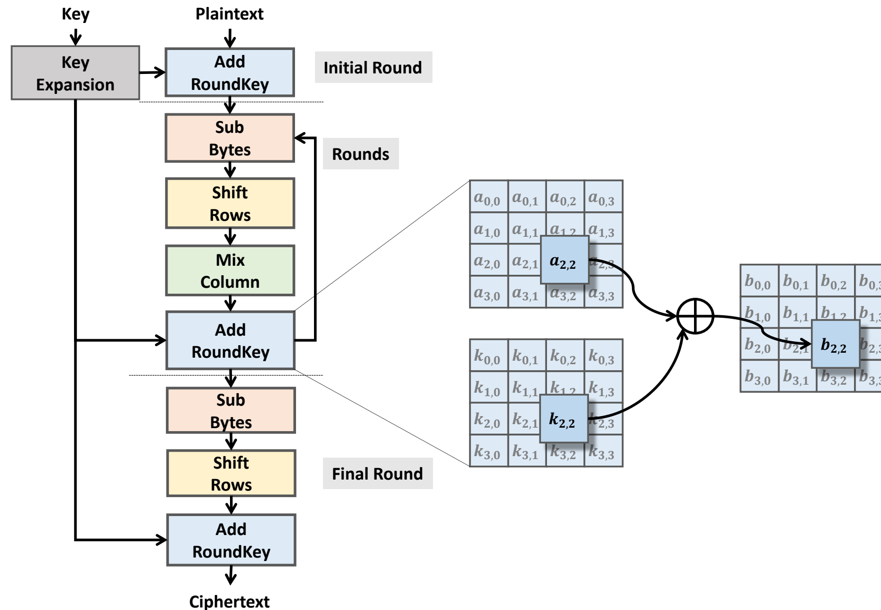


Figure 2-3. AES Encryption Flow

The encryption flow of AES is shown in the Figure 2-3. AES accepts a 128-bit plaintext, 128-bit user key and generates 128-bit ciphertext. The encryption proceeds through an initial round and subsequent 10 round repetition of 4 steps. These steps are *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. In the final round, MixColumns step is skipped. For each of these rounds, separate 128-bit round subkeys are needed. The round subkeys are generated from the initial user key via a key expansion step. The key expansion uses Rijndael's key schedule.

[2] For the rest of the chapter, unless explicitly specified, we will use AES-128 and AES interchangeably.

The plaintext is organized as a 4×4 column-major order matrix, which is operated through the AES rounds. The SubBytes step uses a non-linear transformation on every element of the matrix. The non-linear transformation is defined by an 8-bit substitution box, also known as Rijndael S-box. The ShiftRows step cyclically shifts the bytes in each row by a certain offset. In the MixColumns step, each column is multiplied by a fixed matrix. In the AddRoundKey step, each byte of the matrix is exclusive-OR-ed with each byte of the current round subkey. This is shown graphically in the Figure 2-3.

2.2.2 AES Attacks

Advanced Encryption Standard (AES) algorithm with various key lengths (128, 192 and 256) is widely used. The fact that AES stood the intense scrutiny from attackers over the last 15 years itself makes it an important benchmark for cryptography and cryptanalysis. So far, the best-known attempt against full AES-128, by *algebraic cryptanalysis*, has a computational complexity of $2^{126.1}$, which is slightly better than the brute-force attack and practically infeasible [59]. However, the perspective of *physical cryptanalysis* changes this scenario completely.

Since the pioneering works on differential power analysis [63], numerous side-channel attacks have been developed. Side-channel attacks can be classified into *passive*, *semi-invasive* and *invasive* attacks depending on the level of intrusion necessary for the attacker. The side-channels are of varied forms ranging from the software execution pattern such as cache timing [64] to more detailed hardware-oriented information leakages such as electromagnetic waves [65], acoustic waves [66] and optical fault injections [67]. Recent surveys on timing channels and invasive fault attacks are available in [68] and [69], respectively. Another approach of constructing an invasive attack originates from a malicious hardware, secretly inserted into a chip. These are commonly known as hardware Trojans [94, 95]. Such attacks have broken systems with mathematical security proof. In this scenario, secure implementation is rapidly becoming as important as the mathematical security proofs. For example, an AES implementation with protection against a first-order

side-channel attack is presented in [60]. The protected design is still vulnerable to more sophisticated attacks and even then, incurs 4.6× area- and 3.6× power-overhead, respectively, compared to the unprotected implementation.

Considering the impact that AES has on our everyday communications, many of the attack techniques report their efficacy by demonstrating an attack on AES, which is also the target cipher for the current work. Among the hardware side-channel attacks reported against AES, attacks based on scan-chain [72] and external fault injections [73] are most prominent. For all these attacks, effective countermeasures are proposed and the inherent resilience of various design points [74] is studied. It is also shown that there exists an interplay between the countermeasures of one attack and the consequently increased vulnerability against another attack [75].

In Chapter 6, I propose a novel and effective attack, termed *Trace Buffer Attack*. It identifies the trace buffer as a source of information leakage and shows that an effective security attack on AES is possible by analyzing the trace buffer content.

2.3 Hardware Trojan Attacks and Detection Techniques

In this section we briefly describe the growing threat of hardware Trojan attacks and discuss two broad classes of Trojan detection approaches.

2.3.1 Hardware Trojan Attacks

Malicious modification of IC at different stages of its life cycle, known as hardware Trojan attacks, is an impending threat in the electronics industry. Increased reliance on third party hardware intellectual property (IP) blocks and design automation tools in the IC design flow as well as outsourcing of design/fabrication steps to external parties due to economic reasons is rapidly increasing the vulnerability to Trojan attacks. An adversary can mount such an attack with an objective to cause in-field operational failure or to leak secret information from inside a chip - e.g. the key in a cryptographic IC. Recent investigations have shown that an intelligent adversary can insert tiny Trojans

of numerous forms and sizes into a million-transistor design, which can easily evade conventional manufacturing test that is not designed to isolate the stealthy Trojan attacks.

Depending on their mode of operation and structure, hardware Trojans can be grouped into several broad classes. A common classification of Trojans [96, 102] is based on the activation mechanism (referred as *Trojan trigger*) and the effect on the circuit functionality (referred as *Trojan payload*). As shown in Figure 1-5, Trojans can be both combinationally and sequentially triggered. Typically, an adversary would choose an extremely rare activation condition so that it is highly unlikely for the Trojan to trigger during conventional manufacturing test. *Sequentially triggered* Trojans (the so-called "time bombs"), on the other hand, are activated by the occurrence of a sequence, or after a period of continuous operation. The simplest sequential Trojans are synchronous stand-alone counters, which trigger a malfunction on reaching a particular count. The trigger mechanism can also be *analog* in nature, whereby on-chip sensors are used to trigger a malfunction. For example, the Trojan gets activated when the temperature of a particular region of the IC exceeds a threshold [96]. Trojans can also be classified based on their *payload* mechanisms into two main classes - *digital* and *analog*. *Digital payload* Trojans can either affect the logic values at chosen internal payload nodes, or can modify the contents of memory locations. *Analog payload* Trojans, on the other hand, affect circuit parameters such as performance, power and noise margin.

2.3.2 Trojan Detection Approaches

Detecting hardware Trojan instances in an IC before it is used in an electronic system is of paramount importance. Even though design-for-security (DfS) approaches that aim at hardening a design with respect to Trojan insertion or facilitating Trojan detection during manufacturing test are being actively researched [98], they have several major limitations: (1) they cannot provide provably robust defense against all forms of Trojan attacks; (2) they often incur unacceptable design overhead; and (3) they cannot be applied to legacy designs, which is difficult to change for incorporating DfS features. Hence, a Trojan

detection step for trust validation during post-silicon manufacturing test is becoming crucial to isolate ICs affected with Trojans.

It is worth noting that conventional post-manufacturing test using functional / structural test patterns performs poorly to reliably detect hardware Trojans. This is because manufacturing test generation and application aim at detecting manufacturing defects with well-characterized behavior and model that cause deviation from functional or parametric specifications. They do not aim at detecting additional functionalities incorporated by a Trojan or deviation in circuit behavior triggered by rare events. Hence, conventional testing methods typically provide poor Trojan detection capability, as observed by recent researches [100]. Destructive testing of a chip by de-packaging, de-metallization and micro-photography based reverse-engineering is highly expensive (in time and cost) and not a feasible solution because an attacker may selectively insert Trojan into a small subset of the manufactured ICs [103].

Existing Trojan detection approaches fall into two major classes: (a)*functional testing* based, and (b) *side-channel analysis* based. Most Trojan detection techniques proposed in the literature are characterized by their efficiency in detecting particular classes of Trojan. These approaches typically fail to provide high confidence in detecting generic Trojan of arbitrary operating mode. The enormous variety of Trojans and the inordinately large Trojan population that might be present in a circuit makes it difficult to devise deterministic test patterns for them. The *functional testing* based Trojan detection approaches [100] aim to trigger rare events at internal nodes in the circuit to activate Trojans and then compare the obtained output logic values of the circuit with the expected golden values of the IC. On the other hand, the *side-channel analysis* based Trojan detection approaches [104][107][114] are based on observing the effect that an inserted Trojan is expected to have on a physical parameter such as circuit transient current, power consumption or path delay. If the observed value of the measured

parameter differs by more than a threshold from the golden value, the presence of a Trojan is suspected.

Both classes of Trojan detection techniques have their relative merits and demerits. The main challenge for functional testing based Trojan detection approaches is the enormously large *Trojan design space*, which makes complete enumeration and test generation for all feasible Trojan instances in a moderately-sized circuit computationally infeasible. This makes it extremely difficult to guarantee that an arbitrary Trojan would be activated, cause circuit malfunction and thus get detected during the test application phase. On the other hand, the advantage of the side-channel analysis based approaches lies in the fact that even if the Trojan circuit does not cause observable malfunction in the circuit during test, the presence of the extra circuitry can be reflected in the measured side-channel parameter. Also, such techniques are applicable for arbitrarily complex Trojans because they do not need to make any assumption about the mode of operation of an inserted Trojan. However, the main challenges associated with side-channel analysis are large *process variation* and *design marginality* induced effects in modern nanometer technologies [96], and measurement noise, which can mask the effect of an inserted Trojan circuit, especially for small Trojans.

One promising direction to overcome process variation is to generate functional test patterns that are likely to activate the Trojans. These approaches rely on the fact that an adversary will choose a trigger condition for the Trojan using a set of rare nodes. Various approaches tried to maximize the rare node activation to increase the likelihood of activating Trojans. Some approaches [113][114] use the Design-for-Test (DFT) infrastructure (such as additional scan flip-flop) to increase the transition probability of low-transition nets. MERO [100] takes the advantage of N-detect test [115] to maximize the trigger coverage by activating the rare nodes. The test generation ensures that each of the nodes gets activated to their rare values for at least N times. They have shown that if N is sufficiently large, a Trojan with trigger conditions from

these rare nodes, will be highly likely to be activated by the generated test set. Saha et al. [101] improves the test pattern generation of MERO by using genetic algorithm and Boolean satisfiability, which could more effectively propagate the payload of possible Trojan candidates. However, these functional test generation approaches are not designed for side-channel analysis. Direct application of these test generation approaches for side-channel analysis would not be best for improving side-channel sensitivity for Trojan detection. The objective of increasing side-channel sensitivity is very different from the ones in both MERO as well as in subsequent improvement by Saha et al. [101]. Unlike these existing approaches, we propose a side-channel aware test generation approach in Chapter 7, which can maximize switching activity in an unknown Trojan circuit while minimizing the background switching.

CHAPTER 3
VULNERABILITY AND ENERGY-AWARE CACHE RECONFIGURATION

In this chapter, we propose a methodology for using cache reconfiguration in soft real-time systems. The proposed approach provides an efficient cache tuning strategy based on static profiling and dynamic scheduling of tasks. It explores Vulnerability-aware Energy Optimization (VAEO) opportunity within each task (*intra-task VAEO*) as well as across task sets (*inter-task VAEO*). While traditional approaches (no DCR) uses a fixed cache for all tasks in the system, the inter-task DCR will select (use) the most beneficial cache configuration for each task to improve both vulnerability and energy-efficiency. Intra-task DCR will extend the optimization opportunity further by enabling changes in cache configuration within each task depending on task phases (application requirements). The proposed research is able to balance performance, energy consumption and vulnerability, so that tasks can meet their deadlines and produce energy savings while vulnerability reduction can also be achieved. The configurable cache architecture used in our work is similar to the one in [11]. It contains four cache banks operating as four separate ways. The cache ways can be configured to shut down so as to vary the cache size. The way associativity can be changed by concatenating ways. The line size can be adjusted by configuring the fetch unit to different lengths. This architecture requires very simple hardware augmentation and minor overhead [11]. A light process can be used as the cache tuner, which will make the reconfiguration decision and change the configuration at runtime. There are many prior efforts in developing energy- and performance-aware cache reconfiguration techniques.

The rest of the chapter is organized as follows. Section 3.1 motivates the reader by illustrating the effect of DCR on performance, energy consumption and vulnerability. Section 3.2 presents our cache reconfiguration methodology for inter-task VAEO. Section 3.3 presents intra-task VAEO, which includes phase identification and cache

configuration assignment for phases. Section 3.4 presents the experimental results. Finally, Section 3.5 concludes the chapter.

3.1 Motivation: Illustrative Example

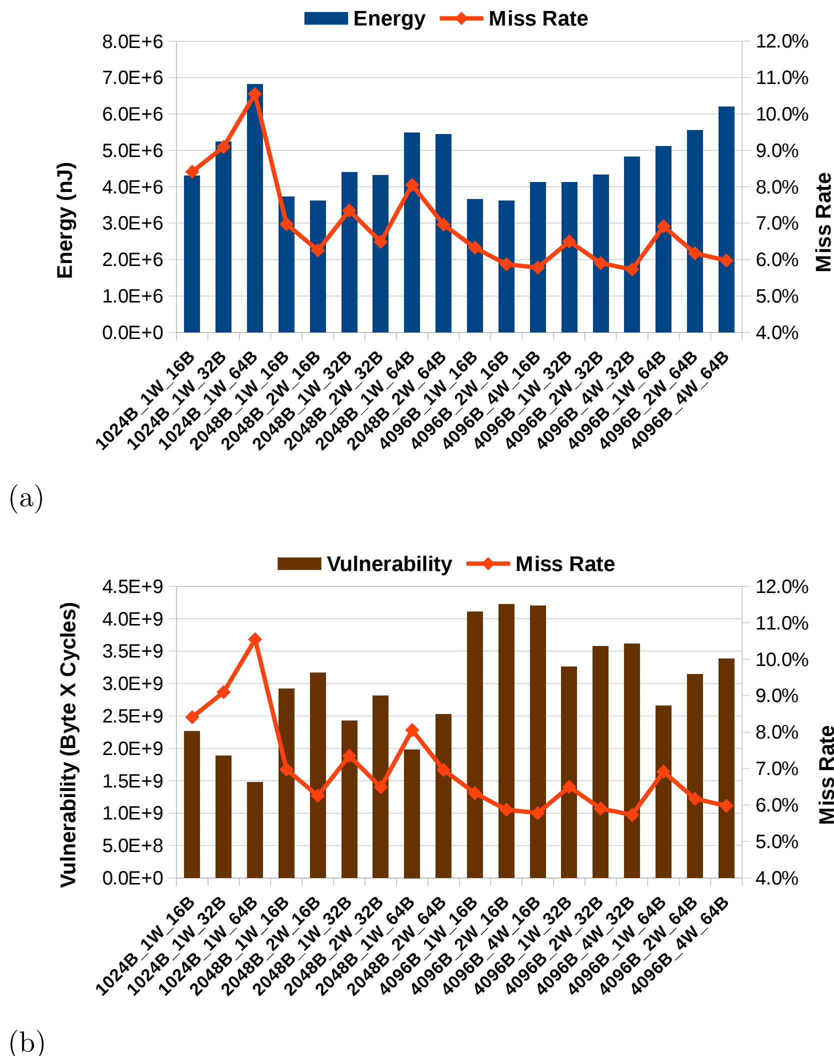


Figure 3-1. Energy and vulnerability values of *pegwit* benchmark using different cache configurations. (a) Energy and miss rate, (b) Vulnerability and miss rate.

Existing techniques for cache reconfiguration do not consider cache vulnerability due to soft errors. Figure 3-1 illustrates the interesting behaviors of vulnerability and energy consumption under different cache configurations. We run the program *pegwit* (a benchmark from MediaBench [22]) for 18 times, and each run uses a different configuration for L1 data cache. Each configuration consists of three parameters: cache

size, associativity and line size. For example, 1024B_1W_64B implies a cache configuration with cache size of 1024 bytes, one way with 64 bytes line size.

Figure 3-1 shows that the energy consumption, vulnerability and miss rate change drastically as we tune cache configurations. Both energy and vulnerability relate to cache miss rates and cache configurations. However, the correlation behaviors are quite different and even conflicting in certain scenarios. In Figure 3-1(a), energy consumption decreases when miss rate decreases (the first 9 cache configurations), but keeps increasing for the last 9 cache configurations even though miss rates are fairly low. The reason is that the total energy consumption is the sum of dynamic and static energy. For the first 9 cache configurations, the total energy is dominated by dynamic energy consumption, thus the total energy decreases when miss rate (dynamic energy consumption) decreases. However, for the last 9 cache configurations with large cache size, the total energy is dominated by static energy consumption even though miss rates are low. In Figure 3-1(b), the relation between vulnerability and miss rate is a little more complex. Cache size has a significant influence on vulnerability. Configurations with cache size of 1024B is much less vulnerable than configurations with cache size of 2048B and 4096B. For configurations with the same cache size, vulnerability decreases when miss rate increases and vice versa. For the same cache size, lower miss rate means that more dirty data is staying in cache for longer time, which contributes to vulnerability.

There are two interesting observations here: (i) small cache size might have high energy consumption but less vulnerable; (ii) low miss rate might be energy friendly but leads to higher vulnerability. These observations motivate us to investigate the trade-off between vulnerability, energy and performance during DCR. In this chapter, we develop a cache reconfiguration framework that considers both energy and cache vulnerability. Since both vulnerability and energy depend on program characteristics and cache configurations, we statically analyze various cache configurations for each application. Such an approach is suitable for embedded systems since applications are known a priori. Based on static

analysis, we propose inter-task as well as intra-task dynamic cache tuning that can select suitable configurations during runtime.

3.2 Inter-task Cache Reconfiguration

3.2.1 System Model

Let us define the reliability-aware DCR problem with consideration of both energy and cache vulnerability. The system we consider can be modeled as:

- A processor with a reconfigurable cache which supports m possible cache configurations $C = \{c_1, c_2, c_3, ..., c_m\}$.
- A set of n independent tasks $T = \{t_1, t_2, t_3, ..., t_n\}$.
- Each task $t_i \in T$ has attributes including arrival time, period and deadline. Non-preemptive execution is employed, which means, a task will continue execution until completion once it starts to execute.

Let $e_{t_i}^{c_j}$, $p_{t_i}^{c_j}$ and $v_{t_i}^{c_j}$ denote the energy, execution time (performance) and vulnerability of task t_i when it is run on cache configuration c_j. The reliability-aware DCR problem is to find a cache assignment for the task set such that energy consumption and vulnerability are minimized with each of the tasks satisfying its deadline. One common practice for dealing with multi-objective optimization problem is to optimize one objective at a time while transforming other objectives into constraints. We introduce the *Vulnerability-aware Energy Optimization* (**VAEO**) problem, which aims at minimizing the total energy consumption, while adding vulnerability of tasks as constraints. A heuristic algorithm based on run-time task scheduling is proposed for solving the VAEO problem.

3.2.2 Vulnerability-Aware Energy Optimization (VAEO)

Let n represent the total number of task arrivals within the least common multiple (hyper-period[1]) of all task periods. $\sum_{i=1}^{n} e_{t_i}^{c_j}$ is the total energy consumption of n tasks[2] . The VAEO DCR problem can be defined as the following:

$$\text{minimize} \sum_{i=1}^{n} e_{t_i}^{c_j} \tag{3–1}$$

subject to

$$v_{t_i}^{c_j} \leq V_{t_i}, \ \forall i \in [1, n] \tag{3–2}$$

$$a_{t_i} + w_{t_i} + p_{t_i}^{c_j} \leq D_{t_i}, \ \forall i \in [1, n] \tag{3–3}$$

Equation 3–1 is the optimization objective. Equation 3–2 and 3–3 contain the vulnerability and timing constraints. V_{t_i} is the upper bound for vulnerability of task t_i. Here a_{t_i}, w_{t_i}, $p_{t_i}^{c_j}$, D_{t_i} denote the arrival time, queuing time, execution time, and deadline of task t_i. The optimization goal is to find a set of cache configuration assignments for all tasks so that the total energy consumption is minimized with vulnerability and timing constraints. We choose V_{t_i} as the vulnerability of task t_i when it is executed with the *base cache*, the most profitable cache configuration decided during design time. In other words, we set the vulnerability as a constraint to ensure that it is always at least as reliable as the *base cache*.

In Equation 3–3, arrival time a_{t_i} and deadline D_{t_i} are known upon the arrival of the task, while queuing time w_{t_i} and execution time $p_{t_i}^{c_j}$ depend on the scheduling and cache

---

[1] A hyper-period is the Least Common Multiple (LCM) of all the periods in the task set. The basic idea of using hyper-period is that once we find a profitable (for energy or vulnerability) schedule for one hyper-period, the exactly same schedule can be applied to subsequent hyper-periods.

[2] It will be precise to call n as the total number of “jobs” as in real-time system terminology. However, for ease of discussion, we do not distinguish between tasks and jobs.

reconfiguration algorithms. Queuing time w_{t_i} depends on the scheduler and is determined by the priority of this task and the other tasks currently in the queue. Execution time $p_{t_i}^{c_j}$ is determined by the cache configuration c_j which will be assigned to this task by the cache reconfiguration algorithm.

3.2.3 Heuristic Approach for VAEO Problem

Tasks arrive periodically and each task is inserted into a list of ready tasks upon arrival. We propose a heuristic approach, which employs Earliest Deadline First (EDF) as our underlying scheduling algorithm. EDF fetches the task with the highest priority (earliest deadline) to execute. The cache configuration selection algorithm will pick a configuration for this task and try to satisfy Equation 3–2 and 3–3 if possible. Our heuristic approach chooses between the VAEO cache configuration and performance optimal (PO) cache configuration for this task.

- ***VAEO cache configuration*** of a task is the configuration which satisfies Equation 3–2 and consumes the least energy among all possible configurations.
- ***PO cache configuration*** of a task is the configuration which has the shortest execution time, but PO configuration might not satisfy Equation 3–2.

The intuition behind our approach of choosing between PO and VAEO configuration are as follows:

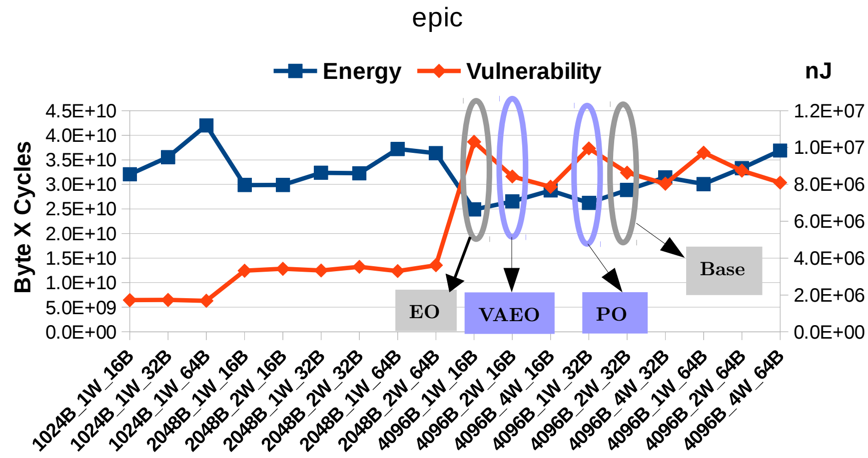
(1) The VAEO configuration satisfies the vulnerability constraint in Equation 3–2 and it is most beneficial for energy savings, although it might have long execution time. We would like to always choose the VAEO configuration for energy optimization, as long as this choice would not cause the current task or any of the subsequent tasks to violate their deadlines.

(2) The PO configuration is aimed on Equation 3–3 for satisfying timing constraints. If the VAEO configuration of a task causes deadline violations, we would conservatively choose the PO configuration instead. With this task running under the PO configuration, the subsequent tasks will have more slack time for scheduling and possibly save energy.

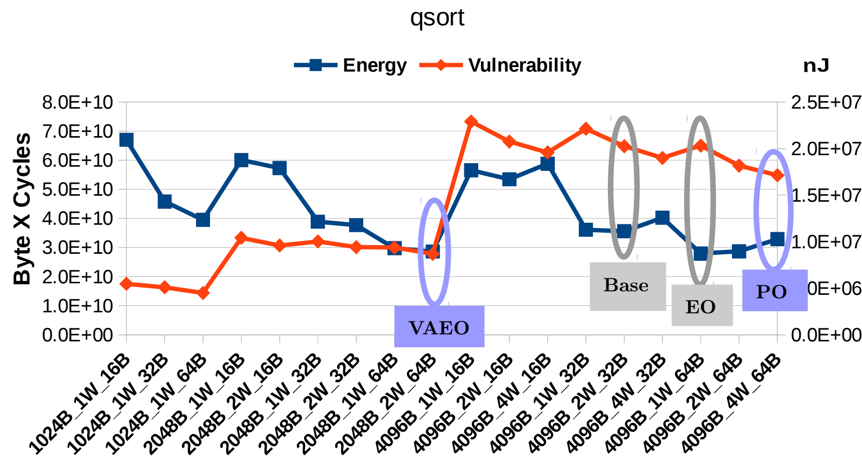
Algorithm 1: Inter-task Cache Reconfiguration

1: **Input**: List of ready tasks (LRT) and task profile table.
2: **Output**: VAEO or PO cache configuration.
3: **Step 1**: Sort all tasks in LRT by priority and fetch the task t_c with highest priority.
4: **Step 2**: t_1 to t_m are tasks left in LRT, from highest to lowest priority. τ represents the current time.
5: /** check the schedulability of each task in LRT **/
6: **for** $j = 1$ *to* m **do**
7:     **if** $\tau + p_{t_c}^{PO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$ **then**
8:         Discard task t_j
9:     **end**
10: **end**
11: **Step 3**: Select cache configuration for current task t_c. Let m' be the number of tasks in LRT left after Step 2.
12: /** test the feasibility of using VAEO config for t_c **/
13: **if** $\tau + p_{t_c}^{VAEO} > D_{t_c}$ **then**
14:     $OK_{VAEO} = false$;
15: **end**
16: **else**
17:     $OK_{VAEO} = true$;
18:     **for** $j = 1$ *to* m' **do**
19:         **if** $\tau + p_{t_c}^{VAEO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$ **then**
20:             $OK_{VAEO} = false$;
21:         **end**
22:     **end**
23: **end**
24: **if** $OK_{VAEO} == true$ **then**
25:     **return** VAEO configuration for task t_c
26: **end**
27: **else**
28:     **return** PO configuration for task t_c
29: **end**

Figure 3-2 shows the VAEO and PO configurations for benchmarks *epic* and *qsort*. The *base cache* configuration is 4096B_2W_32B. For *epic*, the PO configuration (4096B_1W_32B), determined by runtime (which is not shown in this figure), has worse vulnerability than Base. The VAEO configuration (4096B_2W_16B) has the minimum energy consumption, among all candidates which has smaller vulnerability than Base. Cache sizes of 1K and 2K (the first nine configurations) are candidates



(a) epic



(b) qsort

Figure 3-2. VAEO and PO configurations of two benchmarks: (a) *epic* and (b) *qsort*.

with very small vulnerability, however, they are not chosen because of large energy consumption. The configuration with minimum energy consumption (4096B_1W_16B, marked as EO in Figure 3-2a) is not chosen as VAEO, because its vulnerability is higher than the Base. For *qsort*, the VAEO configuration (2048B_2W_64B) finds a sweet spot which has low vulnerability and energy footprint. It is not the one with the minimum energy consumption (EO), while it has much lower vulnerability than Base and PO configurations. VAEO configuration has cache size 2K and line size 64B, which indicates that the data of the program can fit into 2K cache and data is accessed in large chunks.

Algorithm 1 illustrates the runtime cache selection procedure for VAEO approach. Let us assume that our system uses non-preemptive EDF scheduling for the task set. Tasks arrive periodically and currently available tasks will be put into the list of ready tasks (LRT), which is maintained as a priority queue based on the deadlines of tasks. Algorithm 1 is called when the processor is ready to execute a new task. The term $p_{t_i}^{PO}$ stands for the execution time of task t_i using its PO configuration, and $p_{t_i}^{VAEO}$ stands for the execution time using its VAEO configuration.

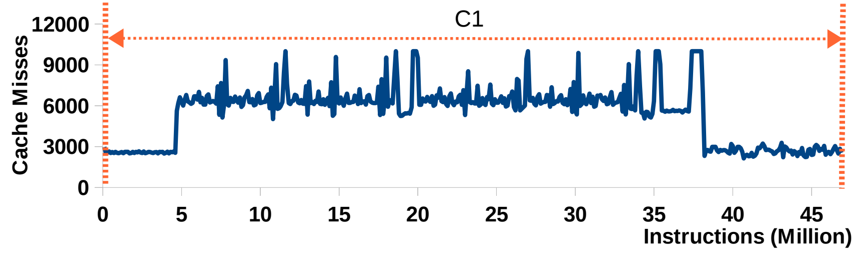
Step 1 fetches the current task t_c to be executed, which is the highest priority task from LRT. Step 2 checks the schedulability of the tasks left in LRT, when the current task t_c is executed with PO cache configuration. The schedulability of each task t_j left in the LRT is checked by $\tau + p_{t_c}^{PO} + \sum_{i=1}^{j} p_{t_i}^{PO} > D_{t_j}$, which tests whether its deadline can be met with the assumption that all preceding tasks (and itself) use PO cache configurations. If t_j cannot satisfy its deadline even with this conservative assumption, t_j should be discarded. The discarding process is done from highest priority to lowest priority, so as to achieve fewest discarded tasks. This step ensures that all tasks in LRT will satisfy their deadlines with their PO configurations, when the current task t_c is executed with its PO configuration. This step will be skipped if LRT is empty. In Step 3, we try to test the feasibility of using its VAEO configuration for the current task t_c, which will help improve vulnerability and energy consumption. The appropriate cache configuration for the current task t_c is selected by checking whether it is safe to use its VAEO configuration. VAEO configuration is safe, only if no tasks in the LRT will fail to meet their deadlines with their PO configurations. If the VAEO configuration is not safe for t_c, we will conservatively execute the current task t_c with its PO configuration, which can ensure all tasks left in the LRT to satisfy their deadlines with their PO configurations (otherwise they would have already been discarded in Step 2). This algorithm has time complexity of $O(m \log m)$ where m is the total number of tasks in LRT, since Step 1 takes $O(m \log m)$ time, Step 2 takes $O(m)$ time and Step 3 takes $O(1)$ time.

Note that Algorithm 1 needs to take the profile information (the performance numbers for PO and VAEO configurations) for all the tasks. The profiling can be done off-line for one specific input pattern for a program. In this work, we assume that the input size (for example, the image size, the compression ratio for *jpeg*) remains the same but input patterns (for example, contents, image format) can vary. This is a reasonable assumption for real-time embedded systems. We performed our offline analysis by varying input patterns (data values) for all the benchmarks and observed that it has a minor impact on the footprint of data access. Since the profile of vulnerability and energy estimation for data pages depends on the data access pattern, our static profiling will still remain effective for different input patterns during runtime. Our observations are consistent with the ones made by existing literature [11].

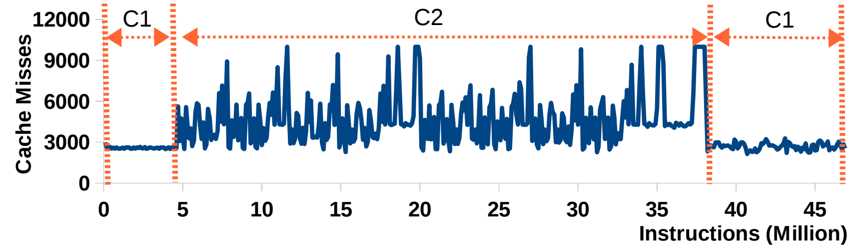
3.3 Intra-task Cache Reconfiguration

The heuristic approach described in Section 3.2 provides solution for the inter-task cache reconfiguration of a task set, which will choose between the PO and VAEO configurations for each task. In this section we find that it is even more beneficial if we can reconfigure inside the task itself (i.e. intra-task reconfiguration). In other words, both intra-task and inter-task can be used simultaneously for improving vulnerability and energy efficiency. A task can have considerably different behaviour depending on which portion of execution is examined.

Figure 3-3(a) shows the data cache misses, when benchmark *qsort* is executed with a fixed cache C1 (1024B_1W_32B). We can observe three program phases based on the number of cache misses per sampling point (100K instructions). The first and third phases (0 $\sim$ 5 million and 38 $\sim$ 47 million) have fewer than 3000 misses per sampling point, while the second phase has more than 6000 misses per sampling point. Based on this observation, a large cache C2 (4096B_2W_32B) is selected for the second phase in Figure 3-3(b), which greatly reduces the cache misses. This example shows that



(a) One phase for the whole task using a fixed cache.



(b) Three intra-task phases using different caches

Figure 3-3. Data cache misses for benchmark *qsort*. (a) Without intra-task reconfiguration, using one cache for the whole task; (b) With intra-task reconfiguration, using a different cache for each of the three phases.

intra-task reconfiguration can reduce cache misses. Our ultimate goal of using intra-task reconfiguration is to further optimize energy and/or vulnerability of the task.

A phase of task (program) can be defined as an interval of execution during which a measured program metric is relatively stable [26], [27], [28]. Intra-task cache reconfiguration aims at finding the sequence of cache assignments to different phases of the task, so that cache misses (energy and/or vulnerability) of this task is further optimized. In order to improve energy consumption and vulnerability, we need to properly define the phases and carefully select the configuration for each phase. Our approach for intra-task reconfiguration is to switch to the most beneficial cache configuration when the task enters a different phase during its execution. At the beginning of a new phase, we choose a configuration based on the characteristics (cache requirement) of the new phase. The cache will be flushed if we decide that the configuration is to be changed for the new phase. The flushing of cache will result in additional cache misses, which cause penalty in performance and energy consumption. However, the flushing of cache is beneficial to

reduce vulnerability, because vulnerable data in cache are prematurely written back to the memory. A beneficial configuration would be one which can save energy and reduce vulnerability, in spite of the additional misses at the very beginning of the new phase.

The problem of intra-task cache reconfiguration boils down to solving the following two problems: (1) how to monitor the execution of the task and define phase partitions; (2) how to decide the cache configuration for each phase. The following sections address these challenges.

3.3.1 Phase Extraction

We introduce our approach for phase extraction in Algorithm 2. We get the cache miss statistics of all intervals for a fixed cache (1024B_1W_32B). We choose such a relatively small cache, because it is easier to identify phases. The algorithm works in two steps: (1) identify potential phase boundaries, and (2) post-process to select profitable phases. Firstly, an interval is marked as a potential starting point of a new phase if the change in number of cache misses exceeds the threshold that we have set (line 5-7). Secondly, each of the potential phases is examined and the ones which mark a relatively stable execution (i.e. longer than the minimum threshold) are kept (line 10-12). A phase will be merged with the previous phase if it lasts no longer than *PhaseLength*. For each benchmark, it is divided into 100 sampling intervals with equal number of instructions. Each interval is profiled with the number of cache misses by simulation using a configuration of 1024B_1W_32B. We used the threshold for change of cache misses (*MissFactor*) as 2, and the threshold for minimum phase length (*PhaseLength*) as 5 intervals.

Figure 3-4 shows the phases identified for 6 benchmarks from the MediaBench [22] and EEMBC [23] automotive benchmark suits. We can observe that benchmarks have very different patterns of data cache misses during execution. We identify 3 phases in epic, 2 phases in dijkstra, 2 phases in cjpeg, 2 phases in BITMNP01, and 4 phases in AIFFTR01, while pegwit has no obvious phases.

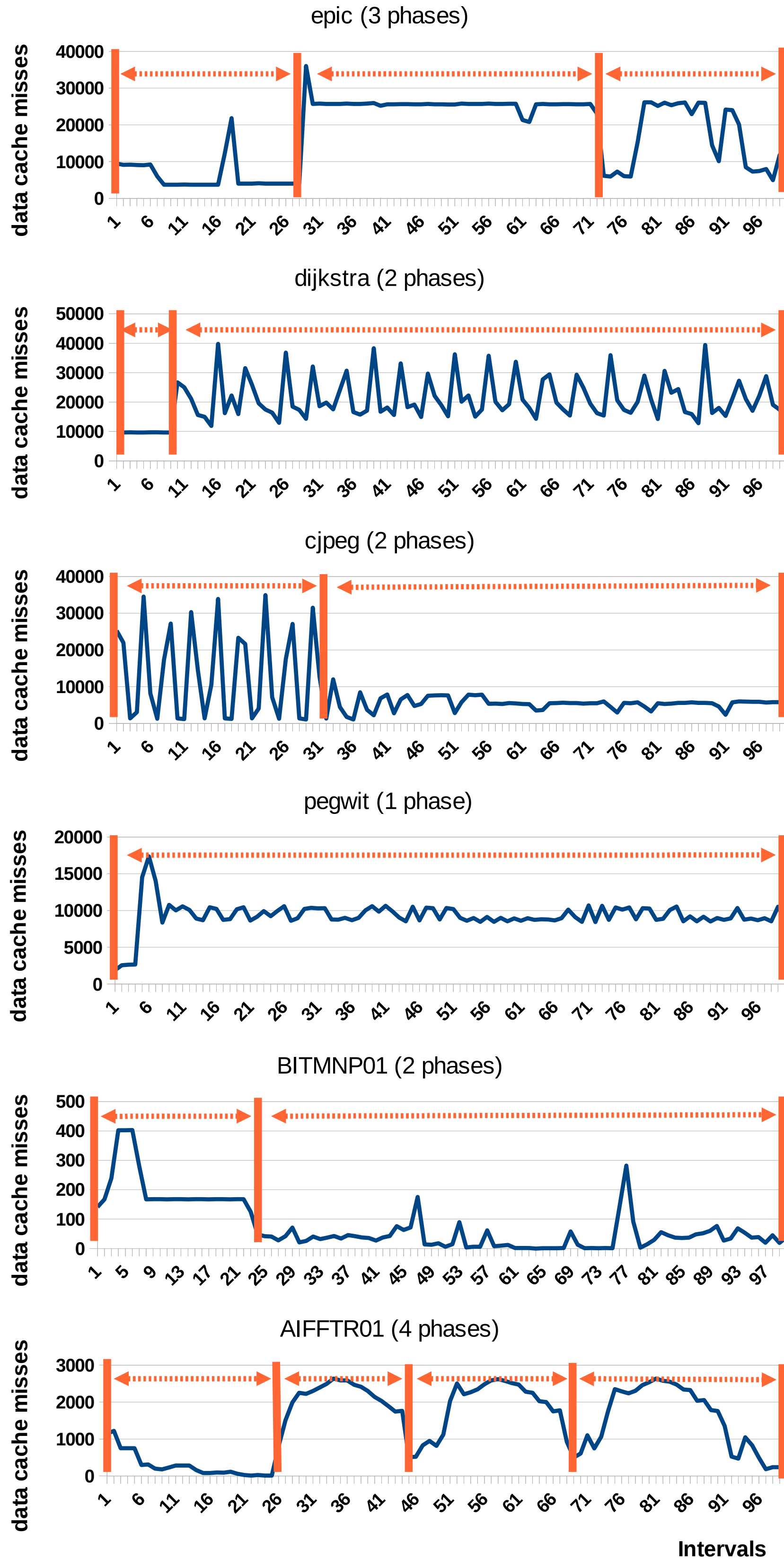


Figure 3-4. Phases identified for different benchmarks.

Algorithm 2: Phase Identification

1: **Input**: Benchmark, threshold $MissFactor$ (changes of cache misses), threshold $PhaseLength$
2: **Output**: Identified phases.
3: Get the cache misses (CM_i) statistics of all intervals
4: **for** *each interval* i **do**
5:     **if** $CM_i < CM_{i-1}/MissFactor$ *or* $CM_i > CM_{i-1} * MissFactor$ **then**
6:         Set interval i as a potential phase boundary
7:     **end**
8: **end**
9: **for** *each potential phase* f_j **do**
10:     **if** $length(f_j) < PhaseLength$ **then**
11:         Merge this phase with previous phase
12:     **end**
13: **end**

3.3.2 Cache Assignment for Phases

In this section, we show the cache assignment for phases of a task for the VAEO problem. We want to minimize the total energy consumption of all phases, with the total vulnerability constrained. The best VAEO cache configuration for m phases (f_1 to f_m) can be defined as:

$$\text{minimize} \sum_{i=1}^{m} e_{f_i}^{c_j} \tag{3–4}$$

$$\text{subject to} \sum_{i=1}^{m} v_{f_i}^{c_j} \leq V \tag{3–5}$$

$e_{f_i}^{c_j}$ and $v_{f_i}^{c_j}$ are the energy consumption and vulnerability of phase f_i using config c_j. V is the threshold for vulnerability, which is the vulnerability of the task when it is executed with the base cache.

For each of the m phases, there are n possible configurations. The time complexity for a brute-force exploration of all possible combinations is $O(n^m)$. We observe that this is essentially a dynamic programming problem, where each phase is dependent on the previous phase. If the current phase chooses a different configuration from the previous phase, the cache needs to be flushed before running the current phase. However, if the

chosen configuration for the current and previous phases are the same, the cache is not flushed and will keep the data. We define the *dynamic programming* problem as follows.

$$E^{c_j}_{(f_1\sim f_i)} = \min_{k\in(1..n)}\{E^{c_k}_{(f_1\sim f_{i-1})} + e^{c_j}_{f_i}\} \tag{3–6}$$

$$\text{where} \quad i > 1, \quad 1 \le j \le n,$$

$$\text{and } V^{c_k}_{(f_1\sim f_{i-1})} + v^{c_j}_{f_i} \le V_{(f_1\sim f_i)}$$

$$\text{with the initial states:} \quad E^{c_j}_{(f_1\sim f_1)} = \begin{cases} e^{c_j}_{f_1}, & v^{c_j}_{f_1} \le V_{f_1} \\ \infty, & \text{otherwise} \end{cases} \tag{3–7}$$

$E^{c_j}_{(f_1\sim f_i)}$ is the minimum total energy consumption for the first i phases ($f_1 \sim f_i$), assuming that the current phase f_i chooses c_j. Equation 3–6 shows the formula to get the current minimum energy consumption, based on the previous iteration step. $V_{(f_1\sim f_i)}$ is the threshold for vulnerability of the first i phases, which is the vulnerability when the task is run with base cache. Equation 3–7 shows the initial states for our dynamic programming.

If we remove the constraints for vulnerability (or set the vulnerability constraints as very large), the DP algorithm can always produce the optimal solution. We can use induction to prove that $E^{c_j}_{(f_1\sim f_i)} = \min_{k\in(1..n)}\{E^{c_k}_{(f_1\sim f_{i-1})} + e^{c_j}_{f_i}\}$ will always find the optimal cache assignments for all phases.

Step 1: Let $i = 1$, $E^{c_j}_{(f_1\sim f_1)} = e^{c_j}_{f_1}$ for all $1 \le j \le n$. This is our base case for induction. This is the minimal energy for phase f_1 using configuration c_j, since there is no previous phase and it is the only possibility.

Step 2: Assume $i = x$, $E^{c_j}_{(f_1\sim f_x)}$ for all $1 \le j \le n$ are the minimal energy solutions for the previous x phases ($f_1 \sim f_x$).

Let $i = x + 1$, $E^{c_j}_{(f_1\sim f_{x+1})} = \min_{k\in(1..n)}\{E^{c_k}_{(f_1\sim f_x)} + e^{c_j}_{f_{x+1}}\}$ provides the energy for phases f_1 to f_{x+1}, which combines the optimal solutions from previous step for phases f_1 to f_x. This is the minimal energy that can be achieved for phases f_1 to f_{x+1}.

When the vulnerability threshold is small, the dynamic programming is not guaranteed to find the optimal solution since the partial solutions in the earlier stages may not be useful if the final vulnerability number crosses the threshold. In this scenario, exhaustive exploration is needed to find the optimal result. It is important to note that, the exhaustive exploration is feasible since the number of phases (m) are small, typically less than 10, with $O(n^m)$ complexity.

Algorithm 3 is an iterative implementation of our cache assignment approach for the phases. We use two arrays to store the energy and vulnerability values ($E[m][n]$ and $V[m][n]$), where m is the number of phases, and n is the number of cache configurations. In line 4-10, we initialize the states of phase f_1, as directed by Equation 3–7. For each configuration c_j, its energy value will be updated only if its vulnerability is smaller than V_{f_1}. In line 12-25, we evaluate the states of phase f_2 to f_m, as outlined by Equation 3–6. In each iteration (phase f_i), we update the optimal energy value ($E[i][j]$, which is $E^{c_j}_{(f_1 \sim f_i)}$ in Equation 3–6) for each configuration (c_j). This is done in line 15-23, which compares all solutions ($E[i-1][k]$ and $V[i-1][k]$) found at the previous iteration for phases $f_1 \sim f_{i-1}$. In the process of comparing previous solutions (line 18), we also ensure that the vulnerability constraints are not violated (line 17). In line 27-30, we iterate through feasible solutions at the last phase f_m, and find the optimal solution with the minimum energy. The initialization process of line 4-10 is of complexity of $O(n)$. The dynamic programming process of line 12-25 has complexity of $O(mn^2)$. The final iteration for output of line 27-30 has complexity of $O(n)$. Thus, the algorithm has an overall time complexity of $O(mn^2)$. This algorithm can be completed in reasonable time since m is typically less than 10 and n is 18 in our framework.

3.3.3 Inter+Intra Cache Reconfiguration

Up to this point, we have introduced both inter-task DCR and intra-task DCR. The inter-task DCR approach optimizes at the task level, where each task is deemed as an atom since our system is non-preemptive. The intra-task DCR approach optimizes

Algorithm 3: Cache Assignment for Intra-task Phases

```
1: Initialize the energy array E[m][n] = {∞, ..., ∞}
2: Initialize the vulnerability array V[m][n] = {∞, ..., ∞}
3: /** Phase 1 **/
4: for config c_j=c_1 to c_n do
5:     Get e_{f_1}^{c_j} and v_{f_1}^{c_j} by running phase f_1 with config c_j
6:     if v_{f_1}^{c_j} ≤ V_{f_1} then
7:         E[1][j] = e_{f_1}^{c_j}
8:         V[1][j] = v_{f_1}^{c_j}
9:     end
10: end
11: /** Phase 2 to Phase m **/
12: for phase f_i=f_2 to f_m do
13:     for config c_j=c_1 to c_n do
14:         E[i][j] = ∞
15:         for config c_k=c_1 to c_n do
16:             Get e_{f_i}^{c_j} and v_{f_i}^{c_j} by running f_i with config c_j
17:             if V[i−1][k] + v_{f_i}^{c_j} ≤ V_{f_i} then
18:                 if E[i−1][k] + e_{f_i}^{c_j} < E[i][j] then
19:                     E[i][j] = E[i−1][k] + e_{f_i}^{c_j}
20:                     V[i][j] = V[i−1][k] + v_{f_i}^{c_j}
21:                 end
22:             end
23:         end
24:     end
25: end
26: /** Find the optimal solution with minimum energy **/
27: for config c_j=c_1 to c_n do
28:     E_min = min(E[m][j], E_min);
29:     V = V[m][j];
30: end
31: The path leading to E_min is the VAEO solution
32: return the VAEO config vector for all phases
```

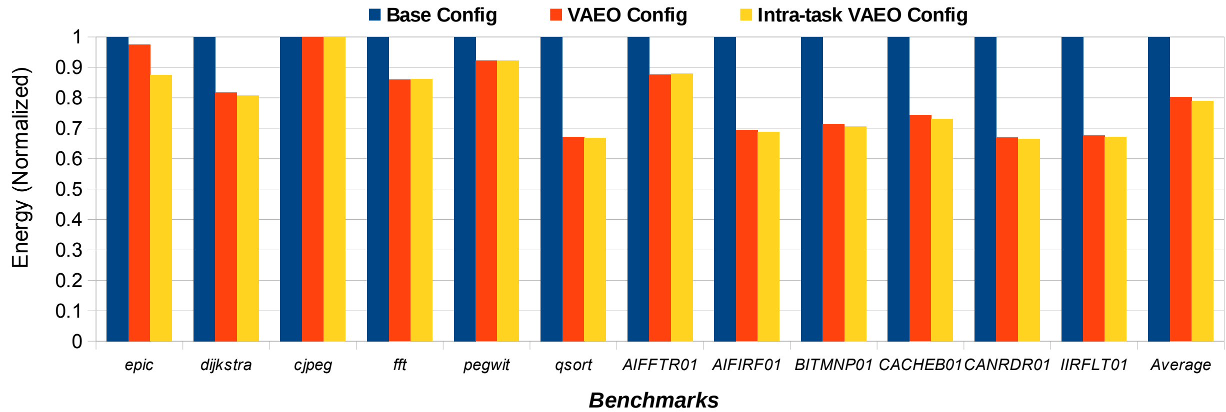
at the phase level (inside a task), where phases can execute with the intra-task VAEO configuration vector (one configuration for each phase). It is straightforward to introduce our (inter+intra)-task DCR approach, which combines these two levels of optimization by applying intra-task DCR on each task for inter-task DCR. Algorithm 4 shows our (inter+intra)-task DCR approach. In Step 1, we generate the profile (i.e., the

intra-task VAEO configuration vector) for each task, which can be obtained by the phase identification and cache assignment methods described earlier in this section. Step 2 shows the (inter+intra)-task cache reconfiguration approach. We fetch the task with the highest priority from the task queue as the current task t_c. The inter-task reconfiguration method (Algorithm 1) is called to make the decision whether the intra-task VAEO configuration is suitable for t_c to satisfy deadline constraints. If the intra-task VAEO configuration is chosen, the system will execute the task with intra-task reconfiguration. Otherwise, the system will execute the task with PO configuration without intra-task reconfiguration.

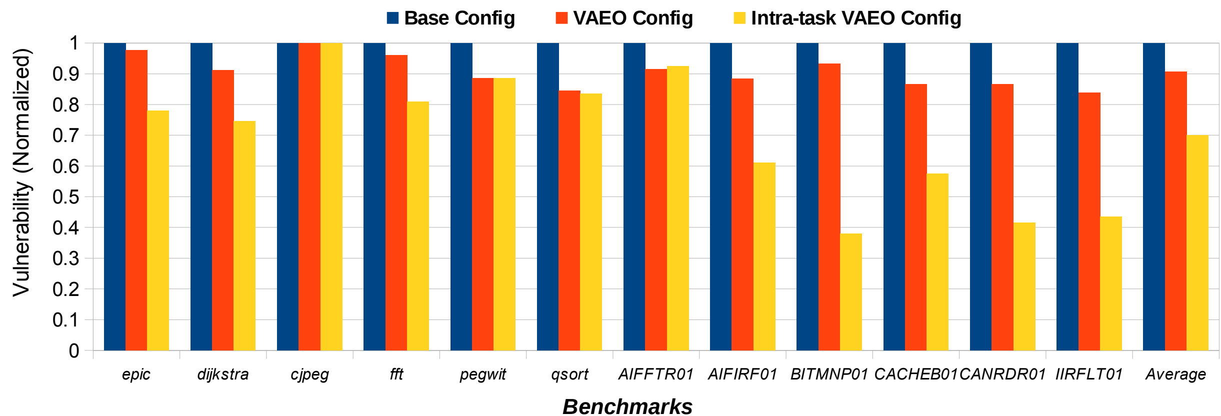
Algorithm 4: (Inter+intra)-task Cache Reconfiguration

```
1:  Step 1: Generate profile for each task.
2:  for each task t_i do
        // Call Algo. 2
3:      phases = PhaseIdentify(t_i)
        // Call Algo. 3
4:      Intra-task VAEO config = CacheAssign(phases)
5:  end
6:  Step 2: (Inter+intra)-task Cache Reconfiguration
7:  while task queue is not empty do
8:      Fetch the current task t_c with highest priority
9:      Use Algo. 1 for inter-task cache reconfiguration
10:     if Intra-task VAEO config is chosen then
11:         Execute t_c with intra-task reconfiguration
12:     end
13:     else
14:         Execute t_c with the PO config
15:     end
16: end
```

Instead of using a fixed VAEO configuration for a task, our (inter+intra)-task DCR approach can use the intra-task VAEO configuration, which has optimal configurations for different phases. There is no context switching or preemption during the execution of a task, even though intra-task optimization is applied. Compared with inter-task DCR, the inter+intra DCR approach introduces overhead in the form of cache flushing when cache configurations change between phases. Since the number of phases in a task is relatively



(a) Data cache energy consumption



(b) Data cache vulnerability

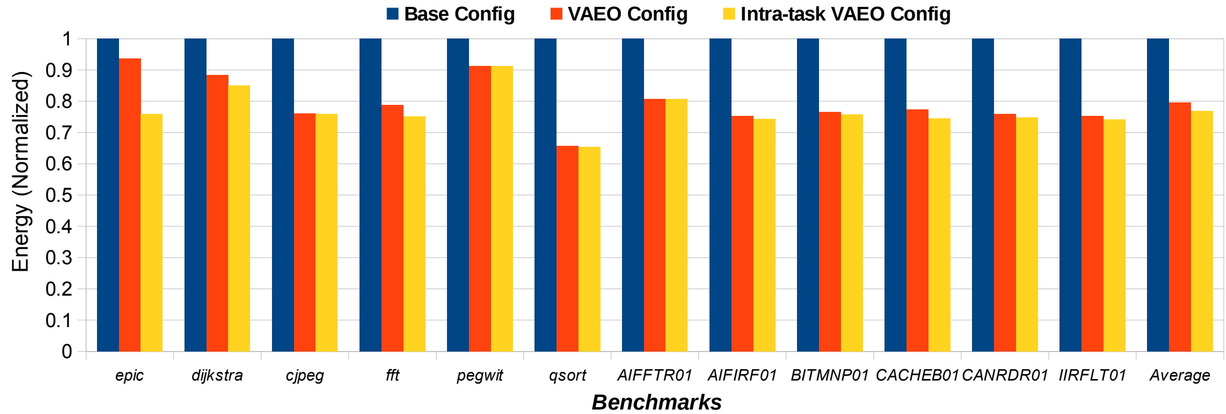
Figure 3-5. Comparison of DL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations. (a) Data cache energy consumption, (b) Data cache vulnerability.

small, the overhead caused by intra-task reconfiguration is negligible (less than 1% penalty for performance). The overhead of cache flushing on energy consumption and vulnerability is far outweighed by the benefits of intra-task reconfiguration, which will be presented in our experiments.

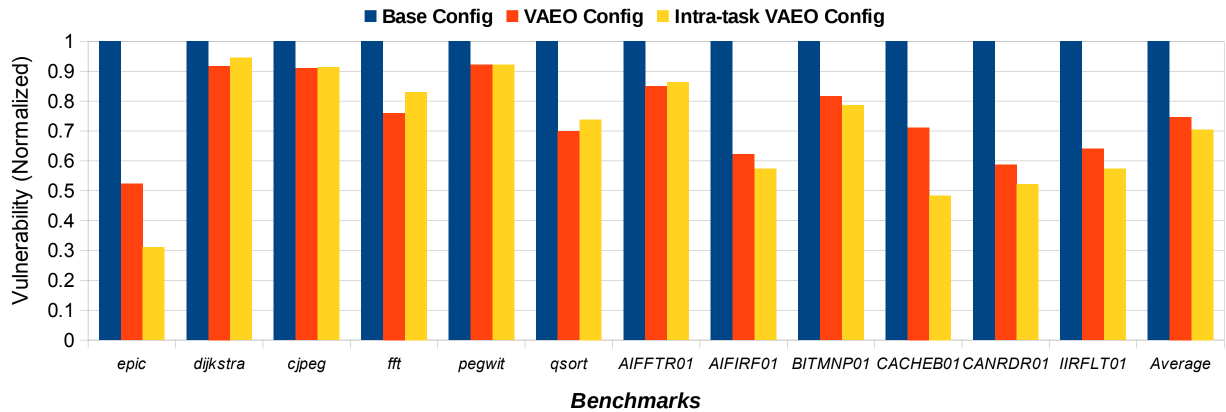
3.4 Experiments

3.4.1 Experimental Setup

The configurable caches used in our work are from the cache architecture introduced in [8]. The underlying cache architecture contains a configurable cache with a four-bank



(a) Instruction cache energy consumption



(b) Instruction cache vulnerability

Figure 3-6. Comparison of IL1 energy consumption and vulnerability of single tasks for Base, VAEO and Intra-task VAEO configurations. (a) Instruction cache energy consumption, (b) Instruction cache vulnerability.

cache with sizes of 1 KB, 2 KB and 4 KB, line sizes of 16 bytes, 32 bytes and 64 bytes, and associativity of 1-way, 2-way and 4-way. In order to quantify reliability-aware DCR trade-off, we selected benchmarks from MediaBench [22] and EEMBC automotive [23] benchmark suites. Table 3-1 shows our four task sets with three selected benchmarks in each set. All of the tasks are executed with the default input parameters provided with the benchmark suites. The benchmarks from MediaBench have about 10∼200 million instructions, while the benchmarks from EEMBC AutoBench have about 1∼10 million instructions. The rationale for us to form a task set is that the tasks are of

comparable size in terms of number of instructions. Both task set 1 and set 2 consist of three tasks from MediaBench. Both task set 3 and set 4 consist of three tasks from EEMBC AutoBench. Thus, set 1 and set 2 have more instructions and can potentially stress the cache with more cache accesses, compared to set 3 and set 4.

Table 3-1. Four task sets with twelve benchmarks

	Task 1	Task 2	Task 3
Task Set 1	epic	dijkstra	cjpeg
Task Set 2	fft	pegwit	qsort
Task Set 3	AIFFTR01	AIFIRF01	BITMNP01
Task Set 4	CACHEB01	CANRDR01	IIRFLT01

We modified the SimpleScalar simulator [24] for cache vulnerability analysis and energy consumption estimation. We performed the vulnerability analysis during cache accesses for each byte in instruction and data cache. The vulnerability estimation function collects all the vulnerable intervals for each valid byte in cache. We applied the same energy model as in [8] to calculate both dynamic and static energy consumption, and the energy consumption was estimated using CACTI 5.3 [25] with 65 nm technology. For static profiling of each task to find the PO, VAEO, and Intra-task VAEO (with intra-task reconfiguration) cache configurations, we developed Perl scripts to search the design space of all possible cache configurations. Since we only consider systems with one level of reconfigurable cache architecture, the space of possible cache configurations is small. The statistics for all possible cache configurations for a task can be collected in a reasonable time (a few hours). Once we have the profile tables for all the tasks, we use an EDF scheduler to simulate the system for a hyper-period. The cache selection algorithms are integrated in the scheduler to make decisions to reconfigure the cache during simulation. The optimization for instruction cache and data cache are independent. In the following subsections, we will first present results for optimization of individual benchmarks, followed by the results for task sets with scheduling and cache selection.

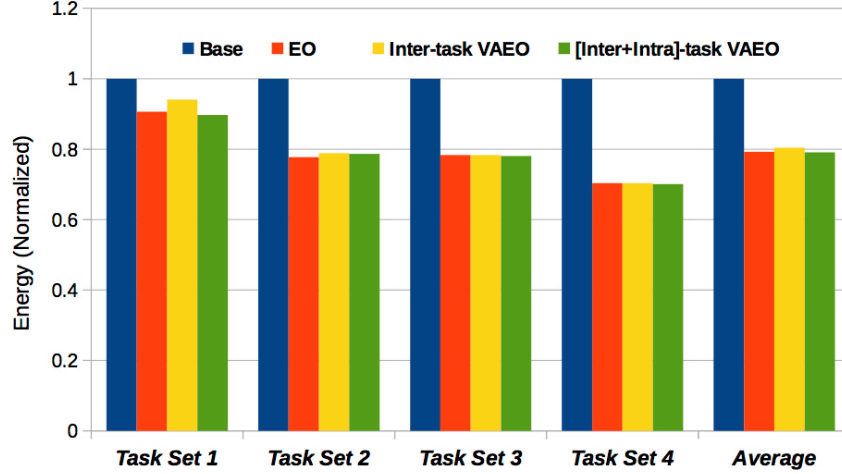
3.4.2 VAEO and Intra-task VAEO Configurations

In this section, we present the results to show the effectiveness of reconfiguration for single tasks. We will compare the energy consumption and vulnerability when a task is executed with the Base, VAEO and Intra-task VAEO configurations.

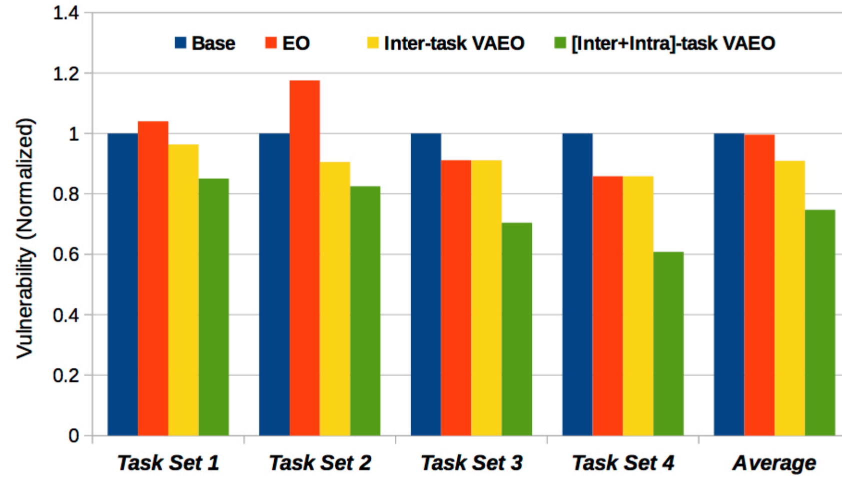
- **Base Config**: the configuration of the *base cache*, which is 4KB, 2-way associative with line size of 32 bytes.
- **VAEO Config**: the vulnerability-aware energy optimal configuration without intra-task reconfiguration.
- **Intra-task VAEO Config**: the VAEO configuration when intra-task reconfiguration is allowed.

Figure 3-5a and 3-5b show the energy and vulnerability of L1 data cache for 12 benchmarks. The VAEO configurations can reduce energy consumption (up to 33.0%, 19.8% on average), as well as vulnerability (up to 16.1%, 9.3% on average), compared with Base configurations. The Intra-task VAEO configurations can reduce energy consumption (up to 33.5%, 21.1% on average), as well as vulnerability (up to 58.4%, 30.0% on average), compared with Base configurations. Generally speaking, the VAEO configurations can greatly reduce energy and vulnerability compared with the Base, and the Intra-task VAEO can further improve energy and vulnerability compared with VAEO.

We observe that the trend of energy and vulnerability improvement in data cache (Figure 3-5) is similar to the trend of instruction cache (Figure 3-6). Figure 3-6a and 3-6b show the energy and vulnerability of L1 instruction cache for 12 benchmarks with their Base, VAEO and Intra-task VAEO configurations. The VAEO configurations can reduce energy consumption (up to 34.3%, 20.4% on average), as well as vulnerability (up to 47.6%, 25.3% on average), compared with Base configurations. The Intra-task VAEO configurations can reduce energy consumption (up to 34.5%, 23.1% on average), as well as vulnerability (up to 68.9%, 29.5% on average), compared with Base configurations.



(a) Data cache energy consumption



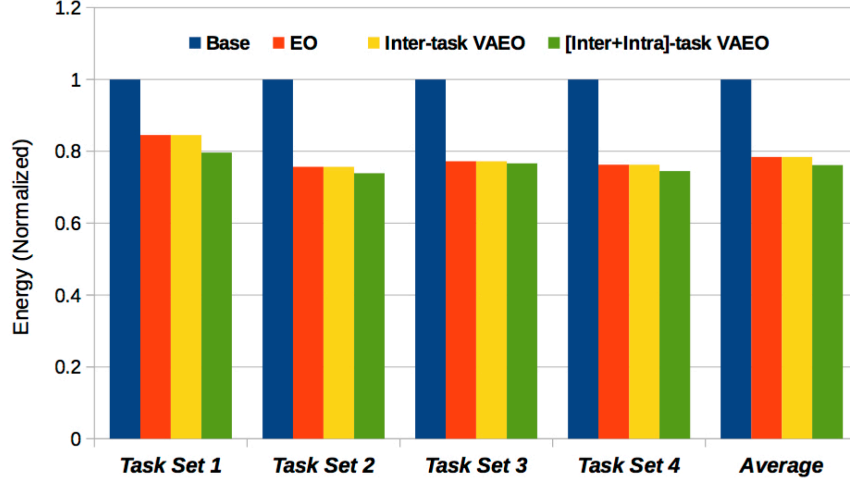
(b) Data cache vulnerability

Figure 3-7. (a) Data cache energy and (b) Data cache vulnerability.

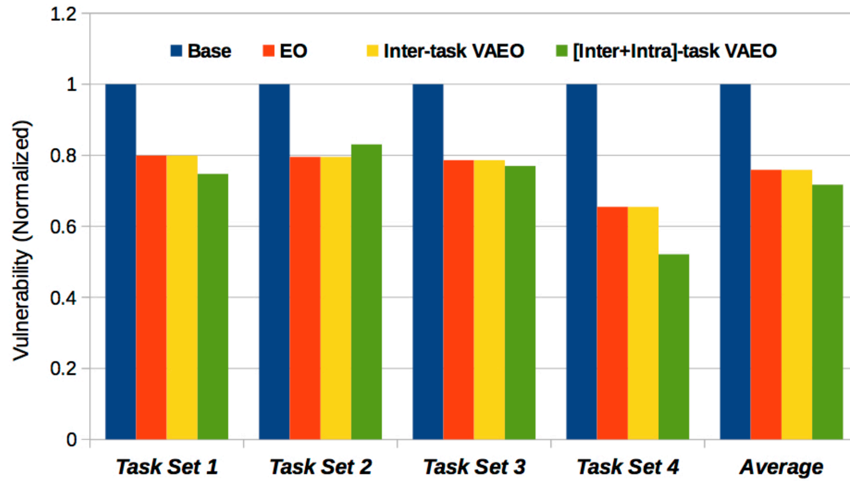
3.4.3 Results for Inter-task VAEO and (Inter+Intra)-task VAEO

In this section, we present the results to show the effectiveness of reconfiguration for task sets using proposed approaches. We profile each task with its PO, EO, VAEO, and Intra-task VAEO configurations. The runtime algorithms (Algorithm 1 for inter-task reconfiguration and Algorithm 4 for (inter+intra)-task reconfiguration) will select between PO and VAEO (or Intra-task VAEO) configurations. In the following section, we compare our proposed VAEO approaches with the base cache system as well as [11]:

- **Base** refers to the base system which uses the fixed *Base Config* for all tasks.



(a) Instruction Cache Energy Consumption



(b) Instruction Cache Vulnerability

Figure 3-8. (a) Instruction cache energy and (b) Instruction cache vulnerability.

- **EO** [11] refers to the Energy-Optimization approach in [11] which chooses between PO and EO configurations.
- **Inter-task VAEO** is our inter-task reconfiguration approach when the runtime algorithm chooses between PO and VAEO configurations.
- **(Inter+Intra)-task VAEO** is our (inter+intra)-task reconfiguration approach when the runtime algorithm chooses between PO and Intra-task VAEO configurations.

Figure 3-7a and 3-7b show the results of L1 data cache for four task sets. *Inter-task VAEO* can improve energy by 19.6% on average and vulnerability by 9.0% on average, compared with *Base*. *(Inter+Intra)-task VAEO* can improve energy by 20.9% on average

and vulnerability by 25.3% on average, compared with *Base*. *(Inter+Intra)-task VAEO*, which takes advantage of both intra-task and inter-task reconfiguration, can further improve energy consumption compared with *Inter-task VAEO*. Compared with EO [11], our VAEO approach can reduce vulnerability by 8.7% on average, while it consumes 1.2% more energy on average. This minor energy penalty is not surprising since EO did not consider any vulnerability threshold during energy minimization, whereas our approach respects the vulnerability constraint. Our (inter+inter)-task VAEO approach reduce vulnerability by 24.9% and it saves 0.1% more energy compared with EO [11]. As shown in Figure 3-7b, EO [11] produces very bad vulnerability for Task Set 1 and Task Set 2, which is even worse than the *Base* system.

Figure 3-8a and 3-8b show the results of L1 instruction cache for four task sets. *Inter-task VAEO* can improve energy by 21.6% on average and vulnerability by 24.1% on average. *(Inter+Intra)-task VAEO* can improve energy by 23.8% on average and vulnerability by 28.2% on average. Compared with EO [11], our VAEO approach produces the exact same results for four task sets. This is because the VAEO configurations are exactly the same as EO configurations for IL1 cache. This suggests that IL1 cache accesses has similar patterns among benchmarks, thus the results for IL1 have fewer variations than that of DL1 cache. Our (inter+inter)-task VAEO approach can improve a little bit further over the VAEO approach. Compared with EO [11], our (inter+inter)-task VAEO approach can reduce vulnerability by 4.1% and save 2.3% more energy.

3.4.4 Hardware Overhead

Cost of implementation involves two factors: (1) the cost of reconfiguration infrastructure (2) the cost of chip area for storing profile table. Dynamic cache reconfiguration (DCR) is an approach widely used in embedded systems for performance improvement and energy saving. This architecture requires very simple hardware augmentation and minor overhead [11]. A light process can be used as the cache tuner, which will make the reconfiguration decision and change the configuration at runtime.

The overhead to implement our VAEO approach is mostly the cost to store the profile table in hardware. The cache tuner will fetch the cache configuration information from the profile table. The size of the table depends on the number of tasks in the system and the information needed to store for each task. For the VAEO approach, we need to store two configurations (i.e., [*Config*, *Runtime*] for the PO and VAEO configurations) for each task. Five bits are used to specify a configuration since the configurable cache architecture used in this study offers 18 possible cache configurations. Another 16 bits are used to store the expected runtime of the task. For 12 benchmarks, the profile table contains 24 entries each with 21 bits. For the VAEO approach with intra-task reconfiguration, we need to store [$Phase_1$, $Config_1$, ..., $Phase_n$, $Config_n$, *Runtime*] for the intra-task VAEO configuration. We use 16 bits ($Phase_i$) to indicate the start instruction number of the phase, and 5 bits to store its cache configuration. For benchmarks used in our experiments, n is at most 4. In total it takes at most 100 (=16*4+5*4+16) bits to store the intra-task VAEO configuration for one task.

Table 3-2. Overhead of profile table (180nm technology)

Approach	Table size (bits)	Area (μm^2)	Dynamic Power (μW)	Leakage Power (μW)
VAEO	504	96730	60.05	180.13
Intra-task VAEO	1032	198067	122.97	368.84

Table 3-3. Overhead of profile table (65nm technology)

Approach	Table size (bits)	Area (μm^2)	Dynamic Power (μW)	Leakage Power (μW)
VAEO	504	10641	19.26	243.37
Intra-task VAEO	1032	21788	39.44	498.33

We used Synopsis Design Compiler with TSMC library to implement the profile table. We estimate a table lookup frequency of once per 3 μs. It is a table lookup every one thousand instructions using a 500 MHz CPU with an average CPI of 1.5. It should be suffice since the benchmarks we used have around 1 to 200 million instructions. Table 3-2 and 3-3 show our results of the area, dynamic power, and leakage power for the profile table using 180nm and 65nm, respectively. We observed that on average for each task set, the energy overhead of our approach accounts for less than 2% (0.067 mJ compared

to 3.38 mJ) of the total energy savings for VAEO approach, and less than 3% (0.14mJ compared to 4.73 mJ) of the total energy savings for intra-task VAEO approach. The (intra+intra)-task VAEO approach has slightly higher overhead than VAEO and also has higher energy savings. Therefore, we can conclude that the overhead for profile tables are negligible compared to the energy saving for both VAEO and (inter+intra)-task VAEO approaches.

3.5 Summary

Dynamic cache reconfiguration is widely used for improving energy and performance in embedded systems. While cache vulnerability is a well studied area, previous research efforts did not explore cache vulnerability in the context of cache reconfiguration. In this chapter, we developed algorithms to reduce cache vulnerability with energy and performance considerations. Our experimental results demonstrated that our approach can significantly improve the reliability of both instruction and data caches. For the data cache, the Inter+Intra DCR approach can improve energy by 20.9% on average and vulnerability by 25.3% on average. For the instruction cache, the Inter+Intra DCR approach can improve energy by 23.8% on average and vulnerability by 28.2% on average. Future work will focus on applying our approach to more flexible systems in broader areas. (1) Our approach can be extended to multi-level caches in single-core systems as well as multicore systems. The only difference would be that heuristic approaches should be used to efficiently explore beneficial cache configurations because exhaustive exploration may not be feasible since the number of possible configurations can be very large. (2) Our approach can be extended to systems allowing preemptive execution. This can be achieved by partitioning tasks into phases and profiling each partition, and preemptive task can resume execution using the configuration for the current phase.

CHAPTER 4
VULNERABILITY-AWARE RECONFIGURATION FOR PARTIALLY PROTECTED CACHES

In this chapter, we apply the idea of cache reconfiguration to the partially protected caches (PPC) architecture. On one hand, our approach has the advantage of PPC, which has a protected cache against soft-error vulnerability. On the other hand, it also has the advantage of DCR, which can reduce energy consumption by proper reconfiguration. Our proposed approach can select the best cache configurations for the two caches of the PPC architecture, and it also searches for the optimal data partitioning scheme to split data into the two caches. Our method is designed for the optimization of single tasks in the reconfigurable PPC architecture, which can be easily extended to task sets.

We propose a reconfigurable cache architecture (as in Figure 4-1) to address the above challenges. While the protected cache reduces vulnerability, the reconfigurability of the two (protected and unprotected) caches enables reduction in both execution time and energy consumption. This chapter makes four important contributions:

1. It presents a reconfigurable cache architecture to improve performance and energy efficiency while maintaining PPC's natural advantage for vulnerability reduction.
2. It develops a greedy algorithm for partitioning data pages between the protected and unprotected caches.
3. It proposes a method for synergistic exploration of cache configurations and data partitioning schemes to trade-off between vulnerability, energy consumption and performance.

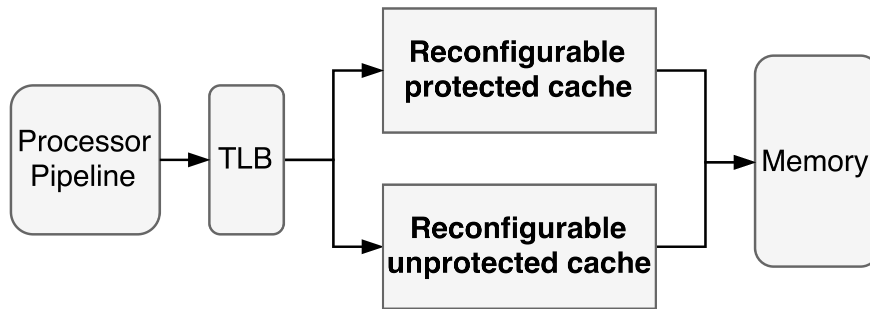


Figure 4-1. A reconfigurable PPC-base architecture with one protected cache and the other unprotected cache at the same level of hierarchy.

4. It proposes two heuristics for fast exploration of cache configurations and data pages without degrading the optimization results.

The remainder of the chapter is organized as follows. Section 4.1 presents the energy and vulnerability models. Section 4.2 describes the data partitioning method and our vulnerability-aware cache reconfiguration framework using exhaustive exploration as well as fast exploration. Section 4.3 presents our experimental results. Finally, Section 4.4 concludes the chapter.

4.1 Energy Models

In this section, we introduce the models used to measure energy consumption and vulnerability of the PPC caches.

4.1.1 Energy Model for Unprotected Cache

The energy model for unprotected cache is adopted from the one used in [11], which calculates both dynamic and static energy consumption and main memory fetch energy. The total cache energy consumption is $E = E_{dyn} + E_{sta}$, where E_{dyn} and E_{sta} denote the dynamic and static energy of the cache subsystem. Let E_{access} and E_{miss} be the energy consumption for per access and per miss, P_{sta} be the power consumption for one clock cycle (CC). Specially, we have:

$$E_{dyn} = Accesses \times E_{access} + Misses \times E_{miss} \tag{4–1}$$

$$E_{miss} = E_{offchip_access} + E_{block_fill} \tag{4–2}$$

$$E_{sta} = P_{sta} \times CC \times t_{cycle} \tag{4–3}$$

4.1.2 Energy Model for Protected Cache

For the ECC protected cache, the dynamic energy calculation also includes energy consumption for ECC encode/decode. Similar to [38], we categorize the cache accesses into *read_hit*, *read_miss*, *write_hit*, and *write_miss* since each operation results in different ECC events. For example, the energy consumption of *read_hit* is the sum of the access energy consumption and the energy consumption of ECC decoding (d), while the energy

consumption of $read_miss$ is the sum of the access energy consumption, the energy consumption of ECC decoding (d) as well as the ECC encoding (e).

$$\Delta E_{dyn} = RH \times d + RM \times (d + e) + WH \times e + WM \times (d + e) \tag{4–4}$$

where RH, RM, WH and WM denote the number of $read_hit$, $read_miss$, $write_hit$ and $write_miss$, respectively.

4.2 Cache Reconfiguration of PPC

There are multiple aspects that have impact on a program executing on a reconfigurable PPC-based architecture. These aspects include: (1) the data page map which partitions data pages into the two caches; (2) the configuration of the protected cache and the configuration of the unprotected cache. Our goal is to optimize *performance*, *energy*, and *vulnerability* for programs running in our system. Specifically, we would like to minimize both vulnerability and energy consumption with acceptable or no degradation on performance. Since different programs have various data access patterns and cache requirements, we need to make wise design decisions to take advantage of the protected cache to reduce vulnerability, while utilize reconfigurability to save energy consumption.

Our problem can be formulated as shown in Equation 4–5, where we would like to minimize both vulnerability and energy with acceptable degradation (less than *rThresh*) on performance.

$$\begin{aligned} &minimize \text{ (Vulnerability, Energy)} \\ &\qquad subject\ to \text{ RuntimeOverhead} < rThresh \end{aligned} \tag{4–5}$$

Figure 4-2 outlines our approach to accomplish the goal of vulnerability and energy optimization without penalizing the performance. The two important design decisions we have to make for each program include: (1) data partitioning to map data pages into

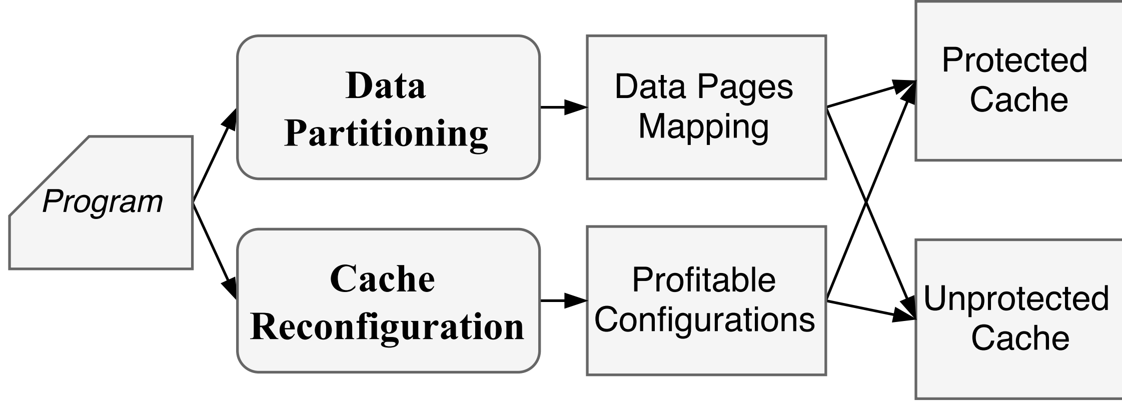
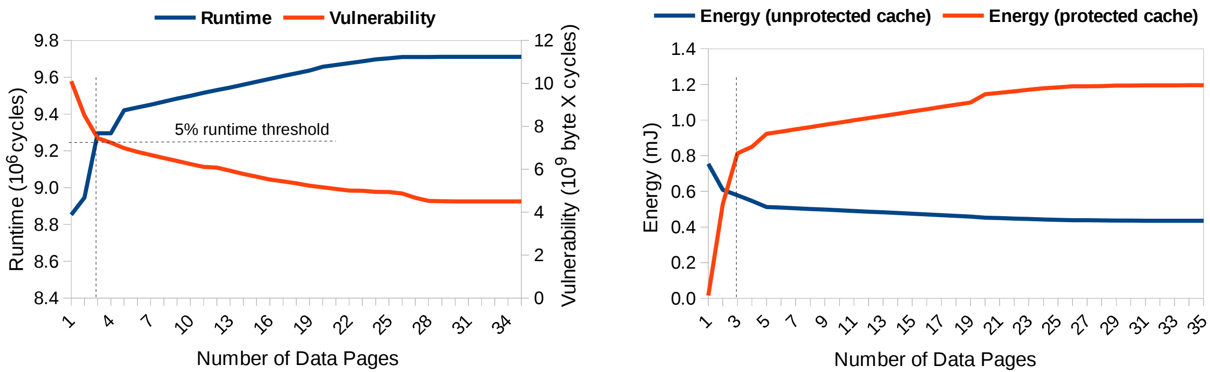


Figure 4-2. Our exploration methodology consists of two design decisions: data partitioning and cache reconfiguration.

the protected and unprotected caches; (2) cache reconfiguration to select the profitable configurations for the two caches. We perform these two steps through off-line (design time) analysis. It is important to note that off-line exploration is applicable and useful for embedded systems design because these systems have well-defined applications (programs) that are known a priori. The remainder of this section is organized as follows. Section 4.2.1 describes efficient partitioning of data pages between protected and unprotected regions. Section 4.2.2 describes exploration of different cache configurations. Finally, Section 4.2.3 presents two fast and effective design space exploration techniques to improve the scalability of our reconfiguration framework.



(a) Vulnerability and runtime trade-off

(b) Energy consumption

Figure 4-3. Trade-off between vulnerability, runtime and energy, for benchmark *cjpeg*. (a) Vulnerability and runtime trade-off, (b) Energy consumption.

4.2.1 Data Partitioning

We have two data caches, with one protected from soft errors, and another unprotected which remains vulnerable. The protected cache is very effective in reducing the vulnerability. For any data mapped to the protected cache, it is protected against soft errors. If we map all data into the protected cache, the vulnerability of the application will be reduced to zero. However, mapping too much data into the small protected cache will increase the cache misses and eventually result in a significant degradation of performance and increase in energy consumption. We performed a simple experiment to show the effectiveness of protected cache in reducing vulnerability, as well as its side-effect on runtime and energy consumption. Figure 4-3 illustrate this exploration for benchmark *cjpeg* from MediaBench [22]. First, we map all the data pages (54 pages in total for *cjpeg*) into the unprotected cache. Then we sort the pages by vulnerability in decreasing order. Each time we map a new page (from the top of the sorted list) into the protected cache if it would reduce the vulnerability. Figure 4-3a shows that after mapping 35 pages into the protected cache, vulnerability is reduced by 55%, while runtime increases by 10%. The runtime is expected to increase because the protected cache, which is much smaller in capacity, will cause a lot of misses when there are many data pages evicting each other's data. Figure 4-3b shows that energy consumption of unprotected cache decreases as we remove pages from the unprotected cache. However, the energy consumption of the protected cache will increase drastically when more data pages are mapped into it. The energy consumption of the protected cache can rise to as much as 2.5 times of that of the unprotected cache. This example suggests that we cannot afford to blindly map data pages into the protected cache, which might result in unacceptable performance and energy penalty.

We introduce a greedy approach with a runtime threshold during data partitioning. For the same example above, if we set a 5% threshold for runtime, we are allowed to map only three pages into the protected cache. The runtime threshold will limit both the

performance degradation and the energy penalty. Algorithm 5 shows our data partitioning approach. It takes the benchmark and the runtime penalty threshold ($rThresh$) as inputs, and produces a data partitioning scheme $PageMap$ as output. $PageMap[i]$ indicates the cache for the $i^{th} Page$: 0 means the unprotected cache, and 1 means the protected cache. We first map all pages into the unprotected cache (the base partitioning scheme) by setting $PageMap$ as all 0's (line 1). We simulate the benchmark with the base partitioning scheme, which provides us with $BaseRuntime$ and a profile of vulnerability of all data pages (line 2-4). After sorting the pages by vulnerability in descending order (line 5), we greedily select each page to test whether it is suitable to be mapped into the protected cache (line 6 to 15). A page is suitable to be protected if it satisfies two conditions: (i) it is favorable for vulnerability reduction and (ii) it would not cause the program to exceed the runtime threshold. If either of the two conditions is not satisfied, the page should be put back into the unprotected cache. For the benchmarks used in our experiments, we have up to 258 pages, on average 58 pages. Let p be the total number of data pages in a benchmark. Although we need to sort the pages in line 5 (which usually has time complexity of $O(p \log p)$ for conventional sorting algorithms), the time for sorting is negligible compared with the time used for simulation. We need to do p rounds of simulation to explore all the data pages and get a most beneficial $PageMap$. The time needed for Algorithm 5 is $O(C \times p)$, where C is the time needed for one simulation, and p is the number of data pages.

4.2.2 Cache Exploration

This section describes how to take advantages of both cache reconfiguration (DCR) and data partitioning (PPC) by synergistic exploration. In our reconfigurable PPC architecture, the **base cache**[1] for the unprotected cache is 4096B_1W_32B (size of 4KB,

[1] The **base cache** refers to the cache configuration that is widely used in the literature for the selected benchmarks.

Algorithm 5: DataPartition

Input: Benchmark, *rThresh*, page vulnerability profile
Output: *PageMap*

```
1:  Set PageMap[n] = (0, 0, ..., 0);
2:  {runtime, vulnerability} = simulate(PageMap)
3:  Set BaseRuntime = runtime;
4:  Set BestVulnerability = vulnerability;
5:  Sort the pages by vulnerability in descending order
6:  for (i = 0; i < PageMap.size; i++) do
7:      Set PageMap[i] = 1;
8:      {runtime, vulnerability} = simulate(PageMap)
9:      if vulnerability < BestVulnerability then
10:         if runtime < BaseRuntime × (1 + rThresh) then
11:             Set BestVulnerability = vulnerability;
12:         else
13:             Set PageMap[i] = 0;
14:         end
15:     else
16:         Set PageMap[i] = 0;
17:     end
18: end
19: return PageMap
```

1-way associative, and line size of 16-byte), which can be reconfigured to the size of 1KB, 2KB and 4KB, associativity of 1-way, 2-way and 4-way, line size of 16-byte, 32-byte and 64-byte. This will lead to 18 valid configurations[2] for the unprotected cache. The base configuration for the protected cache is 512B_1W_32B, which can be reconfigured to be size of 256B and 512B, associativity of 1-way and 2-way, and line size of 16-byte and 32-byte. This will lead to 6 valid configurations for the protected cache.

It is crucial to dynamically select partitioning schemes for different cache configurations. For example, assume that we have a data partitioning solution for the base configuration (<512B, 4096B>) for the protected and unprotected caches. If we use the same data partitioning for another cache configuration (<256B, 4096B>), it would

[2] It is less than 3^3 since not all combinations are valid [11].

likely encounter a lot of cache misses in the protected cache and result in significant performance and energy penalty.

The effectiveness of the data partitioning algorithm described in Section 4.2.1, is influenced by the cache exploration. First, the data partitioning depends on the configuration of the unprotected cache, as it requires the vulnerability profile of data pages. When using a different unprotected cache, the vulnerability of a data page will change, which will directly affect its priority (importance in reducing vulnerability) during data partitioning. Similarly, the data partitioning is a greedy algorithm and is constrained by the configuration of the protected cache. We propose a synergistic exploration of different cache configurations with various partitioning of data pages.

Algorithm 6: Exhaustive Cache Exploration

Input: Benchmark, $rThresh$
Output: Protected and unprotected cache configs

```
1:  for cache size s_u of 1024B, 2048B, 4096B do
2:      for associativity a_u of 1, 2, 4 ways do
3:          for line size l_u of 16B, 32B, 64B do
4:              UnproConfig = (s_u, a_u, l_u)
5:              Generate vulnerability profile
6:              for cache size s_p of 256B, 512B do
7:                  for associativity a_p of 1, 2 ways do
8:                      for line size l_p of 16B, 32B do
9:                          ProConfig = (s_p, a_p, l_p)
                            // Call Algorithm 5
10:                         DataPartition(rThresh)
11:                     end
12:                 end
13:             end
14:         end
15:     end
16: end
17: Collect (runtime, vulnerability, energy) for all configs.
18: Choose the best configs based on system goal.
19: return (UnproConfig, ProConfig, PageMap)
```

Algorithm 6 shows our approach of dynamic data partitioning during the process of cache exploration. We explore the cache size, way associativity, and line size of the unprotected cache in line 1-3. For each unprotected cache configuration, it is necessary to re-evaluate the vulnerability of all pages based on the current configuration of the unprotected cache (line 4-5). Next, we explore the configurations of the protected cache and call Algorithm 5 to get the best data partitioning for the selected pair of cache configurations (line 6-10). The complexity of the algorithm is $O(n_1 \times n_2 \times p)$, where n_1 and n_2 are the number of configurations for the unprotected and protected caches, respectively, and p is the number of data pages in the benchmark.

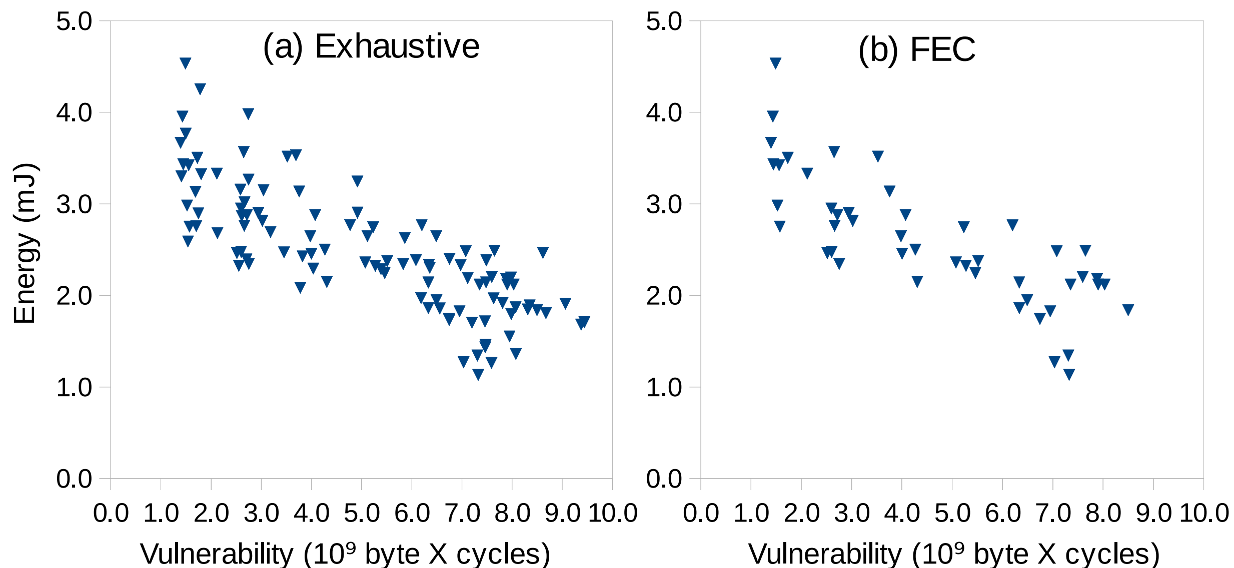


Figure 4-4. Configurations covered by (a) Exhaustive Exploration (108), (b) FEC Exploration (41), for benchmark *cjpeg*.

4.2.3 Fast Exploration

One major drawback of of the exhaustive approach outlined in the previous section is the long exploration time. The total time for exhaustive exploration is $(C \times n_1 \times n_2 \times p)$. For the largest benchmark *epic* with p=258 pages, the time needed for one simulation is C=22 seconds, so the total time is (22×18×6×258) seconds, which is about 7 days. Clearly, exhaustive exploration is not feasible for larger benchmarks and complex processors

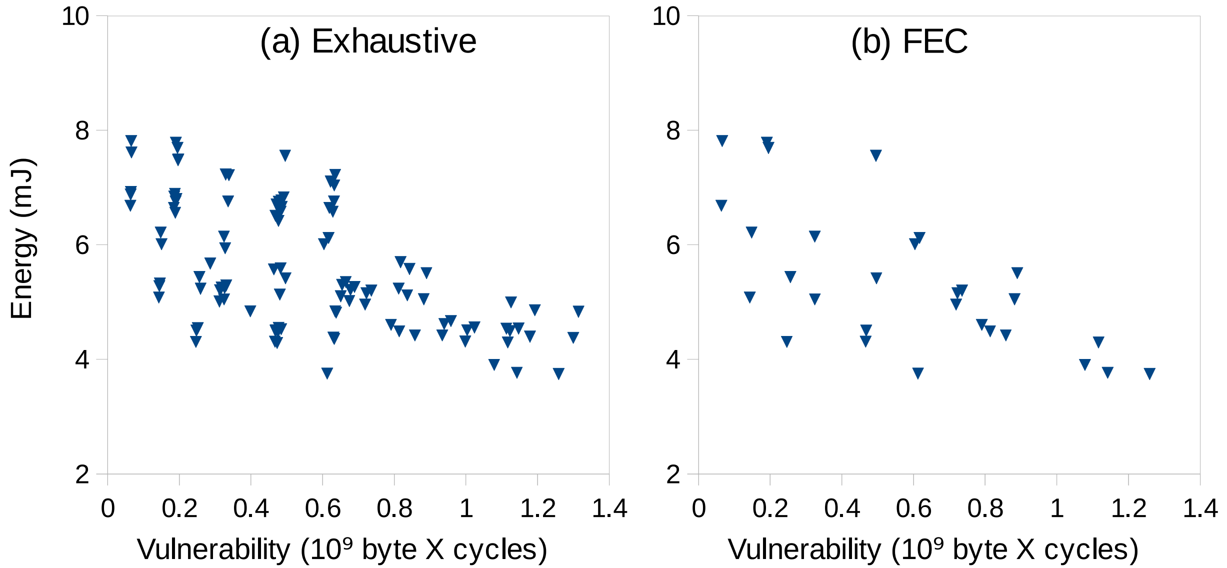


Figure 4-5. Configurations covered by (a) Exhaustive Exploration (108), (b) FEC Exploration (29), for benchmark *pegwit*.

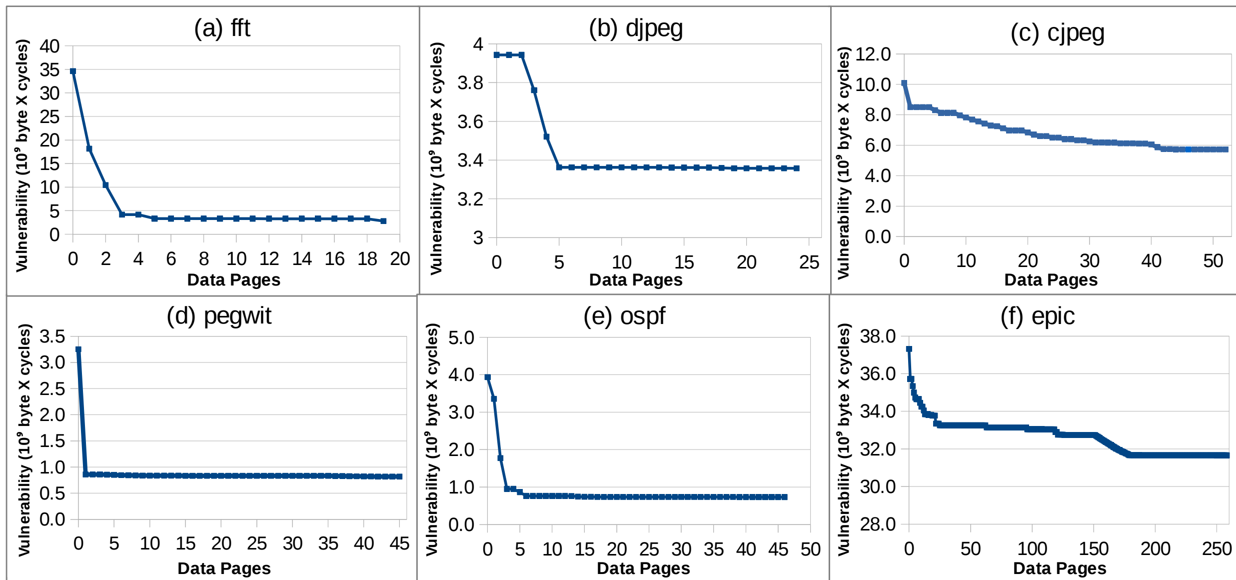


Figure 4-6. Page exploration during data paritioning for six different benchmarks

with a large number of cache configurations. In order to improve the scalability of our reconfiguration framework, we propose two fast and effective exploration heuristics that would explore fewer cache configurations and/or fewer data partitioning schemes without compromising the quality of optimization objectives. The remainder of this sections

describes two fast exploration techniques: Fast Exploration of Caches (**FEC**) and Fast Exploration of Caches and Data pages (**FECD**).

4.2.3.1 Fast exploration of caches (FEC)

By examing the results generated by exhaustive exploration, we find that some very unprofitable cache configurations are also explored. We propose a heuristic to reduce the number of explorations for cache configurations, thus drastically reduce the overall exploration time. We have 18 configurations for the unprotected cache and 6 configurations for the protected one. Both caches (protected and unprotected) have influence on the overall vulnerability and energy consumption. If one configuration for the protected (or unprotected) cache is very bad, it may not be useful to explore the unprotected (or protected) cache at all. We can explore the two caches independently and pick out the Pareto-optimal tradeoff points. Candidates with both vulnerability and energy consumption worse than the Pareto-optimal ones are eliminated during the exploration. Our proposed Fast Exploration of Caches (FEC) heuristic is summarized below:

1. Hold the protected cache as the base configuration. Tune the unprotected cache and record all its Pareto-optimal configurations. Let P_u denote the set of recorded configurations for the unprotected cache.

2. Hold the unprotected cache as the base configuration. Tune the protected cache and record all its Pareto-optimal configurations. Let P_p denote the set of recorded configurations for the protected cache.

3. Explore all the combinations from each set of Pareto-optimal configurations from the previous two steps, and find the best configuration based on the system optimization goal.

The first two steps explore 24 (=18+6) candidates while the last step explores $|P_u| * |P_p|$ candidates. The number of Pareto-optimal points varies for different applications but normally around 2 to 6. In our experiments, the total number of explored cache configurations is 35 on average for the above heuristic approach. The total exploration time for one benchmark will be reduced to $C \times n' \times p$, where $n' = 35$ on average.

The number of cache configuration is reduced from 108(=18*6) to 35, which results in about 3X speed-up. Figure 4-4 and 4-5 show the explored points of the exhaustive exploration and the FEC heuristic for benchmark *cjpeg* and *pegwit*, where FEC explores 41 configuration for *cjpeg* and 29 configurations for *pegwit*.

4.2.3.2 Fast exploration of caches and data (FECD)

Given a combination of configurations of the unprotected and protected caches, the data partitioning in Algorithm 5 will divide data pages between them. The data partitioning process explores each page to decide whether this page should be protected or not, and it takes p (number of pages) simulations. Figure 4-6 shows the data partitioning exploration process when the two caches are set to their base configurations. There is a clear trend for these six benchmarks in Figure 4-6. The first few pages result in drastic vulnerability reduction and the curve will saturate after a certain number of pages are mapped to the protected cache. This is because the pages are sorted by vulnerability, and the partitioning algorithm goes through them one by one. This inspires us that we don't have to explore all the pages. We can save the total exploration time ($C \times n' \times p'$) by reducing the pages explored for each cache configuration. We can explore only the top p' pages for these benchmarks without any compromising of exploration quality. In our experiments for **FECD**, we choose to explore only top 50% of the data pages (with minor impact on *cjpeg* and *epic*), which can enable another 2X speed-up, compared to **FEC**.

4.3 Experiments

Our framework used the *sim-outorder* simulator from the SimpleScalar toolset [24]. The protected cache has an ECC-based technique, while the unprotected cache has no protection against soft errors. We assume that the protected cache is optimized to have the same access time as the unprotected one. Both caches are reconfigurable, and the reconfigurable cache model is described in Section 4.2.2. The ten benchmarks that are used in our experiments are from the MediaBench [22] and MiBench [46], which are representative of embedded system applications. The energy model and vulnerability

model are detailed in Section 4.1. The energy consumption for cache accesses is estimated using CACTI 4.2 [25] with a 0.18 μm technology. We implemented the vulnerability calculation in the simulator for the unprotected cache.

4.3.1 Synergistic Exploration

Figure 4-7 and 4-8 show the exploration results by applying Algorithm 6 for benchmark *cjpeg* and *pegwit*, with runtime penalty threshold to be 5%. The figures show the exhaustive exploration of all the 108 (=18×6) cache configurations, each of which has full exploration of all its data pages.

There are several interesting observations that we would like to highlight:

(1) If we observe the curves for 18 configurations of the unprotected cache, they have very consistent trends for six different protected cache configurations, especially for the benchmark *pegwit*. For example in Figure 4-8, the protected cache configuration $256B_1W_32B$ always has worse vulnerability (Figure 4-8a), worse energy consumption(Figure 4-8b), worse runtime(Figure 4-8c) than $512B_2W_32B$, if the same unprotected cache configuration is used. This motivates us to perform fast exploration of cache configurations. We can avoid exploration of a certain configuration if we know it will always produce worse results than another configuration.

(2) As shown in Figure 4-7a and 4-8a, vulnerability is dominated by the unprotected cache. This is as expected, since vulnerability only comes from data maintained in the unprotected cache. The vulnerability of the first nine unprotected cache configurations (of size 1024B and 2048B), is smaller than that of the last nine configurations (of size 4096B). This is due to the fact that a large unprotected cache can retain more data and the data usually stay in the cache for longer time because of lower miss rates, compared with a small unprotected cache.

(3) Figure 4-7 (*cjpeg*) has more fluctuation than Figure 4-8 (*pegwit*). This is because different configurations for *cjpeg* have very different data partitioning, while *pegwit* has the same data partitioning for all configurations. This is consistent with what we observed

in Figure 4-6 on page exploration for *cjpeg* and *pegwit*. The vulnerability of *cjpeg* is spread among many pages, while that of *pegwit* is dominated by the first page. So the data partitioning is the same for *pegwit* (just protect the first page) no matter what cache configuration is used. While the data partitioning changes greatly for *cjpeg* when different cache configurations are used.

4.3.2 Comparison with Previous Works

In this section, we perform exploration in two directions depending on the primary optimization objective: the energy-aware vulnerability-minimization exploration and the vulnerability-aware energy-minimization exploration. The first one is vulnerability minimization, where energy constraint is determined by the energy consumption of the two fixed base caches. In other words, we are trying to find the cache configuration with lowest vulnerability while the energy consumption is equal or better than two (protected and unprotected) fixed base caches (no DCR capability). Similarly, the second one is energy minimization with vulnerability constraint (determined by the vulnerability of two base caches). We refer to these two optimal configurations as **DCR+PPC(VulMin)** and **DCR+PPC(EnergyMin)**, respectively. [5] and [39] are the two most related works in reducing cache vulnerability for embedded systems. We compare our results with [5] and [39] to demonstrate the effectiveness of our approach.

4.3.2.1 Improvement to DCR

The approach in [5] uses dynamic cache reconfiguration (DCR without PPC) to reduce vulnerability. We use our *DCR+PPC(VulMin)* configuration to compare with them, as shown in Table 4-1. The results for DCR [5] are from the EAVO approach in [5], which uses only one reconfigurable cache. Our approach can reduce vulnerability by up to 96.5%, on average 61.9%, for the ten benchmarks. Our approach is able to provide drastic improvement in vulnerability compared to [5], because we take advantage of PPC's ability to reduce vulnerability. For benchmark *rijndael*, our vulnerability number is the same as [5]. This is because our DCR+PPC configuration is the same as DCR [5], which means

we map all the data into the unprotected cache while the protected one is not used at all. The reason is that *rijndael* has only 17 data pages (the smallest among all benchmarks), and mapping any of the data pages into the protected cache resulted in exceeding the performance threshold.

Table 4-1. Vulnerability reduction compared to [5]

Benchmark	DCR [5]	Our Approach	**Improvement**
fft	2.5E+10	2.2E+09	**91.3%**
cjpeg	1.0E+10	3.8E+09	**62.6%**
djpeg	1.9E+09	1.7E+09	**13.4%**
pegwit	2.3E+09	8.0E+07	**96.5%**
rijndael	6.8E+10	6.8E+10	**0.0%**
stringsearch	2.0E+09	2.9E+08	**85.9%**
untoast	2.5E+10	9.4E+08	**96.2%**
epic	6.4E+09	3.3E+09	**48.0%**
ospf	2.4E+09	2.9E+08	**87.9%**
susan	9.8E+09	6.2E+09	**37.1%**
Average	-	-	**61.9%**

4.3.2.2 Improvement to PPC

In this subsection, we compare with the partially protected caches (PPC) as proposed in [39], which has a fixed unprotected cache and a fixed protected cache (no DCR). We use our exhaustive exploration approach considering data partitioning and cache configurations as described in Algorithm 6. We present two sets of results based on two different primary optimization goals:

- **DCR+PPC(VulMin)**: vulnerability minimization, where energy constraint is determined by the energy consumption of the two fixed base caches.
- **DCR+PPC(EnergyMin)**: energy minimization with vulnerability constraint set as the vulnerability of two fixed base caches.

Figure 4-9a shows the vulnerability improvement for all benchmarks. *DCR+PPC(VulMin)* reduces the vulnerability by 52.8% on average, while *DCR+PPC(EnergyMin)* reduces the vulnerability by 28.9% on average. Figure 4-9b shows that *DCR+PPC(VulMin)* reduces the energy by 9.6% on average, while *DCR+PPC(EnergyMin)* reduces the energy by 18.4% on average. As expected based on

their optimization goals, *DCR+PPC(VulMin)* provides better vulnerability reduction whereas *DCR+PPC(EnergyMin)* provides better energy reduction results.

Table 4-2 and Table 4-3 show the detailed results (these results were summarized, in normalized form, in Figure 4-9). The first column indicates the benchmark. The second, fifth and eighth columns provide the vulnerability, energy and execution time, respectively, for the base configuration with PPC [39]. The third, sixth and ninth columns provide vulnerability, energy and execution time, respectively, using our approach (DCR+PPC). The fourth, seventh and tenth columns indicate the improvement produced by our approach compared to [39]. A positive number implies improvement whereas a negative number means overhead (penalty). Table 4-2, *DCR+PPC(VulMin)*, shows that our approach can provide significant reduction (on average 52.8%, up to 90.3%) in vulnerability, modest reduction (on average 9.6%, up to 38.9%) in energy, and minor performance penalty (0.3% on average). Table 4-3, *DCR+PPC(EnergyMin)*, demonstrates significant energy (on average 18.4%, up to 39.3%) and vulnerability reduction (on average 28.9%, up to 90.3%), with minor performance improvement (0.7% on average).

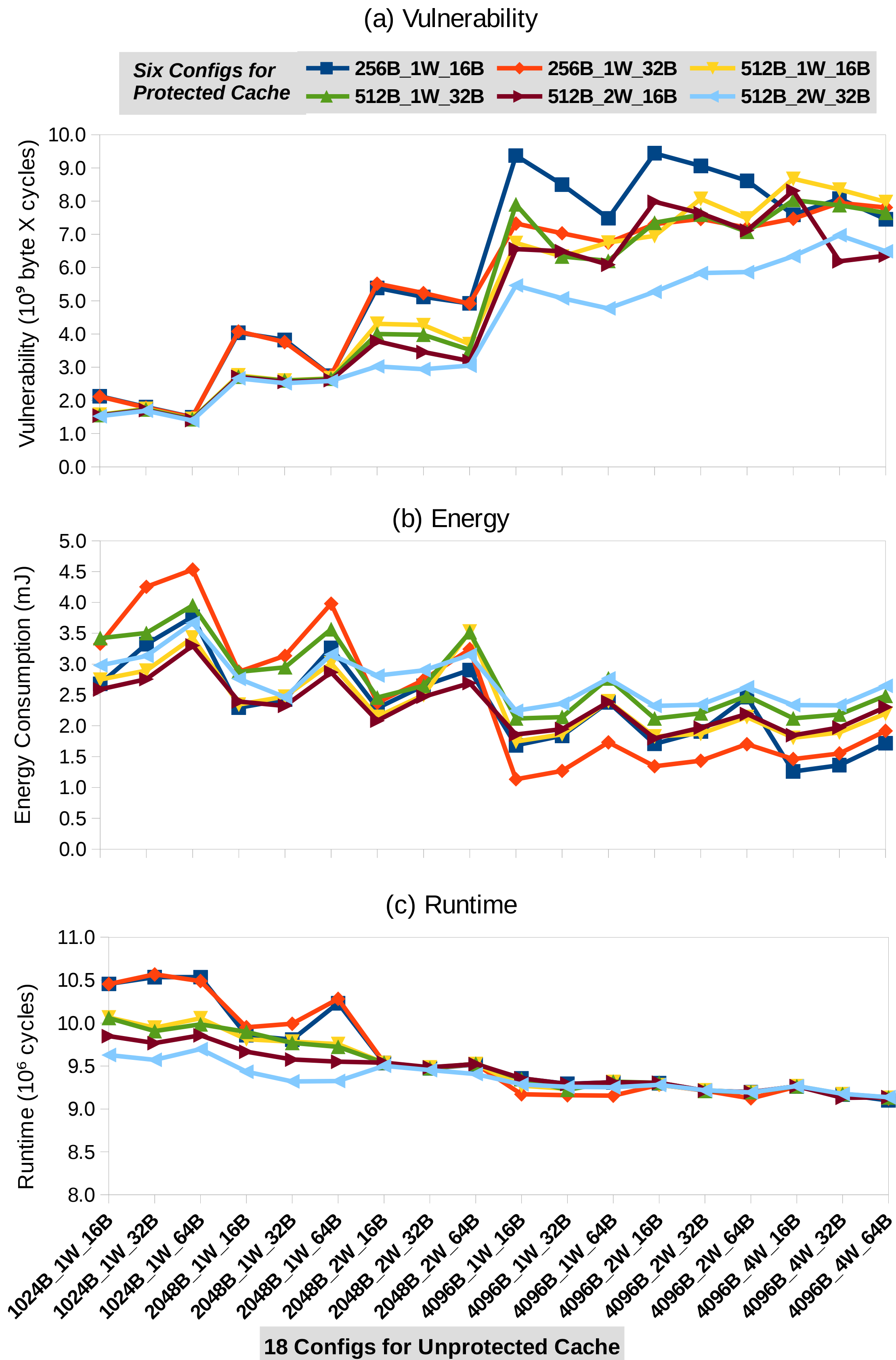


Figure 4-7. Exploration of all cache configurations for benchmark *cjpeg*, with *rThresh* = 5%.

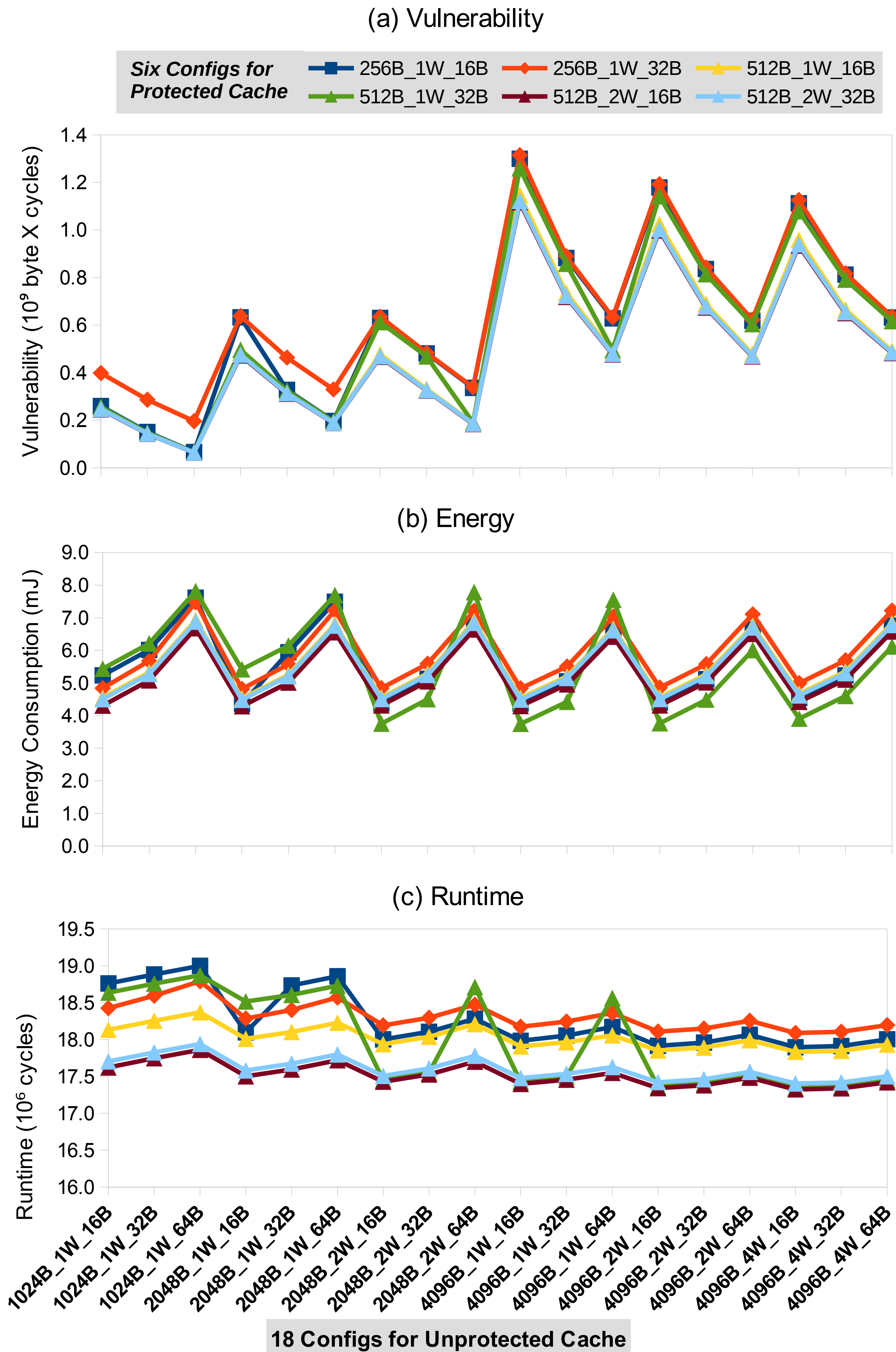
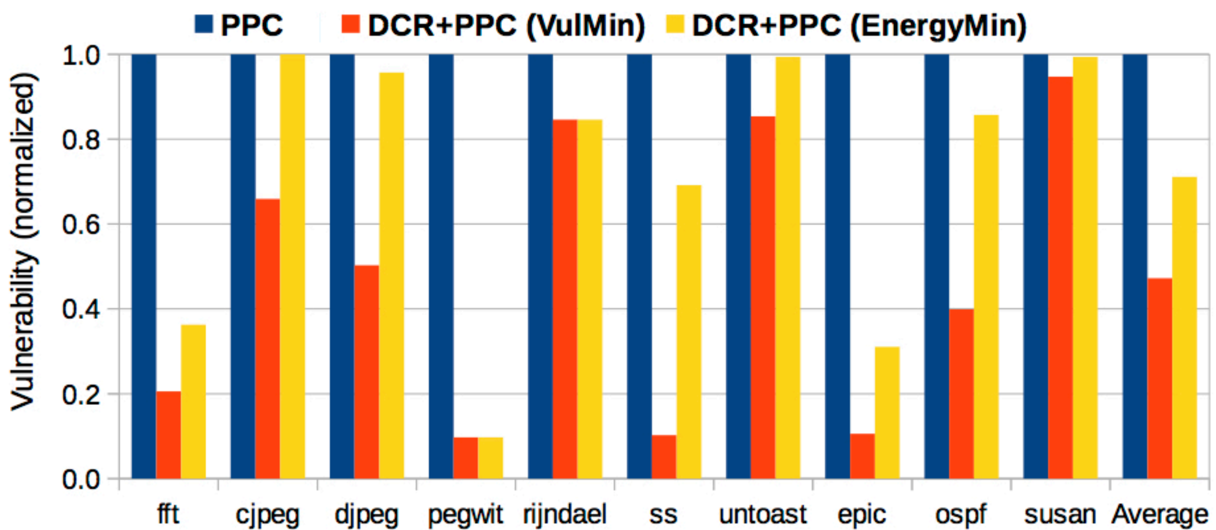
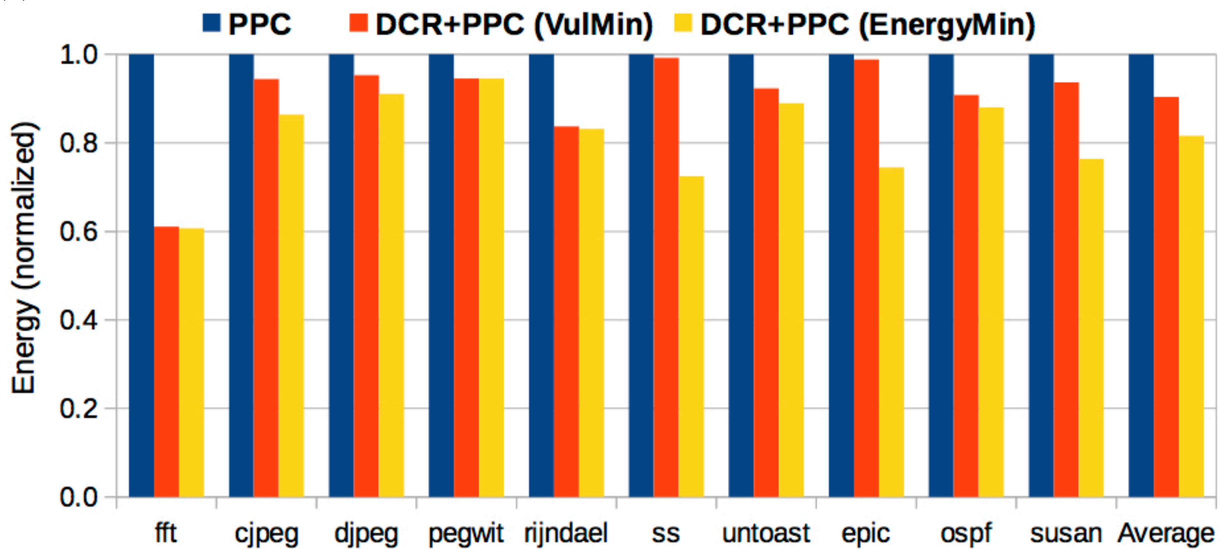


Figure 4-8. Exploration of all cache configurations for benchmark *pegwit*, with $rThresh = 5\%$.



(a) Vulnerability Improvement



(b) Energy Improvement

Figure 4-9. Comparison of *DCR+PPC(VulMin)* and *DCR+PPC(EnergyMin)* with PPC[39]. (a) Vulnerability improvement, (b) Energy improvement.

Table 4-2. *DCR+PPC(VulMin)*: vulnerability minimization under energy constraints

Benchmark	**Vulnerability** (10^9 *byte* × *cycles*)			**Energy** (*mJ*)			**Runtime** (10^6 *cycles*)		
	PPC [39]	Our Approach	**Improvement**	PPC [39]	Our Approach	**Improvement**	PPC [39]	Our Approach	**Improvement**
fft	10.81	2.22	**79.4%**	8.54	5.22	**38.9%**	52.60	51.21	**2.6%**
cjpeg	5.72	3.77	**34.1%**	2.21	2.08	**5.6%**	9.26	9.54	**-3.0%**
djpeg	3.36	1.69	**49.7%**	0.34	0.33	**4.7%**	2.63	2.69	**-2.2%**
pegwit	0.82	0.08	**90.3%**	4.93	4.66	**5.5%**	17.85	17.83	**0.1%**
rijndael	80.09	67.75	**15.4%**	3.84	3.22	**16.3%**	46.23	46.25	**-0.1%**
stringsearch	2.80	0.29	**89.8%**	0.81	0.81	**0.7%**	6.82	6.69	**1.9%**
untoast	1.10	0.94	**14.6%**	2.60	2.40	**7.7%**	29.52	29.27	**0.9%**
epic	31.66	3.35	**89.4%**	5.00	4.94	**1.1%**	36.35	37.43	**-3.0%**
ospf	0.73	0.29	**60.1%**	1.54	1.40	**9.2%**	5.19	5.25	**-1.1%**
susan	6.53	6.18	**5.3%**	1.18	1.10	**6.3%**	15.55	15.35	**1.3%**
Average	-	-	**52.8%**	-	-	**9.6%**	-	-	**-0.3%**

Table 4-3. *DCR+PPC(EnergyMin)*: energy minimization under vulnerability constraints

Benchmark	**Vulnerability** (10^9 *byte* × *cycles*)			**Energy** (*mJ*)			**Runtime** (10^6 *cycles*)		
	PPC [39]	Our Approach	**Improvement**	PPC [39]	Our Approach	**Improvement**	PPC [39]	Our Approach	**Improvement**
fft	10.81	3.92	**63.8%**	8.54	5.19	**39.3%**	52.60	51.16	**2.7%**
cjpeg	5.72	5.72	**0.0%**	2.21	1.91	**13.6%**	9.26	9.26	**-0.1%**
djpeg	3.36	3.21	**4.3%**	0.34	0.31	**8.9%**	2.63	2.60	**1.2%**
pegwit	0.82	0.08	**90.3%**	4.93	4.66	**5.5%**	17.85	17.83	**0.1%**
rijndael	80.09	67.75	**15.4%**	3.84	3.20	**16.8%**	46.23	46.25	**-0.1%**
stringsearch	2.80	1.93	**30.9%**	0.81	0.59	**27.5%**	6.82	6.67	**2.1%**
untoast	1.10	1.09	**0.6%**	2.60	2.31	**11.0%**	29.52	29.32	**0.7%**
epic	31.66	9.83	**68.9%**	5.00	3.72	**25.5%**	36.35	37.28	**-2.6%**
ospf	0.73	0.63	**14.3%**	1.54	1.36	**12.0%**	5.19	5.07	**2.3%**
susan	6.53	6.48	**0.7%**	1.18	0.90	**23.6%**	15.55	15.39	**1.0%**
Average	-	-	**28.9%**	-	-	**18.4%**	-	-	**0.7%**

Table 4-4. Simulation time for three exploration strategies: *Exhaustive*, *FEC* and *FECD*

Benchmark	**Exhaustive**		**FEC**			**FECD**		
	Num. of Simations	Exploration Time (hh:mm)	Num. of Simations	Exploration Time (hh:mm)	**Improv. to Exhaustive**	Num. of Simations	Exploration Time (hh:mm)	**Improv. to Exhaustive**
fft	1728	15:58	464	04:01	**73.1%**	232	02:01	**86.6%**
cjpeg	5616	12:28	2132	05:44	**62.0%**	1326	03:57	**76.4%**
djpeg	2592	02:36	984	01:37	**62.0%**	468	00:17	**81.9%**
pegwit	4860	20:25	1305	05:29	**73.1%**	652.5	03:45	**86.6%**
rijndael	1620	09:51	495	03:42	**69.4%**	247.5	01:21	**84.7%**
stringsearch	1728	02:01	480	01:34	**72.2%**	240	00:17	**86.1%**
untoast	1404	08:57	377	02:08	**73.1%**	188.5	01:04	**86.6%**
epic	27864	169:38	8256	50:58	**70.4%**	3741	23:38	**86.6%**
ospf	4968	07:47	2162	03:57	**56.5%**	1081	01:29	**78.2%**
susan	7452	23:11	2691	08:22	**63.9%**	1345.5	04:11	**81.9%**
Average	5983	27:41	1934	08:09	**67.6%**	952	04:54	**83.6%**

There are three important aspects in our results:

(1) Although we set the runtime threshold to be 5% (same as the threshold used in [39]) in our data partitioning algorithm, all benchmarks have far better performance than the threshold. In fact, many of them even have performance improvement. The is due to the fact that although PPC has the potential to cause performance degradation, DCR can find the suitable cache configurations to hide the performance penalty. This further demonstrates the effectiveness of our DCR+PPC approach.

(2) *DCR+PPC(VulMin)* can also reduce energy consumption, and *DCR+PPC(EnergyMin)* can also reduce vulnerability. That is because *DCR+PPC(VulMin)* and *DCR+PPC(EnergyMin)* are two Pareto-optimal points among all explored configurations. For example, *DCR+PPC(VulMin)* has vulnerability as the primary goal (the first dimension), while energy is still constrained to be better than base configuration (the second dimension).

(3) For benchmark *pegwit*, the two tables report the same numbers because the two optimization explorations have chosen the same cache configurations (also with the same partitioning scheme).

4.3.3 Fast Exploration

This subsection presents results for fast exploration of fewer cache configurations and/or fewer data partitioning schemes without compromising the quality of optimization goal. The three exploration strategies are:

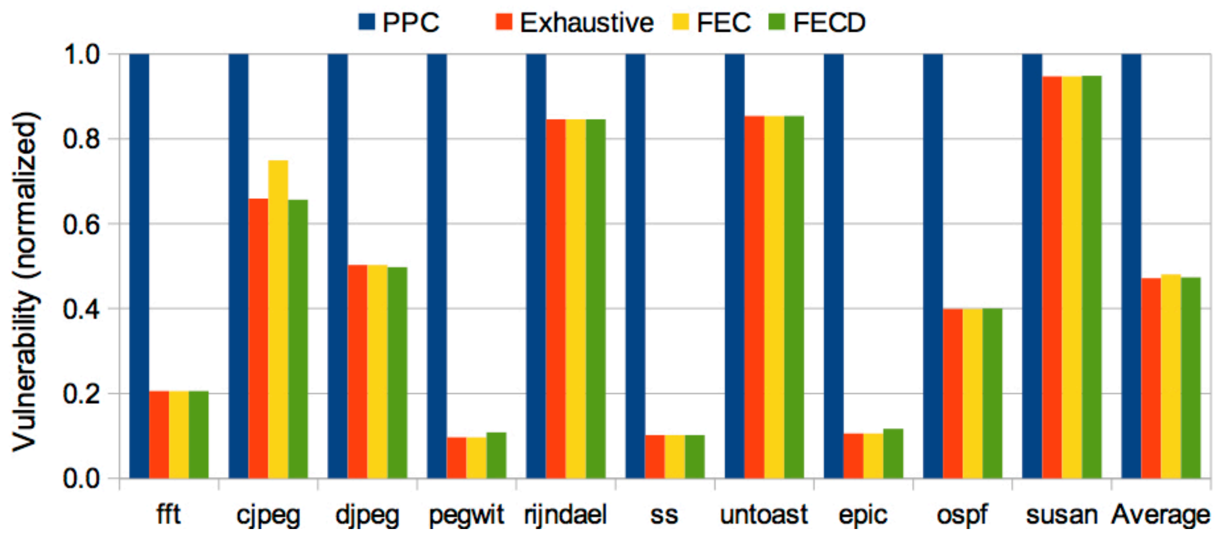
- **Exhaustive**: Exhaustive exploration on both cache configurations and data pages.
- **FEC**: Fast exploration on cache configurations.
- **FECD**: Fast exploration on both cache configurations and data pages.

Table 4-4 shows the required number of simulations and exploration time for each of the benchmarks. The average number of simulations is 5983 for *Exhaustive*, 1934 for *FEC*, and 952 for *FECD*. The average exploration time is 27 hours for *Exhaustive*, 8 hours for *FEC*, and 5 hours for *FECD*. *FEC* can reduce time by 67.6% (3 times speed-up), while *FECD* can reduce time by 83.6% (6 times speed-up), compared to *Exhaustive*.

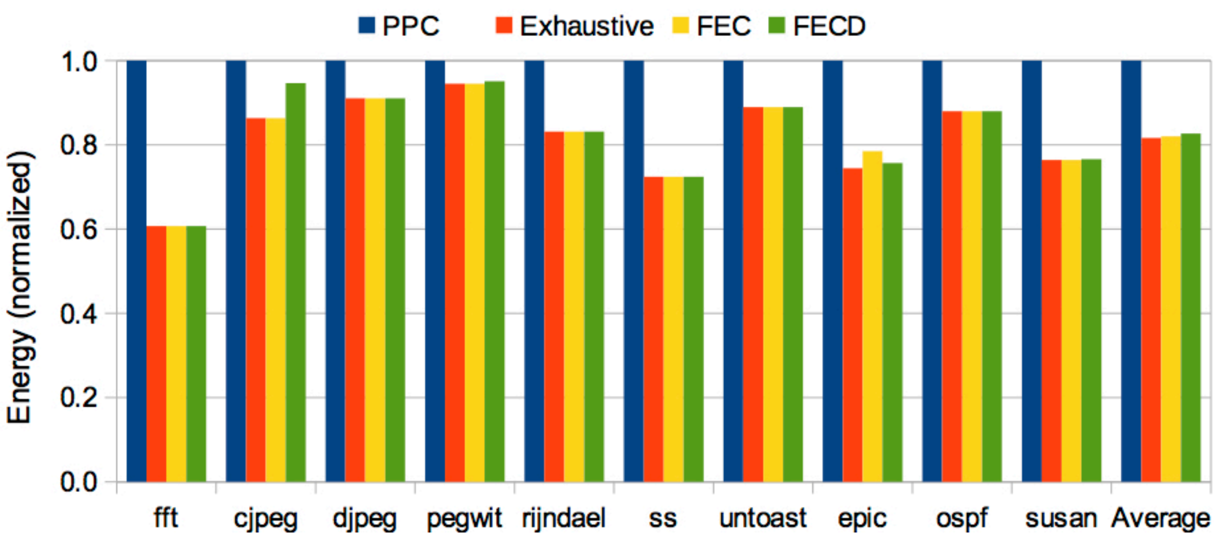
Figure 4-10 shows the quality of results for the three exploration strategies. For vulnerability minimization (Figure 4-10a), *FEC* achieves a vulnerability about 9% worse than *Exhaustive* for *cjpeg*, while it gets exactly the same vulnerability numbers for the other nine benchmarks; *FECD* can achieve a vulnerability almost as good (less than 1% worse) for *pegwit* and *epic*, while it gets exactly the same results for other eight benchmarks. Similarly for energy minimization (Figure 4-10b), *FEC* achieves energy consumption about 5% worse than *Exhaustive*, while it gets exactly results for other nine benchmarks; *FECD* achieves energy about 9% worse for *cjpeg* and less than 1% for *epic* and *susan*, while it gets exactly results for other seven benchmarks. In summary, our fast exploration techniques (*FEC* and *FECD*) can reduce the exploration time by (3X and 6X) with minor (less than 1% on average) impact on design quality.

4.4 Summary

Designing reliable embedded systems needs to consider cache vulnerability due to soft errors. In this chapter, we presented a reconfigurable cache architecture to combine the advantages of PPC (vulnerability reduction) and cache reconfiguration (energy and performance improvement). Synergistic integration of cache reconfiguration and data partitioning improves both vulnerability and energy efficiency. For vulnerability minimization, our approach can significantly reduce vulnerability (up to 90.3%, on average 52.8%), and also reduce energy consumption (up to 38.9%, on average 9.6%), with minor (on average 0.3%) performance penalty. In order to improve the scalability of our reconfiguration framework, we presented two fast exploration strategies that can achieve up to 6X speed-up, with negligible impact on the quality of exploration results.



(a) DCR+PPC(VulMin)



(b) DCR+PPC(EnergyMin)

Figure 4-10. Comparison of three exploration strategies: *Exhaustive*, *FEC* and *FECD*. (a) DCR+PPC(VulMin), (b) DCR+PPC(EnergyMin)

CHAPTER 5
VULNERABILITY-AWARE CACHE TUNING FOR MULTICORE SYSTEMS

Multicore architectures consist of multiple processor cores to improve execution performance of application programs. Multicore processor usually has on-chip caches to resolve the performance bottleneck caused by the increasing gap between processor and memory speed. In a typical multicore system, each core maintains its private L1 caches while all cores share the same L2 cache. There are many optimization techniques for multi-level on-chip caches to improve performance and energy consumption of the overall system [12, 13, 51]. With the increasing demand for high reliability and availability, vulnerability of caches due to soft errors is gaining increasing importance. Data corruption caused by soft errors can change the behavior of applications and may eventually result in a system failure. As for performance and energy improvement, it is beneficial to maintain a useful data longer in the cache. However, longer data retention can negatively impact the vulnerability or probability of data corruption due to soft errors. It is a great challenge to keep vulnerability under control while we optimize the cache subsystem for improvement in performance and energy consumption.

In this chapter, I propose a vulnerability-aware energy optimization technique which integrates cache reconfiguration (DCR) of private L1 caches and cache partitioning (CP) of the shared L2 cache. This chapter makes four important contributions: (i) We explore the inter-dependence of L1 DCR and L2 CP for performance, energy consumption as well as vulnerability; (ii) We are able to minimize energy consumption without violating both vulnerability and real-time constraints; (iii) Our fast and scalable static profiling algorithm can efficiently search the design space of L1 configurations and L2 partitions, making it feasible to find the optimal result using dynamic programming; and (iv) Our results demonstrate that our approach can provide significant energy savings compared with the base configuration as well as drastic reduction in vulnerability compared to the state-of-the-art techniques.

The remainder of the chapter is organized as follows. The architecture model and an motivational example are presented in Section 5.1. Section 5.2 presents our approach for vulnerability-aware optimization. Section 5.3 presents the experimental results. Section 5.4 concludes the chapter.

5.1 Modeling Systems with Reconfigurable Caches

In this section, we describe the modeling of multicore systems with reconfigurable caches. First, we describe the underlying multicore architecture. Next, we present the energy and vulnerability models. Then, we provide an illustrative example to motivate the need for the proposed exploration framework. Finally, we present the problem formulation.

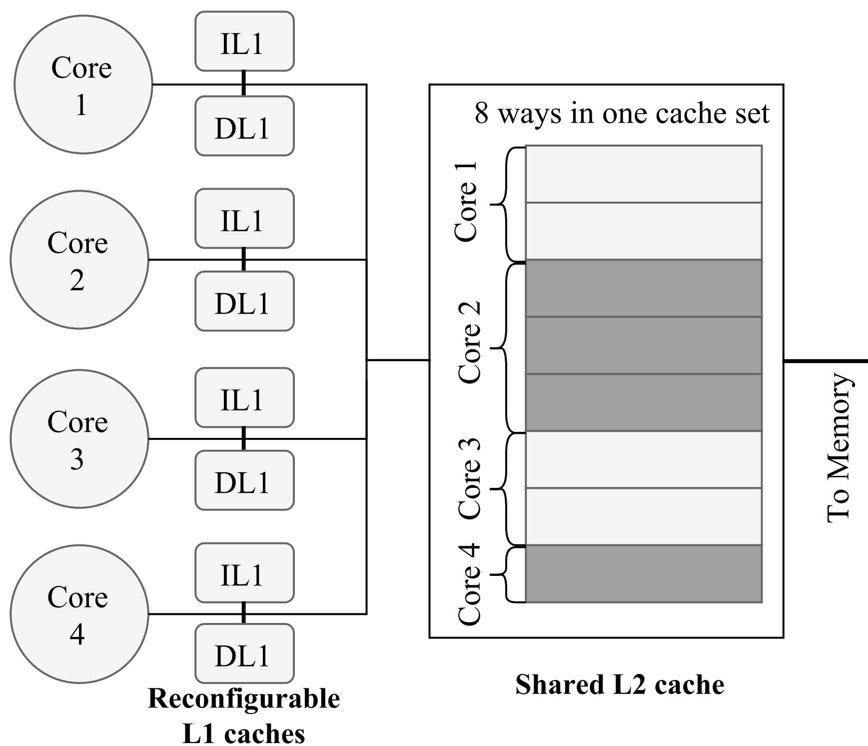


Figure 5-1. A multicore system with reconfigurable L1 caches and a partition-enabled shared L2 cache.

5.1.1 Multicore Architecture Model

Figure 5-1 shows a typical multicore system with a shared on-chip L2 cache and private L1 caches for each core. In this chapter, we assume that the private L1 caches (both IL1 and DL1) are reconfigurable, and the shared L2 cache is equipped with

way-based partitioning. The L1 caches can reconfigure its cache size, associativity, and line size. The reconfigurable cache architecture is the same as [11, 50]. The cache size is tuned by selectively shutting down the banks with gated-V_{dd} techniques. The associativity is reconfigured by logically concatenating ways. The line size can be changed by fetching multiple unit-length blocks in one access. The reconfigurable architecture is lightweight, which introduces negligible overhead [11].

The shared L2 cache with way-based partitioning [52] is illustrated in Figure 5-1. Each L2 cache set (8-way associativity as in this example) is partitioned into four parts, each of which will be assigned to one core. Each core will access only the assigned portion of the cache sets and enforce the LRU replacement policy among its individual group of ways. The number of ways assigned to a core is referred to as its *partition factor*. As shown in Figure 5-1, Core 1 has a L2 partition factor of 2. In this chapter, we use dynamic reconfiguration of the L1 caches and static partitioning of the shared L2 cache. In other words, L1 cache configurations can be tuned for each application on each core during runtime. While L2 partition factors are pre-determined for each core and remain unchanged during runtime, all applications running on that core have the same L2 partition factor.

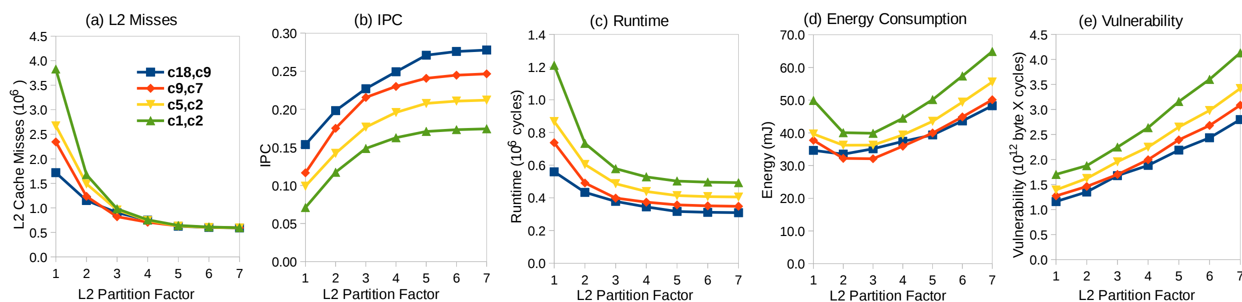


Figure 5-2. Inter-dependence of L1 DCR and L2 CP on (a) L2 Misses, (b) IPC, (c) Runtime, (d) Energy and (e) Vulnerability.

5.1.2 Energy and Vulnerability Models

The *Energy Model* is adopted from the one used in [11]. The cache energy consumption consists of static and dynamic energy: $E = E_{sta} + E_{dyn}$. The static energy

dissipation E_{sta} is computed as $E_{sta} = P_{sta} \times t$, where P_{sta} is the static power of cache. Dynamic energy dissipation E_{dyn} comes from both cache accesses and cache misses.

$$E_{dyn} = Accesses \times E_{access} + Misses \times E_{miss} \tag{5–1}$$

$$E_{miss} = E_{offchip_access} + E_{block_fill} \tag{5–2}$$

where E_{access} and E_{miss} are the energy required per cache access and per cache miss, respectively. E_{access} and E_{miss} are constant values for one specific configuration. $E_{offchip_access}$ is the energy for accessing the lower level of the memory hierarchy, and E_{block_fill} is the energy for filling the cache block with fetched data.

The *Vulnerability Model* is based on per-byte analysis of cache data with respect to the sequence of operations during its lifetime in the cache. Operations on a byte include "fill", "read", "write" and "evict". Similar to [5], the vulnerability analysis divides the lifetime of a byte into vulnerable and un-vulnerable intervals. The *vulnerable intervals* are of four types: "fill-to-read", "read-to-read", "write-to-read", "write-to-evict". We measure the vulnerability of cache as the summation of vulnerable intervals of all bytes in all cache blocks.

5.1.3 Illustrative Example

Figure 5-2 shows the impact of L1 DCR and L2 CP for benchmark *qsort* from MiBench [46]. The L2 partition factor can change from 1 to 7 in a 8-way associative L2 cache. Four pairs of cache configurations[1] for IL1 and DL1 are randomly chosen. We observe that different L1 configurations will lead to different L2 cache misses (Figure 5-2a) and pipeline throughput (i.e. IPC in Figure 5-2b). This is expected since L1 configuration determines the number accesses to the L2 cache, as well as the pipeline throughput. Secondly, as L2 partition factor w increases, L2 cache misses will decrease and eventually

[1] Here *c18* and *c9*, for example, stands for the IL1 and Dl1 using the 18-th and 9-th configuration, respectively.

converge (for $w \geq 4$) for different L1 configurations. However, the IPC shows great diversity even when L2 partition factor is large.

Figure 5-2(c-e) show the runtime, energy consumption and cache vulnerability of the benchmark, respectively. It is interesting to see that they have different patterns as L2 partition factor w increases. Runtime will decrease drastically as w increases, which is accordant with the pattern of IPC. Energy consumption will decrease to a minimal point (for $w = 3$), but it will increase when w becomes larger. This is because dynamic energy (caused by a lot of cache misses) dominates the total energy consumption when w is small, while static energy dominates when w is too large. However, vulnerability will increase with w. This is expected for two reasons: (1) a large w means that L2 cache has more valid area and is holding more data, which remains vulnerable to soft errors; (2) the decrease in cache misses (data replacement) also indicates that data are residing in the cache for longer time, which means data will have longer vulnerable intervals. While a large L2 partition facilitates performance, it might jeopardize energy consumption and vulnerability. This shows that performance, energy and vulnerability have very different (often conflicting) cache requirements.

Given the above observations, both L1 DCR and L2 CP have major impact on performance, energy consumption and vulnerability. The interesting trade-offs between them is the motivation of this chapter to explore for optimization. We exploit L1 DCR and L2 CP simultaneously for vulnerability-aware energy optimization for real-time multi-core systems.

5.1.4 Problem Formulation

We model our multicore system as follows:

- The multicore processor has m cores $\mathbb{P}$ $\{p_1, p_2, ..., p_m\}$.
- Each core has private IL1 and DL1, both of which can be reconfigured to r configurations $\mathbb{C}$ $\{c_1, c_2, ..., c_r\}$.
- The shared L2 cache is ω-way associative, which supports way-based partitioning.

- A set of n independent tasks $\mathbb{T}$ $\{\tau_1, \tau_2, ..., \tau_n\}$ with a common deadline D.

Our optimization goal is to find a reconfiguration scheme **R** for the private L1 caches and a partitioning scheme **P** for the shared L2 cache such that the overall energy consumption E is minimized without violating vulnerability constraints and task deadlines. Assume that we are given the following:

- A task mapping scheme **M**: $\mathbb{T} \rightarrow \mathbb{P}$, which assigns tasks to each core. In this chapter, we assume that the task mapper **M** is given, which can ensure that the total runtime on each core is comparable. ρ_k is the number of tasks mapped to core k.
- A reconfiguration scheme **R** for L1 caches: $C_I, C_D \rightarrow \mathbb{T}$, which assigns one IL1 and DL1 configuration to each task.
- A partitioning scheme **P** for L2 cache: $\mathbf{P} = \{w_1, w_2, ..., w_m\}$, which allocates w_k ways to core k.

For task $\tau_{k,i} \in \mathbb{T}$ (the i_{th} task on core k), $e_{k,i}(c_I, c_D, w_k)$ denotes the energy consumption of the cache subsystem when the task is executed with L1 configurations (c_I, c_D) and L2 partition factor w_k. Similarly, let $t_{k,i}(c_I, c_D, w_k)$ and $v_{k,i}(c_I, c_D, w_k)$ denote the execution time and the total vulnerability. Our minimization problem can be formulated as follows:

$$E = \sum_{k=1}^{m} \sum_{i=1}^{\rho_k} e_{k,i}(c_I, c_D, w_k) \tag{5–3}$$

is minimized subject to:

$$\max_{k=1..m} (\sum_{i=1}^{\rho_k} t_{k,i}(c_I, c_D, w_k)) \leq D \tag{5–4}$$

$$\sum_{i=1}^{\rho_k} v_{k,i}(c_I, c_D, w_k) \leq V_k, \forall k \in [1, m] \tag{5–5}$$

$$\sum_{k=1}^{m} w_k = \omega; w_k \geq 1, \forall k \in [1, m] \tag{5–6}$$

Equation 5–4 guarantees that all tasks will meet the deadline D. Equation 5–5 guarantees that the total vulnerability of the tasks on each core is constrained by the threshold V_k, which is chosen as the base case vulnerability. Equation 5–6 verifies that the partitioning scheme is valid.

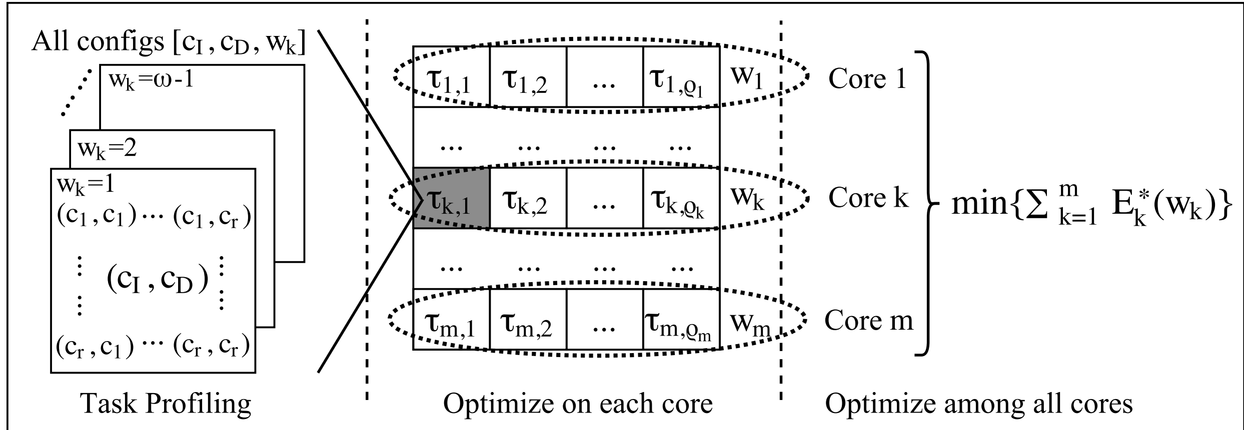


Figure 5-3. Three-step optimization: the first step statically profiles each task, the second step optimizes for each partition factor on each core to find the best L1 cache configurations, the third step combines the optimal solution on all cores to find the best L2 partition scheme.

5.2 Vulnerability-Aware DCR+CP

In this section, we present our approach which utilizes the static profiles of tasks to efficiently search the design space for the optimal energy solution. Our three-step optimization approach is illustrated in Figure 5-3, with the **first step** to profile each task, the **second step** to use a dynamic programming algorithm to optimize for all cache configurations on each core, and the **third step** to combine the optimal solutions on each core by trying out all feasible L2 partition schemes.

5.2.1 Task Profiling

Theoretically, we can do static profiling for the whole task set $\mathbb{T}$ for all possible L1 reconfiguration schemes $\mathbf{R}$ and all possible L2 partition schemes $\mathbf{P}$. However, it is not feasible to do this exhaustive exploration because of excessive simulation time. Assume that we have a four-core processor with an 8-way associative L2 cache. Each core is assigned with three tasks and the IL1 and DL1 cache each has 18 configurations [11]. The total number of architectural simulations would be $((18^2)^3)^4 * 35$. To be specific, $((18^2)^3)^4$ would be all L1 configurations (both IL1 and DL1) for the tasks (three on each core) across four cores. This needs to be multiplied by 35, which is the total number of valid L2

partition schemes according to Equation 5–6 with m=4 and ω=8. If each simulation takes only 1 minute, the total simulation time is longer than the age of the universe.

Fortunately, we can drastically reduce the complexity of static profiling by exploiting the inherent independence in our system. Tasks running on different cores are independent (with no inter-task data sharing). After introducing L2 partitioning, each task is essentially isolated on a separate core with private L1 caches and a dedicated L2 partition. Therefore, we can profile each task as if it is executed independently on a uniprocessor with a w_i-way associative L2 cache (with capacity equal to w_i/ω of the original L2). The total number of simulations required for the entire task set would be $r^2 \cdot (\omega - 1) \cdot n$, where r^2 is the number of IL1 and DL1 combinations, $(\omega - 1)$ is the number of possible L2 partition factors, and n is the total number of tasks. Using the same example above, it takes $18^2 \times 7 \times 12$ simulations with 12 tasks. For benchmarks used in our experiments, the static profiling can finish within three days. We simulate each task with all possible IL1 and DL1 cache configurations, along with all possible L2 partition factors. After static profiling, each task has a profile table with $r^2 \cdot (\omega - 1)$ entries, each of which contains the runtime, energy consumption, vulnerability for the specified L1 configurations and L2 partition factor.

Note that the profiling can be done off-line for one specific input pattern for a program. In this work, we assume that the input size remains the same but content can vary. This is a reasonable assumption for real-time embedded systems. We performed our offline analysis by varying input patterns (data values) for all the benchmarks and observed that it has minor impact on the footprint of data access. Since profile of vulnerability and energy estimation for data pages depends on the data access pattern, our static profiling will still remain effective for different input patterns. Our observations are consistent with the ones made by existing literature [11].

5.2.2 Optimization on Each Core

In order to find the optimal solution under deadline and vulnerability constraints, we first optimize on each core (find profitable L1 configurations), and then optimize across all cores (find the best L2 partition scheme). In this subsection, we explain our approach for optimization on each core. Since static partitioning of L2 is used, tasks on the same core share the same L2 partition factor w_k. This fact enables us to treat each core as a subproblem, which optimizes the energy consumption for a given core under different L2 partition factors. In other words, we find cache assignment $\mathbf{R}$ to minimize $E_k(w_k) = \sum_{i=1}^{\rho_k} e_{k,i}(c_I, c_D, w_k)$ constrained by $\sum_{i=1}^{\rho_k} t_{k,i}(c_I, c_D, w_k) \leq D$ and $\sum_{i=1}^{\rho_k} v_{k,i}(c_I, c_D, w_k) \leq V_k$, with k and w_k fixed for $\forall k \in [1, m]$ and $\forall w_k \in [1, \omega - 1]$.

This subproblem is to choose L1 configurations for each task so that the total energy is optimized with constraints. The optimization goal is to minimize energy, which can be discretized to simplify the problem. We can use a dynamic programming algorithm to search for the optimal solution. Let $e_k^{min}(w_k)$ and $e_k^{max}(w_k)$ denote the minimum possible energy ($\sum_{i=1}^{\rho_k} min\{e_{k,i}(c_I, c_D, w_k)\}$) and the maximum possible energy ($\sum_{i=1}^{\rho_k} max\{e_{k,i}(c_I, c_D, w_k)\}$) on core k, respectively. The energy consumption $E_k(w_k)$ of core k using partition factor w_k is bounded by $[e_k^{min}(w_k), e_k^{max}(w_k)]$. Let S_i^E denote the current solution found for the first i tasks. It has a cumulative energy consumption of E while the execution time and vulnerability are minimized. The execution time $T[i][E]$ for S_i^E is stored in a two-dimensional table T. The vulnerability for S_i^E is stored in another two-dimensional table V. As we try out all possible (c_I, c_D) configurations, we update the solution for S_i^E whenever runtime or vulnerability can be improved. The dynamic programming process uses the recursive formula shown in Figure 5-4 to update the two tables. The solutions for the first i tasks (the ith row in the two tables) are built upon the previous step, i.e., the $(i-1)^{th}$ row. All entries in T and V are initialized to some very large value. Based on the above recursive formula, we update the tables one row at a time for all energy values in $[e_k^{min}(w_k), e_k^{max}(w_k)]$. When the ith row is calculated, all previous

If $(T[i][E] > T[i-1][E - e_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k)$ && $V[i][E] > V[i-1][E - e_{k,i}(c_I, c_D, w_k)] + v_{k,i}(c_I, c_D, w_k))$
{
$T[i][E] = T[i-1][E - e_{k,i}(c_I, c_D, w_k)] + t_{k,i}(c_I, c_D, w_k)$
$V[i][E] = V[i-1][E - e_{k,i}(c_I, c_D, w_k)] + v_{k,i}(c_I, c_D, w_k)$
}

Figure 5-4. Recursive formula for dynamic programming

$(i-1)$ rows are already computed. The final optimal energy consumption $E^*_k(w_k)$ can be found by:

$$E^*_k(w_k) = \min\{E_k \mid T[\rho_k][E_k] \le D \;\&\&\; V[\rho_k][E_k] \le V_k\} \tag{5–7}$$

Equation 5–7 provides the solution for core k with partition factor w_k, which has minimum energy consumption with deadline and vulnerability constraints satisfied.

5.2.3 Optimization Across All Cores

In this step, we combine the solutions found on each core and search for the minimum total energy consumption E^* of all cores within all L2 partition schemes $\mathbf{P}$. For a given partition factor w_k on core k, the optimal energy $E^*_k(w_k)$ is already calculated in the first step. A valid partition scheme $\{w_1, w_2, ..., w_m\}$ is one that complies with Equation 5–6. The final total energy E^* can be found by:

$$E^* = min\{\sum_{k=1}^{m} E^*_k(w_k)\}, \quad \forall\{w_1, w_2, ..., w_m\} \in \mathbf{P} \tag{5–8}$$

Since the number of valid partition schemes is small (35 for 4-core processor with an 8-way associative L2 cache), an exhaustive search on all partition schemes is feasible. In our experiment, we assume that after the tasks on a core finish execution the core along with its private L1 caches and the designated L2 partition is turned off. Thus, E^* will be the final energy consumption for all cores running with the optimal configuration and partitioning scheme.

Algorithm 7 shows the major steps of our cache reconfiguration and partitioning approach. In the **first step** (line 1-10), for each task $\tau_{k,i}$, we simulate the task with all possible configurations $[c_I, c_D, w_k]$. We collect the energy, vulnerability and runtime

numbers of the task using the configurations and save them in its profile table. In the **second step**, our algorithm iterates to find the best L1 configurations for all tasks in core k with partition factor w_k. During each iteration (line 11 to 37), all discretized energy values (e) and all L1 cache configurations (1 to r^2) for current task $\tau_{k,i}$ are examined. The dynamic programming process of the first task on a core is shown in line 13 to 22, and that of task 2 to ρ_k is in line 23 to 34. Line 35 gets the optimal solution $E_k^*(w_k)$ for core k with partition factor w_k. In the **third step** (line 38 to 41), our algorithm iterates over all valid partitioning schemes to find the global optimal energy consumption. Line 39 gets the energy consumption for partition scheme P_j, and line 40 updates the final solution E^* with the minimal energy consumption. The time complexity for the first step is $O(m \cdot \rho_k \cdot \omega \cdot r^2)$, where m is the number of cores, ρ_k is number of tasks on each core, ω is the number of ways in L2 cache, r^2 is the number of L1 configurations. The time complexity for the second step is $O(m \cdot \omega \cdot \rho_k \cdot r^2 \cdot (e^{max} - e^{min}))$, where $e^{max} - e^{min}$ is the energy range. The time complexity for the third step is $O(m \cdot |\mathbf{P}|)$, where m is the number of cores and $|\mathbf{P}|$ is the number of partition schemes. In our experiments, our proposed approach can find the optimal solution in less than three days, which is mostly the time of the first step for profiling. Since our approach is based on static (offline) analysis and one-time effort, this is a reasonable time.

5.3 Experiments

In order to evaluate the effectiveness of our approach, we use the architectural simulator gem5 [56] in system emulation (SE) mode to simulate the multicore system as shown in Figure 5-1. We enhanced the simulator to support reconfiguration of L1 caches and way-based partitioning of the shared L2 cache. We also embedded our measurement for vulnerability of caches in the simulator, while the energy estimation of the cache subsystem is calculated with a script after simulation. We configured our system with a four-core processor running at 500MHz on each core with the TimingSimpleCPU model in gem5. The shared L2 cache supports 32KB, 8-way associative with 32-byte lines. There

are 35 valid schemes to partition the L2 ways among the four cores. The L1 caches have a base configuration as 4KB, 2-way associative with 32-byte lines, which offers effective size of 1KB, 2KB, and 4KB, and associativity of 1-way, 2-way, and 4-way, and line size of 16-byte, 32-byte and 64-byte. There are 18 configurations in total for the L1 caches[2] . We used 20 applications from the MiBench [46] and SPEC CPU2000 [57] benchmark suites as our tasks for evaluation. Table 5-1 shows the task sets used in our experiments. We choose 4 task sets which contain 2 tasks running on each core, 3 task sets which contain 3 tasks on each core, and 2 task sets which contain 4 tasks on each core. The task assignment on cores is based on the rule that each core will have comparable execution time and vulnerability.

In our results, we will compare the following three approaches:

- **CP Only**: the base configuration, which has L1 in base configurations and uniform L2 cache partitioning among cores.
- **DCP+CP[12]**: the energy-aware approach in [12] using DCR on L1 and CP on L2.
- **Our Approach**: our vulnerability-aware energy optimization approach using DCR on L1 and CP on L2.

Here **CP Only** refers to the base configuration of the system, which has uniform L2 cache partitioning among the four cores with all the L1 caches in base configuration. For our vulnerability-aware approach, the vulnerability threshold on each core is set as that of the base system (**CP Only**). We want to minimize the energy consumption while ensure that the vulnerability be at least better than the base system.

5.3.1 Deadline and Vulnerability Threshold

It is meaningful to see how deadline and vulnerability threshold affect the optimization process. Figure 5-5 shows the optimal energy consumption (i.e. $E_1^*(w_1)$ as in Equation 5–7) of core 1 using partition factor ($w_1 = 2$) for task set 9, under different

[2] It is fewer than 3^3 since not all combinations are valid [11].

deadline and vulnerability constraints. In Figure 5-5a, as we gradually vary the deadline from 4600 ms to 3600 ms, the optimal energy found by the dynamic programming algorithm will become worse. When the deadline is shorter than 3690 ms, there is no feasible solution. In Figure 5-5b, as we gradually reduce the vulnerability threshold from 8.4×10^{12} to 7.2×10^{12} bytes-cycles, the optimal energy solution will also become worse. There is no solution when vulnerability threshold is set smaller than 7.3×10^{12} bytes-cycles. In this example, we can get a converged optimal energy solution (2753 mJ) with a deadline larger than 4300 ms and a vulnerability threshold larger than 8.0×10^{12} bytes-cycles. Note that in Figure 5-5a we removed the vulnerability constraint (i.e. set vulnerability threshold as infinity) to solely investigate the effect of deadline and vice versa for Figure 5-5b.

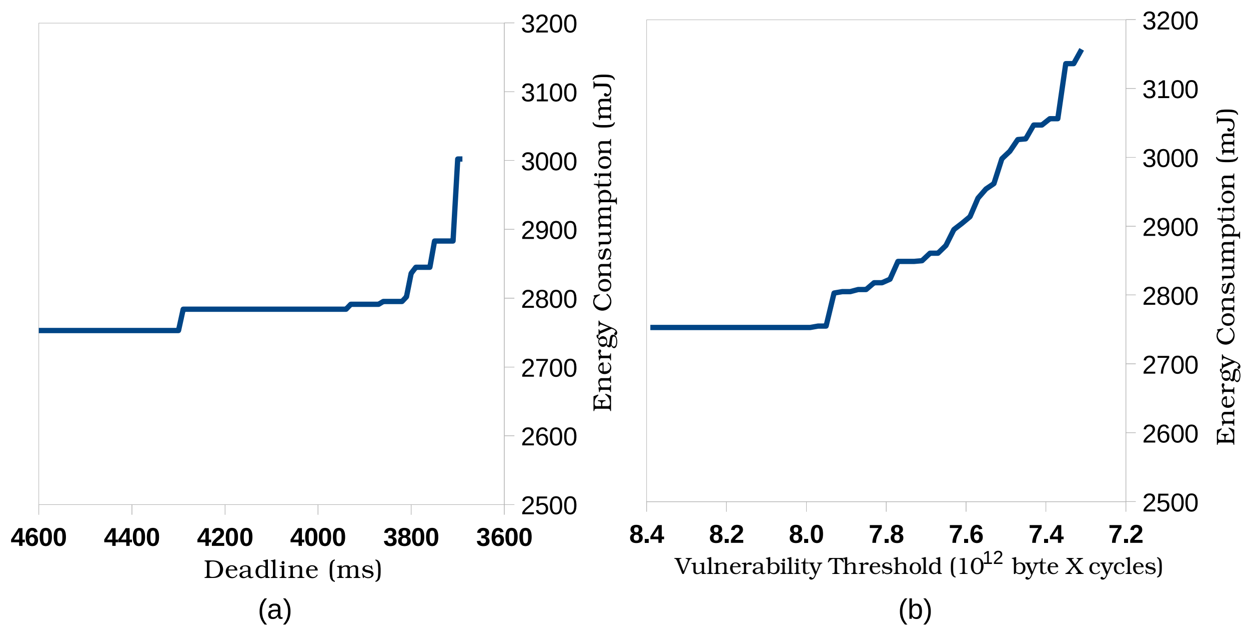
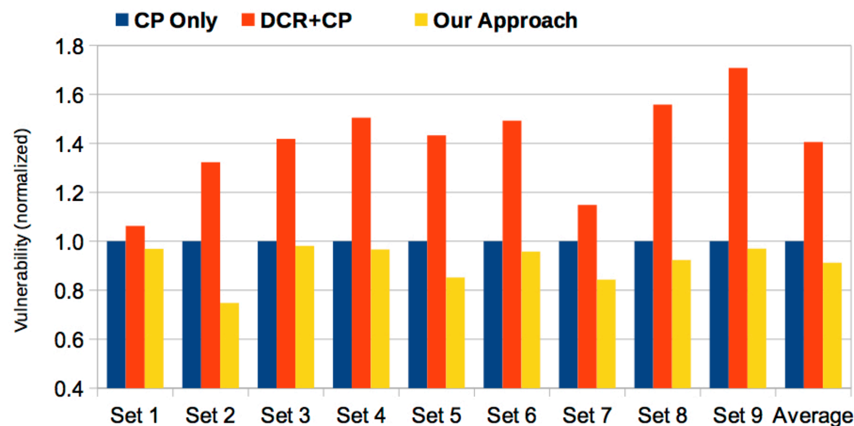


Figure 5-5. Effects of Deadline and Vulnerability Threshold.

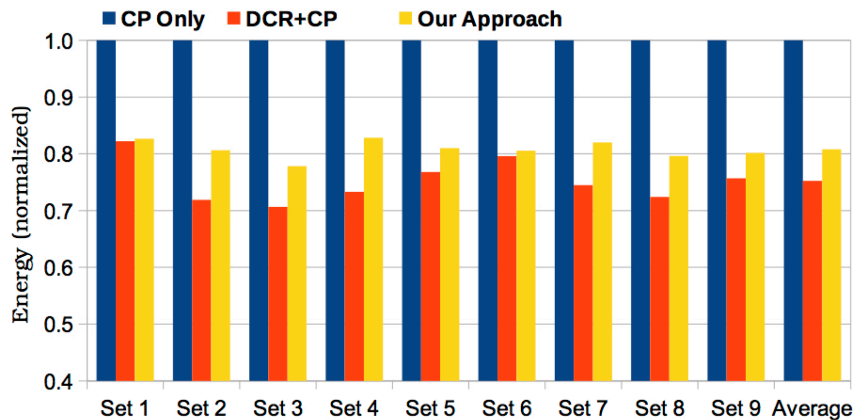
This example suggests that the choice of deadline and vulnerability threshold can affect the optimal energy solution. In our experiments, the deadline is chosen in a way so that each core can reach the converged minimum energy under the base configuration setting. The vulnerability threshold on each core is also same as the base system which

runs with uniform L2 partition and the base configuration for L1s. These settings are performed under the assumption that our approach should not be more vulnerable than the base system while improving the energy profile. This assumes that our system should be at least less vulnerable than the base system. In other words, we want our energy optimization process to be vulnerability-aware.

5.3.2 Vulnerability-Aware Energy Reduction



(a) Vulnerability



(b) Energy consumption

Figure 5-6. Comparison of vulnerability and energy consumption for the cache hierarchy. (a) Vulnerability, (b) Energy consumption.

Figure 5-6 illustrates the comparison of vulnerability and energy consumption of the nine task sets in Table 5-1. Here the vulnerability is the maximum vulnerability among four cores while energy consumption is the total energy consumption of all L1 caches and

L2 partitions. The maximum vulnerability provides an indication of the overall reliability of the cache subsystem since all the cores are independent with its private L1 caches and designated L2 partition.

Figure 5-6a shows the results for vulnerability reduction. Compared with **CP Only**, our approach reduces vulnerability by up to 25.2% and on average 8.8%. Compared with [12], our approach achieves up to 73.9% reduction in vulnerability and 49.3% on average. Figure 5-6b shows the energy savings. Compared with **CP Only**, our approach reduces energy consumption by up to 22.2% and 19.2% on average. Compared with [12], our approach consumes on average 5.6% and up to 9.5% more energy. In summary, our vulnerability-aware energy optimization can significantly reduce energy (on average 19.2%) compared with the base system. Compared with the state-of-art approach for energy optimization, we gain significant vulnerability reduction (on average 49.3%) with minor energy overhead (on average 5.6%).

In order to understand the rationale of above improvement, we would like to analyze the optimal solutions returned by Algorithm 1 for two different tasks sets. Table 5-2 and Table 5-3 show the results of L2 partition factors and [IL1, DL1] cache configurations found by our approach for task set 1 and task set 9, respectively. Task set 1 has two tasks on each core, with a partition scheme of [2,2,1,3] ways dedicated for each core. Task set 9 has four tasks on each core, with a partition scheme of [2,2,2,2]. We can see that different tasks have very different L1 configurations, which shows the necessity of DCR to suit the unique needs of a task. For a certain task, the best [IL1, DL1]configurations depend not only on the task itself (i.e. its data access patterns), but also the L2 partition factor as well as the deadline and vulnerability threshold. There are a few tasks appearing in both Set 1 and Set 9. For benchmarks *qsort*, *vpr*, *parser*, and *toast*, they have the exact same L2 partition factor and L1 configurations for the two sets. For benchmark *untoast*, Set 1 and Set 9 have chosen different L1 configurations when Set 1 (Core 3) uses a partition factor of 1 and Set 9 (Core 2) uses a partition factor of 2. Because Set 9

assigns a larger partition factor, *untoast* can execute with smaller L1 cache sizes ([1KB, 1KB]) for reducing energy under the deadline and vulnerability constraints.

Vulnerability-constrained systems can tolerate up to certain vulnerability level due to its implemented mitigation solution. Therefore, existing energy-optimization techniques (such as [12]) are not applicable on them. For example, if a system can tolerate up to 20% more vulnerability compared to the base configuration, most of the energy savings (except for Set 1 and Set 7) are meaningless since they crossed the vulnerability threshold. In other words, apparent energy benefit of [12] is not useful in practice. Therefore, our vulnerability-aware energy optimization approach is vital for multicore systems with vulnerability constraints.

5.4 Summary

Cache vulnerability is a major concern in embedded systems design due to increasing cache size and soft errors. While both vulnerability and energy optimization have received considerable attention in recent years, there are no existing works on vulnerability-aware energy optimization for multicore systems. In this chapter, we presented a vulnerability-aware energy optimization technique for real-time multicore systems. Our approach integrates dynamic cache reconfiguration (DCR) of private L1 caches and cache partitioning (CP) of the shared L2 cache. L2 CP is effective in reducing inter-core interference, while applying L1 DCR can further reduce the energy consumption under the performance and vulnerability constraints. Our task profiling technique based on the independence between tasks can drastically reduce the complexity of space exploration. Our proposed algorithm uses dynamic programming by discretizing the energy values, which can efficiently search the space to find optimal L1 cache configurations for each task and L2 cache partition factors for each core. Experimental results demonstrated that we can achieve 19.2% average energy savings compared with the base system, while drastically reduce the vulnerability (49.3% on average) compared with the state-of-art technique [12].

Algorithm 7: Vulnerability-aware DCR+CP

```
1:  for k = 1 to m do
2:      for i = 1 to ρ_k do
3:          for w_k = 1 to ω − 1 do
4:              for c_I, c_D ∈ ℂ do
5:                  Simulate task τ_{k,i} with config=[c_I, c_D, w_k]
6:                  Collect t_{k,i}(config) e_{k,i}(config) v_{k,i}(config)
7:              end
8:          end
9:      end
10: end
11: for k = 1 to m do
12:     for w_k = 1 to ω − 1 do
13:         for e = e_k^{min}(w_k) to e_k^{max}(w_k) do
14:             for c_I, c_D ∈ ℂ do
15:                 if e_{k,1}(c_I, c_D, w_k) == e then
16:                     if t_{k,1}(c_I, c_D, w_k) < T[1][e] && v_{k,1}(c_I, c_D, w_k) < V[1][e] then
17:                         T[1][e] = t_{k,1}(c_I, c_D, w_k)
18:                         V[1][e] = v_{k,1}(c_I, c_D, w_k)
19:                     end
20:                 end
21:             end
22:         end
23:         for i = 2 to ρ_k do
24:             for e = e_k^{min}(w_k) to e_k^{max}(w_k) do
25:                 for c_I, c_D ∈ ℂ do
26:                     e′ = e − e_{k,i}(c_I, c_D, w_k)
27:                     if T[i − 1][e′] + t_{k,i}(c_I, c_D, w_k)¡T[i][e]
                        &&V[i − 1][e′] + v_{k,i}(c_I, c_D, w_k)¡V[i][e]
28:                     then
29:                         T[i][e]=T[i − 1][e′] + t_{k,i}(c_I, c_D, w_k)
30:                         V[i][e]=V[i − 1][e′] + v_{k,i}(c_I, c_D, w_k)
31:                     end
32:                 end
33:             end
34:         end
35:         E_k^*(w_k) = min{e_k | T[ρ_k][e_k] ≤ D & V[ρ_k][e_k] ≤ V_k}
36:     end
37: end
38: for all P_j = {w_1, w_2, ..., w_m} ∈ P do
39:     E_j^* = Σ_{k=1}^{m} E_k^*(w_k)
40:     E^* = min(E^*, E_j^*)
41: end
42: return E^*
```

Table 5-1. TASK SETS FROM MIBENCH [46] AND CPU2000 [57] BENCHMARKS

Task set	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7	Set 8	Set 9
Core 1	qsort vpr	mcf sha	applu lucas	mgrid FFT	mcf toast sha	mgrid parser gcc	vpr sha FFT	sha mcf untoast toast	gcc stringsearch parser dijkstra
Core 2	parser toast	gcc bitcount	dijkstra swim	dijkstra parser	gcc parser stringsearch	toast FFT mcf	CRC32 lucas untoast	applu gcc bitcount ammp	untoast mcf ammp bitcount
Core 3	untoast swim	patricia lucas	ammp FFT	CRC32 swim	patricia qsort vpr	bitcount ammp applu	mgrid bitcount qsort	lucas FFT CRC32 patricia	lucas patricia qsort vpr
Core 4	dijkstra sha	basicmath swim	basicmath stringsearch	applu bitcount	basicmath CRC32 ammp	qsort dijkstra patricia	applu parser stringsearch	vpr basicmath mgrid swim	basicmath toast applu CRC32

Table 5-2. TASK SET 1: CACHE CONFIG ($[c_I, c_D, w_k]$)

Set 1	**Core 1** $w_1 = 2$	**Core 2** $w_2 = 2$	**Core 3** $w_3 = 1$	**Core 4** $w_4 = 3$
Task 1	[4KB_4W_16B, 2KB_2W_32B] **qsort**	[2KB_2W_64B, 4KB_4W_16B] **parser**	[2KB_2W_32B, 2KB_2W_16B] **untoast**	[2KB_2W_64B, 2KB_2W_16B] **dijkstra**
Task 2	[1KB_1W_64B, 4KB_4W_16B] **vpr**	[4KB_1W_64B, 1KB_1W_16B] **toast**	[4KB_4W_32B, 2KB_2W_32B] **swim**	[1KB_1W_64B, 1KB_1W_32B] **sha**

Table 5-3. TASK SET 9: CACHE CONFIG ($[c_I, c_D, w_k]$)

Set 9	**Core 1** $w_1 = 2$	**Core 2** $w_2 = 2$	**Core 3** $w_3 = 2$	**Core 4** $w_4 = 2$
Task 1	[1KB_1W_64B, 2KB_2W_16B] **gcc**	[1KB_1W_64B, 1KB_1W_16B] **untoast**	[4KB_4W_16B, 2KB_2W_32B] **lucas**	[1KB_1W_64B, 4KB_4W_16B] **basicmath**
Task 2	[4KB_1W_32B, 4KB_4W_16B] **stringsearch**	[1KB_1W_32B, 1KB_1W_16B] **mcf**	[1KB_1W_64B, 1KB_1W_16B] **patricia**	[4KB_1W_64B, 1KB_1W_16B] **toast**
Task 3	[2KB_2W_64B, 4KB_4W_16B] **parser**	[1KB_1W_64B, 1KB_1W_16B] **ammp**	[4KB_4W_16B, 2KB_2W_32B] **qsort**	[1KB_1W_64B, 1KB_1W_16B] **applu**
Task 4	[2KB_2W_64B, 2KB_2W_16B] **dijkstra**	[1KB_1W_32B, 1KB_1W_32B] **bitcount**	[1KB_1W_64B, 4KB_4W_16B] **vpr**	[2KB_1W_32B, 2KB_2W_16B] **CRC32**

CHAPTER 6
TRACE BUFFER ATTACK ON AES CIPHER

It is of utmost importance to remain fully aware of the design vulnerabilities, in the form of precise information leakage. In this chapter, we introduce **Trace Buffer Attack** (TBA), a novel attack that can be mounted with the help of post-silicon debug facilities present in a chip. System-on-Chip (SoC) designs have in-built trace buffer that traces a small set of internal signals during execution, and the traced signal values are used during post-silicon (off-line) debug. There is an inherent conflict between security and observability. While debug engineers would like to have better observability, the security experts would like to enforce limited or no visibility with respect to the security modules in a SoC design. A trade-off is typically made where trace signals are carefully selected to maintain security while providing reasonable debug capability. To the best of our knowledge, the vulnerability of trace buffers in cryptographic implementation has not been studied in the literature. We conclusively show that to achieve a certain quantifiable level of debugging ability, security is compromised. We consider AES as the benchmark algorithm for demonstrating the efficacy of this attack though, the attack can be mounted on other ciphers following the same principles outlined in this work. Our experimental results demonstrate that we can fully recover the secret key for AES-128 (iterative) implementation whereas we can partially recover the secret key for various pipelined AES implementations.

The rest of this chapter is organized as follows. Section 6.1 describes our trace buffer attack with knowledge of the RTL implementation. Section 6.2 analyzes the process to attack without any knowledge of the RTL implementation. Section 6.3 presents the experimental studies followed by proposed countermeasures in Section 6.4. The chapter is concluded in Section 6.5.

6.1 Trace Buffer Attack with RTL Knowledge

In this section, we launch the trace buffer attack assuming that the register-transfer level (RTL) implementation is available. We first describe the attack model and then introduce the proposed attack in two phases. In the first phase, we attempt to establish the correspondence between the signal values in trace buffer and variables in the AES design. In the second phase, depending on the trace buffer size and the number of cycles for which each signal is dumped, the signal values are fed to the restoration algorithm. The restoration algorithm attempts to restore internal signals and eventually recover bits in the user-specified primary key. Details of each step are elaborated in the following sections.

6.1.1 Attack Model

The proposed trace buffer attack has the following assumptions:

1. The primary key is stored in secure memory and properly maintained by key management.
2. The attacker knows the AES encryption algorithm as it is open to public.
3. High level timing information, as well as the RTL implementation of the AES, is known to the attacker.
4. The attacker has access to trigger the trace buffer recording at any time and dump out the traced content via the JTAG port after designated clock cycles.
5. The attacker does not know which signals are recorded in the trace buffer.

The assumptions that we have made for trace buffer attack are similar to the ones made in the literature for scan-chain attacks [84–86]. The primary key is assumed to be properly maintained by key management. The attacker knows the algorithmic details and the high level timing information of the cryptosystem being implemented in the device. In Assumption (3), we assume that the attacker knows the RTL implementation of AES for the attack proposed in Section 6.1. We realize that this is a very strong assumption. We remove this assumption to make it a more realistic attack in Section 6.2. Assumption (4) is similar to scan chain attacks concerning the debugging JTAG port. The attacker has

the ability to run the device under test mode, i.e. trigger the recording of trace buffer and dump out the buffer content. He can feed the circuit with designated inputs (plaintexts and fake keys) for cryptanalysis. For Assumption (5), the attacker does not know which signals are recorded in the trace buffer. It is the first challenge to be resolved if the attacker wants to launch an attack.

Compared with the scan chain attacks, the trace buffer attack introduces the following additional challenges. (1) The first step for trace buffer attack is similar to scan-based attack, which is to identify signals. For scan-based attack, the attacker knows what signals are in the scan flip-flops. The attacker's problem is to identify the structure (order) of the scan chain. However, for trace buffer attack, the attacker does not even know which signals are selected to be recorded in the buffer. (2) The number of signals traced is usually much smaller compared to the length of scan chains (especially if it is full-scan). The number of traced signals is limited, which makes it more challenging for the second step of signal restoration for trace buffer attack. (3) The trace buffer can record values over a continuous interval. Trace buffer attack has the advantage to analyze the signal values between clock cycles (between encryption rounds, while the scan chain can only scan out the signal values at one clock cycle. (4) Protection against scan-based attacks mostly focuses on scrambling the structure of scan chain. For trace buffer attack, the countermeasures have to focus on scrambling or direct encryption of the recorded signals.

6.1.2 Determine Trace Buffer Signals

If an attacker wants to steal the primary key, signal values in the trace buffer are the starting point of hacking. Unless the traced data is encrypted or debugging is authentication based, the attacker can easily dump traced data through JTAG interface. The challenge for *Trace Buffer Attack* is that the attacker does not know what signals are recorded in the trace buffer. In this section, we assume that the attacker has access to a few test chips and the RTL description of the AES design. The one-to-one mapping

between the traced signals and the registers in RTL description can be established by running some test chips and matching with RTL simulation.

Algorithm 8: MAP SIGNALS TO REGISTERS IN RTL

```
    Input: AES RTL implementation, AES test chip
    Output: Identified signals in trace buffer
 1: while true do
 2:   Select a random plaintext T_itr, a random key K_itr
 3:   Run RTL simulation with T_itr and K_itr for c cycles
 4:   Run the test chip with T_itr and K_itr for c cycles
 5:   for Each traced signal S_i in trace buffer do
 6:     Represent S_i as a vector of c values
 7:     for Each register R_j in RTL do
 8:       Represent R_j as a vector of c values
 9:       if the vectors of S_i and R_j are the same then
10:         (S_i, R_j) is a possible match
11:       end
12:     end
13:     if S_i has a unique match R_j then
14:       (S_i, R_j) is a verified match
15:     end
16:   end
17:   if Every signal in S has a unique match then
18:     Break
19:   end
20: end
21: return Identified signals in trace buffer
```

Algorithm 8 shows the process to match signals in trace buffer with registers in the RTL implementation. For each iteration, we select a random input plaintext T_{itr} and a random key K_{itr}. We run the test chip and the RTL simulation with the same key and input text for c cycles. Each traced signal will have a vector of c values stored in the trace buffer. For each traced signal, we compare its vector with vectors of all the registers from RTL simulation. If a unique match is found in the RTL simulation, this traced signal is identified in the RTL description. We repeat the process until all the traced signals are uniquely identified.

6.1.3 Signal Restoration

Let us assume that the attacker has finished the preparation in the previous step and successfully identified the signals in the trace buffer. The next step is to run the chip in the working mode with the secret primary key and take advantage of the trace buffer to initialize the attack. The attacker dumps out the signal states recorded in the buffer during online encryption, and tries to analyze the design so as to recover as many other signals as possible, and eventually obtain the primary key. In post-silicon debug, restoration of unknown signals based on trace buffer data is a crucial step in debugging. This section describes signal restoration based on trace buffer.

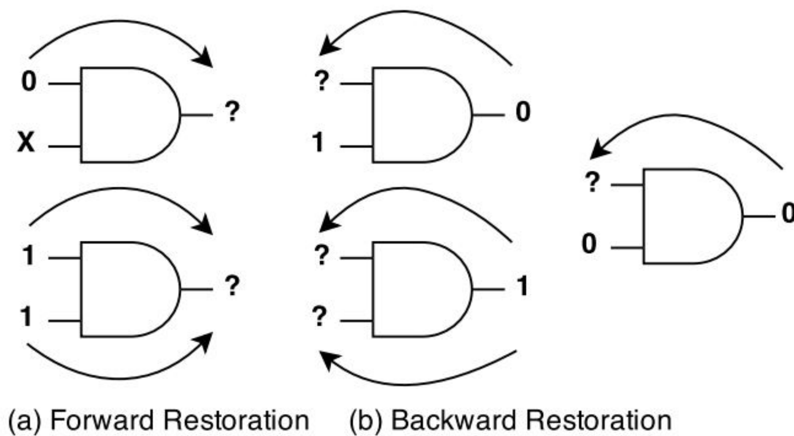


Figure 6-1. Illustration of signal restoration for an AND gate

The signals can be reconstructed from the traced signals in two directions: forward and backward restoration. Forward restoration pushes the restoration of signals from input to output, which is the process of inferring output values if some inputs are known. Backward restoration infers input values if some outputs are known. Figure 6-1 illustrates forward and backward restoration with a simple example of AND gate. Figure 6-1(a) shows forward restoration: if one of the inputs is 0, the output can be inferred to be 0; if both of the inputs are 1, the output can be inferred to be 1. Figure 6-1(b) and (c) shows backward restoration: if the output is 1, both of the inputs can be inferred to be 1. However, if the output is 0, backward restoration might not be successful as shown

in (c). The restoration process for other logic components is similar to AND gate. The restoration for registers (flip-flops) is that the state at current cycle is related to the state at previous cycle as specified by their truth tables.

Algorithm 9 outlines the major steps in a typical restoration algorithm. We first read in the AES circuit and form a hypergraph. Based on the trace buffer content, we perform forward and backward restorations to construct value assignments for un-traced nodes. We use a queue *UnderProcess* to keep track of nodes which have new values been restored. The queue is initialized with nodes from the trace buffer. Each node in the queue is processed by backward and forward restoration and nodes with newly assigned values will be put to the end of the queue. This process continues until no new assignments are created, i.e., the queue becomes empty. Although this algorithm has exponential complexity, in reality, it completes the process very fast (as demonstrated in Section 6.3) since the number of new values created decreases significantly after each iteration.

6.2 Trace Buffer Attack without RTL Knowledge

In this section, we aim to attack the AES cipher without the knowledge of its RTL implementation. We consider iterative AES-128 with a trace buffer (32×512) of width 32 and depth 512. We first identify as many signals as we can in the trace buffer, by referring to the variables in the AES encryption algorithm. We then show that the primary key can be retrieved by taking advantage of Rijndael's key scheduling,

6.2.1 Mapping Signals to Algorithm Variables

Suppose we have the AES-128 chip as in Case Study 1 (Section 6.3), we don't have the implementation at RTL level, which means that we cannot run the RTL simulation and the chip side-by-side to compare which signals are recorded in the trace buffer. What we know from the chip datasheet is that it takes 13 cycles to complete one encryption. The trace buffer contains values of 32 internal signals. The intermediate encrypted text and the round key are most beneficial for signal restoration to recover the primary key bits. The code snippet of C implementation of AES-128 is shown in Figure 6-2. The

Algorithm 9: SIGNAL RESTORATION ALGORITHM

```
Input: Trace buffer content, AES netlist
Output: Restored signal (node) values
1:  Read in the AES circuit and form a hypergraph
2:  Put all traced nodes into the UnderProcess queue
3:  Update the traced nodes with their known values (0/1)
4:  Update all other nodes with unknown values (x)
5:  while UnderProcess is not empty do
6:  |   Take a node N from the UnderProcess queue
7:  |   for each node in N's BackwardNeighbors do
8:  |   |   Backward Restoration for this neighbor node
9:  |   |   if value at any cycle is restored then
10: |   |   |   Add this neighbor node to UnderProcess
11: |   |   end
12: |   end
13: |   for each node in N's ForwardNeighbors do
14: |   |   Forward Restoration for this neighor node
15: |   |   if value at any cycle is restored then
16: |   |   |   Add this neighbor node to UnderProcess
17: |   |   end
18: |   end
19: end
20: return
```

variables $\{state[0], state[1], state[2], state[3]\}$ represent the intermediate encrypted text, and the variables $[w0\ w1\ w2\ w3]$ represent the 128-bit round key. The *for* loop of 10 iterations represents the 10 encryption rounds.

We would like to find out whether any signals (bits) in the trace buffer are from the intermediate encrypted text or the round key. It takes 13 cycles for the AES chip to complete one encryption operation, which is one initial round and 10 subsequent rounds in the C program of AES. If one bit is from the round key $[w0\ w1\ w2\ w3]$, we can represent the values of this bit over 10 rounds as a 10-bit binary string by running the C program. The resulting 10-bit string would be a substring of the same bit/signal in the trace buffer. If we apply a fixed key and a fixed plaintext when we run the test chip, a matching algorithm variable bit from the C program would actually be a 10-bit substring repeating with a period of 13 in the string of the same signal in the trace buffer.

```
void AES128(word state[], word key[]){
/* the initial round */
state[0] ^= key[0];
state[1] ^= key[1];
state[2] ^= key[2];
state[3] ^= key[3];
word y, p0, p1, p2, p3;
byte rcon = 1;
/* ten encryption rounds */
word w0 = key[0];
word w1 = key[1];
word w2 = key[2];
word w3 = key[3];
for(int i=1; i<=10; i++) {
// round-key generation
...
// four-step encryption
...
}
}
```

Figure 6-2. C Code Snippet for AES-128

Algorithm 10 shows the details about how we matched the signals from trace buffer bits to algorithm variable bits. We run the C program and the test chip with a same random plaintext T_{itr}, and a same random key K_{itr}. Each signal S_i in trace buffer is represented as a 512-bit binary string and each variable bit $V_{j,k}$ as a 10-bit binary string. We decide that $(S_i, V_{j,k})$ is a possible match if variable bit $V_{j,k}$ is a repeating pattern of S_i. The algorithm tries to identify as many signals of S as possible, and it will terminate when matched signals are uniquely identified and no more unique match can be found. The complexity of the matching algorithm is $O(W * \sum_j B_j)$, where W is the buffer width, B_j is the number of bits in variable V_j, $\sum_j B_j$ is the total number of candidate variable bits.

By applying the above method, we can identify 30 out 32 signals in the trace buffer. These 30 signals include two bits from the intermediate register, and another 28 bits from the round key register can be matched. The 28 bits from the 128-bit round key register include 1 bit from the first word, 2 bits from the third word, and 25 bits from the fourth word. More details about these signals are presented in Section 6.3.1. Note that we apply

Algorithm 10: MAP SIGNALS TO BITS IN AES VARIABLES

Input: AES C implementaion, AES test chip
Output: Identified signals in trace buffer

```
1:  while true do
2:      Select a random plaintext T_itr, a random key K_itr
3:      Run the C program of AES-128 with T_itr and K_itr
4:      Run the test chip with T_itr and K_itr
5:      for Each traced signal S_i in trace buffer do
6:          Represent S_i as a 512-bit binary string
7:          for Each variable V_j in AES algorithm do
8:              Extract V_j across the 10 encryption rounds
9:              for Each bit V_(j,k) in V_j do
10:                 Represent V_(j,k) as 10-bit binary string
11:                 if V_(j,k) is a repeating pattern in S_i then
12:                     (S_i, V_(j,k)) is a possible match
13:                 end
14:             end
15:         end
16:         if S_i has a unique match V_(j,k) then
17:             (S_i, V_(j,k)) is a verified match
18:         end
19:     end
20:     if Every signal in S have either unique or no match then
21:         Break
22:     end
23: end
24: return Identified signals in trace buffer
```

the state-of-the art signal selection algorithm to select the trace signals. In other words, we did not choose signals that would help us in trace buffer attack. This research also points to the need for having security-aware signal selection.

6.2.2 Attack by Taking Advantage of Rijndael's Key Expansion

As shown in Figure 2-3, the last step of each round is XOR with a round key. The initial round takes the primary key, and each of the following 10 rounds uses a different round key. The round key generation follows the Rijndael's key expansion algorithm to generate the next 4-word round key $[RK_{i+1,1}, RK_{i+1,2}, RK_{i+1,3}, RK_{i+1,4}]$ based on the current 4-word round key $[RK_{i,1}, RK_{i,2}, RK_{i,3}, RK_{i,4}]$.

$$
\begin{aligned}
RK_{(i+1,1)} &= RK_{(i,1)} \oplus sbox(lcs(RK_{(i,4)})) \oplus RC_i \\
RK_{(i+1,2)} &= RK_{(i,2)} \oplus RK_{(i+1,1)} \\
RK_{(i+1,3)} &= RK_{(i,3)} \oplus RK_{(i+1,2)} \\
RK_{(i+1,4)} &= RK_{(i,4)} \oplus RK_{(i+1,3)}
\end{aligned}
\tag{6–1}
$$

Equation 6–1 shows the Rijndael's key expansion algorithm. For the $(i+1)^{th}$ round, the first word $RK_{(i+1,1)}$ is the XOR of three items: the first word of ith round, the substituted word by applying a one-byte *lcs* (*Left Circular Shift*) operation and a byte-wise *sbox* substitution on the fourth word of ith round, and the round constant RC_i. The *sbox* function is byte-to-byte substitution according to a 16×16 lookup table as shown in Fig. 6-4. For the other three words $RK_{(i+1,2)}$,$RK_{(i+1,3)}$, $RK_{(i+1,4)}$, they follow the same pattern: the XOR of the word itself at i^{th} round and the previous word at $(i+1)^{th}$ round. Based on the above observation, we generalize two rules as shown in Equation 6–2, which will be useful for signal values restoration between cycles (rounds).

$$
\begin{aligned}
&\text{Rule 1:} \quad sbox(lcs(RK_{(i,4)})) = RK_{(i,1)} \oplus RK_{(i+1,1)} \oplus RC_i \\
&\text{Rule 2:} \quad RK_{(i+1,j-1)} = RK_{(i,j)} \oplus RK_{(i+1,j)},\ j = 2, 3, 4
\end{aligned}
\tag{6–2}
$$

In Rijndael's round key expansion, the fourth word of current round key is the seed word for generating the next round key, which is shown in Equation 6–1 and 6–2. The *lcs* and *sbox* operations on the fourth word are the sources to introduce unpredictable randomness to round keys. We have figured out that the trace buffer contains bits from the fourth word in the round key register in Section 6.2.1, which would be critical for us to retrieve the full key.

(1) Analysis: Assume the Fourth Word Known

Table 6-1 shows that if all the 32 bits of the fourth word of the round keys are known, the 128-bit primary key, which is all 0's for this example, can be recovered. Round keys are represented as hexadecimal digits and 'X' means 'unknown'. Assume we know

the fourth word of the round keys as shown in Table 6-1(A). We first apply Rule 2 on $RK_{(1\sim10,4)}$ (the fourth column of the Table 6-1(B)), we can retrieve the third word of all round keys except the first round, which is $RK_{(2\sim10,3)}$ (the third column). Similarly, we can retrieve the second column and the first column ($RK_{(3\sim10,2)}$ and $RK_{(4\sim10,1)}$). Now we have successfully retrieved the full round key RK_4. The Rijndael's key expansion defines the relation between two consecutive round keys, which means we can use Equation 6–1 to get the previous round key if we have the current round key. With RK_4 already retrieved, we can then get RK_3, RK_2, RK_1 and eventually RK_0, which is the primary key.

Table 6-1 shows that we would be able to retrieve RK_4 if all the 32 bits of the fourth word of the round key register are known. In fact, only the first four rows in Table 6-1 are needed to retrieve RK_4. With Rule 2, any four consecutive rounds with the fourth word known will be able to retrieve a full round key, i.e. the value of $RK_{(i\sim i+3),4}$ will lead to the recovery of full round key RK_{i+3}.

(2) Restoration from Partial Information in Trace Buffer

However, the trace buffer contains only 25 bits of the fourth word as shown in Section 6.3.1. The above approach needs four consecutive rounds with the fourth word known, while we have 7 bits missing for the fourth word of each round. If we try to brute-force all possibilities, the time complexity is $2^{7*4} = 2^{28}$. While this brute-force attack is within reasonable computation limit, we show that even that is not required if we put the *sbox* bijection property into use. The *sbox* lookup table as shown in Figure 6-4 is the core of Rijndael's key expansion. It is a bijective mapping between the bytes before and after $sbox$ substitution. The bijection property of this byte-to-byte mapping makes it possible to recover missing bits in round keys.

Rule 1 shows the relationship between $RK_{(i,1)}$, $RK_{(i+1,1)}$, $RK_{(i,4)}$ and RC_i. Given that $RK_{(i,1)}$, $RK_{(i+1,1)}$ and $RK_{(i,4)}$ are partially obtained from the trace buffer content and RC_i is a known constant. We use the example in Figure 6-3 to illustrate how Rule 1 can help recover missing bits in $RK_{(i,4)}$. In this example, $RK_{(4,1)}$ and $RK_{(5,1)}$ have 4 bits missing

Table 6-1. Recover when the fourth word of the round key register are known.

(A) Assume the fourth word of all rounds known				
RK_1	XXXXXXXX	XXXXXXXX	XXXXXXXX	62636363
RK_2	XXXXXXXX	XXXXXXXX	XXXXXXXX	F9FBFBAA
RK_3	XXXXXXXX	XXXXXXXX	XXXXXXXX	0B0FAC99
RK_4	XXXXXXXX	XXXXXXXX	XXXXXXXX	7E91EE2B
RK_5	XXXXXXXX	XXXXXXXX	XXXXXXXX	F34B9290
RK_6	XXXXXXXX	XXXXXXXX	XXXXXXXX	6AB49BA7
RK_7	XXXXXXXX	XXXXXXXX	XXXXXXXX	C61BF09B
RK_8	XXXXXXXX	XXXXXXXX	XXXXXXXX	511DFA9F
RK_9	XXXXXXXX	XXXXXXXX	XXXXXXXX	4C664941
RK_{10}	XXXXXXXX	XXXXXXXX	XXXXXXXX	6F8F188E
(B) Apply Rule 2 to recover RK_4				
RK_1	XXXXXXXX	XXXXXXXX	XXXXXXXX	62636363
RK_2	XXXXXXXX	XXXXXXXX	9B9898C9	F9FBFBAA
RK_3	XXXXXXXX	696CCFFA	F2F45733	0B0FAC99
RK_4	EE06DA7B	876A1581	759E42B2	7E91EE2B
RK_5	7F2E2B88	F8443E09	8DDA7CBB	F34B9290
RK_6	EC614B85	1425758C	99FF0937	6AB49BA7
RK_7	21751787	3550620B	ACAF6B3C	C61BF09B
RK_8	0EF90333	3BA96138	97060A04	511DFA9F
RK_9	B1D4D8E2	8A7DB9DA	1D7BB3DE	4C664941
RK_{10}	B4EF5BCB	3E92E211	23E951CF	6F8F188E
(C) Use Equation 6–1 to get $RK_3 \sim RK_1$, and RK_0 (the primary key)				
RK_0	00000000	00000000	00000000	00000000
RK_1	62636363	62636363	62636363	62636363
RK_2	9B9898C9	F9FBFBAA	9B9898C9	F9FBFBAA
RK_3	90973450	696CCFFA	F2F45733	0B0FAC99
RK_4	EE06DA7B	876A1581	759E42B2	7E91EE2B

Assume we have the fourth word of all rounds known, we can apply Rule 2 in a cascaded way and recover all bits in RK_4. From RK_4, we can use Equation 6–1 to get RK_3, RK_2, RK_1, and RK_0, which is the primary key.

and $RK_{(4,4)}$ has 5 bits missing. We apply Rule 1 as shown in line 7 and derive with the partially known $RK_{(4,1)}$, $RK_{(5,1)}$ and $RK_{(4,4)}$. After some bit-manipulation, we get *sbox* mapping from a 32-bit word to another 32-bit word in line 12∼13. In line 15, the first byte is $1001x0x1$ and we would like to figure out two unknown bits. If we decompose the byte in half, the left part is 9 and the right part could be four choices: 1, 3, 9 or B. This means we need to consider sbox(9,1), sbox(9,3), sbox(9,9) or sbox (9,B) as potential

1: Given partial round key bits:
2: $RK_{(4,1)} = (111x11100000x1101101101001xx1011)_b$
3: $RK_{(5,1)} = (011x11110010x1100010101110xx1000)_b$
4: $RK_{(4,4)} = (01xx11101001x0x11x10111000xx1011)_b$
5: $RC_4 = (00010000000000000000000000000000)_b$
6: Apply Rule 1:
7: $\mathbf{sbox}(\mathbf{lcs}(RK_{(4,4)})) = RK_{(4,1)} \oplus RK_{(5,1)} \oplus RC_4$
8: $\longrightarrow$
9: $\mathbf{sbox}(\mathbf{lcs}(01xx11101001x0x11x10111000xx1011))$
10: $= 100x00010010x0001111000111xx0011$
11: $\longrightarrow$
12: $\mathbf{sbox}(1001x0x1\ 1x101110\ 00xx1011\ 01xx1110)$
13: $= 100x0001\ 0010x000\ 11110001\ 11xx0011$
14: For each byte, check the *sbox* lookup table
15: $sbox([1001x0x1]) = [100x0001]$
16: $\rightarrow sbox([10010001] = [10000001]$
17: $sbox([1x101110]) = [0010x000]$
18: $\rightarrow sbox([11101110] = [00101000]$
19: $sbox([00xx1011]) = [11110001]$
20: $\rightarrow sbox([00101011] = [11110001]$
21: $sbox([01xx1110]) = [11xx0011]$
22: $\rightarrow sbox([01111110] = [11110011]$
23: Missing bits of $RK_{(4,4)}$ recovered by $sbox$ lookup table:
24: $RK_{(4,4)} = (01111110100100011110111000101011)_b$

Figure 6-3. An example showing the recovery of missing bits in $RK_{(4,4)}$ by using *sbox* lookup table (Rule 1).

matches. However, the right hand side of line 15 indicates that the expected value $100x0001$ can be either 81 or 91. Among the four possible choices, only sbox(9,1) fits the requirement. Therefore we get the unique mapping for the first byte as sbox(9,1) = 81, i.e., $sbox([10010001]) = [10000001]$ as shown by the yellow circle (Row 9 and Column 1) in the lookup table. As Figure 5 shows, using similar lookup, we can identify the other three bytes, i.e., $sbox([11101110]) = [00101000]$, $sbox([00101011]) = [11110001]$, and $sbox([01111110]) = [11110011]$.

Algorithm 11 shows the steps to restore the primary key from the available trace buffer content. In Step 1, we first apply Rule 2 in a cascaded way to get partial bits of the first word in round keys. In Step 2, we then apply Rule 1 and the unique mapping

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 6-4. AES *sbox* lookup table (the numbers are in hexadecimal format)

property of *sbox* to further recover missing bits of the fourth word in round keys. After using *sbox*, the following steps will be very similar to the example shown in Table 6-1. In Step 3, we re-apply Rule 2 in a cascaded way to get a full round key. Finally in Step 4, we use Equation 6–1 to push back from that round key and eventually get the primary key. Section 6.3-A shows detailed experimental results after each step.

The most critical part in Algorithm 11 is Step 2, i.e., applying Rule 1 to recover missing bits in $RK_{(i,4)}$. Note, it is also possible that multiple candidate mappings are available in the lookup table for the partially known bytes. In that case, we have to evaluate all possible mappings. Our experiments with different random numbers suggest that the chances of multiple candidates are very rare.

6.3 Experimental Results

We applied our trace buffer attack on the AES Verilog implementations (the iterative AES-128, and the pipelined AES-128, AES-192 and AES-256 [?]) from the OpenCores

Algorithm 11: RESTORE MISSING BITS IN ROUND KEYS

Input: Identifid signals in round keys from trace buffer
Output: Restored round key bits

```
1:  Update identified bits with values (1/0) from trace buffer
2:  Update all other bits with unknown values (x)
    /* Step 1: Apply Rule 2 */
3:  for j ← 4 to 2 do
4:      for i ← 1 to 9 do
5:          RK(i+1,j−1) = RK(i,j) ⊕ RK(i+1,j)
6:      end
7:  end
8:  /* Step 2: Apply Rule 1 */
9:  for i ← 4 to 9 do
10:     Use the bijection property of sbox to recover missing bits in RK(i,4)
11: end
12: /* Step 3: Apply Rule 2 one more time */
13: for j ← 4 to 2 do
14:     for i ← 1 to 9 do
15:         RK(i+1,j−1) = RK(i,j) ⊕ RK(i+1,j)
16:     end
17: end
18: /* Step 4: Use Equation 6–1 to get the primary key */
19: for i ← 9 to 1 do
20:     RK(i−1,2) = RK(i,2) ⊕ RK(i,1)
21:     RK(i−1,3) = RK(i,3) ⊕ RK(i,2)
22:     RK(i−1,4) = RK(i,4) ⊕ RK(i,3)
23:     RK(i−1,1) = RK(i,1) ⊕ sbox(lcs(RK(i−1,4))) ⊕ RC_{i−1}
24: end
25: return PrimaryKey = RK_0
```

website. The Synopsys Design Compiler is used to synthesize the RTL implementation into a gate-level netlist. We developed C++ code to simulate the gate-level circuits and implement the signal restoration algorithms. The experiments were conducted on a computer with AMD Opteron 2.4GHz core and 32GB memory. We used signal selection algorithm in [79] to select trace signals for the trace buffers since it produces signals that can maximize observability compared to the other signal selection techniques.

6.3.1 Case Study 1: Iterative AES-128

The iterative AES-128 design has 530 flip-flops and about 25,000 basic logic gates. The 530 flip-flops (registers) include:

- *ld_r*, *done*, which are one-bit control signals.
- *dcnt[0..3]*, which is a 4-bit register keeping track of the encryption rounds.
- *text_in_r[0..127]*, which is a 128-bit register holding the plaintext.
- *w0[0..31]*, *w1[0..31]*, *w2[0..31]*, and *w3[0..31]*, which are 32-bit each, holding the round keys.
- *sa00[0..7], sa01[0..7], sa02[0..7], sa03[0..7], sa10[0..7], sa11[0..7], sa12[0..7], sa13[0..7], sa20[0..7], sa21[0..7], sa22[0..7], sa23[0..7], sa30[0..7], sa31[0..7], sa32[0..7], sa33[0..7]*, which are 8-bit registers holding intermediate encrypted text in bytes.
- *u0.rcon[24..31]* and *u0.r0.rcnt[0..3]*, which are 12 temporary registers in the key expansion unit.
- *text_out[0..127]*, which is a 128-bit register holding the ciphertext.

(1) Attack without RTL Implementation

As described in Section 6.2, we assume that we don't have the RTL implementation. We first use Algorithm 10 to guess which bits of the round keys are recorded in the (32×512) trace buffer. We are able to recognize 28 bits from the round key, including 1 bit from the first word (w0[14]), 2 bits from the third word (w2[17] and w2[29]), and 25 bits from the fourth word (w3[0-3, 6-13, 15-16, 18, 20-27, 30-31]). We conduct the restoration process according to Algorithm 11. Table 6-2 shows intermediate results after each step of the restoration process. First, we apply Rule 2 (the relation between different words) to restore missing bits in the fourth word, which results in Table 6-2(B). We then apply Rule 1 (the unique mapping property of $sbox$ lookup table) to get Table 6-2(C). We apply Rule 2 again to get a full round key RK_7, which results in Table 6-2(D). From RK_7, we can use Equation 6–1 to get $RK_6 \sim RK_1$ and eventually get the primary key RK_0, which is all 0's in this case.

Table 6-2. Restore the primary key from the available trace buffer content.

(A) Step 0: Bits from the trace buffer

RK_1	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxxx0xxxxxxxxxxxxxxxxx	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx0xxxxxxxxxxxxxxxxx	00xx10110000x1x11x10110010xx1001
RK_4	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx11101001x0x11x10111000xx1011
RK_5	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	11xx00110100x0x11x01001010xx0000
RK_6	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx10101011x1x01x01101110xx0111
RK_7	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	11xx01100001x0x11x11000010xx1011
RK_8	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx00010001x1x11x11101010xx1111
RK_9	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx0xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx11000110x1x00x00100101xx0001
RK_{10}	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx0xxxxxxxxxxxxxxxxx	01xx11111000x1x10x01100010xx1110

(B) Step 1: Apply Rule 2 in a cascaded way

RK_1	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01xx11101001x0x11x10111000xx1011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	100x11011101x0100x11110010xx1011	11xx00110100x0x11x01001010xx0000
RK_6	111x11000110x0010100101110xx0101	000x01000010x1010x11010110xx1100	100x10011111x1110x00100100xx0111	01xx10101011x1x01x01101110xx0111
RK_7	001x00010111x1010001011110xx0111	001x01010101x0000x10001000xx1011	101x11001010x1110x10101100xx1100	11xx01100001x0x11x11000010xx1011
RK_8	000x11101111x0010000001100xx0011	001x10111010x0010x10000100xx1000	100x01110000x1100x00101000xx0100	01xx00010001x1x11x11101010xx1111
RK_9	101x00011101x1001101100011xx0010	100x10100111x1011x11100111xx1010	000x11010111x0111x11001111xx1110	01xx11000110x1x00x00100101xx0001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110

(C) Step 2: Apply Rule 1 and sbox to recover

RK_1	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01111110100100011110111000101011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	100x11011101x0100x11110010xx1011	11110011010010111001001010010000
RK_6	111x11000110x0010100101110xx0101	000x01000010x1010x11010110xx1100	100x10011111x1110x00100100xx0111	01101010101101001001101110100111
RK_7	001x00010111x1010001011110xx0111	001x01010101x0000x10001000xx1011	101x11001010x1110x10101100xx1100	11000110000110111111000010011011
RK_8	000x11101111x0010000001100xx0011	001x10111010x0010x10000100xx1000	100x01110000x1100x00101000xx0100	01010001000111011111101010011111
RK_9	101x00011101x1001101100011xx0010	100x10100111x1011x11100111xx1010	000x11010111x0111x11001111xx1110	01001100011001100100100101000001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110

(D) Step 3: Apply Rule 2 again to recover

RK_1	xxxxxxxxxxxxxxxxx1xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xx1xxxxxxxxxxx1xxxxxxxxxxxxxxxxx	01xx00100110x0x10x10001101xx0011
RK_2	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	100x10111001x0001x01100011xx1001	11xx10011111x0x11x11101110xx1010
RK_3	xxxxxxxxxxxxxxxxx0xxxxxxxxxxxxxx	011x10010110x1001x00111111xx1010	111x00101111x1000x01011100xx0011	00xx10110000x1x11x10110010xx1001
RK_4	111x11100000x1101101101001xx1011	100x01110110x0100x01010110xx0001	011x01011001x1100x00001010xx0010	01111110100100011110111000101011
RK_5	011x11110010x1100010101110xx1000	111x10000100x1000x11111000xx1001	10001101110110100111110010111011	11110011010010111001001010010000
RK_6	111x11000110x0010100101110xx0101	00010100001001010111010110001100	10011001111111110000100100110111	01101010101101001001101110100111
RK_7	00100001011101010001011110000111	00110101010100000110001000001011	10101100101011110110101100111100	11000110000110111111000010011011
RK_8	00001110111110010000001100110011	00111011101010010110000100111000	10010111000001100000101000000100	01010001000111011111101010011111
RK_9	10110001110101001101100011100010	10001010011111011011100111011010	00011101011110111011001111011110	01001100011001100100100101000001
RK_{10}	101x01001110x1110101101111xx1011	001x11101001x0101x10001000xx0001	001x00111110x0010x01000111xx1111	01xx11111000x1x10x01100010xx1110

(E) Step 4: Apply Equation 6–1 to push back to get RK_0 (which is the primary key)

RK_0	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
RK_1	01100010011000110110001101100011	01100010011000110110001101100011	01100010011000110110001101100011	01100010011000110110001101100011
RK_2	10011011100110001001100011001001	11111001111110111111101110101010	10011011100110001001100011001001	11111001111110111111101110101010
RK_3	10010000100101110011010001010000	01101001011011001100111111111010	11110010111101000101011100110011	00001011000011111010110010011001
RK_4	11101110000001101101101001111011	10000111011010100001010110000001	01110101100111100100001010110010	01111110100100011110111000101011
RK_5	01111111001011100010101110001000	11111000010001000011111000001001	10001101110110100111110010111011	11110011010010111001001010010000
RK_6	11101100011000010100101110000101	00010100001001010111010110001100	10011001111111110000100100110111	01101010101101001001101110100111

There are 28 bits available in the trace buffer, as shown in Table 6-2(A). We first apply Rule 2 to get Table 6-2(B). We then apply Rule 1 to get Table 6-2(C), and apply Rule 2 again to get a full round key RK_7 shown in Table 6-2(D). From RK_7, we can use Equation 6–1 to get RK_6, ..., RK_1 and eventually get the primary key (RK_0), which is all 0's in this case.

The attack can successfully retrieve the full key without RTL knowledge. There are two major reasons why this attack works so well. Firstly, 28 bits of the round key registers are recorded in the trace buffer. The signal selection algorithm in [79] only greedily choose signals that are best for observability. The signals in the round key registers

happen to be of highest restoration capability for observing other internal signals. The blindness of selection of these round key signals contributes to information leakage, as well as high observability. Secondly, the bijection property of *sbox* function plays a critical role in recovering the missing bits in the fourth word of the round keys. However, if too many bits from the round key were not recorded in the buffer, we might need a lot more brute-force effort in Step 2 of Algorithm 11 to verify missing bits when using the *sbox* lookup table. In the next section, we will see that the attack with RTL knowledge is more powerful in recovering primary key bits.

(2) Attack with RTL Implementation

We explore different trace buffer sizes with buffer widths of 8, 16, and 32, buffer depth (traced cycles) of 64, 128, 256 and 512 in our experiments. The signals recorded in the trace buffer are identified by using methods detailed in Section 6.1.2 with the help of RTL implementation. The identified signals for each buffer width is as follows:

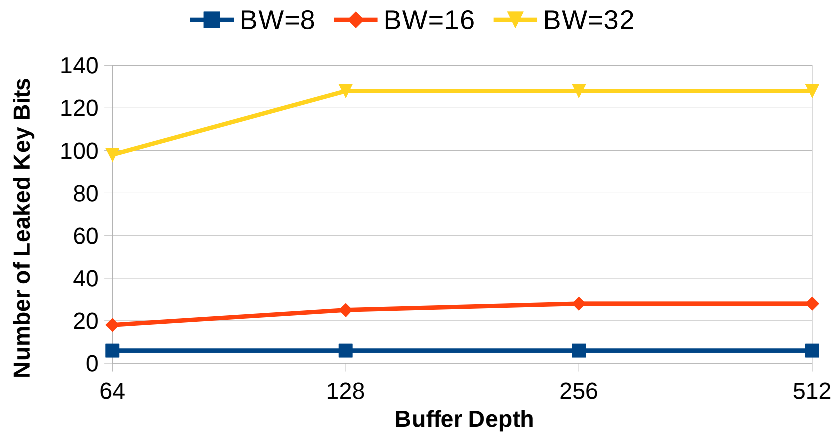
- BufferWidth=8: {dcnt[2], ld_r, w3[2], w3[1], w3[30], w3[27], w3[17], w3[13]}
- BufferWidth=16: {dcnt[2], ld_r, w3[4], w3[29], w3[27], w3[23], w3[22], w3[18], w3[16], w3[15], w3[14], w3[13], w3[12], w3[10], w1[9], w3[8]}
- BufferWidth=32: {dcnt[2], ld_r, sa03[7], sa13[7], w3[7], w3[6], w3[3], w3[2], w3[1], w3[31], w3[30], w2[29], w3[27], w3[26], w3[25], w3[24], w3[23], w3[22], w3[21], w3[20], w3[18], w2[17], w3[16], w3[15], w0[14], w3[13], w3[12], w3[11], w3[10], w3[9], w3[8], w3[0]}

Table 6-3 shows our results of trace buffer attack on the iterative AES-128 cipher. The trace buffers with a buffer width of 32 and a buffer depth no less than 128 are able to recover the full primary key in a few minutes.

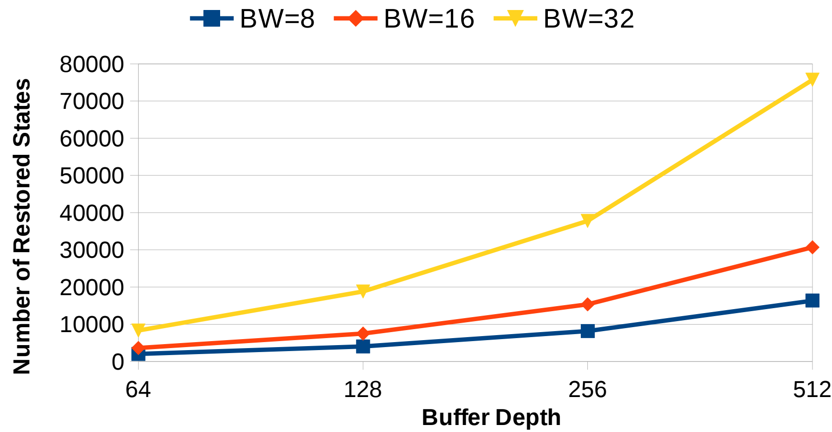
Figure 6-5(a) shows the number of bits in the user key leaked with different buffer sizes. Figure 6-5(b) shows the total number of internal states restored (debug observability) during restoration. The number of restored primary key bits increases with bigger buffer width. For the same buffer width, the number of restored key bits increases slightly as the trace cycles increase, and it will be saturated after buffer depth is

Table 6-3. Signal restoration for iterative AES-128.

BufferWidth \ BufferDepth		64	128	256	512
8	**leaked key** (bits)	6	6	6	6
	memory (MB)	116.4	161.4	252.0	432.0
	time (mm:ss)	0:27.75	0:56.07	1:50.35	3:43.26
16	**leaked key** (bits)	18	25	28	28
	memory (MB)	116.4	161.4	252.0	432.0
	time (mm:ss)	0:27.82	0:55.94	1:51.00	3:44.10
32	**leaked key** (bits)	98	128	128	128
	memory (MB)	116.4	161.4	252.0	432.0
	time (mm:ss)	0:28.01	0:55.98	1:52.81	3:51.38



(a) Number of primary key bits leaked



(b) Number of flip-flop states restored

Figure 6-5. Security and observability trade-off using different buffer widths and buffer depths.

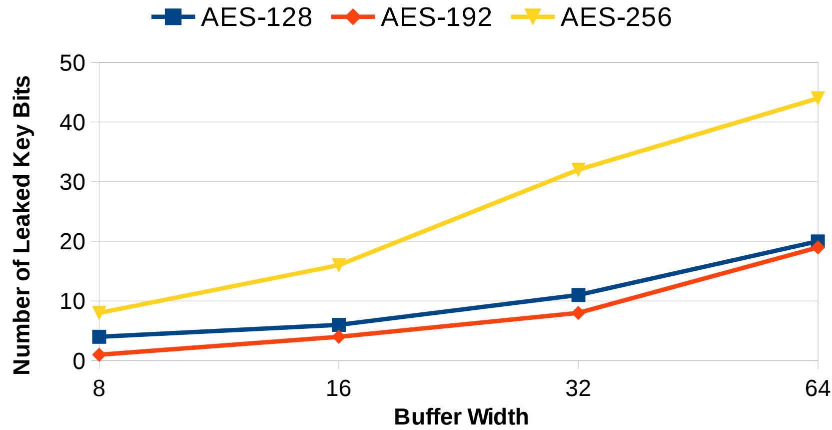
big enough (256 cycles or more). The 8×512, 16×512 and 32×512 trace buffer can respectively restore 6, 28 and 128 bits of the primary key. The fact that the 32×512 trace buffer can restore all 128-bit primary key is not surprising. The success of recovering the full primary key is due to the observability provided by the trace buffer. The iterative AES-128 design[1] has relatively short pathways with only 530 flip-flops in total. The 32 signals selected out of the 530 flip-flops is the set of signals which could offer best observability to the debugger.

The attack with RTL implementation is more powerful than without RTL in two ways. First, the attack with RTL can identify all signals traced in the buffer, which means the attack with RTL has more information to start with. Second, the restoration in Algorithm 9 (with RTL knowledge) can deterministically propagate values forward and backward in the AES circuit, while the restoration in Algorithm 11 (without RTL) would need a lot more brute-force effort to test and verify all possible mappings if the *sbox* lookup table cannot find a unique mapping.

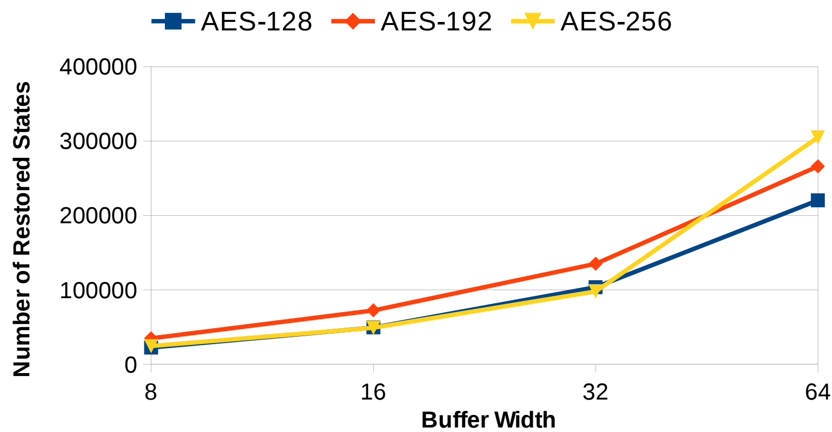
6.3.2 Case Study 2: Pipelined AES Ciphers

The main difference from the iterative version is that the pipelined implementation unrolls all the encryption rounds to be independent hardware units, which makes the pipelined version about 10-15 times as large as the iterative. For example, the pipelined AES-128 cipher has 6720 flip-flops and about 290,000 logic gates, which is roughly 10 times (10 encryption rounds) as large as the iterative AES-128. This poses a greater challenge for the restoration process, because many signal values are not inferable due to the long pathways between the known signals. Only signals that are very close to the input can be propagated backward and possibly restore the primary key bits.

[1] For iterative implementation, the restoration is clearly able to recover the key and we expect the same trend to follow for AES-192 and AES-256.



(a) Number of primary key bits leaked



(b) Number of flip-flop states restored

Figure 6-6. Pipelined AES-128, AES-192, and AES-256 ciphers: security and observability trade-off.

We explore different trace buffer sizes with buffer widths of 8, 16, 32 and 64, buffer depth of 512 in our experiments. We set the buffer depth to be 512 cycles, which should be suitable for the pipelined AES ciphers. Table 6-4 shows the experimental results on the pipelined implementation of AES-128, AES-192, and AES-256 ciphers by using the attack method with RTL knowledge. For a buffer width of 64, we are able to respectively restore 20, 19 and 44 bits of the primary key for AES-128, AES-192 and AES-256 in a few hours.

Figure 6-6 shows our experimental results of pipelined AES ciphers as we increase the trace buffer width. As the trace buffer width increases, both observability and the

leaked number of key bits increase. The restoration algorithm is not able to restore the full primary key for any of the pipelined AES ciphers. Nevertheless, considerable knowledge about the key is gained, which does not suffice to recover the secret though, can aid other modes of cryptanalysis.

Table 6-4. Pipelined AES-128, AES-192 and AES-256.

BufferWidth \ AESciphers		AES-128	AES-192	AES-256
8	**leaked key** (bits)	4	1	8
	memory (GB)	4.66	5.37	6.56
	time (h:mm:ss)	3:51:45	4:29:05	6:38:06
16	**leaked key** (bits)	6	4	16
	memory (GB)	4.66	5.37	6.56
	time (h:mm:ss)	3:44:14	4:12:22	6:22:59
32	**leaked key** (bits)	11	8	32
	memory (GB)	4.66	5.37	6.56
	time (h:mm:ss)	3:19:12	4:10:25	6:31:08
64	**leaked key** (bits)	20	19	44
	memory (GB)	4.66	5.37	6.56
	time (h:mm:ss)	3:42:02	4:08:43	6:03:15

6.4 Proposed Countermeasures

Trace buffer attack is possible because the attacker can observe the internal values of the circuit by taking advantage of the trace buffer used for DfT. One approach to obtain a secure IC is to blow test circuitry [83] after production test. This technique is broadly used in the smartcard community, which guarantees that the chip secrecy will not be abused as a test engineer could do. However, it is not acceptable from the SoC point of view because this technique disables the test mode activation after production test. This contradicts the purpose of trace buffer for online monitoring and offline debugging in post-silicon debug.

In scan chain based DfT, several solutions have been proposed so that scan chains can provide visibility without compromising security [86–88]. Most of these approaches scrambles the structure of scan chain and make the scanned outputs difficult or impossible for the attacker to comprehend. Paul et al. [87] scramble the scan chain by reordering

the scan cells and only the authorized user can get the correct order. Sengar et al. [88] insert inverters to scan chains to make it difficult for attackers to understand the internal scan structure. Another secure scan architecture is proposed in [86], which reorganizes the scan chain in a tree structure. However, these techniques cannot be directly applied to trace buffer because we don't have a chain-like structure in the trace buffer. The ultimate goal of the countermeasure is to protect the content of the trace buffer. In fact, any approach that can encrypt a block of memory will be applicable here. We explore a LFSR-based approach and a PUF-based approach and compare the pros and cons of these two approaches.

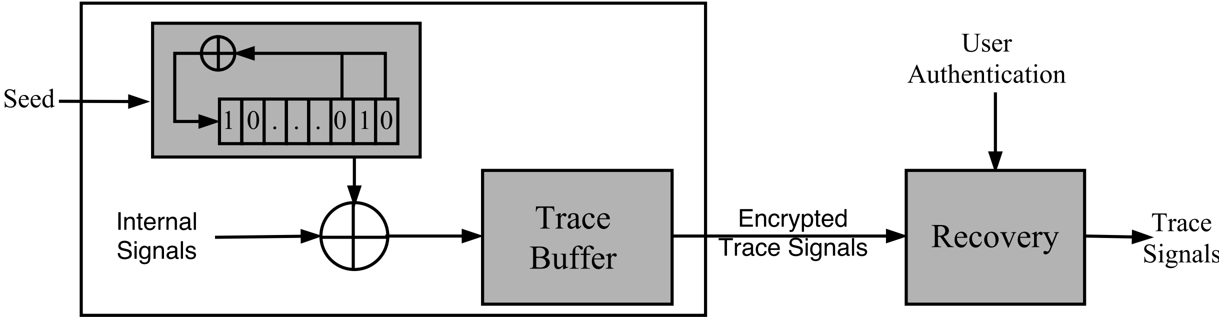


Figure 6-7. LFSR-based Countermeasure

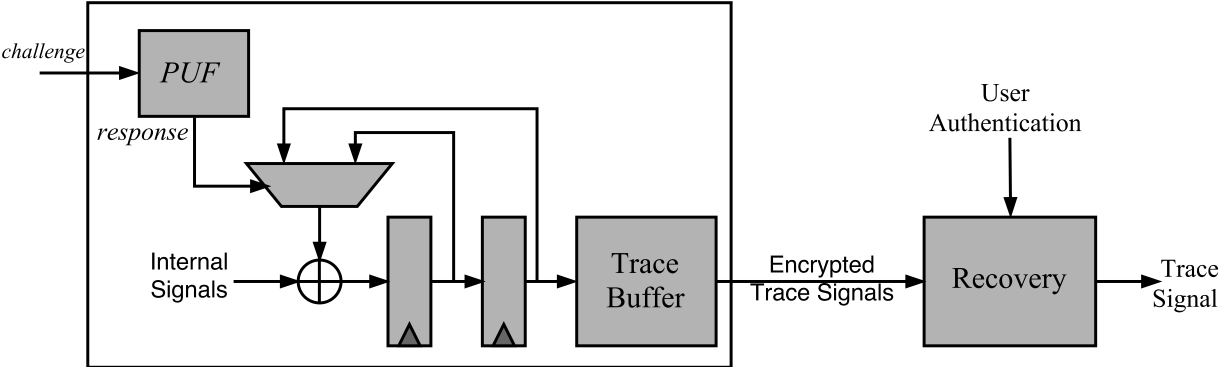


Figure 6-8. PUF-based Countermeasure

The LFSR-based approach, as shown in Figure 6-7, uses a Left Feedback Shift Register (LFSR) to scramble the traced signals before they are recorded in the trace buffer. LFSR requires an initial value (i.e. the seed) to set the initial state of the shift

register, and a well-chosen feedback function (XOR of some bits as in this example). LFSR can produce a sequence of pseudo-random numbers based on the seed and feedback function. The pseudo-random number will be added to the traced signals at each clock cycle. The structure of the LFSR is simple and the overhead for implementing LFSR would be minimal. Considering a trace buffer of 32-bit width, we need a 32-bit LFSR. Suppose the feedback function is XOR of eight selected bits, we would need 32 flip-flops for the shift register, and 7 two-input XOR gates for feedback function. We also need 32 XOR gates for adding the pseudo-random number with the original trace signals. Thus, the overhead is 32 flip-flops and 39 gates, which is minimal. The drawback with LFSR-based approach is that the pseudo-random sequence depends on the secrecy of the seed. The seed needs to be properly maintained as a secret by key management.

The PUF-based approach, as shown in Figure 6-8, uses a Physical Unclonable Function (PUF) to introduce built-in randomness into the traced signals. The idea of this countermeasure closely follows a similar countermeasure proposed for scan-chain attacks [82]. The signals from consecutive clock cycles are XOR-ed according to a PUF response. Since the PUF response is only known to the valid user, he/she can recover the trace signal values easily. For a malicious user, recovering the original trace signal values is hard. PUF provides a challenge-response mechanism, where the mapping from a challenge to a response is controlled by the manufacturing process as well as the nature of the Integrated Circuit (IC). This complex control makes PUF structures hard to clone and at the same time a unique device identification can be obtained. Compared to the look-up table-based storage of key, PUF provides a large set of challenge-response keys with a storage requirement that increases linearly with the number of challenge bits. Only a valid user is aware of the challenge-response sets. The drawback with PUF-based approach is that a reliable PUF ("strong" PUF) has very high overhead. An arbiter-based strong PUF [90] has been implemented in $0.02mm^2$ chip area in 180 nm fabrication technology.

Another SRAM-based strong PUF [91] is implemented with $0.08mm^2$ chip area in 65 nm technology.

Table 6-5. Comparison of LFSR-based and PUF-based countermeasures

Countermeasures	Protection Level	Area Overhead
LFSR-based approach	Low	Low
PUF-based approach	High	High

As shown in Table 6-5, PUF-based approach provides stronger protection than LFSR-based method but incurs higher area overhead. An LFSR is a linear system, which leads to easy cryptanalysis [92]. A recent work on fault countermeasures [93] has shown that a bad choice of internal randomness source can lead to the complete failure of the countermeasure itself. For the PUF-based countermeasure, the randomness comes from the manufacturing process as well as the nature of the Integrated Circuit (IC). Compared with block-cipher or stream-cipher (including LFSR) based countermeasures, the PUF-based countermeasure will be the most robust. To protect from trace buffer attack, the designer needs to trade-off between protection level and area overhead of different countermeasures. PUF-based countermeasure should always be chosen if area overhead is acceptable.

6.5 Summary

In this chapter, we introduce a novel attack, *Trace Buffer Attack*, on the AES cipher. The attack is mounted with the help of trace buffers, which provides observability for post-silicon debug. We identify this as a source of information leakage and experimentally demonstrate that AES, the currently dominant block cipher, is vulnerable. We show that we can mount a strong attack with the knowledge of the RTL implementation. We are also able to take advantage of the patterns in Rijndael's key expansion, and restore the primary key even when RTL implementation is not available. With a trace buffer size of 32×128, the full key of the iterative AES-128 can be restored in a few minutes. For pipelined AES, partial key can be restored in a few hours. This work illustrates the need for security-aware trace signal selection, and highlights the need for further research in understanding the trade-off between security and debug observability.

CHAPTER 7
STATISTICAL TEST GENERATION FOR TROJAN DETECTION

In this chapter, we propose a Trojan detection approach that can take advantage of both side channel analysis and functional test generation. Designed for improving side channel sensitivity for Trojan detection, our approach is significantly more effective than the test generation methods purely for functional testing. Instead of aiming on finding a vector to activate a set of rare nodes, we focus on creating a set of vector pairs to maximize switching in rare nodes.

The goal of our work is to generate efficient test vectors for Trojan detection using side-channel analysis. Functional test can detect Trojan effect only when it is fully triggered and its payload is propagated to the primary outputs, which makes functional test infeasible to detect Trojans in most cases. Side channel analysis can detect well-hidden Trojans by inspecting the side channel signals, for example, transient current in the circuit. If the switching effect introduced by the Trojan circuit is distinguishable, in the presence of process variation, the Trojan will get caught. In this paper we propose a comprehensive test generation framework to assist side channel analysis for hardware Trojan detection. Our algorithm creates multiple excitation of rare switching which is important in making side-channel based Trojan detection effective. Moreover, we also try to simultaneously minimize the background switching to maximize the relative switching.

We use the relative switching of the Trojan with respect to the whole circuit to indicate the sensitivity of the side channel signals. The statistical test patterns can maximize relative Trojan detection sensitivity under any process noise. Process variation is not expected to affect our side channel sensitivity computation since we consider switching activity instead of actual current or power values. The assumptions we have made are similar to the state-of-the-art side-channel analysis based Trojan detection approaches. The proposed method can be combined with any existing process calibration approaches

(such as one in [116] or [117]) to minimize the false positives/negatives - i.e. maximize Trojan coverage.

To make side channel analysis successful in detecting Trojans, we need to: (1) maximize the switching activity in the Trojan circuit; (2) minimize the switching activity in other parts of the circuit so that the relative switching effect is maximized. The main idea of this paper is to generate high quality test patterns which can achieve these two goals and increase the sensitivity of side channel analysis. The following are the major contributions of this paper:

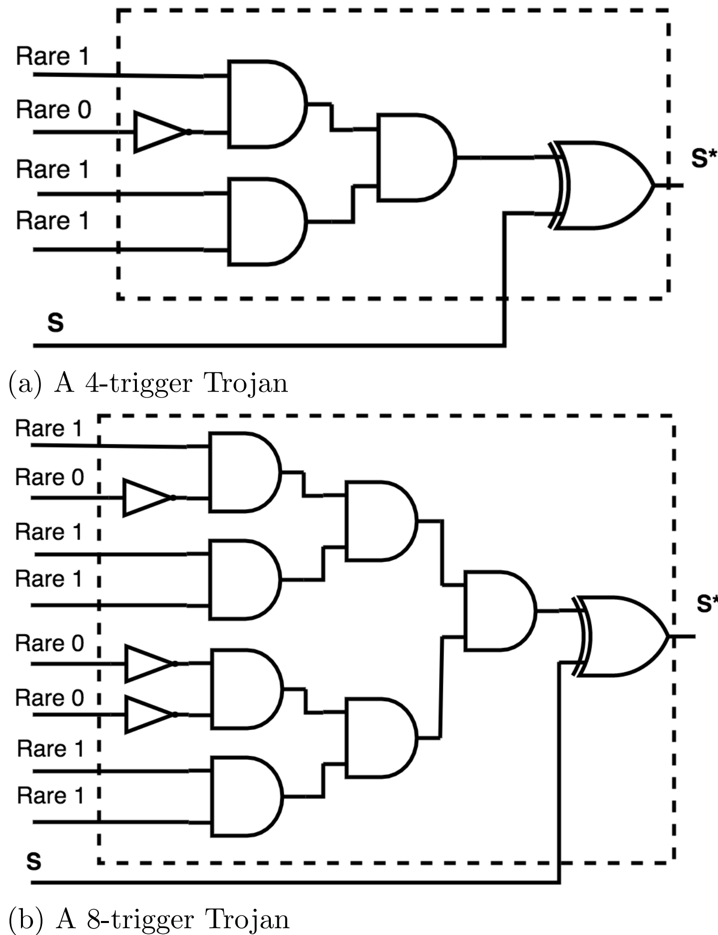
1. It presents, for the first time in our knowledge, a statistical test generation approach for increasing side-channel analysis based Trojan detection sensitivity. The proposed approach can be applicable to any transient current based Trojan detection approach.

2. The methodology, referred to as MERS (Multiple Excitation of Rare Switching) for statistical test generation, can derive a compact testset that can trigger each of the rare nodes to satisfy *rare switching* for multiple times. MERS can have a good coverage of all rare nodes and greatly increase the switching effect inside arbitrary Trojans in unknown locations of the circuit.

3. Two reordering methods are proposed to reduce the total switching of the circuit and thus further increase the sensitivity of side channel analysis. First, a simple and low-cost method based on Hamming distance of input vector pairs is introduced to reorder the tests. Next, we develop another simulation based method to more effectively balance switching in rare nodes and the total switching.

Our side-channel based approach is targeted towards detecting unknown Trojans, which means it will remain equally effective even if the adversary is aware of the proposed method. This is due to the following two reasons: (1) the proposed test generation method is statistical in nature - so, unlike conventional deterministic test approaches, it maximizes the activation probability for arbitrary Trojans designed with any trigger condition; and (2) it maximizes the detection sensitivity of unknown Trojans, however "stealthy", by amplifying its effect in side-channel signature. Our simulation platform inserts large number of arbitrary Trojans in a design and shows that the proposed approach is highly effective in detecting them.

The rest of the chapter is organized as follows. Section 7.1 presents the MERS test generation algorithm and the test reordering algorithms to improve sensitivity of side channel analysis. Section 7.2 describes the experiment setup and presents results on a set of ISCAS benchmarks with detailed analysis. Section 7.3 concludes the paper.



(a) A 4-trigger Trojan

(b) A 8-trigger Trojan

Figure 7-1. Trojans with rare nodes as trigger conditions. The 4-trigger Trojan will only be activated by the rare combination 1011 and the 8-trigger Trojan will only be activated by the rare combination 10110011.

7.1 MERS: Increasing the Trojan Detection Sensitivity

In this section, we present the proposed methodology for side-channel aware test generation in details. The methodology is based on the concept of statistically maximizing the switching activity in all the rarely triggered circuit nodes.

The effectiveness of a test pattern for side channel analysis is measured in two ways: (1) the ability to create most switching inside a Trojan or fully activate a Trojan; (2) the ability to create high Trojan-to-circuit switching. We measure *DeltaSwitch* as the switching introduced by the Trojan, which is the difference of number of switches between the golden circuit and the Trojan-infected circuit. We measure *RelativeSwitch* as the ratio of *DeltaSwitch* to the total number of switches (*TotalSwitch*) in the golden circuit. An effective test vector should be capable to create large *DeltaSwitch*, and more importantly to have large *RelativeSwitch*, as it is directly related to the sensitivity for side channel analysis.

$$RelativeSwitch = DeltaSwitch / TotalSwitch \quad (7\text{–}1)$$

The major challenges for generating high-quality test vectors are (1) we are not sure of the location where the Trojan is inserted in the circuit; (2) the Trojan is stealthy and has very low activity when it is not triggered. These characteristics have made random tests not effective in magnifying the side channel signal for Trojan detection. Fig. 7-1 shows two example Trojan instances. The 4-trigger Trojan will only be activated by the rare combination 1011 and the 8-trigger Trojan will only be activated by the rare combination 10110011. If the possibility of each rare node to take its rare value is 0.1, the probability to have these two Trojans fully triggered is 10^{-4} and 10^{-8}, respectively.

Our test generation approach (MERS) is based on creating a set of test vectors for each candidate rare node individually to have *rare switching* for multiple (at least N) times. Our approach utilizes the principle of n-detect [115] tests to increase the likelihood of partially or fully activating a Trojan. MERS can generate a high-quality testset for these rare nodes individually to have rare switching for N times. If N is sufficiently large, a Trojan with triggering conditions from these rare nodes is likely to have high switching activity even though it might not be fully activated.

Algorithm 12: Multiple Excitation of Rare Switching (MERS)

Input: Circuit netlist, rare switching requirement (N), list of rare nodes ($R = \{r_1, r_2, ..., r_m\}$), list of random patterns ($V = \{v_1, v_2, ..., v_n\}$)

Output: MERS test patterns (T)

```
    // simulate and sort random vectors
1:  for each random vector v in V do
2:      Simulate the circuit with the input vector v
3:      Count the number of nodes (R_V) in R with their rare values satisfied
4:  end
5:  Sort vectors in V in descending order of R_V
6:  for each node r_i in R do
7:      Set its rare switching counter (S_i) to 0
8:  end
    // mutate vector to find improved vector pairs
9:  Initialize previous vector t_p as a vector of all 0's
10: for each vector v_j in V do
11:     Simulate the circuit with vector pair (t_p, v_j)
12:     Count the number of rare switches (R_S)
13:     Set v'_j = v_j
14:     for each bit in v'_j do
15:         Mutate the bit and re-simulate the circuit with vector pair (t_p, v'_j)
16:         Count the number of rare switches (R'_S)
17:         if R'_S > R_S then
18:             Accept the mutation to v'_j
19:         end
20:     end
21:     Update S_i for all nodes in R due to vector v'_j
22:     if v'_j increases S_i for at least one rare node then
23:         Add the mutated vector v'_j to T
24:         Set t_p = v'_j
25:     end
26:     if S_i ≥ N for all nodes in R then
27:         Break
28:     end
29: end
30: return MERS test patterns T
```

7.1.1 Multiple Excitation of Rare Switching

The basic idea of MERS is that if we have a rare node switch N times where N is sufficiently large, it significantly improves the chances of switching in a Trojan associated

with that rare node. The **rare switching** in our algorithm specially refers to a rare node switching from its non-rare value to its rare value. The reason to choose this criteria is two-fold: (1) it is more difficult to switch from non-rare to rare value than from rare to non-rare value; (2) it defines the switching between the previous vector and the current vector, and it usually helps to create an extra switching between the current vector and the next vector. This will increase the probability of switching of a Trojan which has rare nodes as its trigger conditions. Our approach is also applicable to sequential Trojans, which requires the rare combinations to happen a certain number of times to be fully triggered.

Algorithm 12 shows the steps of MERS to generate high quality tests for creating switches in rare nodes, so as to assist side channel analysis for hardware Trojan detection. The algorithm is fed with the golden circuit netlist, the list of random test patterns (V) and a list of rare nodes (R) (which is obtained by random simulation beforehand). First, we simulate each random pattern and count the number of rare nodes (R_V) that take their rare values. We sort the random patterns in descending order of R_V, which means that the vector with ability to activate the most number of rare nodes goes first. Next, we initialize the rare switching counter S_i for each rare node to 0. In the next step, we mutate vectors from the random pattern set to generate high quality tests. We mutate the current vector one bit at a time and we accept the mutated bit only if the mutated vector can increase the number of nodes to have rare switching. In this step, only those rare nodes with $R_S < N$ are considered. The mutation process repeats until each rare node has achieved at least N rare switches. The output of the test generation process is a compact set that improves the switching capability in rare nodes, compared to random patterns. The complexity of the algorithm is $O(nm)$, where n is the total number of test vectors mutated during the process, and m is the number of bits in primary inputs. The runtime to generate MERS tests can be found in Table 7-1.

The testset generated by MERS is expected to be very effective in increasing the likelihood of rare nodes to switch and thus increasing the activities in Trojans. In other words, MERS testset is capable of maximizing the DeltaSwitch (the numerator in Equation 7–1). MERS testset is already a very high quality testset in terms of criteria for DeltaSwitch. However, MERS testset also creates more switching in other parts of the circuit, when it is making efforts to switch rare nodes. This characteristic of increased TotalSwitch would be further illustrated in the Section 7.2. In order to maximize relative switching, we need to have TotalSwitch in control as well. In the following subsections, we propose two methods to tune the MERS testset, so that it can: (1) still be effective for DeltaSwitch, (2) reduce TotalSwitch and improve the effectiveness for RelativeSwitch. The first method is a heuristic approach based on hamming distance of test vectors, which can reduce the total switching. The second one is simulation based, in which we try to balance the rare switching and the total switching while we explore all the candidate vectors.

7.1.2 Hamming Distance Based Reordering

If two consecutive input vectors have the same values in most bits, it is very possible that the internal nodes will also have a lot of values in common. A simple heuristic to reduce total switching in circuit is to have similar input vectors. We use the Hamming distance between two vectors to represent the similarity. Algorithm 13 shows our approach to reorder the testset by Hamming distance. The algorithm is a greedy approach to explore all candidate vectors and take the best one in terms of Hamming distance. We first check the Hamming distances between the previous vector and all the remaining vectors, then we select the vector which has the minimum Hamming distance as the next following vector. The time complexity of Algorithm 13 is $O(n^2)$, where n is the testset size. Fortunately, it is of low cost to calculate the Hamming distance between two input vectors. The actual run-time is very short because n (number of test patterns produced by MERS) is small, in the order of tens of thousands.

Algorithm 13: TESTS REORDERING BY HAMMING DISTANCE (MERS-h)

Input: List of Test Patterns ($T_{orig} = \{t_1, t_2, ..., t_n\}$) produced by Algorithm 1
Output: Improved Test Patterns (T_{hamm})

1: *Initialize* $T_{hamm} = \{\}$
2: *Initialize* previous test t_p as a vector of all 0's
3: **while** T_{orig} *is not empty* **do**
4: $min_{dist} = int_max$
5: $best_{idx} = -1$
6: **for all** *remaining tests* t_j *in* T_{orig} **do**
7: **if** $min_{dist} > hamming_dist(t_p, t_j)$ **then**
8: $min_{dist} = hamming_dist(t_p, t_j)$
9: $best_{idx} = j$
10: **end**
11: **end**
12: *Add* $t_{best_{idx}}$ to the end of T_{hamm}
13: *Remove* $t_{best_{idx}}$ from T_{orig}
14: *Update* $t_p = t_{best_{idx}}$
15: **end**
16: **return** T_{hamm}

7.1.3 Simulation Based Reordering

The reordering problem to improve the relative switching is actually a multi-objective optimization problem: maximize the *DeltaSwitch* and minimize the *TotalSwitch* as in Equation 7–1. We don't know the *DeltaSwitch*, because the location and type of the Trojan is unknown. However, rare switching between two vectors is a good indicator for *DeltaSwitch*, which means a large number of rare switching would imply a large *DeltaSwitch* in Trojan. We redefine the optimization goal as to maximize the rare switching and minimize the total switching at the same time between vector pairs. We formalize the problem as shown in Equation 7–2. We need to explore the best weights to balance between the two objectives:

$$maximize \quad (w_1 * RareSwitch - w_2 * TotalSwitch) \tag{7–2}$$

We propose an approach as shown in Algorithm 14 based on real simulation of the test vectors to maximize the combined objective. We introduce a concept of *profit* to

Algorithm 14: TESTS REORDERING BY SIMULATION (MERS-s)

Input: List of Test Patterns ($T_{orig} = \{t_1, t_2, ..., t_n\}$) produced by Algorithm 1
Output: Improved Test Patterns (T_{sim})

```
1:  Initialize T_sim = {}
2:  Initialize previous test t_p as a vector of all 0's
3:  while T_orig is not empty do
4:      max_p = int_min
5:      best_idx = −1
6:      for all remaining tests t_j in T_orig do
7:          Simulate the circuit with vector pair (t_p, t_j)
8:          Count the number of RareSwitch and TotalSwitch
9:          profit = C * RareSwitch − TotalSwitch
10:         if max_p < profit then
11:             max_p = profit
12:             best_idx = j
13:         end
14:     end
15:     Add t_best_idx to the end of T_sim
16:     Remove t_best_idx from T_orig
17:     Update t_p = t_best_idx
18: end
19: return T_sim
```

indicate the fitness of a test vector to follow the previous test vector. *profit* is defined as ($C * RareSwitch - TotalSwitch$), where C is the ratio of two weights w_1 and w_2. It is meant to maximize the rare switching (activity in Trojan circuits) and minimize the total switching of the whole circuit. In the experiment section, we will explore different weight ratios and check the influence of weight ratios on side channel sensitivity.

Algorithm 14 shows our approach to tune the testset by simulation with *profit* as a reordering criterion. By exhaustively checking the *profit* between the previous vector and all the remaining vectors, we select the vector which has the maximum *profit* as the next following vector. The time complexity of Algorithm 14 is $O(n^2)$, where n is the test length. However, it is much slower than Algorithm 13, because it is time-consuming to simulate input vector pairs and calculate *profit*.

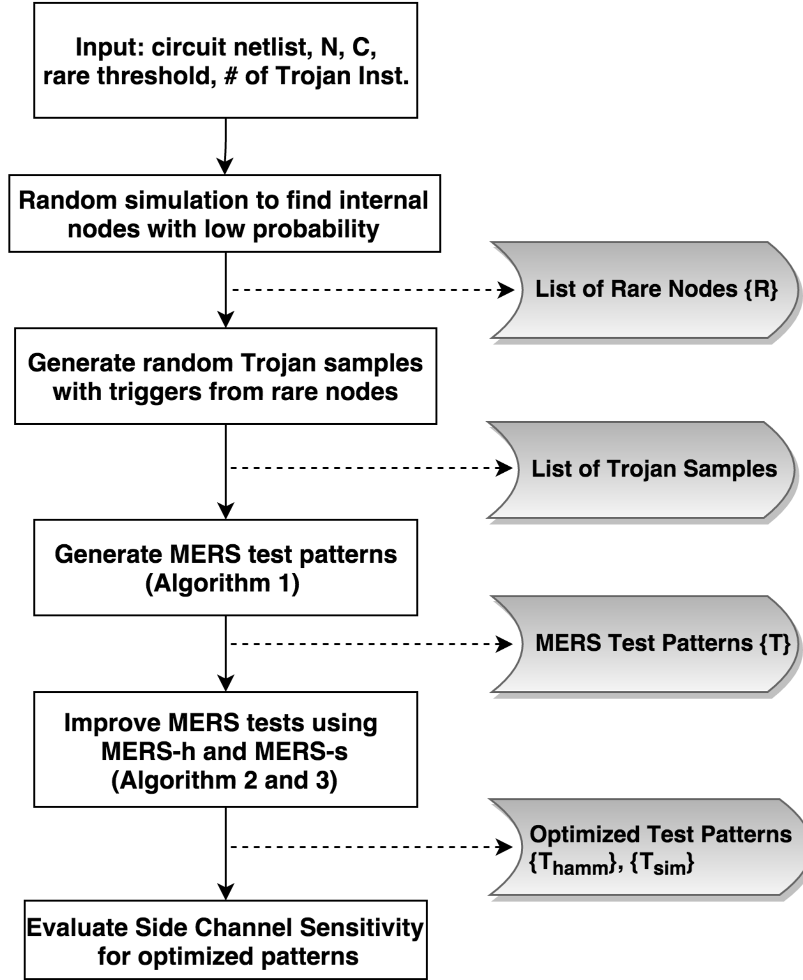


Figure 7-2. Test generation framework for side-channel analysis based Trojan detection.

7.2 Experiments

7.2.1 Experimental Setup

The test generation framework, including the MERS core algorithms and the evaluation framework, is implemented using C. As shown in Fig. 7-2, the test generation framework can simulate circuit netlists, generate MERS testset, further tune the testset, and evaluate the effectiveness of testsets on random Trojans. We evaluated our approach on a subset of ISCAS-85 and ISCAS-89 benchmark circuits. The sequential circuits are converted into full scan mode. We also implemented the MERO [100] approach with parameter N of 1000 for comparison We did our experiments on a server with AMD Opteron Processor 6378 (2.4GHz). The runtime for different benchmarks and different

methods is shown in Table 7-1. The table also shows the number of rare nodes in each benchmark, and we used 0.1 as the rare threshold to select rare nodes.

Table 7-1. Runtime for MERS test generation and test reordering.

Benchmark	**Nodes (rare / total)**	**Run-time (s)**		
		MERS	MERS-h reordering	MERS-s reordering
c2670	63 / 1010	13370.86	7.24	4925.23
c3540	331 / 1184	6097.51	9.43	18166.94
c5315	255 / 2485	45595.97	11.04	39073.81
c6288	45 / 2448	4154.62	0.31	2802.85
c7552	306 / 3720	81405.89	25.2	63502.19
s13207	592 / 2504	12511.95	365.02	29064.72
s15850	679 / 3004	19903.44	728.14	38181.49
s35932	896 / 6500	7295.74	39.53	31201.04

7.2.2 Evaluation Criteria

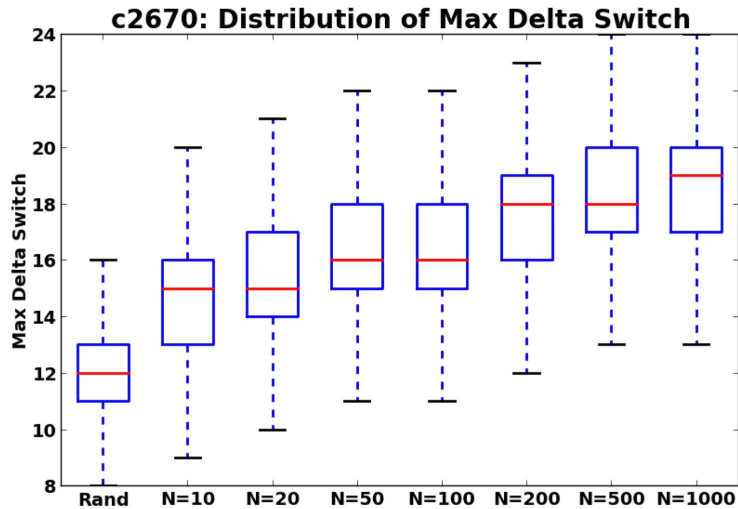
When applying a testset to a circuit with Trojan, there are four criteria to evaluate the effectiveness of the testset:

- **AvgDeltaSwitch**: the average delta switch when applying the testset on this Trojan-infected circuit.
- **MaxDeltaSwitch**: the maximum delta switch when applying the testset.
- **AvgRelativeSwitch**: the average relative switch when applying the testset.
- **MaxRelativeSwitch**: the maximum relative switch when applying the testset. We choose this criterion as the **Side Channel Sensitivity** because this directly determines whether a Trojan can be detected through side-channel analysis.

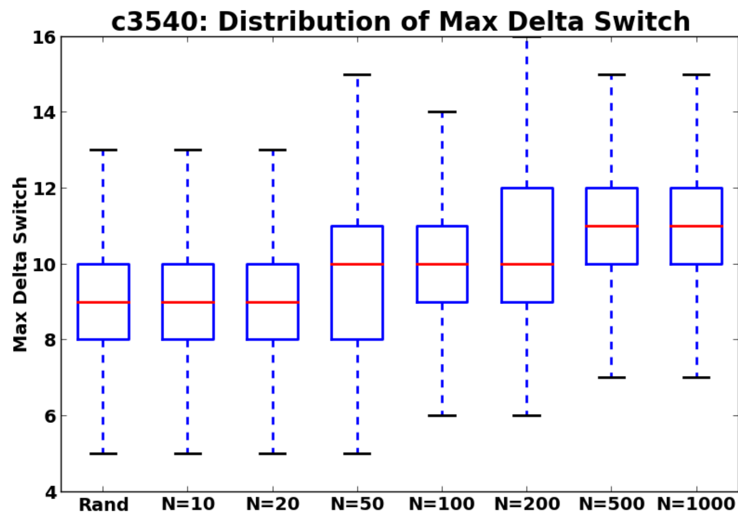
AvgDeltaSwitch and *MaxDeltaSwitch* reflect the activity in Trojan, and *AvgRelativeSwitch* as *MaxRelativeSwitch* reflect the sensitivity of the side channel signal created by the Trojan. Among these four metrics, we care the most about *MaxRelativeSwitch*, which is most important in side-channel analysis for Trojan detection.

As for evaluation of testsets, we would expect a high-quality testset to have a good coverage over all possible Trojans. In our experiments, we apply the testset to 1000 randomly inserted Trojan samples and compute these four values for each Trojan instance. We would then take the average of these four metrics, which would reflect the

capability of the testset to assist side channel analysis for different Trojans. The average *MaxRelativeSwitch* would be most suitable for Side Channel Sensitivity evaluation, which is to maximize the sensitivity for an arbitrary Trojan in unknown circuit location.



(a) c2670: Distribution of MaxDeltaSwitch over 1000 random samples of 8-trigger Trojans.



(b) c3540: Distribution of MaxDeltaSwitch over 1000 random samples of 8-trigger Trojans.

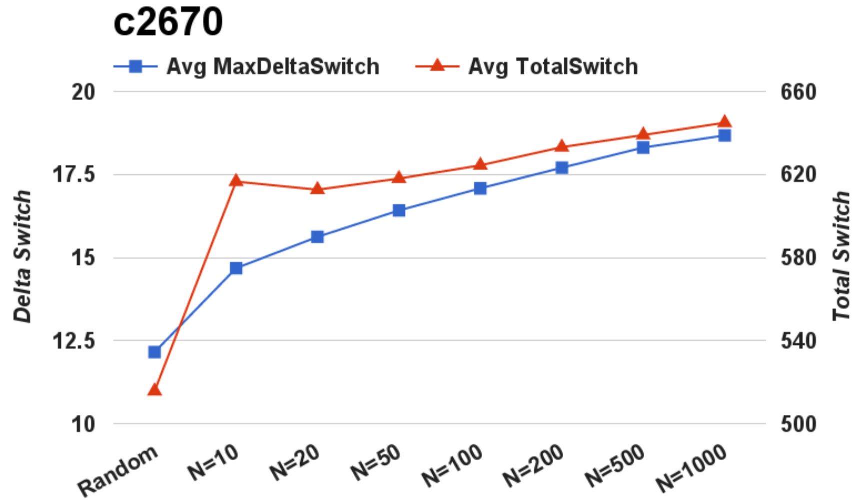
Figure 7-3. Impact of N (number of times that a rare node have rare switching) on MaxDeltaSwitch for benchmarks (a) c2670 and (b) c3540.

7.2.3 Exploration of N

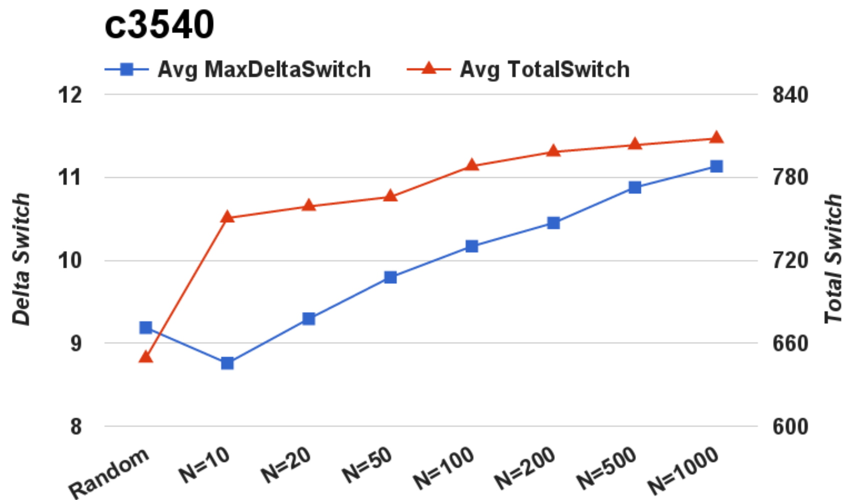
Fig. 7-3 shows the distribution of MaxDeltaSwitch over 1000 random 8-trigger Trojan samples for two ISCAS-85 benchmarks. We choose different N to generate MERS testsets, to compare with the Random (10K vectors) testset. For each testset, the box plot shows (minimum, first quartile, median, third quartile, maximum) values of MaxDeltaSwitch of the 1000 Trojan samples. It is clear from these plots that the distribution of MaxDeltaSwitch is constantly improving with increasing N. For $c2670$, the average MaxDeltaSwitch (as shown by the red lines) can reach 18.67 for MERS ($N = 1000$), while Random testset is only 12.15. For $c3540$, the average MaxDeltaSwitch can reach 11.13 for MERS ($N = 1000$), while Random testset is only 9.19. The fact that the quality of MERS tests improves with increasing N is not surprising. It is similar to N-detect tests for stuck-at faults, where fault coverage is expected to improve with increasing N. The testset size also increases with N. The sizes of testsets for MERS ($N =$ 10, 20, 50, 100, 200, 500, 1000) are (71, 140, 347, 656, 1262, 3142, 6199) for $c2670$, and (161, 302, 742, 1441, 2858, 7070, 14250) for $c3540$. In most of our experiments, we will choose a value of N = 1000, which is a good balance between testset quality and testset size. For fair comparison with Random testset, we will only take the first 10K vectors of MERS testset if it is larger than 10K.

7.2.4 Side-effect of MERS: Increased TotalSwitch

Fig. 7-4 shows the average $MaxDeltaSwitch$ and the average $TotalSwitch$ of the testsets for 1000 8-trigger Trojan samples for different values of N. For both of the two benchmarks, the average $TotalSwitch$ increases with N as well as the average $MaxDeltaSwitch$. It is obvious that all the MERS testsets have much larger average $TotalSwitch$, compared with the Random testset. For $c2670$, the average $TotalSwitch$ for MERS ($N = 1000$) is 644.9, which is about 1.25X times of that of the Random testset (515.7). For $c3540$, the average $TotalSwitch$ for MERS ($N = 1000$) is 808, while Random testset is only 649.2. The insight that we can get from here is that MERS tends



(a) c2670



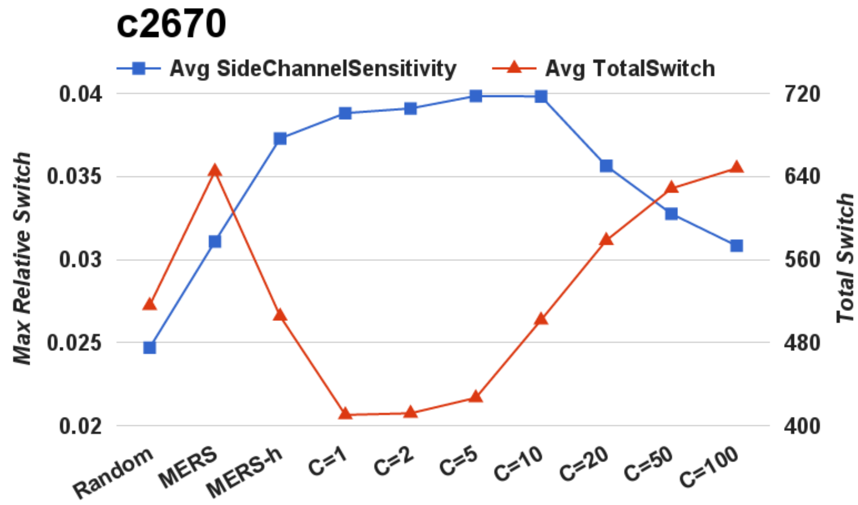
(b) c3540

Figure 7-4. MaxDeltaSwitch versus TotalSwitch for different N for benchmarks (a) c2670 and (b) c3540.

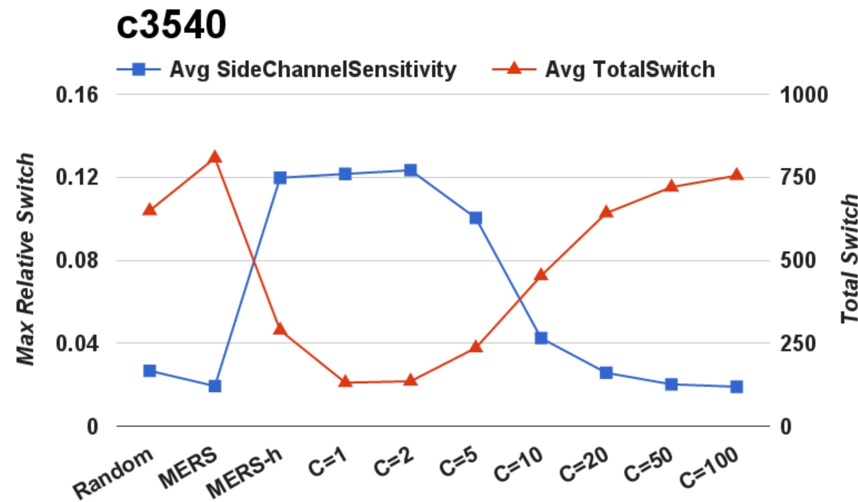
to increase the *TotalSwitch* of the circuit, although it is designed to increase switches in rare nodes. The following subsection will show that the proposed reordering methods would be effective to reduce *TotalSwitch* and thus increase side channel sensitivity.

7.2.5 Reordering and Exploration of C

The effectiveness of the two reordering methods can be observed in Fig. 7-5 and Fig. 7-6. As shown in Fig. 7-5, MERS-h can reduce *TotalSwitch* and thus increase the relative



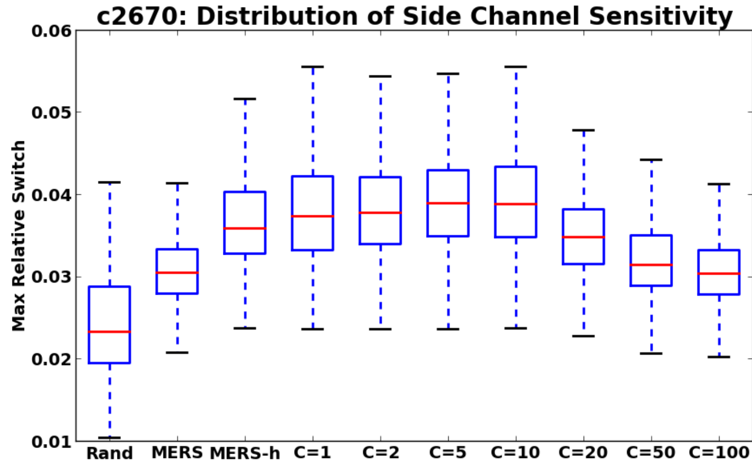
(a) c2670



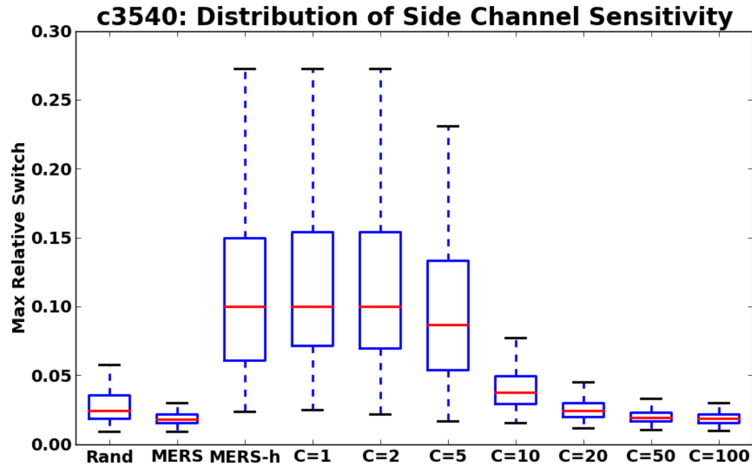
(b) c3540

Figure 7-5. Side Channel Sensitivity versus Total Switch for Random, the original MERS, MERS-h and MERS-s (with different C) for benchmarks (a) c2670 and (b) c3540.

switching (i.e. the Side Channel Sensitivity), compared with the original MERS testset. For MERS-s with different weight ratio C, side channel sensitivity improves steadily with a small C, and then goes down when C is too large. As the weight ratio to balance *DeltaSwitch* and *TotalSwitch*, a large C will outweigh the influence of *TotalSwitch*, which will make it not much different from the original MERS testset. In the following



(a) c2670: Distribution of Side Channel Sensitivity over 1000 random samples of 8-trigger Trojans.



(b) c3540: Distribution of Side Channel Sensitivity over 1000 random samples of 8-trigger Trojans.

Figure 7-6. Distribution of Side Channel Sensitivity for Random, the original MERS, MERS-h and MERS-s (with different C) for benchmarks (a) c2670 and (b) c3540.

experiments, we choose the weight ratio as $C = 5$, as it reaches a good balance between the total switching and rare switching.

Fig. 7-6 shows detailed distribution of Side Channel Sensitivity for 1000 8-trigger Trojan samples with different choices of C. The reordering methods are working well to

improve Side Channel Sensitivity, which is built on the fact that the original MERS testset is already of high quality in terms of *DeltaSwitch*, or switching in Trojans.

7.2.6 Effectiveness of MERS in Creating Trojan Activity

Table 7-2 shows that MERS (N=1000) is very effective in creating *DeltaSwitch* caused by Trojan. The average *Max Delta Switch* increases by 31.11% and the average *Avg Delta Switch* by 187.33% on average for different benchmarks, compared with Random testset. This shows the effectiveness of MERS in creating Trojan activity.

Table 7-3 shows that MERS is also helpful in improving RelativeSwitch. The average AvgRelativeSwitch increased by 158.16%, compared with Random testsets. For average MaxRelativeSwitch (Side Channel Sensitivity), MERS has an average improvement of 18.89%. However, Side Channel Sensitivity for benchmark *c3540* and *c6288* is not doing as well as that of Random testsets. This is due to the fact that the MERS testset also increase the total switching, when it is making efforts to make rare nodes switching. This phenomenon is illustrated and explained in Fig. 7-4 and Fig. 7-5, and this side effect can improved by the two reordering algorithms as shown in Table 7-4 and 7-5.

Table 7-2. Comparison of MERS (N=1000) with Random (10K) for average MaxDeltaSwitch and average AvgDeltaSwitch.

Benchmark	Average MaxDeltaSwitch			Average AvgDeltaSwitch		
	Random	MERS	Improve.	Random	MERS	Improve.
c2670	12.15	18.67	53.67%	1.4289	6.8561	379.83%
c3540	9.19	11.13	21.16%	1.3716	2.9058	111.85%
c5315	9.51	13.80	45.16%	1.3116	3.9300	199.64%
c6288	6.63	7.26	9.63%	1.0636	4.8448	355.50%
c7552	8.53	12.00	40.76%	1.3488	2.7700	105.36%
s13207	6.63	8.83	33.18%	0.6428	0.9771	52.01%
s15850	7.53	10.84	43.99%	0.7465	1.3609	82.29%
s35932	15.16	15.37	1.35%	2.1803	6.8060	212.16%
Avg. Improve.	–	–	31.11%	–	–	187.33%

Table 7-3. Comparison of MERS (N=1000) with Random (10K) for average *MaxRelativeSwitch* (Side Channel Sensitivity) and average *AvgRelativeSwitch*.

Benchmark	Average MaxRelativeSwitch (Side Channel Sensitivity)			Average AvgRelativeSwitch		
	Random	MERS	Improvement	Random	MERS	Improvement
c2670	0.02469	0.03108	25.90%	0.00255	0.01054	314.14%
c3540	0.02670	0.01933	-27.59%	0.00214	0.00361	69.12%
c5315	0.00526	0.00766	45.72%	0.00075	0.00200	165.65%
c6288	0.00534	0.00395	-26.06%	0.00059	0.00219	270.68%
c7552	0.00452	0.00852	88.48%	0.00058	0.00113	94.65%
s13207	0.00756	0.00844	11.64%	0.00066	0.00085	28.22%
s15850	0.00593	0.00716	20.70%	0.00053	0.00082	54.25%
s35932	0.00523	0.00587	12.29%	0.00060	0.00223	268.54%
Avg. Improve.	–	–	18.89%	–	–	158.16%

7.2.7 Side Channel Sensitivity Improvement

To this point, we have explored the parameters: N for MERS and C for MERS-s, we choose $N = 1000$ and $C = 5$ in the following experiment to compare our proposed schemes with Random testset and MERO. Table 7-4 and 7-5 show the improvement of proposed approaches on Side Channel Sensitivity for 4-trigger and 8-trigger Trojans.

Table 7-4 shows that MERS, MERS-h and MERS-s have 10.37%, 138.44% and 152.26% improvement over the Random testsets, respectively. While the original MERS testsets is 23.95% worse than MERO testsets, MERS-h and MERS-s have 52.62% and 62.01% improvement over MERO. Table 7-5 shows the results for 8-trigger Trojans. Compared with Random testsets, MERS, MERS-h and MERS-s can have 18.89%, 107.53% and 96.61% improvement, respectively. The original MERS testsets is 12.43% worse than MERO testsets. MERS-h and MERS-s testsets can improve the Side Channel Sensitivity by 40.79% and 38.50%.

In this section, we explored the impact of different values of N for MERS and observed the effectiveness of MERS to maximize Trojan activity as N increases. We confirmed the superiority of MERS testsets over Random testsets in Section 7.2.6 on creating switching activity in randomly sampled Trojans. We observed that the total

switching was also likely to increase while MERS was making efforts to maximize rare switching in Trojans. The two reordering methods (MERS-h and MERS-s) successfully had the total switching under control while maintaining the rare switching high. The comparison with Random and MERO testsets shows the effectiveness of our test generation framework in maximizing Side Channel Sensitivity for Trojan detection.

Table 7-4. Comparison of average *Side Channel Sensitivity* between Random (10K), MERO, and MERS testsets, N=1000, C=5 for MERS-s, over 1000 random samples of 4-trigger Trojans.

Benchmark	**Comparison Testsets**		**Proposed Schemes**			**Improvement to Random**			**Improvement to MERO**		
	Random	MERO	MERS	MERS-h	MERS-s	MERS	MERS-h	MERS-s	MERS	MERS-h	MERS-s
c2670	0.01703	0.02571	0.02231	0.03035	0.03308	31.01%	78.27%	94.31%	-13.23%	18.07%	28.69%
c3540	0.02144	0.04238	0.01336	0.10677	0.11067	-37.71%	397.97%	416.16%	-68.48%	151.96%	161.16%
c5315	0.00445	0.01082	0.00747	0.01287	0.01586	67.79%	188.97%	256.29%	-30.97%	18.89%	46.59%
c6288	0.00480	0.00395	0.00313	0.00741	0.00896	-34.81%	54.47%	86.85%	-20.88%	87.50%	126.80%
c7552	0.00351	0.00737	0.00491	0.01250	0.01168	39.61%	255.63%	232.38%	-33.46%	69.50%	58.42%
s13207	0.00568	0.00617	0.00619	0.00773	0.00826	9.07%	36.24%	45.49%	0.31%	25.29%	33.80%
s15850	0.00447	0.00487	0.00474	0.00691	0.00634	6.14%	54.83%	42.06%	-2.75%	41.86%	30.17%
s35932	0.00354	0.00463	0.00361	0.00500	0.00512	1.89%	41.17%	44.53%	-22.12%	7.90%	10.48%
Avg. Improve.	–	–	–	–	–	10.37%	138.44%	152.26%	-23.95%	52.62%	62.01%

Table 7-5. Comparison of average *Side Channel Sensitivity* between Random (10K), MERO, and MERS testsets, N=1000, C=5 for MERS-s, over 1000 random samples of 8-trigger Trojans.

Benchmark	**Comparison testsets**		**Proposed Schemes**			**Improvement to Random**			**Improvement to MERO**		
	Random	MERO	MERS	MERS-h	MERS-s	MERS	MERS-h	MERS-s	MERS	MERS-h	MERS-s
c2670	0.02469	0.03204	0.03108	0.03729	0.03984	25.90%	51.05%	61.40%	-3.01%	16.37%	24.35%
c3540	0.02670	0.05532	0.01933	0.11974	0.10037	-27.59%	348.53%	275.96%	-65.05%	116.47%	81.44%
c5315	0.00526	0.00875	0.00766	0.01020	0.01129	45.72%	94.03%	114.78%	-12.38%	16.66%	29.14%
c6288	0.00534	0.00412	0.00395	0.00649	0.00790	-26.06%	21.55%	47.97%	-4.20%	57.49%	91.72%
c7552	0.00452	0.00914	0.00852	0.01437	0.01149	88.48%	217.78%	154.00%	-6.70%	57.31%	25.74%
s13207	0.00756	0.00838	0.00844	0.01053	0.01112	11.64%	39.24%	47.05%	0.69%	25.58%	32.63%
s15850	0.00593	0.00722	0.00716	0.00923	0.00818	20.70%	55.69%	37.94%	-0.87%	27.86%	13.28%
s35932	0.00523	0.00638	0.00587	0.00692	0.00700	12.29%	32.39%	33.80%	-7.90%	8.58%	9.74%
Avg. Improve.	–	–	–	–	–	18.89%	107.53%	96.61%	-12.43%	40.79%	38.50%

7.2.8 Calibration and Multiple-Parameter Side-Channel Analysis

MERS can be combined with any existing process calibration approaches [116][117][118] to minimize the false positives/negatives - i.e. maximize Trojan coverage. Most side-channel analysis based approaches perform process variation calibration by using golden chips at different process corners. This helps us obtain the limiting threshold values, beyond which any chip is classified as Trojan-infected. MERS simultaneously

maximize the switching in Trojan and minimize the background switching, so as to maximize the relative switching. It is intuitive to use the dynamic current (I_{DDT}) as the side channel signature, since MERS tests would contribute greatly to the dynamic current if a Trojan exists. By calibration or reference to that of a golden chip, MERS helps side channel analysis to reduce the intra-die systematic process variations. Moreover, as shown in [118], various measurable parameters can be used for multiple-parameter side-channel-based Trojan detection where at least one parameter is affected by the Trojan and other parameters are used to calibrate the process noise. For example, the quiescent or leakage current (I_{DDQ}) and the maximum operating frequency (F_{max}) will also be influenced when there is a Trojan. Although MERS might not contribute to I_{DDQ} nor F_{max}, they can serve as side channel references to calibrate process noise. [118] has shown the joint effect of these three variables (I_{DDT}, I_{DDQ} and F_{max}). MERS can increase I_{DDT}, which would greatly improve the accuracy of [118] to isolate a Trojan-infected chip in the multiple-parameter space from process induced variations.

7.2.9 Scalability to Large Designs

For a large design, the supply current of a golden chip for a high-activity vector can be large compared with the additional current consumed by a small Trojan. The variation in the current value due to process noise can be very large, which would mask the effect of the Trojan on the measured current and lead to difficulty for accurate Trojan detection. MERS is scalable to large circuits and can be combined with region-based test generation approaches, which segment a circuit into nearly-isolated regions (i.e. with low connectivity between them). In this case, MERS can be applied separately to each region. For example, in case of a processor, MERS can be employed separately to its regions, such as, integer execution unit, floating point datapaths, control logic, and result bus logic. MERS can work with [118] to isolate a region and prevent unwanted switching in independent functional modules, by taking advantage of the power gating techniques conventionally used by low-power designs, such as clock gating, supply gating, or operand

isolation. MERS can also be applied a more flexible region-based side channel analysis approach proposed in [119]. They perform a functional decomposition to divide a large design into several small blocks or regions, so that they can activate them one region at a time. MERS can be used as the test generation algorithm to generate vectors that maximize the activity within each region. The decision to report a chip as Trojan-infected would be based on the dispersion of its region current matrix of all regions with respect to the golden chip. Future work will include integration of MERS with region-based circuit partitioning techniques to further enhance its effectiveness and its evaluation on larger industry-standard designs.

7.3 Summary

We have presented a framework for statistical test generation, called MERS, which can significantly improve the Trojan detection sensitivity in side-channel analysis based Trojan detection. The approach aims at statistically increasing switching activity in an unknown Trojan to amplify the Trojan effect in presence of large process variations. Such a test generation approach will, in general, be effective for any side-channel analysis approaches that rely on activity in Trojan circuits (e.g. transient current or EM based methods). Furthermore, MERS is effective for any Trojan forms/sizes, where a Trojan is implanted through alterations in a circuit structure - the most dominant mode of Trojan implantation. Our simulation results on a set of benchmark circuits show that the proposed approach can improve the side channel sensitivity by more than 96.61%, compared with random tests for a large set of arbitrary Trojans. It shows that a judicious statistical test generation such as MERS can serve as an essential component in a side-channel Trojan detection approach.

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

SoCs are widely used in designing both IoT devices and embedded systems in a wide variety of domains. The vulnerabilities of SoCs pose unique threats to the reliability and security of the system. This dissertation described a set of novel techniques and methodologies for SoC vulnerability analysis and mitigation. This chapter concludes this dissertation and outlines possible directions for future research.

8.1 Conclusions

To design reliable and secure SoCs, it is crucial to analyze the possible vulnerabilities of different SoCs and find countermeasures for them. We investigated cache vulnerability due to soft errors, utilization of debug infrastructure (e.g., trace buffer) for attack, and malicious modifications (hardware Trojans) in SoC designs. This dissertation developed techniques to investigate and protect against these vulnerabilities.

In Chapter 3, we aimed at reducing cache vulnerability during dynamic cache reconfiguration (DCR). By tuning the cache configuration to satisfy the specific requirement of each application, conventional DCR approaches focus on improving the performance and/or energy consumption. But conventional DCR approaches may result in unacceptable cache vulnerability. We developed algorithms to reduce vulnerability with energy and performance considerations during DCR. We proposed a vulnerability-aware energy-optimization (VAEO) approach, which can significantly improve the reliability of both instruction and data caches.

In Chapter 4, we applied DCR on the partially protected caches (PPC) architecture. The PPC architecture has two data caches at the same level of memory hierarchy, one protected cache and one unprotected cache. PPC provides an effective mechanism to reduce cache vulnerability. However, it introduces unacceptable energy and performance overhead. We presented a reconfigurable cache architecture to combine the advantages of PPC (vulnerability reduction) and cache reconfiguration (energy and performance

improvement). Synergistic integration of cache reconfiguration and data partitioning improves both vulnerability and energy efficiency. We also presented two fast exploration strategies that can achieve up to 6X speed-up, with negligible impact on the quality of exploration results.

In Chapter 5, we applied cache reconfiguration on multicore systems. We presented a vulnerability-aware energy optimization technique for real-time multicore systems. Our approach integrates dynamic cache reconfiguration (DCR) of private L1 caches and cache partitioning (CP) of the shared L2 cache. L2 CP is effective in reducing inter-core interference, while applying L1 DCR can further reduce the energy consumption under the performance and vulnerability constraints. Our task profiling technique based on the independence between tasks can drastically reduce the complexity of design space exploration. Our proposed algorithm uses dynamic programming by discretizing the energy values, which can efficiently search the space to find optimal L1 cache configurations for each task and L2 cache partition factors for each core.

In Chapter 6, we investigated the vulnerability of trace buffer in post-silicon debug, and introduced a novel attack, *Trace Buffer Attack*, on the AES cipher. The attack is mounted with the help of trace buffers, which provides observability for post-silicon debug. We identify this as a source of information leakage. We show that we can mount a strong attack with the knowledge of the RTL implementation. We are also able to take advantage of the patterns in Rijndael's key expansion, and restore the primary key even when RTL implementation is not available. Our work illustrates the need for security-aware trace signal selection, and highlights the need for further research in understanding the trade-off between security and debug observability. We also proposed two countermeasures to protect trace buffer against this kind of attack.

In Chapter 7, we presented a framework for statistical test generation, called MERS, for hardware Trojan detection. Our test generation strategy creates multiple excitations of rare switching (MERS) on rare nodes to increase Trojan switching activity. It statistically

increases switching activity in an unknown Trojan to amplify the Trojan effect in the presence of large process variations. It can significantly improve the Trojan detection sensitivity in side-channel analysis based Trojan detection. Our approach will be effective for any side-channel analysis approaches that rely on activity in Trojan circuits (e.g. transient current or EM based methods).

In conclusion, this dissertation presented a comprehensive study of different SoCs on vulnerability analysis and mitigation. We developed a set of techniques to reduce cache vulnerability, protect against trace buffer attack, and detect hardware Trojans. Our research will enable designers and security engineers to improve the reliability and security of SoCs.

8.2 Future Research Directions

As more and more SoCs are deployed, vulnerabilities of SoCs will continue to undermine the reliability and security of our systems. The research described in this dissertation can be extended in the following directions:

Our approaches for cache vulnerability reduction can be extended to systems allowing preemptive execution. This can be achieved by partitioning tasks into phases and profiling each partition, and preempted task can resume execution using the configuration for the current partition. In case of statically scheduled systems, we exactly know the preemption points, therefore, we can profile accurately even for preemptive tasks. However, for dynamically scheduled systems, we need to profile a task at phase boundaries and during runtime choose the configuration using nearest neighbor approach.

Our approaches for cache vulnerability reduction are based on static profiling of applications before runtime. Static profiling takes very long simulation time and it might not be suitable for systems where applications have very different runtime behaviors. A natural extension is to make our approach work online without static profiling beforehand. This can be achieved by means of intermittent execution and periodic prediction of vulnerability behavior.

We have shown that the trace buffer can be a source of information leakage for hardware cryptographic devices. Scan chain is also a popular debug facility in post-silicon debug. If an attacker takes advantage of both the trace buffer and scan chain, he/she can get more internal information of the cryptographic devices. It is therefore important to consider the situation that the attacker might employ both the trace buffer and scan chain for a more sophisticated attack.

Our hardware Trojan detection approach can increase the relative switching of the unknown Trojan circuit. Our experiments based on simulation have shown that the side-channel sensitivity is greatly increased. The next step is to use our approach on both ASIC and reconfigurable systems. In this way, we can see the effect of process variation and measurement noise on side-channel sensitivity, as well as the detection accuracy of Trojans.

APPENDIX
LIST OF PUBLICATIONS

Conference Papers

1. Farimah Farahmandi, **Yuanwen Huang**, Prabhat Mishra. Hardware Trojan Localization using Polynomials. Asia and South Pacific Design Automation Conference (ASPDAC), January 2017.

2. **Yuanwen Huang**, Swarup Bhunia, Prabhat Mishra. MERS: Statistical Test Generation for Side-Channel Analysis based Trojan Detection. ACM Conference on Computer and Communications Security (CCS), October 2016.

3. **Yuanwen Huang** and Prabhat Mishra. Reliability and Energy-aware Cache Reconfiguration for Embedded Systems. International Symposium on Quality Electronic Design (ISQED), March 2016. [**Best Paper Award**]

4. **Yuanwen Huang**, Anupam Chattopadhyay and Prabhat Mishra. Trace Buffer Attack: Security versus Observability Study in Post-Silicon Debug. IEEE International Conference on Very Large Scale Integration (VLSI-SoC), October 2015.

5. **Yuanwen Huang**, Prabhat Mishra. Vulnerability-aware Energy Optimization using Cache Reconfiguration and Partitioning in Multicore Systems. IEEE International Conference On Computer Aided Design (ICCAD), 2017. (Under Review)

6. **Yuanwen Huang**, Prabhat Mishra. Vulnerability-aware Cache Reconfiguration for Partially Protected Cache. IEEE International Conference on Computer Design (ICCD), 2017. (Under Review)

Journal Papers

1. **Yuanwen Huang** and Prabhat Mishra. Trace Buffer Attack on the AES Cipher. Springer Journal of Hardware and Systems Security (HaSS), 2017.

2. **Yuanwen Huang** and Prabhat Mishra. Vulnerability-aware Energy Optimization for Reconfigurable Caches in Multitasking Systems. IEEE Transaction on Computers (ToC), 2017. (Major Revision)

3. **Yuanwen Huang**, Swarup Bhunia, Prabhat Mishra. A Scalable Side-Channel Analysis-Aware Test Generation for Trojan Detection. IEEE Transaction on Information Forensics and Security (TIFS), 2017. (Under Review)

Book Chapters

1. Formal Approaches to Hardware Trust Verification. Farimah Farahmandi, **Yuanwen Huang** and Prabhat Mishra. The Hardware Trojan War: Attacks, Myths, and Defenses, S. Bhunia and M. Tehranipoor (editors), Springer, 2017.

2. Test Generation for Detection of Malicious Parametric Variations. **Yuanwen Huang** and Prabhat Mishra. Hardware IP Security and Trust, P. Mishra, S. Bhunia and M. Tehranipoor (editors), Springer, 2016.

REFERENCES

[1] Dave Evans. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. Cisco, 2011.

[2] Internet of things research study. 2015 report. Hewlett Packard Enterprise.

[3] S. Bhunia, D. Agrawal and L. Nazhandali, "Guest Editors' Introduction: Trusted System-on-Chip with Untrusted Components," in IEEE Design & Test, vol. 30, no. 2, pp. 5-7, 2013.

[4] The top 10 IoT vulnerabilities. Open Web Application Security Project, 2014. https://www.owasp.org/index.php/Top_IoT_Vulnerabilities

[5] Y. Huang and P. Mishra,"Reliability and energy-aware cache reconfiguration for embedded systems," In Proceedings of the International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, 2016, pp. 313-318.

[6] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (p. 76). IEEE Computer Society Press, 2012.

[7] R. Jeyapaul and A. Shrivastava, "Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors," In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), Taipei, 2011, pp. 105-114.

[8] W. Wang, S. Ranka, P. Mishra, "Dynamic Reconfiguration in Real-Time Systems - Energy, Performance, Reliability and Thermal Perspectives," Springer, 2012.

[9] C. Ekelin, "Clairvoyant non-preemptive EDF scheduling," In Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06), Dresden, 2006, pp. 7.

[10] X. Qin, W. Wang and P. Mishra, "TCEC: Temperature- and Energy-Constrained Scheduling in Real-Time Multitasking Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 31(8), pages 1159-1168, August 2012.

[11] W. Wang, P. Mishra and A. Gordon-Ross, "Dynamic Cache Reconfiguration for Soft Real-Time Systems," ACM Transactions on Embedded Computing Systems (TECS), volume 11, issue 2, 2012.

[12] W. Wang, P. Mishra and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multi-Core Systems", In Proceedings of the ACM/IEEE Design Automation Conference (DAC), pp. 948-953, 2011.

[13] P. Hsu and T. Hwang, “Thread-criticality aware dynamic cache reconfiguration in multi-core system,” In Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD), pp. 413-420, 2013.

[14] Y. Cai, M. Schmitz, A. Ejlali, B. Al-Hashimi and S. Reddy, “Cache size selection for performance, energy and reliability of time-constrained systems,” In Proceedings of Asia and South Pacific Conference on Design Automation (ASP-DAC), 2006, pp. 6.

[15] S. Mittal and J. S. Vetter, “A Survey of Techniques for Modeling and Improving Reliability of Computing Systems,” in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 4, pp. 1226-1238, April 2016.

[16] Y. Ko, R. Jeyapaul, Y. Kim, K. Lee and A. Shrivastava, “Guidelines to design parity protected write-back L1 data cache,” In Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2015, pp. 1-6.

[17] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor.” In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2003, p. 29-.

[18] W. Zhang et al. “An analysis of microarchitecture vulnerability to soft errors on simultaneous multithreaded architectures.” IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS) 2007.

[19] V. Sridharan, H. Asadi, M. B. Tahoori and D. Kaeli, “Reducing Data Cache Susceptibility to Soft Errors,” in IEEE Transactions on Dependable and Secure Computing, vol. 3, no. 4, pp. 353-364, Oct.-Dec. 2006.

[20] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli, “Balancing Performance and Reliability in the Memory Hierarchy,” In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS), pp. 269-279.

[21] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, “Computing Architectural Vulnerability Factors for Address-Based Structures,” In Proceedings of the 32nd annual International Symposium on Computer Architecture, 2005 (ISCA), pp. 532-543.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “MediaBench: a tool for evaluating and synthesizing multimedia and communication systems,” In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30). 1997, pp. 330-335.

[23] http://www.eembc.org. EEMBC, The Embedded Microprocessor Benchmark Consortium.

[24] http://www.simplescalar.com. SimpleScalar: An Infrastructure for Computer System Modeling. Computer, 2002.

[25] http://www.hpl.hp.com/research/cacti/. CACTI, An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model.

[26] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," In Proceedings of the 30th annual International Symposium on Computer Architecture (ISCA). ACM, 2003, pp. 336-349.

[27] H. Hajimiri and P. Mishra, "Intra-Task Dynamic Cache Reconfiguration," In Proceedings of the 25th International Conference on VLSI Design, 2012, pp. 430-435.

[28] T. Adegbija, A. Gordon-Ross and A. Munir, "Dynamic phase-based tuning for embedded systems using phase distance mapping," In Proceedings of the IEEE 30th International Conference on Computer Design (ICCD), 2012, pp. 284-290.

[29] B. Jacob et al. Memory systems: cache, DRAM, disk (Book), Morgan Kaufmann Publishers, 2007.

[30] J. L. Autran et al. Real-time neutron and alpha soft-error rate testing of CMOS 130nm SRAM: Altitude versus underground measurements. IEEE International Conference on IC Design & Technology (ICICDT) 2008.

[31] L. Borucki et al. Comparison of Accelerated DRAM Soft Error Rates Measured at Component and System Level. IEEE International Reliability Physics Symposium (IRPS) 2008.

[32] V. Sridharan et al. Memory Errors in Modern Systems The Good, The Bad, and The Ugly. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2015.

[33] Charlie Slayman. Presentation at IEEE Reliability Society, Santa Clara Valley Chapter. http://www.ewh.ieee.org/r6/scv/rl/articles/Soft%20Error%20mitigation.pdf

[34] J. Suh et al. Soft error benchmarking of L2 caches with PARMA. ACM SIGMETRICS Performance Evaluation Review 2011.

[35] ARM Cortex-A9 and Cortex-A53 Technical Reference Manuals. http://infocenter.arm.com/help/index.jsp

[36] FreeRTOS, the market leading real time operating system. http://www.freertos.org/

[37] Rodriguez, S. and Jacob, B. "Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)." ACM International Symposium on Low Power Electronics and Design (ISLPED), 2006.

[38] K. Lee et al. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[39] K. Lee et al. Partitioning Techniques for Partially Protected Caches in Resource-Constrained Embedded Systems. TODAES, 2010.

[40] A. Shrivastava et. al. Compilation techniques for energy reduction in horizontally partitioned cache architectures. CASES, 2005.

[41] K. Lee et al. Data Partitioning Techniques for Partially Protected Caches to Reduce Soft Error Induced Failures. DIPES, 2008.

[42] M. M. Sabry et al. OCEAN: An Optimized HW/SW Reliability Mitigation Approach for Scratchpad Memories in Real-Time SoCs. ACM TECS, 2014.

[43] N. N. Sadler and D. J. Sorin. Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache. ICCD, 2006.

[44] J.-F. Li and Y.-J. Huang. An Error Detection and Correction Scheme for RAMs with Partial-Write Function. MTDT, 2005.

[45] Doe Hyun Yoon and Mattan Erez. Memory mapped ECC: low-cost error protection for last level caches. SIGARCH Comput. Archit., 2009.

[46] Guthaus, Matthew R., et al. MiBench: A free, commercially representative embedded benchmark suite. WWC, 2001.

[47] W. Wang and P. Mishra. Dynamic Reconfiguration of Two-Level Caches in Soft Real-Time Embedded Systems. ISVLSI, 2009.

[48] A. Gordon-ross and F. Vahid. Automatic tuning of two level caches to embedded applications. DATE, 2004.

[49] C. Zhang et al. A highly configurable cache for low energy embedded systems. ACM TECS, 2005.

[50] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. DAC, 2007.

[51] S. Mittal et al. Master: A multicore cache energy-saving technique using dynamic cache reconfiguration. IEEE TVLSI, 2014.

[52] A. Settle et al. A dynamically reconfigurable cache multithreaded processors. J. of Embed. Comput., vol. 2, pp. 221-233, 2006.

[53] D. Kaseridis et al. Bank-aware dynamic cache partitioning for multicore architectures. ICPP, 2009.

[54] R. Reddy and P. Petrov. Cache partitioning for energy-efficient and interference-free embedded multitasking. ACM TECS, 2010.

[55] M. Maghsoudloo and H. Zarandi. Cache vulnerability mitigation using an adaptive cache coherence protocol. Supercomputing, 2014.

[56] N. Binkert et al. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2, August 2011.

[57] CPU2000. SPEC CPU2000. http://www.spec.org/cpu2000.

[58] Y. Huang et al., "Trace Buffer Attack: Security versus observability study in post-silicon debug," VLSI-SoC, 2015.

[59] A. Bogdanov et al., "Biclique cryptanalysis of the full AES," in *Advances in Cryptology: ASIACRYPT 2011*.

[60] A. Moradi et al., "Pushing the limits: A very compact and a threshold implementation of AES," in *Advances in Cryptology: EUROCRYPT 2011*.

[61] *FIPS 197, Advanced Encryption Standard*, 2001. [Online]. Available: csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[62] *ARM Embedded Trace Buffer*, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0168b/ar01s03s03.html

[63] P. C. Kocher et al., "Differential power analysis," in CRYPTO , 1999.

[64] D. Osvik et al., "Cache attacks and countermeasures: The case of AES," CT-RSA, 2006.

[65] D. Genkin et al., "Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation," Cryptology ePrint Archive, Report 2015/170, 2015.

[66] D. Genkin et al., "RSA key extraction via low-bandwidth acoustic cryptanalysis," CRYPTO, 2014.

[67] S. Skorobogatov and R. Anderson, "Optical fault induction attacks,"Springer, 2003.

[68] C. Rebeiro et al., *Timing Channels in Cryptography: A Micro-Architectural Perspective*, Springer, 2015 edition.

[69] A. Barenghi et al., "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, 2012.

[70] S. Bhunia et al., "Hardware trojan attacks: Threat analysis and countermeasures," *Proceedings of the IEEE*, 2014.

[71] B. Ege et al., "Differential scan attack on AES with x-tolerant and x-masked test response compactor," in *Digital System Design (DSD), 2012*.

[72] S. Ali et al., "Test-mode-only scan attack using the boundary scan chain," in ETS, 2014.

[73] D. Mukhopadhyay, "An improved fault based attack of the advanced encryption standard," in *Progress in Cryptology: AFRICACRYPT 2009*.

[74] S. Ali et al., "AES design space exploration new line for scan attack resiliency," in VLSI-SoC, 2014.

[75] F. Regazzoni et al., "Interaction between fault attack countermeasures and the resistance against power analysis attacks," in *Fault Analysis in Cryptography*, 2012.

[76] K. Basu and P. Mishra, "RATS: restoration-aware trace signal selection for post-silicon validation," *IEEE Trans. VLSI Syst.*, 2013.

[77] D. Chatterjee et al., "Simulation-based signal selection for state restoration in silicon debug," in ICCAD, 2011.

[78] M. Li and A. Davoodi, "A hybrid approach for fast and accurate trace signal selection for post-silicon debug," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2014.

[79] K. Rahmani et al., "Efficient trace signal selection using augmentation and ILP techniques," in ISQED 2014.

[80] *OpenCores AES-128 cipher*, [Online]. Available: http://opencores.org/project,aes_core

[81] *OpenCores AES ciphers (all key sizes)*, [Online]. Available: http://opencores.org/project,tiny_aes

[82] S. Banik et al., "Cryptanalysis of the double-feedback xor-chain scheme proposed in Indocrypt 2013", *INDOCRYPT 2014*.

[83] D. Hely et al., "Scan design and secure chip [secure IC testing]," IEEE International On-Line Testing Symposium, 2004, pp. 219-224.

[84] B. Yang et al., "Scan Based Side Channel Attack on Data Encryption Standard," IACR Cryptology ePrint Archive 2004: 83.

[85] B. Yang et al., "Secure Scan: A design-for-test architecture for crypto chips," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2006, 25(10), 2271-2276.

[86] D. Mukhopadhyay et al., "CryptoScan: A Secured Scan Chain Architecture," 14th Asian Test Symposium (ATS), 2005, pp. 348-353.

[87] S. Paul et al., "VIm-Scan: A Low Overhead Scan Design Approach for Protection of Secret Key in Scan-Based Secure Chips," IEEE VLSI Test Symposium (VTS), 2007, pp. 455460.

[88] G. Sengar et al., "Secured Flipped Scan-Chain Model for Crypto-Architecture," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2007), 26, 2080-2084.

[89] C. Herder et al., “Physical unclonable functions and applications: A tutorial,” Proceedings of the IEEE (2014), 102(8), pp. 1126-1141.

[90] S. Devadas et al., “Design and implementation of PUF-Based unclonable RFID ICs for anti-counterfeiting and security applications,” IEEE Int. Conf. RFID, (2008), pp. 5864.

[91] M. Bhargava and K. Mai, “An efficient reliable PUF-based cryptographic key generator in 65nm CMOS,” Design, Automation & Test in Europe (DATE 2014), pp. 1-6.

[92] E. Zenner. “Cryptanalysis of LFSR-based pseudorandom generators-a survey,” Technical Report Informatik 04-004, University of Mannheim, 2004.

[93] S. Banik and A. Bogdanov, “Cryptanalysis of Two Fault Countermeasure Schemes”, Progress in Cryptology – INDOCRYPT 2015, pp 241-252.

[94] F. Farahmandi, Y. Huang, P. Mishra. Trojan Localization using Symbolic Algebra. Asia and South Pacific Design Automation Conference (ASPDAC), 2017.

[95] Y. Huang, S. Bhunia, P. Mishra. MERS: Statistical Test Generation for Side-Channel Analysis based Trojan Detection. ACM Conference on Computer and Communications Security (CCS), 2016.

[96] R. Chakraborty, S. Narasimhan and S. Bhunia. Hardware Trojan: Threats and emerging solutions. IEEE International High-Level Design Validation and Test Workshop (HLDVT), 2009.

[97] DARPA: TRUST in Integrated Circuits (TIC), 2007. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA503809

[98] R. Chakraborty and S. Bhunia. Security against hardware Trojan through a novel application of design obfuscation. ACM International Conference on Computer-Aided Design (ICCAD), pp. 113-116, 2009.

[99] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi and V. De. Parameter variations and impact on circuits and microarchitecture. ACM/IEEE Design Automation Conference (DAC), pp. 338-342, 2003.

[100] R. Chakraborty, F. Wolff, S. Paul, C. Papachristou and S. Bhunia. MERO: A Statistical Approach for Hardware Trojan Detection. International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp. 396-410, 2009.

[101] S. Saha, R. Chakraborty, S. Nuthakki, Anshul, and D. Mukhopadhyay. Improved Test Pattern Generation for Hardware Trojan Detection Using Genetic Algorithm and Boolean Satisfiability. International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp. 577-596 (2015).

[102] Y. Jin and Y. Makris. Hardware Trojan detection using path delay fingerprint. IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2008.

[103] M. Banga and M. Hsiao. A region based approach for the identification of hardware Trojans. IEEE International Workshop on Hardware-Oriented Security and Trust (HOST), 2008.

[104] M. Banga, M. Chandrasekar, L. Fang and M. Hsiao. Guided test generation for isolation and detection of embedded Trojans in ICs. ACM Great Lakes Symposium on VLSI (GLSVLSI), pp. 363-366, 2008.

[105] Y. Jin and Y. Makris. Hardware Trojan detection using path delay fingerprint. IEEE International Workshop on Hardware-Oriented Security and Trust (HOST), 2008.

[106] S. Wei and M. Potkonjak. Scalable hardware Trojan diagnosis. IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 20(6), pp. 1049-1057, 2012.

[107] R. Rad, J. Plusquellic and M. Tehranipoor. A sensitivity analysis of power signal methods for detecting hardware Trojans under real process and environmental conditions. IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 18(12), pp. 1735-1744, 2010.

[108] H. Salmani and M. Tehranipoor. Layout-Aware Switching Activity Localization to Enhance Hardware Trojan Detection. IEEE Transactions on Information Forensics and Security, 7(1), pp. 76-87, 2012.

[109] S. Dupuis, P. Ba, G. Natale, M. Flottes, and B. Rouzeyre. A novel hardware logic encryption technique for thwarting illegal overproduction and Hardware Trojans. IEEE 20th International On-Line Testing Symposium (IOLTS), pp. 49-54, 2014.

[110] J. Rajendran, Y. Pino, O. Sinanoglu and R. Karri. Security analysis of logic obfuscation. ACM/IEEE Design Automation Conference, pp. 83-89, 2012.

[111] S. Shekarian, M. Zamani and S. Alami. Neutralizing a design-for-hardware trust technique. International Symposium on Computer Architecture and Digital Systems (CADS), pp. 73-78, 2013.

[112] X. Mingfu, H. Aiqun and L. Guyue: Detecting Hardware Trojan through Heuristic Partition and Activity Driven Test Pattern Generation. Communications Security Conference (CSC), pp. 1-6, 2014.

[113] H. Salmani, M. Tehranipoor and J. Plusquellic. A novel technique for improving hardware Trojan detection and reducing Trojan activation time. IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 20(1), pp. 112-125, 2012.

[114] B. Zhou, W. Zhang, S. Thambipillai, and J. Teo. A low cost acceleration method for hardware Trojan detection based on fan-out cone analysis. ACM International Conference on Hardware Software Codesign and System Synthesis, p. 28, 2014.

[115] I. Pomeranz and S. Reddy. A measure of quality for n-detection test sets. IEEE Transactions on Computers, 53(11), pp. 1497-1503, 2004.

[116] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi and B. Sunar. Trojan Detection using IC Fingerprinting. IEEE Symposium on Security and Privacy, pp. 296-310, 2007.

[117] X. Wang, H. Salmani, M. Tehranipoor and J. Plusquellic. Hardware Trojan Detection and Isolation Using Current Integration and Localized Current Analysis. IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems, pp. 87-95, 2008.

[118] S. Narasimhan, D. Du, R. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy and S. Bhunia. Hardware Trojan detection by multiple-parameter side-channel analysis. IEEE Transactions on Computers, 62(11), pp. 2183-2195, 2013.

[119] D. Du, S. Narasimhan, R. Chakraborty and S. Bhunia. Self-referencing: a scalable side-channel approach for hardware trojan detection. International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp. 173-187, 2010.

BIOGRAPHICAL SKETCH

Yuanwen Huang received the B.E. degree in Electrical Engineering from the Department of Control Science and Engineering, Huazhong University of Science and Technology, China, in 2012. He received the Ph.D. degree in Computer Engineering from the Department of Computer and Information Science and Engineering, University of Florida, USA, in 2017. His research interests included system reliability and security, hardware security validation, energy optimization of caches in embedded systems. He was a recipient of the Best Paper Award from the International Symposium on Quality Electronic Design in 2016.