UC San Diego UC San Diego Electronic Theses and Dissertations

Title

Defusing the Tension between Security and Performance with Secure Microarchitectures

Permalink

https://escholarship.org/uc/item/0dr022gj

Author Taram, Mohammadkazem

Publication Date 2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Defusing the Tension between Security and Performance with Secure Microarchitectures

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Mohammadkazem Taram

Committee in charge:

Professor Dean Tullsen, Chair Professor Ryan Kastner Professor Farinaz Koushanfar Professor Alex Snoreen Professor Deian Stefan

2022

Copyright Mohammadkazem Taram, 2022 All rights reserved. The Dissertation of Mohammadkazem Taram is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

Dissertation Approval Page	iii	
Table of Contents iv		
List of Figures vi		
List of Tables	xi	
Acknowledgements	xii	
Vita	xiv	
Abstract of the Dissertation	xvi	
Introduction1Novel Microarchitectural Attacks2Architecture Support for Security3Secure Speculative Execution4Secure Multi-Threading Architectures	1 3 4 5 6	
5 Outline	8	
 Chapter 1 Packet Chasing	9 11 13 13 14 16 17	
 1.2.1 Deconstruction of the NIC Driver 1.2.2 Recovering the Cache Footprint of the Ring Buffer 1.2.3 Chasing Packets over the Cache 1.3 Packet Chasing: Receiving Packets without Network Access 1.4 Packet Chasing: Exploiting Packet Size 1.5 Potential Software Mitigation 1.6 Adaptive I/O Cache Partitioning Defense 1.7 Conclusions 	17 20 24 27 34 36 37 41	
 Chapter 2 Context Sensitive Decoding	44 47 51 51 52	

TABLE OF CONTENTS

2.2.4Performance Guidelines and Optimizations552.2.5Potential Applications572.3Case Study 1:Side-Channel Defense582.3.1Assumptions and Threat Model582.3.2Stealth-Mode Translation592.3.3Securing the Instruction Cache632.3.4Securing the Data Cache632.3.5Securing other Side-Channels642.4Case Study II: Unit-Level Power Gating682.5.1Performance Evaluation682.5.2Security Evaluation702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Mitogode1003.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.6.1Security Discussion1033.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Sakground and Related Work1174.2.3Microarchitectural Covert/ Side Channels1244.3Assumptions and Threat Model125 <t< th=""><th></th><th>2.2.3 Microcode Update and Auto-Translation</th><th>54</th></t<>		2.2.3 Microcode Update and Auto-Translation	54
2.2.5Potential Applications572.3Case Study LSide-Channel Defense582.3.1Assumptions and Threat Model582.3.2Stealth-Mode Translation592.3.3Securing the Instruction Cache622.3.4Securing the Data Cache632.3.5Securing other Side-Channels.642.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1053.6.2Performance1074.2.3Microarchitectural Overview1074.2.4SMT Covert and Side Channels1214.2.4SMT Covert / Side Channels1214.2.5Side-Channel Mitigation1244.2.4SMT Covert / Side Channels1214.2.4SMT Covert / Side Channels1		2.2.4 Performance Guidelines and Optimizations	55
2.3Case Study I:Side-Channel Defense582.3.1Assumptions and Threat Model582.3.2Stealth-Mode Translation592.3.3Securing the Instruction Cache622.3.4Securing the Data Cache632.3.5Securing other Side-Channels642.4Case Study II: Unit-Level Power Gating652.5.1Performance Evaluation682.5.2Security Evaluation702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization1003.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion1154.1Introduction1154.2Background and Related Work1154.2Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.2 <td></td> <td>2.2.5 Potential Applications</td> <td>57</td>		2.2.5 Potential Applications	57
2.3.1Assumptions and Threat Model582.3.2Stealth-Mode Translation592.3.3Securing the Instruction Cache622.3.4Securing the Data Cache632.3.5Securing other Side-Channels642.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization Flow Tracking1003.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1224.3Assumptions and Threat Model1224.4Side-Channel Mitigation1244.3Assumptions and Threat Model1224.4Side-Channel Mitigation<	2.3	Case Study I:Side-Channel Defense	58
2.3.2Stealth-Mode Translation592.3.3Securing the Instruction Cache622.3.4Securing the Data Cache632.3.5Securing other Side-Channels642.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.2Decoder-Level Information Flow Tracking1003.4.2Decoder-Level Information Flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion1154.1Introduction1154.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1224.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Mitigation1244.3 <td>-</td> <td>2.3.1 Assumptions and Threat Model</td> <td>58</td>	-	2.3.1 Assumptions and Threat Model	58
2.3.3Securing the Instruction Cache622.3.4Securing the Data Cache632.3.5Securing other Side-Channels.642.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1174.2.1The x86 Pipeline Resources/Structures1174.2.2Side-Channel Mitigation1224.2.3Microarchitectural Covert / Side Channels1224.2.4SMI Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Mod		2.3.2 Stealth-Mode Translation	59
2.3.4 Securing the Data Cache 63 2.3.5 Securing other Side-Channels. 64 2.4 Case Study II: Unit-Level Power Gating 65 2.5 Methodology 68 2.5.1 Performance Evaluation 68 2.5.2 Security Evaluation 70 2.6 Results 70 2.6.1 Stealth Mode 70 2.6.2 Selective Devectorization 74 2.7 Conclusion 77 Chapter 3 Context-Sensitive Fencing 79 3.1 Background and Related Work 82 3.2 Assumptions and Threat Model 89 3.4 Design and Implementation 92 3.4.1 Microcode Customization 93 3.4.2 Decoder-Level Information Flow Tracking 100 3.4.3 Mitigations for Control-flow Mistraining 102 3.5 Methodology 104 3.6 Evaluation 105 3.6.1 Security Discussion 106 3.6.2 Performance 107 3.7		2.3.3 Securing the Instruction Cache	62
2.3.5Securing other Side-Channels.642.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work.823.2Assumptions and Threat Model.893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining.1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.2.2<		2.3.4 Securing the Data Cache	63
2.4Case Study II: Unit-Level Power Gating652.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2.3Microarchitectural Covert/Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.2.2Side-Channel Mutigation124		2.3.5 Securing other Side-Channels	64
2.5Methodology682.5.1Performance Evaluation682.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Sidcroarchitectural Covert/ Side Channels1214.2.3Microarchitectural Covert/ Side Channels1224.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	2.4	Case Study II: Unit-Level Power Gating	65
2.5.1 Performance Evaluation 68 2.5.2 Security Evaluation 70 2.6 Results 70 2.6.1 Stealth Mode 70 2.6.2 Selective Devectorization 74 2.7 Conclusion 77 Chapter 3 Context-Sensitive Fencing 79 3.1 Background and Related Work 82 3.2 Assumptions and Threat Model 89 3.3 Architectural Overview 90 3.4 Design and Implementation 92 3.4.1 Mitogations for Control-flow Mistraining 100 3.4.2 Decoder-Level Information Flow Tracking 100 3.4.3 Mitigations for Control-flow Mistraining 102 3.5 Methodology 104 3.6 Evaluation 105 3.6.1 Security Discussion 106 3.6.2 Performance 107 3.7 Conclusion 113 Chapter 4 Secure Simultaneous Multithreading 115 4.1 Introduction 115 4.2	2.5	Methodology	68
2.5.2Security Evaluation702.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		2.5.1 Performance Evaluation	68
2.6Results702.6.1Stealth Mode702.6.2Selective Devectorization742.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6I Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4.1Overview1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		2.5.2 Security Evaluation	70
2.6.1 Stealth Mode 70 2.6.2 Selective Devectorization 74 2.7 Conclusion 77 Chapter 3 Context-Sensitive Fencing 79 3.1 Background and Related Work 82 3.2 Assumptions and Threat Model 89 3.3 Architectural Overview 90 3.4 Design and Implementation 92 3.4.1 Microcode Customization 93 3.4.2 Decoder-Level Information Flow Tracking 100 3.4.3 Mitigations for Control-flow Mistraining 102 3.5 Methodology 104 3.6 Evaluation 105 3.6.1 Security Discussion 106 3.6.2 Performance 107 3.7 Conclusion 113 Chapter 4 Secure Simultaneous Multithreading 115 4.1 Introduction 115 4.1 Rackground and Related Work 117 4.2.2 Simultaneous Multithreading 117 4.2.3 Microarchitectural Covert/ Side Channels 121	2.6	Results	70
2.6.2 Selective Devectorization 74 2.7 Conclusion 77 Chapter 3 Context-Sensitive Fencing 79 3.1 Background and Related Work 82 3.2 Assumptions and Threat Model 89 3.3 Architectural Overview 90 3.4 Design and Implementation 92 3.4.1 Microcode Customization 93 3.4.2 Decoder-Level Information Flow Tracking 100 3.4.3 Mitigations for Control-flow Mistraining 102 3.5 Methodology 104 3.6 Evaluation 105 3.6.1 Security Discussion 106 3.6.2 Performance 107 3.7 Conclusion 113 Chapter 4 Secure Simultaneous Multithreading 115 4.1 Introduction 117 4.2.2 Simultaneous Multithreading 119 4.2.3 Microarchitectural Covert/ Side Channels 121 4.2.4 SMT Covert and Side Channels 122 4.2.5 Side-Channel Mitigation 124 <td></td> <td>2.6.1 Stealth Mode</td> <td>70</td>		2.6.1 Stealth Mode	70
2.7Conclusion77Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		2.6.2 Selective Devectorization	74
Chapter 3 Context-Sensitive Fencing 79 3.1 Background and Related Work 82 3.2 Assumptions and Threat Model 89 3.3 Architectural Overview 90 3.4 Design and Implementation 92 3.4.1 Microcode Customization 93 3.4.2 Decoder-Level Information Flow Tracking 100 3.4.3 Mitigations for Control-flow Mistraining 102 3.5 Methodology 104 3.6 Evaluation 105 3.6.1 Security Discussion 106 3.6.2 Performance 107 3.7 Conclusion 113 Chapter 4 Secure Simultaneous Multithreading 115 4.1 Introduction 115 4.2 Background and Related Work 117 4.2.1 The x86 Pipeline Resources/Structures 117 4.2.2 Simultaneous Multithreading 119 4.2.3 Microarchitectural Covert/ Side Channels 121 4.2.4 SMT Covert and Side Channels 122 4.2.5 Side-Channel Mi	2.7	Conclusion	77
Chapter 3Context-Sensitive Fencing793.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Sackground and Related Work1174.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	-		
3.1Background and Related Work823.2Assumptions and Threat Model893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	Chapter	r 3 Context-Sensitive Fencing	79
3.2Assumptions and Threat Model.893.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Overview1274.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.1	Background and Related Work	82
3.3Architectural Overview903.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.2	Assumptions and Threat Model	89
3.4Design and Implementation923.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading4.1Introduction1154.2Background and Related Work1174.2.3Simultaneous Multithreading1194.2.4SMT Covert and Side Channels1214.2.5Side-Channel Mitigation1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.3	Architectural Overview	90
3.4.1Microcode Customization933.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading4.1Introduction4.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.4	Design and Implementation	92
3.4.2Decoder-Level Information Flow Tracking1003.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		3.4.1 Microcode Customization	93
3.4.3Mitigations for Control-flow Mistraining1023.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading4.1Introduction1154.14.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		3.4.2 Decoder-Level Information Flow Tracking	100
3.5Methodology1043.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		3.4.3 Mitigations for Control-flow Mistraining	102
3.6Evaluation1053.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Overview1274.4.2Instruction Fetch Bandwidth129	3.5	Methodology	104
3.6.1Security Discussion1063.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading4.1Introduction1154.24.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.6	Evaluation	105
3.6.2Performance1073.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		3.6.1 Security Discussion	106
3.7Conclusion113Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		3.6.2 Performance	107
Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	3.7	Conclusion	113
Chapter 4Secure Simultaneous Multithreading1154.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Overview1274.4.1Overview1274.4.2Instruction Fetch Bandwidth129			
4.1Introduction1154.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	Chapter	r 4 Secure Simultaneous Multithreading	115
4.2Background and Related Work1174.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Overview1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	4.1	Introduction	115
4.2.1The x86 Pipeline Resources/Structures1174.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	4.2	Background and Related Work	117
4.2.2Simultaneous Multithreading1194.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		4.2.1The x86 Pipeline Resources/Structures	117
4.2.3Microarchitectural Covert/ Side Channels1214.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		4.2.2 Simultaneous Multithreading	119
4.2.4SMT Covert and Side Channels1224.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		4.2.3 Microarchitectural Covert/ Side Channels	121
4.2.5Side-Channel Mitigation1244.3Assumptions and Threat Model1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		4.2.4 SMT Covert and Side Channels	122
4.3Assumptions and Threat Model.1254.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129		4.2.5 Side-Channel Mitigation	124
4.4Covert Channel Characterization1264.4.1Overview1274.4.2Instruction Fetch Bandwidth129	4.3	Assumptions and Threat Model	125
4.4.1Overview1274.4.2Instruction Fetch Bandwidth129	4.4	Covert Channel Characterization	126
4.4.2Instruction Fetch Bandwidth129		4.4.1 Overview	127
		4.4.2 Instruction Fetch Bandwidth	129

	4.4.3	Decode/Issue Bandwidth	131
	4.4.4	Register File	132
	4.4.5	Reservation Station (RS)/Scheduler	134
	4.4.6	Reorder Buffer, Load/Store Queues	135
	4.4.7	Covert Channel on Partitioned Structures	136
	4.4.8	Other Pipeline Resources	139
4.5	Mitiga	tions	140
	4.5.1	Static Partitioning	140
	4.5.2	Adaptive Partitioning	141
	4.5.3	Asymmetric SMT	142
4.6	Metho	dology	150
4.7	Result	s	153
	4.7.1	Security Evaluation	153
	4.7.2	Performance Evaluation	154
	4.7.3	Parameter Search for Adaptive	159
	4.7.4	Partitioning Effects on Other Resources	162
	4.7.5	RTL Model of Asymmetric SMT	164
4.8	Conclu	ision	165
~ .	-		
Chapter	$\frac{15}{2}$ Co	onclusion and Future Directions	167
5.1	Future	e Directions	168
D:1-1:	1		
Dibliogr	apny .	•••••••••••••••••••••••••••••••••••••••	170

LIST OF FIGURES

Figure 1.1.	The shared ring buffers (FIFO) between NIC and the device driver.	12
Figure 1.2.	Intel's complex indexing of modern last level cache	15
Figure 1.3.	The IGB driver function that adds the contents of an incoming buffer to a socket_buffer which will be passed to the higher levels of networking stack. The function returns true if the buffer can be reused by the NIC.	19
Figure 1.4.	The IGB driver function that checks if the driver can reuse a page and put it back into the rx ring buffer	19
Figure 1.5.	An example of how the NIC ring buffers are mapped to to the page-aligned cache sets.	21
Figure 1.6.	Frequency of the ring buffers that map to same sets, measured for 1000 instances. Zero represents the number of sets that are page aligned but none of the ring buffers is mapped to those	22
Figure 1.7.	Monitoring all the page-aligned sets while receiving packets. A white dot shows at least one miss (activity) on a cache set in a sample interval	22
Figure 1.8.	Cache footprint of packets with different sizes while probing the addresses that map to the location of the first three blocks in the packet buffer page. A white dot indicates at least a miss in a set	23
Figure 1.9.	Pruning and sequencing of the set graph to get the order of ring buffers. Each node represents a set in the attacker address space. Numbers in squares are the sequence number of the associated ring buffers that map to same set.	25
Figure 1.10.	Spy process decodes the transmitted sequence based on the mon- itored activity on the probed sets. Set 1 acts as the clock and the activity is one for a set if we find at least one miss in the blocks in the eviction set of the probed set.	30
Figure 1.11.	Bandwidth and error rate of the remote covert channel for binary and ternary encoding and various cache probe rates.	30
Figure 1.12.	(a) and (b) show the channel capacity for the remote covert channel where the spy that uses n buffers of the ring's sequence information.(c) and (d) show the out of sync rate and error rate for the case where spy uses all the buffers in the sequence information	33

Figure 1.13.	Detecting successful login for hotcrp.com. Shows original packet sizes vs. the recovered packet sizes by Packet Chasing for the first	24
		34
Figure 1.14.	Performance impact of our adaptive partitioning defense on Nginx web server.	40
Figure 1.15.	Memory Traffic and LLC miss rate of our adaptive partitioning defense vs. DDIO.	40
Figure 1.16.	Comparison of our defenses in terms of response (tail) latency of HTTP requests to the Nginx web server. Randomization period is the interval (measured in number of packets) that we wait between two ring buffer randomizations	41
Figure 2.1.	Intel Front End with CSD Support	52
Figure 2.2.	Auto-translation Procedure	56
Figure 2.3.	Context-Sensitive Decoding with stealth mode	60
Figure 2.4.	Translation of MOV instruction in stealth mode	60
Figure 2.5.	CSD with Selective Devectorization	65
Figure 2.6.	Devectorization of add byte instruction	67
Figure 2.7.	Effect of the cache attacks on AES and RSA with stealth-mode translation enabled.	69
Figure 2.8.	The execution time impact of context-sensitive decoding when implementing secure cache obfuscation normalized to insecure execution mode. The watchdog timer is set so that secure mode is re-entered every 500 microseconds	71
Figure 2.9.	Micro-op expansion due to CSD.	73
Figure 2.10.	Number of cache misses without and with CSD	73
Figure 2.11.	Effect of CSD timer on execution time	73
Figure 2.12.	Total energy consumption of CSD's devectorizing mode normalized to that of power-gating	74
Figure 2.13.	Execution time for different power gating policies, normalized to always on policy	75

Figure 2.14.	Micro-op expansion due to context sensitive decoding normalized to native mode	76
Figure 2.15.	Percentage of time that CSD power gates VPUs	77
Figure 2.16.	Breakdown of vector activity	77
Figure 3.1.	Example Spectre Variant-1 Gadget	84
Figure 3.2.	Mitigating Spectre-v1 using a Fence Instruction	85
Figure 3.3.	Architectural Overview	91
Figure 3.4.	Fence Enforcement Points	93
Figure 3.5.	DLIFT integration with a CSD-enabled pipeline	100
Figure 3.6.	Execution Time of Different Fence Enforcement Levels (normalized to insecure execution)	107
Figure 3.7.	Execution Time of Early and Late Commit of CFENCE (normalized to insecure execution)	108
Figure 3.8.	Effects of injecting CFENCE on Cache Miss Rate	109
Figure 3.9.	Impact of Fence Frequency Optimizations on Execution Time	111
Figure 3.10.	Accuracy of DLIFT: Overtainting Rate of Loads	111
Figure 3.11.	Coverage of DLIFT: Undertainting Rate of Loads	112
Figure 3.12.	Ratio of instructions marked as tainted by DLIFT	113
Figure 4.1.	Simplified Architecture of a Modern X86 Processor	120
Figure 4.2.	Reverse Engineering the physical register file sharing mechanism	134
Figure 4.3.	Reverse Engineering the Reservation Station Sharing Mechanism	135
Figure 4.4.	Reverse Engineering the SQ Sharing Mechanism. In single-thread mode, we see a spike in the execution time at 36, exactly the size of our SQ. In SMT mode, T2 only executes NOPs, but still causes T1's SQ to be halved.	137
Figure 4.5.	Adaptive Partitioning Examples.	143

Figure 4.6.	Borrowing a Physical Register in Asymmetric SMT. It shows the state of the physical register file over time for two scenarios 146
Figure 4.7.	Partitioning Schemes for Execution Units/Ports. Our adaptive partitioning and Asymmetric SMT architecture can reclaim the unused execution slots caused partitioning
Figure 4.8.	Covert Channels between a Spy (To) and a Trojan (T1) Thread. In a fully shared pipeline the instructions executed on T1 have a clear effect on To's execution time. <i>Partitioned, Adaptive</i> and <i>Asymmetric</i> are always constant and share the same straight line
Figure 4.9.	Performance of the Proposed Schemes 154
Figure 4.10.	Running Trusted Cryptography Computation with Untrusted JavaScript Code
Figure 4.11.	Partitioning Schemes for Functional Units 156
Figure 4.12.	Partitioning Schemes for Fetch Bandwidth 157
Figure 4.13.	Partitioning Schemes for Caches 158
Figure 4.14.	Parameter Search for Adaptive Partitioning of Three Example State- ful Resources 160
Figure 4.15.	Parameter Search for Adaptive Partitioning of Two Example State- less Resources. We only increase the share of a thread, if the full counter of that thread is larger than <i>Mult</i> times of the counter of the other thread
Figure 4.16.	Impact of Partitioning Schemes on Individual Resource 163

LIST OF TABLES

Table 1.1.	Summary of experiments for sequence recovery	27
Table 1.2.	Architecture Detail for Baseline Processor	36
Table 2.1.	Architecture detail for the baseline x86 core	68
Table 3.1.	Speculative Attacks Variants	84
Table 3.2.	List of Intel's Serializing Instructions	92
Table 3.3.	Characteristics of Different Fence Types	97
Table 3.4.	Architecture Detail for the Baseline x86 Core	103
Table 3.5.	Benchmarks Description	105
Table 4.1.	Sharing Mechanism of Pipeline Resources	127
Table 4.2.	Architecture Detail for the Baseline x86 Core	150
Table 4.3.	Delay, Area, and Power Results for Different Implementation of the Dispatch Unit.	163

ACKNOWLEDGEMENTS

This dissertation marks the end of my PhD journey. A journey full of challenges, joys, and learning opportunities. A journey that I could not complete without the support and help from a large number of individuals who helped, supported, and inspired me along the way.

First and foremost, I would like to express my immense gratitude to my advisor Dean Tullsen for his unstinting support, his invaluable guidance, and his seemingly endless patience. Dean has been an ingenious researcher and teacher and an inspiring role-model. My PhD journey would not be as successful and memorable were it not for Dean and his support. I am forever indebted to Dean as he shaped my academic and research personality in the best possible way.

Many thanks are due to my PhD committee members, Ryan Kastner, Farinaz Koushanfar, Alex Snoreen, and Deian Stefan for their valuable insights and feedback. I would like to particularly thank Ryan and Deian for also helping me with my academic job search.

I would like to extend my sincere thanks to my lab-mates who have been supporting friends. In particular I am extremely grateful to Ashish Venkat for being an excellent mentor and a caring friend. This dissertation would not be as successful without his support and mentorship.

I also wish to thank my dear friends for their support and kindness throughout my PhD and beyond that.

I would like to thank my parents and my sisters for their unconditional support throughout my life. Thanks should also go to my host family, George and Carol who kindly hosted me in the first few weeks of my PhD and helping me when I needed it most.

Finally, a very special thanks to my partner during these year, Niloofar who supported me in every way throughout my PhD.

The Introduction, in part, uses material from all works listed below.

Chapter 1, in full, is a reprint of the material as it appears in International Symposium on Computer Architecture (ISCA) 2020. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in full, is a reprint of the material as it appears in International Symposium on Computer Architecture (ISCA) 2018. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2019. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in USENIX Security Symposium (USENIX Security) 2022. Taram, Mohammadkazem; Ren, Xida; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

VITA

2013	Bachelor of Science, Computer Engineering, Shahid Beheshti University
2015	Master of Science, Computer Engineering, Sharif University of Technology
2016–2022	Research Assistant, Department of Computer Science and Engineering University of California San Diego
2021	Lecturer, Department of Computer Science and Engineering University of California San Diego
2022	Doctor of Philosophy, Computer Science (Computer Engineering) University of California San Diego

PUBLICATIONS

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen, "Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency," International Symposium on Computer Architecture (ISCA), June. 2018.

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April. 2019.

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen, "Context-Sensitive Decoding: On-Demand Microcode Customization for Security and Energy Management," in IEEE Micro Special Issue on Top Picks from Computer Architecture Conferences, vol. 39, no. 3, pp. 75-83, May-June 2019.

Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh, "Shredder: Learning Noise Distributions to Protect Inference Privacy," Architectural Support for Programming Languages and Operating Systems (ASPLOS), March. 2020.

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen, "Packet Chasing: Spying on Network Packets over a Cache Side-Channel," International Symposium on Computer Architecture (ISCA), June. 2020.

Fatemehsadat Mireshghallah, Mohammadkazem Taram, Ali Jalali, Ahmed Taha Elthakeb, Dean Tullsen, and Hadi Esmaeilzadeh, "Not All Features Are Equal: Discovering Essential Features for Preserving Prediction Privacy," The Web Conference (WWW), April. 2021.

Xida Ren, Logan Moody, Mohammadkazem Taram, Dean Tullsen, and Ashish Venkat "I See Dead μ ops: Leaking Secrets via Intel/AMD Micro-Op Caches," International Symposium on Computer Architecture (ISCA), June. 2021.

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen, "Mitigating Speculative Execution Attacks via Context-Sensitive Fencing," IEEE Design & Test, vol. 39, no. 4, pp. 49-57, August. 2022.

Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen, "SecSMT: Securing SMT Processors against Contention-Based Covert Channels," USENIX Security Symposium (USENIX Security), August. 2022.

ABSTRACT OF THE DISSERTATION

Defusing the Tension between Security and Performance with Secure Microarchitectures

by

Mohammadkazem Taram

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2022

Professor Dean Tullsen, Chair

The pursuit of secure computation has always featured a tension between performance and security. Security mitigations often come with a high performance cost that can be manifested in serious environmental and economic impacts if they are employed, and in disastrous security and privacy breaches, if not. In the context of processor architectures, the security-performance tension is only growing as new attacks appear, each exploiting a crucial performance optimization, threatening to unwind decades of architectural gains. These hosts of attacks on microarchitectural optimizations painfully coincide with an era in which those performance optimizations are needed most – an era when Moore's law is fading and Denard's scaling is gone.

In this dissertation we strive to defuse this security-performance tension by deepening our understanding of vulnerabilities in modern processors, providing efficient hardware support to enable security, and designing new high-performance secure architectures. We first show how performance optimizations can have devastating security implications by introducing a novel microarchitectural side-channel attack that targets Data Direct IO, a network packet processing optimization implemented by Intel (Chapter 1). Then, we propose Context-Sensitive Decoding (CSD), a framework that takes advantage of the instruction-to-micro-op translation that exists in most modern processors to provide security features (Chapter 2). Finally, we propose novel secure and fast architectures to mitigate vulnerabilities in two of the most crucial performance optimizations in modern processors: Speculative Execution (Chapter 3) and Simultaneous Multithreading (Chapter 4).

Introduction

Security and performance are two fundamental goals in the design of modern computing systems. These two, unfortunately, have often conflicting demands. For example, isolation, an intrinsic security property, is fundamentally at odds with sharing, an often desired performance property [155]. Therefore, as we progress towards systems with higher performance, security becomes more challenging [155]. Modern high-performance processors share a plethora of resources (e.g., caches, branch predictors, functional units, etc.) potentially between different security domains. The security implications of this sharing, however, remained historically overlooked until the 2000s, when Percival [212] showed a cache-based side-channel attack capable of successfully breaking the AES encryption. In addition, software security and isolation mechanisms are built upon a set of assumptions that are often violated by complex performance optimizations in modern processors. As a result, any break in those assumptions can break the whole system. Notable examples of such a break are the first two transient execution attacks, i.e., Spectre [148] and Meltdown [165], that were disclosed in early 2018. These attacks can break fundamental software security mechanisms such as memory isolation and leak important information of different kinds – from cryptography keys to saved passwords in the browsers.

Additionally, in order to be widely adopted, security solutions in most cases need to incur acceptably low performance costs. The security community has observed many security mechanisms that have been proposed, even implemented, but that are often disabled by users unwilling to bear the performance burden [3,41]. Although low performance cost is a requirement for many security solutions, it is particularly challenging for security defenses that mitigate attacks on important performance features (e.g., attacks on speculative execution or shared caches) as simply disabling those features will have a direct hit on performance.

At the microarchitecture level, this performance-security tension is becoming particularly apparent. Followed by the disclosure of the first transient execution attacks, we saw a host of new attacks appear [26, 146, 177, 230, 270], each exploiting a crucial performance optimization in modern processors, threatening to unwind decades of architectural gains. Unfortunately, these attacks coincide with an era in which those performance optimizations are needed most – an era when Moore's law is fading and Denard's scaling is gone. Therefore, the computer architecture and computer security communities have started to seek new architectures and techniques to continue to enable these optimizations but with higher levels of protection.

This dissertation is a step forward in this direction. More specifically, it addresses the question of *can we disentangle the inherent tension between security and performance*? This dissertation shows that, at the microarchitecture level, where the sources of these vulnerabilities reside, it is possible to ease this tension with secure microarchitectural techniques. This dissertation proposes techniques that allow us to flexibly reach desirable points in the security-performance space.

Particularly, this dissertation eases the tension in three complementary ways: First, it provides new insights and a deeper understanding of the ways that microarchitectural optimizations can be exploited in practical attacks. This not only leads to more secure and more efficient defenses but also allows us to provide timely mitigations. We expose the security side effects of a sophisticated, high performance microarchitectural technique – Intel Data Direct I/O (DDIO) implemented in most server-grade Intel processors to accelerate network packet processing. Second, it designs new secure architectures that thwart microarchitectural attacks without disabling our performance optimizations. We propose secure designs for two of the most critical performance optimizations in today's processors: speculative execution and simultaneous multithreading. Finally, in addition to high-performance mitigation for microarchitectural vulnerabilities, this dissertation utilizes microarchitectural techniques to improve the performance of other security mitigations (software-based or hardware-based). We propose a framework that utilizes the internal translation of instructions to micro-ops in the processor, offering on-demand security protection for various vulnerabilities.

1 Novel Microarchitectural Attacks

Attention to, concerns over, mitigations for, and research focused on microarchitectural security vulnerabilities have all increased significantly in recent years. It is clear that we face real dangers while these vulnerabilities remain. Of course, to defuse the security-performance tension and to create secure, vulnerability-free architectures, we first need to ferret out these vulnerabilities.

Modern processors employ increasingly complex microarchitectural techniques that are carefully optimized to deliver high performance. However, this complexity can sometimes breed security vulnerabilities. In this dissertation we develop an entirely new, powerful, and evasive microarchitectural attack dubbed *Packet Chasing*. Packet Chasing is an attack on the network that does not require access to the network, and works regardless of the privilege level of the process receiving the packets.

On a DDIO host, incoming network packets from a remote client contend for the shared last-level cache with application data structures from processes on the local host. We show that such contention provides significant leakage, allowing cache side channel attacks to perform covert communication and/or infer network behavior, even when given zero access to the network stack. This dissertation shows that the domain of impact of microarchitectural vulnerabilities is much larger than previously

3

understood. The impacts are not isolated to the microarchitecture, to the processor, or to the server. Microarchitectural vulnerabilities expose the entire reach of the network. With packet chasing, we present a test case for understanding the risk of introducing a new performance feature without a careful evaluation of the security implications. This dissertation also proposes and evaluates a hardware-based high-performance defense against this vulnerability, shifting our security-performance trade-off towards high-performance, secure systems.

2 Architecture Support for Security

Microarchitectural security techniques should not be limited to mitigating microarchitectural vulnerabilities. Sometimes, with a small investment in hardware/architecture we can provide huge improvement in performance of a software-based, or even another hardware-based, mitigation. Guided by our goal of easing the securityperformance tension, this dissertation designs schemes that provide architectural support for new security defenses.

Most modern processors employ translated ISAs. The Intel and AMD x86 processors feature translation from the native instruction set into internal micro-ops that enter the pipeline for execution. These architectures enjoy the dual benefits of a versatile backward-compatible CISC front-end and a simple cost-effective RISC back-end. However, for those architectures the translation has been always static, changing at most once per generation. To unlock the full potential of translated ISAs, this dissertation proposes *context-sensitive decoding* (*CSD*), a technique that allows native instructions to be decoded/translated into a different set of custom micro-ops based on their current execution context. This can be easily done because the processor performs the translation via table lookup. As a result, we can make that translation dynamic simply by incorportating a flag or set of flags into the lookup hash function.

This presents operating systems and runtime systems with the unique opportunity of triggering different custom translation modes, at microsecond or finer granularity, by simply configuring a set of model-specific registers. In this way, for example, an insecure executable can instantly become a secure executable, or performance-optimized code can become energy-optimized, without recompilation or binary translation. This feature is particularly useful for fast and efficient deployment of hardware security defenses in response to new attacks.

Via CSD, this dissertation introduces a simple, yet powerful change to modern architecture pipelines, exploiting the existing decoupling of the internal and external ISAs, which now enables the OS, hardware, or even users to add a suite of dynamic specialization capabilities including on-demand security, performance, energy-conserving, usability, and/or diagnostic features to existing code, without the need to wait for software changes or hardware changes.

3 Secure Speculative Execution

Spectre and related attacks have exposed new dimensions of the securityperformance tension by exploiting a performance optimization integral to modern high-performance architectures: speculative execution. These transient execution attacks bypass existing isolation mechanisms because they exploit instructions that execute only on speculative paths, not on the committed path. Mitigating most variants of Spectre requires highly intrusive changes to existing out-of-order processors, severely limiting performance. Although Intel has announced microcode updates to mitigate certain variants of the attack, a majority of the high impact vulnerabilities still largely rely on software patching. Software mitigations rely on *fence* instructions that mute specific effects of speculative execution by constraining the order of certain memory operations, or in some cases by completely serializing a portion of the dynamic instruction stream, but doing so severely hurts performance. We propose *Context-Sensitive Fencing* (*CSF*) [255, 257], a high-performance microcode-level defense against Spectre. CSF secures modern processors against Spectre attacks while maintaining high levels of performance, flexibility, and programmability.

Additionally, we comprehensively study and evaluate existing fences in the context of security applications. None of the fences in Intel processors were designed for security. Several were designed for synchronization, and thus with different priorities. Others are just unintentional side effects of various other instructions. In this dissertation, we find ample opportunity to significantly shift and reduce the aforementioned tension between performance and security – identifying and rearchitecting several key dimensions of these fences that restrict performance with no actual benefit for security.

CSF is also one of the first defenses to examine the rich body of work on Dynamic Information Flow Tracking (DIFT) in the context of speculative execution. A classic taint tracking mechanism operates in the backend of the pipeline–usually at the execute and commit stages. However, execute- or commit-based taint tracking comes too late in the pipeline for any speculation-based attack, and is of little use. We designed a novel architecture that makes DIFT work in the decode stage of the pipeline. With this, we further established the viability of speculative information flow tracking as an effective attack detection mechanism in the Spectre era.

4 Secure Multi-Threading Architectures

Simultaneous Multithreading (SMT) is a performance optimization that enables a processor core to issue instructions from multiple threads to the execution units in the same core in the same cycle [193, 265, 266]. The substantial benefits of SMT have led to its widespread adoption by virtually all the major players in the high-performance processor market, i.e., Intel, AMD, IBM, and ARM. In an SMT processor, virtually every part of the pipeline is potentially shared and contended for in some way. This creates a performance coupling between threads that is an enormous challenge for security. That's why, despite all the performance benefits, after the wave of speculative execution attacks, Google (in Chrome OS), OpenBSD, and Red Hat, among others, have suggested turning off SMT altogether. This dissertation addresses SMT's sharing-based security challenges while retaining its performance benefits.

Naturally, the first step is to understand to what extent modern SMT processors suffer from information leakage – i.e., to understand *how vulnerable they are*. Thus, we conduct the first comprehensive and exhaustive analysis of resource contention across the entire pipeline for recent offerings from both Intel and AMD. We design a covert-channel discovery framework that deconstructs how the processor manages resource sharing between the SMT threads and measures the potential information leakage resulting from sharing of each of these pipeline resources.

In this dissertation, then, we design and develop secure approaches to multithreading. In particular, we design two novel partitioning approaches that can be applied to all contended resources with slight variation: *Adaptive Partitioning* provides a temporary, hard partition between threads for a particular resource, but that partition can move at regular intervals to adapt to long-term program behavior. *Asymmetric SMT* enables the system to prevent leakage to an untrusted thread, but without sacrificing the performance of the trusted thread. Just one example where this is useful is sandboxing in web browsers. While it is not secure to leak information from the browser thread to the sandbox thread, it is safe to leak information from the sandbox to the browser. Asymmetric SMT recovers the lost resource utilization due to partitioning and uses that to accelerate the execution of the trusted thread. This dissertation addresses contention-based side channels in all pipeline structures, enabling continued use of these performance-critical structures while executing securely. We show that SMT contention-based vulnerabilities can be reduced below the level of other known vulnerabilities, making SMT execution a viable alternative for high-performance secure execution.

5 Outline

Chapter 1 presents a novel microarchitectural side-channel attack that can leak important information from receiving network packets without having access to the network. This chapter also presents novel mitigations for the discovered vulnerability.

Chapter 2 introduces a new architectural framework that enables customization of the micro-op translation based on the current execution context. While there are many potential applications, this chapter demonstrates two use cases: a novel highperformance security defense to thwart instruction/data cache-based side-channel attacks; and a power management technique that performs selective devectorization to enable efficient unit-level power gating.

Chapter 3 presents a high-performance mitigation against multiple Spectre variants. This chapter describes multiple architectural techniques including speculative information flow tracking and new security-oriented fence prmitives, that collectively reduce performance overhead of software-based mitigations by a factor of 6.

Chapter 4 describes our novel secure and high-performance multithreading architectures.

Finally, Chapter 5 concludes the dissertation and discusses possible future directions.

Chapter 1 Packet Chasing

Modern processors employ increasingly complex microarchitectural techniques that are carefully optimized to deliver high performance. However, this complexity often breeds security vulnerabilities, as evidenced recently by Meltdown [165] and Spectre [148]. This chapter explores the vulnerable side effects of another sophisticated high performance microarchitectural technique – *Intel® Data Direct I/O* (DDIO) [127] implemented in most server-grade Intel processors to accelerate network packet processing. Further, it presents new high resolution covert and side channel attacks on the network I/O traffic, which while possible without DDIO, are considerably more effective in the presence of DDIO.

The widespread adoption of multi-gigabit Ethernet and other high-speed network I/O technology such as Infiniband has highlighted the critical importance of processing network packets at high speed in order to sustain this newly available network throughput, and further improve the performance of bandwidth-intensive datacenter workloads. Consequently, most Intel server-class processors today employ DDIO technology that allows the injection and subsequent processing of network packets directly in the processor's last level cache (LLC), bypassing the traditional DMA (Direct Memory Access) interface. DDIO is invisible to software, including OS drivers, and is always enabled by default. The key motivation behind DDIO is the fact that modern server-class processors employ large LLCs (~20MB in size), thereby allowing the network stack to host hot data structures and network packets in-process completely within the LLC, reducing trips to main memory. By eliminating redundant memory transfers, DDIO has been shown to provide substantial improvements in I/O bandwidth and overall power consumption [22,127,163,180]. Although Intel restricts allocating more than 10% of the LLC for DDIO to prevent cache pollution, it neither statically reserves nor dynamically partitions a dedicated portion of the cache for DDIO.

However, despite its good intention to accelerate network packet processing, DDIO has a previously unknown vulnerable side effect that this chapter exposes. On a DDIO host, incoming network packets from a remote client and application data structures from processes on the local host contend for the shared LLC, potentially evicting each other in the event of a cache conflict. In this chapter, we show that such contention provides significant leakage, allowing cache side channel attacks to perform covert communication and/or infer network behavior, with virtually zero access to the network stack. In particular, we describe a new class of covert- and side-channel cache attacks, called *packet chasing*, that exploit this contention by creating arbitrary conflicts in the LLC using carefully constructed memory access patterns and/or network packet streams.

We further show that the location (in cache) of packet buffers used by the network driver, *and the order in which they are filled*, are easily discovered by an attacker, greatly minimizing the amount of probing necessary to follow the sequence of packets being chased.

The *packet chasing*-based covert channel we describe in this chapter allows a spy process running covertly alongside a server daemon on the local DDIO host to receive secret messages from a trojan process running on a remote client across the network, by causing deterministic contention in the last-level cache. We show that such

a covert means of communication is feasible, and is achievable at a high bandwidth, despite the fact that the trojan process only sends broadcast packets and that the spy process is completely isolated from the network-facing server daemon (potentially cross-container and cross-VM), and further lacks any access to the network stack.

In addition to the covert channel, we describe a novel *packet chasing* based side-channel attack that leverages a local spy process running alongside (or, within) a web browser. In our experiments, the spy is on the client side alongside of a browser like Mozilla Firefox, enabling it to fingerprint a remote victim's website accesses without having access to the network. In particular, this attack enables an attacker to recognize the web activity of the victim based on packet size patterns. This type of web fingerprinting could be used by an oppressive government to identify accesses to a banned site, or an attacker could identify members of a secure organization (to then target more directed attacks) simply by fingerprinting a successful login session.

Further, this chapter describes a software-based short-term mitigation, called ring buffer randomization, as well as a hardware defense mechanism that adaptively partitions the LLC into I/O and CPU partitions, preventing I/O packets from evicting CPU/adversary cache blocks. The adaptive partitioning defense that we describe in this chapter has a performance overhead of less than 2.7% compared to the vulnerable DDIO baseline.

1.1 Background and Related Work

This section provides background on network packet handling, DDIO and related network optimizations, network and I/O based attacks, and cache attacks.

1.1.1 Journey of a Network Packet

When an application sends data through the network, it usually sends a stream of data; and it is the responsibility of the transfer layer to break large messages into



Figure 1.1. The shared ring buffers (FIFO) between NIC and the device driver.

smaller pieces that the network layer can handle. The Maximum Transferable Unit (MTU) is the largest contiguous block of data which can be sent across a transmission medium. For example, the Ethernet MTU is 1500 bytes, which means the largest IP packet (or some other payload) an Ethernet frame can carry is 1500 bytes. Adding 26 bytes for the Ethernet header results in a maximum frame of 1526 bytes.

When the NIC driver initializes, it first allocates a buffer for receiving packets and then creates a descriptor which includes the receive buffer size and its physical memory address. It then adds the receive descriptor to the receive ring (rx ring), a circular buffer shared between the driver and the NIC to store the incoming packets until they can be processed by the driver. The driver then notifies the NIC that it placed a new descriptor in the rx ring. The NIC reads the content of the new descriptor and copies the size and the physical address of the buffer into its internal memory. At this step, the initialization is done and the NIC is ready to receive packets.

As shown in Figure 1.1, upon receiving incoming packets, the NIC, using Direct Memory Access (DMA), copies packets to the physical addresses provided in the *rx* ring, and then sends an interrupt to notify the driver. The driver drains the new packets from the *rx* ring and places each of them in a kernel data structure called a socket buffer (*skb*) to begin their journey through the kernel networking stack up to the application which owns the relevant socket. Finally, the driver puts the receive buffer back in the *rx* ring to be used for future packets.

1.1.2 Direct Cache Access and Data Direct I/O

Modern processors and operating systems employ a number of network I/O performance enhancements that address packet processing bottlenecks in the memory subsystem [116, 127]. Huggahalli, et al. [116], present Direct Cache Access (DCA), which enables the NIC to provide prefetch hints to the processor's hardware prefetcher. DCA requires that memory writes go to the host memory and then the processor prefetches the cache lines specified by the memory write. The Intel Sandy-Bridge-EP microarchitecture introduced the Data Direct I/O (DDIO) technology [127] which transparently pushes the data from the NIC or other I/O devices directly into the last level cache. Before DDIO, I/O data was always sent through the main memory; inbound data is written by the I/O device into memory, and then the data is either prefetched before access or demand fetched into the cache upon access by the processor. With DDIO, however, DMA transactions for an I/O region go directly to the last level cache, and they will be in dirty mode and will get written back to memory only upon eviction [81, 137].

While DCA and DDIO have been shown to improve packet processing speeds by reducing the cache miss rates in many scenarios [116, 127], if the device has large descriptor rings, they could potentially degrade performance by evicting useful data out of the LLC [251]. In addition, as we show in this chapter, these technologies potentially open up new vulnerabilities since the packets are brought directly into the LLC, which is shared by all cores in the processor.

1.1.3 Network-Based Covert-Channels

The literature abounds with network-based covert channels that leverage network protocols as carriers by encoding the data into a protocol feature [2,89,107,174, 232,310]. For example, covert channels can be constructed by encoding data in unused or reserved bits of the frame or packet headers [104, 107, 151, 174], such as the TCP Urgent Pointer which is used to indicate high priority data [107]. In the TCP protocol, Initial Sequence Number (ISN) is the first sequence number and is selected arbitrarily by the client. Rowland [226] proposed shifting each covert byte by 26 bits to the left and directly using it as the TCP ISN. Abad [2] shows that the IP header checksum field can also be exploited for covert communication, and further proposes encoding the secret information into the checksum field and adding the content of an IP header extension to compensate the checksum modification, chosen such that the modified checksum will be correct. Other header fields such as address fields [89] and packet length [174] are also exploited to build covert channels. In addition to the header field, packet rate and timing [89, 151], packet loss rate [232], and packet sorting [37] are also used to build covert channels.

Many of these covert channels are based on non-standard or abnormal behaviour of the protocol and can be detected and prevented by simple anomaly detection methods [310]. In addition, all of these network-based covert channels require the receiver to have access to the network and be able to receive packets, while the receiver in our packet chasing attack does not need any access or permission to the network.

1.1.4 Cache Attacks

Cache-Based side-channel attacks are the most common class of architectural timing channel attacks, that leverage the cache as their sole medium of covert communication [23, 167, 207, 297]. These attacks have the potential to reveal sensitive information such as cryptographic keys [61,70,98,167,191,304], keystrokes [99,282], and web browsing data [205,237], by exploiting timing variations that arise as a result of a victim process and a spy process contending for a shared cache. For example, in the PRIME+PROBE [167,205] attack, the spy process infers the secret by learning the temporal secret-dependent cache access patterns of a victim, by contending for the same cache sets as the victim and measuring the timing variations that arise due to



Figure 1.2. Intel's complex indexing of modern last level cache.

such contention. In the PRIME step, the attacker fills one or more cache sets with its own cache blocks, simply by accessing its data. Then, in the IDLE step, the attacker waits for a time interval and lets the victim execute and use the cache, possibly evicting the attacker's blocks. Finally, in the PROBE step, the attacker measures the time it takes to load each set of cache blocks. If it is noticeably slow, she can infer that the victim has accessed a block in that set, replacing the attacker's block.

To perform these attacks in a fine time granularity, the attacker has to target specific sets in the last level cache. As such, she has to know how the addresses map into the sets in the LLC. However, starting with the Sandy Bridge microarchitecture [124], Intel has employed a new LLC design, in which the LLC is split into multiple slices, one for each core (See figure 1.2), with an unpublished hash function mapping physical addresses to slices, supposedly distributing the physical addresses uniformly among the cores. This hash function has since been successfully reverse-engineered for many different processors, including Intels Sandy Bridge [140, 183, 305], Ivy Bridge [120, 183], and Haswell [130, 183] architectures.

In addition to PRIME+PROBE, multiple other variants of cache attacks are also proposed [70, 98, 304]. FLUSH+RELOAD [304] uses Intels CLFLUSH instruction to flush a target address out of the cache, and then, at the measurement phase, the attacker reloads the target address and measure its access time. However, it relies on shared memory between the spy and the victim, and requires access to precise timers. PRIME+ABORT [70] exploits Intel's transactional memory extension (TSX) hardware to mount a timer-free last level cache attack.

Several defenses have been proposed in the literature to mitigate cache timing channels [139, 145, 164, 166, 179, 201, 202, 217, 229, 253–255, 283, 284, 287, 312]. These mitigation strategies include identifying the leakage in software [283], observing anamalous cache behavior [252, 312], closing channels at hardware design time [139, 164, 179, 201, 284], dynamic cache partitioning [145, 284, 287], strictly reserving physical cores to security-sensitive threads [202], randomization [287], memory trace obliviousness [166, 217], and cache state obfuscation using decoy load micro-ops [253, 254].

1.1.5 Security of I/O Devices and Drivers

A number of security attacks have been published that target device drivers [92, 181,314]. Thunderclap [181] describes an attack that subverts the Input-Output Memory Management Unit (IOMMU) protections to expose the shared memory available to DMA-enabled I/O peripherals. Zhu, et al. [315] demonstrate another attack that bypasses IOMMU and compromises the GPU driver to exploit GPU microcode to gain full access to CPU physical memory. To address these vulnerabilities, researchers focus on isolating device drivers, and to make operating systems secure when a device driver is buggy or has code which is intentionally malicious [32, 250]. Tiwari, et al. [259] propose a full system which includes an I/O subsystem and a micro-kernel that enable isolation and secure communication by monitoring and controlling the information flow of the system.

NetCat [152] is a concurrent work to our Packet Chasing attack. It describes an attack that exploits a similar underlying vulnerability. However, this work differs in many important ways. First, NetCat only detects the arrival time of packets, whereas Packet Chasing has the ability to detect both arrival time and size of each packet – the latter is more reliable and less noisy. This gives Packet Chasing-based attacks the opportunity to mount more powerful attacks such as the web fingerprinting attack

that we describe in this chapter (Section 1.4). Second, unlike Packet Chasing, NetCat requires DDIO and RDMA technologies to be present, limiting its generality. Therefore, to mitigate NetCat, it is sufficient to disable DDIO or RDMA. However, as we show in this chapter, the Packet Chasing attack is practical even in the absence of those technologies. Therefore, we also present a more sophisticated yet high-performance defense that mitigates the attacks.

1.2 Packet Chasing: Setting up the Attack

We perform our analysis and attack on Intel's Gigabit Ethernet (IGB) driver version 5.3.5.22 [129] loaded into Linux Kernel version 4.4.0-142. We run the attack on a Dell PowerEdge T620 [65] server which uses Intel I350 network adapter [128] and is operated by two Intel Xeon CPU E5-2660 processors. Each processor has a 20 MB last level cache with 16384 sets. To perform PRIME+PROBE on the last level cache, we use the Mastik Micro-Architectural Side-Channel Toolkit Version 0.02 [303]

Our attack consists of two phases. One is an offline phase where the attacker recovers the sequence of the buffers and an online phase where the attacker uses that information to monitor the incoming packets.

1.2.1 Deconstruction of the NIC Driver

While the code samples of this subsection are specific to Intel's Gigabit Ethernet (IGB) driver, we note that the insights are generalizable. The original Ethernet IEEE 802.3 standard defines the minimum Ethernet frame size as 64 bytes and the maximum as 1518 bytes, with the maximum being later increased to 1522 bytes to allow for VLAN tagging. Since the driver and the NIC don't know the size of incoming packets beforehand, the NIC has to allocate a buffer that can accommodate any size. The IGB driver allocates a 2048 byte buffer for each frame and packs up to two buffers into one 4096 byte page which will be synchronized with the network adapter. For compatibility,

it is recommended [30] that when the device drivers map a memory region for DMA, they only map memory regions that begin and end on page boundaries, which are guaranteed also to be cache line boundaries. Further, the *rx* ring buffer is used to temporarily hold packets while the host is processing them. While employing more buffers in the ring could reduce the packet drop rate, it could also increase the host memory usage and the cache footprint. Therefore, although the maximum size supported by Intel's I350 adapter is 4096 buffers, the default value in the IGB driver is set to 256.

The linux kernel, in the DMA API, provides two different types of DMA memory allocation for device drivers. Coherent (or consistent) memory and streaming DMA mappings. Coherent memory is a type of DMA memory mapping for which a write by either the device or the processor can be visible and read by the processor or device without the need to explicitly synchronize and having to worry about caching effects. However, the processor has to flush the write buffers before notifying devices to read that memory [30]. Therefore, consistent memory can be expensive on some platforms as it invariably entails a wait due to write barriers and flushing of buffers [30]. While the buffers themselves are mapped using streaming DMA mapping, the ring descriptors are mapped using coherent memory. Thus, the device and the driver have the same view of the ring descriptors. Also, this makes the writes to the *rx* descriptors, drivers after receiving packets usually reuse the buffers instead of allocating new buffers. So the drivers.

Figure 1.3 shows the part of the IGB driver code that is called upon receiving packets and whose job is to add the contents of the *rx* buffer to the socket buffer which will be passed to the IP layer. If the size of the packet is less than a predefined threshold (256 by default), then the driver copies the contents of the buffer and then tries to
```
static bool igb_add_rx_frag(rx_buffer, skb){
    . . .
    size = rx_buffer->size;
    page = rx_buffer->page;
    if (likely(size <= IGB_RX_HDR_LEN)) {</pre>
        memcpy(__skb_put(skb, size), page, size);
        /* we can reuse buffer as-is,
        just make sure it is local */
        if (likely(page_to_nid(page) == numa_node_id()))
            return true;
        /* this is a remote page and cannot be reused*/
        put_page(page);
        return false;
    }
    /* only if packet is large */
    skb_add_rx_frag(skb, page);
    return igb_can_reuse_rx_page(rx_buffer, page);
}
```

Figure 1.3. The IGB driver function that adds the contents of an incoming buffer to a socket_buffer which will be passed to the higher levels of networking stack. The function returns true if the buffer can be reused by the NIC.

```
bool igb_can_reuse_rx_page(rx_buffer, page){
    /* avoid re-using remote pages */
    if (unlikely(page_to_nid(page) != numa_node_id()))
        return false;
    /* if we are only owner of page we can reuse it */
    if (unlikely(page_count(page) != 1))
        return false;
    /* flip page offset to other buffer */
    rx_buffer->page_offset ^= IGB_RX_BUFSZ;
    /* bump page refcount before it's given to stack */
    get_page(page);
    return true;
}
```

Figure 1.4. The IGB driver function that checks if the driver can reuse a page and put it back into the rx ring buffer.

recycle the same buffer for future packets. If the buffer is allocated on a remote NUMA node, then the access time to that buffer is much more than if the buffer was allocated in a local NUMA node. Therefore, to improve performance, the driver deallocates the remote buffer and re-allocates a new buffer for that *rx* ring descriptor. If the packet size is larger than 256, then instead of the direct copy, the IGB driver attaches the page as a fragment to the socket buffer. It then calls the *igb_can_reuse_page* function shown in Figure 1.4. This function checks for two conditions that are unlikely to be met. The first condition is that the buffer is allocated on a remote NUMA node. The second condition is that the kernel is still preparing the packet in the other half and that the driver is not the sole owner of the page. If neither condition is met, the driver flips the *page_offset* field, so that the device only uses the second half of the page.

To summarize, in the common scenarios, the driver uses a small number of ring buffers (256) on 256 distinct pages, each of them half-page aligned and it continually reuses these buffers typically until the next system reboot or networking restart. In addition, to maintain high (and consistent) packet processing speeds, the order of the ring descriptors does not change throughout the execution of the driver code. Therefore, as long as the driver reuses the buffers for descriptors, the order of the buffers remains constant.

1.2.2 Recovering the Cache Footprint of the Ring Buffer

The ultimate goal of the Packet Chasing attacker is to gain size and temporal information about incoming packets by spying on the last level cache. To this end, we mount a PRIME+PROBE attack on the last-level cache. However, blindly probing all cache sets doesn't give us much information. This is because the probe time is limited by the time it takes to access the entire cache, which in this case is about 12 million CPU cycles, too long to gain any useful information about incoming packets. Long probe time also makes the attack more susceptible to background noise, as the



Figure 1.5. An example of how the NIC ring buffers are mapped to to the page-aligned cache sets.

probability of observing irrelevant activity on the cache line increases.

However, from the previous subsection, we know that the buffers that store packets in kernel memory are page-aligned. That means we only need to probe the sets that the page-aligned addresses are mapped to. Having 4KB page size implies that the lowest 12 bits of the starting addresses are zero. So the lowest 6 bits of the set indices are zero (also see Figure 1.2). That limits us to 32 sets in each slice for a total of 256 possible sets. Using the Mastik toolkit, we find these sets and construct eviction sets for them, which are essentially a stream of addresses guaranteed to replace all other data from all the cache blocks in a set. With these, we have the ability to monitor all 256 cache sets that are potential candidates for buffer locations.

While all the NIC *rx* buffers map to one of the page-aligned cache sets that we obtain, the distribution of this mapping is not uniform, which means that some of the *rx* buffers are mapped to the same cache set. To show an example of such conflict in the cache sets, we instrument the driver code to print the physical addresses of the ring buffers, which we then map to cache set indices. Figure 1.5 shows this non-uniform mapping for just one instance of the buffer allocation in the NIC. Horizontal axes shows one of the page-aligned cache sets and on the Y axis, we show the number of NIC buffers that map to each page-aligned cache set. In this example, we see that 5 NIC buffers are mapped to cache set number 165 while none of the NIC buffers are mapped to cache set.

Figure 1.6 further analyzes this mapping which shows the result of performing



Figure 1.6. Frequency of the ring buffers that map to same sets, measured for 1000 instances. Zero represents the number of sets that are page aligned but none of the ring buffers is mapped to those.



Figure 1.7. Monitoring all the page-aligned sets while receiving packets. A white dot shows at least one miss (activity) on a cache set in a sample interval

the same experiment across multiple instances of driver initialization. For around 35% of the page-aligned sets, there is no co-mapped NIC buffer, while there are only 5 out of 1000 instances in which we see more than 4 buffers mapped to the same page-aligned cache set.

By narrowing down the number of monitored cache sets to only the 256 possible buffer starting locations, we are able to see a clear footprint in the cache when the NIC device is receiving packets, as shown in Figure 1.7. In this experiment, we rely on a remote sender who is on the same network with the spy and constantly sends broadcast Ethernet frames to the network. To this end, we use Linux raw socket [141] which generates broadcast Ethernet frames with arbitrary sizes. These frames get discarded in the driver since the protocol field is unknown. Thus, the effect that we



Figure 1.8. Cache footprint of packets with different sizes while probing the addresses that map to the location of the first three blocks in the packet buffer page. A white dot indicates at least a miss in a set.

see is only caused by the driver/adaptor accessing the buffers, without any activity of the kernel networking stack. At around sample 25k, the sender starts sending packets and it continues to do so until sample 100k. In some cache sets, e.g., cache set number 53, we don't see any activity and that is because none of the NIC buffers are mapped to those sets.

The packet chasing attacker, with the ability to distinguish between an idle system vs. when there are incoming packets, establishes a leaking channel that can be exploited to covertly communicate secret data over the network. We can further increase the bandwidth of this channel by differentiating the receiving streams based on frame sizes. Since the incoming packets are stored in contiguous *rx* buffers, using the same way that we construct the eviction sets for the page-aligned cache sets, we construct eviction sets for the second cache blocks in the page. All the second cache blocks in the pages are mapped to one of these 256 cache sets. Similarly we find the sets for the third and fourth cache blocks of the pages. This now allows us to recognize not just the presence of a packet, but also the size of the packet.

Figure 1.8 shows the result of a simple experiment where we send packets of different sizes and test our ability to detect packet size. On the columns, we have four different runs with constant packet sizes being sent, from one cache block (64 bytes)

to four cache blocks (256 bytes). On the rows we show detection on four different cache eviction sets, block o to block 3 which are targeting the first to fourth blocks in the page-aligned buffers. As expected, we see clear activity on the diagonal and above, and no activity below the diagonal. The only exception is 1-block packets which exhibit activity on block 1 as well as on block o. This is because there is a performance optimization in the driver code that prefetches the second block of the packet regardless of the packet sizes. The reason for this optimization is that most Ethernet packets have at least two blocks, and 64-byte packets (o-Block Packet) are only common in control packets that don't have payloads such as TCP acknowledge packets.

The attack distinguishes a stream of packets with different sizes from each other, and that could be used to construct a remote covert channel (more details in Sec 1.3) with 1950 bytes-per-second bandwidth by only detecting a stream of small packets vs. a stream of large packets (essentially, a binary signal). However, we can turn this to a more powerful channel if we differentiate sizes with finer granularity, essentially sending multiple bits of information per packet. The following subsection describes the method that we use to further narrow down the monitored sets while we perform PRIME+PROBE.

1.2.3 Chasing Packets over the Cache

The attacker has to probe all 256 page-aligned sets at once to detect incoming packets only because she doesn't know which buffers get filled first, and then probe more sets to detect packet size. However, if we know the order in which the buffers get filled in the driver, then we can actually chase the packets over the cache by only probing the cache sets corresponding to the next expected buffer, building a powerful high-resolution attack. We show that it is possible to almost fully recover the sequence of the buffers, in a one-time statistical analysis phase. Since the buffers are always



Figure 1.9. Pruning and sequencing of the set graph to get the order of ring buffers. Each node represents a set in the attacker address space. Numbers in squares are the sequence number of the associated ring buffers that map to same set.

recycled and then returned to the ring, the order of the buffers in the ring is maintained during the lifetime of the driver.

Algorithm 1 describes the SEQUENCER procedure that we use to recover the sequence. It consists of three steps. First, in the GET_CLEAN_SAMPLES step, we gather cache probe samples for *Nsets* cache sets. To this end, we start with constructing the eviction sets for the page-aligned NIC buffers. However, sometimes we have always-miss scenarios on some sets, which is easily observed a priori. For those sets, we simply use the second cache block of a page-aligned buffer instead of the first one.

After that, we start building a complete weighted graph with the nodes being the monitored cache sets and the weights on the edge that connect node x to node yare the number of times that we observe an activity on set y which was immediately followed by an activity on set x, as illustrated in the leftmost graph in Figure 1.9. To deal with the problem that multiple buffers can map to the same cache set, when we build the graph, we maintain one node history for each edge. This allows the algorithm to distinguish between the activity on two or more different buffers that map to the same cache set by their successor cache sets. So, for example in Figure 1.9, two different buffers are mapped to cache set number 2. These buffers occupy location numbers 93 and 193 in the ring buffer. Therefore, in the final sequence, we have two different instances of cache set number 2, one that is followed by cache set 3 and the other that is followed by cache set 1.

The final step, MAKE_SEQUENCE, is to traverse the graph we build in the previous

steps, starting from a random node, and continuing to move forward until we reach the same node. Note that since the final sequence is a *ring* in which the *in-degree* and the *out-degree* of each node is exactly one, the choice of the starting node doesn't change the outcome.

While this procedure can recover the sequence of the buffers that are mapped into *Nset*, it can only do so if we monitor a limited portion of the page-aligned cache sets (we were successful up to 64 cache sets). This is because the probe time gets longer than what is required to detect the order of the incoming packets, if we include more sets in our monitor list. So we first find the sequence for 32 cache sets, then we repeat the SEQUENCER procedure with the first 31 nodes (node o to node 30) plus a candidate node (e.g, 32) and we try to find the location of the candidate in the sequence. Then, we repeat the same procedure, moving through the node sequence, until we find a place in the sequence for all cache sets.

Sometimes two consecutive buffers are mapped into one set. For example, consider the case that buffers number 93 and 98 are mapped into the set 2 in Figure 1.9. With our approach, we don't capture these cases in the first round, but starting from the beginning, when we do encounter a buffer that is between the two, we can split the two in our graph. If they are truly consecutive in the final ring (unlikely), the buffers are essentially merged, but this has no impact on our mechanism to create a covert channel, and will have minimal effect on the overall fingerprint we observe in the web fingerprinting leakage attack.

We measure Levenshtein distance [135] to quantify the distance between the sequence that we obtain and the ground truth actual sequence that we get from driver instrumentation. The Levenshtein distance between two sequences is the minimum number of single-character edits (i.e., insertions, deletions or substitutions) needed to change one sequence into another. We see the results of this experiment in Table 1.1. ine-tuning the probe rate is a rather challenging task as it needs to be long enough

Results				
Measure	Value	CI		
Levenshtein Distance	25.2	[22, 35]		
Error Rate (%)	9.8	[8.5, 13.6]		
Longest Mismatch	5.2	[3, 9]		
Time (Minutes)	159	[153, 167]		
Parameters				
Parameter	Value			
Number of Samples	100,000			
Number of Monitored Set	32			
Packet Rate (packet/s)	0.2M			
Probe Rate (probe/s)	8000			

Table 1.1. Summary of experiments for sequence recovery

that the activity of each incoming packet touches only one sample, and needs to be small enough to not lose the temporal relation between the incoming consecutive packets. Otherwise, we see a drop in accuracy of the obtained sequence. However, in our covert-channel construction, in many of our attack scenarios, we only need to find buffers that are sufficiently far apart in the ring, so small errors in the sequence are tolerable.

During the profiling period we rely on a remote sender whose only job is to constantly send packets. However, the spy can recover the sequence even without the help of the external sender, as long as the system is receiving packets. In fact, noise (extra packets not sent by co-operating sender) in this step only helps the spy.

1.3 Packet Chasing: Receiving Packets without Network Access

In this section, we show the effectiveness of the Packet Chasing attack by constructing a covert channel over the network. We assume a simple threat model where a remote trojan attempts to send covert messages through the network, to a spy process located in the same physical network. The spy process can be inside a container and does not have root privileges, neither in the container nor in the host OS, and is also not permitted to use the networking stack. The trojan process has the ability to send packets to the physical network, but there is no authorized method to communicate with the spy.

Channel Capacity

To build a framework for quantitatively comparing different encoding and synchronization schemes, we follow the methodology described by Liu, et al. [167] to measure our channel bandwidth and error rate, while transferring a long pseudo-random bit sequence with a period of $2^{15} - 1$. The pseudo-random bit sequence is generated using a 15-bit wide linear feedback shift register (LFSR) that covers all the 2^{15} sequences, except the case that all bits are zeros. This allows us to spot various errors that might happen during the transmission including bit loss, multiple insertion of bits, or bit swaps [167].

Data Encoding and Synchronization

The spy first chooses x, one of the page-aligned cache sets that only one of the ring buffers is mapped to. Finding such a page-aligned cache set is not challenging using the approaches described in Section 1.2. Then the spy process finds the cache sets to which the addresses x + 64, x + 2 * 64, and x + 3 * 64 are mapped. In other words, it finds the cache sets for the second, third, and fourth cache blocks of the page-aligned buffer. As described in Section 1.2, the spy process knows the set index bits for these sets, but the outcome of the hash function (slice bits) is not known. To find out the exact slice, the spy process executes a trial and error procedure in which it selects one of the eight slices based on the activity on the sets. After this step, the initialization is done and the spy process constantly monitors the found cache lines.

The spy process, at time frame *n*, sends 256 (the length of the ring buffer) packets of size (S + 2) * 64 to transmit the symbol S. Since we operate on the network and the latency is fluctuating frequently, we cannot use return-to-zero self-clocking encoding [167], rather we choose to use a synchronized clock encoding scheme in

which the first block of the buffer acts as a clock to synchronize the spy with the trojan. We measure the bandwidth and the error rate for two cases. First, we encode one binary symbol in each packet, i.e., we send either 64-byte packets that encode "o", or we send 256-byte packets that encode "1". Second, we send a ternary symbol in each packet, i.e., we send 64-byte packets to encode "o", 192-byte packets to encode "1", and 256-byte packets to encode "2".

For example, Figure 1.10 shows a part of a sequence that the spy receives in a real experiment. In this experiment, the trojan transmits sequence "2012012012..." and the spy collects one sample from the three cache sets, every 200,000 cycles. When decoding, the spy uses a window of three samples to distinguish between different values. This is because sometimes we see the cache activity of one packet (one symbol) that spans across two cycles (the wide peaks in the figure). The spy process should not decode these cases into two different symbols. In addition, having a window helps if the activity on the sets get skewed because of the delay of arriving packets. The first set of the buffer is used as a clock to synchronize the parties, and activity on the other two sets can show the transmitted values. Monitoring the activity of the two sets only gives us three different symbols because by sending a 3-block packet, we have a compulsory activity on set 2.

To estimate the error rate, we again use edit Levenshtein distance [135] between the sent and received data for the pseudo random bit sequence. Figure 1.11 shows the bandwidth and the error rate of our coding schemes, as well as the effect of varying the probe rate, i.e., the time we wait between consecutive probes. The bandwidth of the channel is almost constant with different probe rates. This is because the limitation here is the line rate. We are using 1 Gb/s Ethernet link and transmitting a collection of packets whose average size is 192 bytes. The maximum frame rate for the packets with frame size of 192 is around 500,000 frames per second [204]. Since we are sending one symbol per 256 packets, our maximum bandwidth is theoretically bounded at 1953



Figure 1.10. Spy process decodes the transmitted sequence based on the monitored activity on the probed sets. Set 1 acts as the clock and the activity is one for a set if we find at least one miss in the blocks in the eviction set of the probed set.



Figure 1.11. Bandwidth and error rate of the remote covert channel for binary and ternary encoding and various cache probe rates.

symbols per second. By coding three symbols, this packet chasing covert channel can reach a bandwidth of 3095 bps. The error rate, however, is reduced as we reduce the probe time. That is because with a longer wait time between two consecutive probes, we raise the probability of capturing irrelevant background activity on the sets. When we use binary encoding, we use the samples from both *set 2* and *set 3* and if they both have activity during a window, we decode as "1". Therefore, the error rate is slightly less than the ternary encoding.

Exploiting Ring Buffer Sequence Information

If we know the ordering of the buffers, this mechanism is easily extended to send more than one symbol per 256 packets. In this case, the trojan can send one covert message every 256/n packets by dividing the ring buffer into n sections of similar sizes by selecting n buffers that are ideally 256/n apart. The selected buffers should be among the buffers that are mapped to only one of the page-aligned cache sets. Then the spy starts monitoring the selected sets and their adjacent blocks to detect the size of the packets that are filling these buffers.

This process can multiply the capacity of the covert channel as shown in Figures 1.12a and 1.12b. These figures show the bandwidth and the error rate for the cases that the spy monitors a different number of buffers in the ring. For each of these buffers the spy probes three cache sets that are associated with the first, third, and fourth cache blocks of the packets that fill these buffers. For the case that there is only one monitored buffer, the trojan sends one covert message with 256 packets, and for the case of 16, the trojan sends a new message every 256/16 = 16 packets. The bandwidth of the channel almost doubles when we double the number of monitored buffers and it goes up to 24.5 kbps for the case of 16 monitored buffers. The error rate remains almost constant until the time between incoming packets gets close to the time between two consecutive probes. Note that when we have more sets in our monitored list, each probe takes more time and this decreases the probe rate. Furthermore, with the increased number of monitored buffers, it becomes harder to find the buffers that are *n* buffers apart in the ring and also do not share the cache set with any other buffer in the ring. For these reasons, we see a jump in the error rate when we monitor 16 buffers of the ring. Note that these and subsequent results also account for inaccuracies incurred when we deconstruct the ring sequence.

Figures 1.12c and 1.12d show the result of another experiment in which we actually chase the packets using the sequence. We probe one buffer at a time and as

soon as we detect an activity on the probed buffer, we move to the next buffer in the sequence. The *out-of-sync* rate is the rate by which packet chasing misses one packet, and therefore it has to wait until completion of the whole ring, or the next time a packet fills that buffer, to get synchronized again. The bandwidth is controlled by the rate at which the sender transmits the packets and the error rate is calculated on the synchronized regions of the transmission. The figure shows that the out-of-sync rate is almost constant for different packet rates. This is because when we probe just one set, the resolution of probing is higher than the time between consecutive packets. In addition, the frequency at which we get out-of-sync is a function of the quality of the sequence that we obtain. The error rate jumps at 640 kbps because at that speed the packets start to arrive out-of-order at the receive side.

Detectability and Role of DDIO/DCA

In the presence of DDIO, the packets that carry the covert messages are hard to detect and filter (e.g., by a firewall system that drops the packets that are sent to the victim node) as they can be regular broadcast packets, e.g., DHCP and ARP, and they are not even required to be destined for the machine that hosts the spy. This is because, with DDIO/DCA, the network adapter directly transfers the packets into the last level cache of the processor, and only after this will the driver examine the header of each frame and discard the packets that do not target any protocol that is hosted in that machine. That is, with DDIO enabled, we can establish a channel between machine A and B even with A only sending packets to machine C on the same network.

DDIO enables Packet Chasing to get a clearer signal as the cache blocks of the payload appear in the cache as the same time as the cache blocks that belong to the header of the packet. This enables the attack to probe the adjacent cache blocks and quickly detect the size of each packet. However, if DDIO is disabled or not present, the journey of a packet would be different. First, the NIC stores the header of the packet in



Figure 1.12. (a) and (b) show the channel capacity for the remote covert channel where the spy that uses n buffers of the ring's sequence information. (c) and (d) show the out of sync rate and error rate for the case where spy uses all the buffers in the sequence information.

the memory, then the driver reads the header and processes the packet according to the header fields. This brings the cache blocks containing the header into the cache. For most of the common higher level protocols (e.g., http) the software stack will access other parts of the packet shortly after the header comes [116].

The latency between I/O writes and driver reads now becomes a factor in the attack without DDIO. In this scenario, the attacker should set the probe time to be larger than that latency. When that latency is accounted for, the cache footprint of the packet remains the same as the DDIO case. However, increasing the probe interval can result in more noise captured in each interval. But the latency, as characterized in [116], is less than 20k cycles for almost 100% of the packets. This latency also depends on the size of the packets. For small packets with less than 5 cache blocks, the payload will be touched almost immediately after the header, as the driver copies such packets into another buffer. In this case, the attack without DDIO detects packet sizes for the small packets as readily as the attack with DDIO.

In short, DDIO makes the attack stealthier, and more reliable (less noise). But the attack is fully possible without DDIO. As an example, the web fingerprinting attack presented in the next section is mounted on a system both with and without DDIO.



Figure 1.13. Detecting successful login for hotcrp.com. Shows original packet sizes vs. the recovered packet sizes by Packet Chasing for the first 100 packets of the responses.

1.4 Packet Chasing: Exploiting Packet Size

In this section, we present a sample application for a Packet Chasing attack, in which we use the high resolution samples of packet sizes to gain information on the co-located user's browsing data. For example, the spy could be waiting for the victim to enter a particular website before initiating some action such as a password detection attack.

This simple attack consist of two phases. First is the offline phase in which the adversary generates traces of packet sizes for different websites of interest, then processes these traces and calculates a representative trace for each website. This is just a point-wise average of the packet sizes, resulting in a vector of these points (average packet size) over time.

In preparation for the attack, the attacker builds the sequence of the ring buffers, as previously described. After that, the attacker enables spy mode in which she constantly monitors the first two cache blocks of the first buffer in the sequence until she finds a window in which there are activities on both block 0 and block 1. This indicates that a packet is filling that buffer. Then, similar to the receiver in the covert channel, on each detected activity the attacker moves to the next buffer in the sequence. Each time, the attacker monitors the first four blocks of the first half-page of the buffer as well as the first four blocks of the the second half-page of the buffer. This is because the driver switches between the halves of the pages when there is a large packet (see Section 1.2.1). This enables the attacker to distinguish between packets with four level of sizes. After collecting the samples of packet sizes, the spy feeds the collected vector into a simple correlation-based classifier which calculates cross-correlation [291] of the collected samples with the representatives of different targets.

Figure 1.13 shows an example of the signals that we obtain by Packet Chasing and the actual packet sizes that are captured using the tcpdump [131] packet analyzer. The websites are accessed using Mozilla Firefox version 68.0.1. The figure shows how packet size, even in cache block granularity, can be an identifier for the webpages that are being accessed. The packets are usually congested on the two sides of the spectrum, they are either carrying a very large message that got fragmented into MTU-sized frames, or they are small control packets [238]. But the last packet of the large messages can fall anywhere between 1-block to MTU, giving us a good indicator of the webpages. In addition, combining packet sizes with the temporal information that Packet Chasing obtains from the packets, gives us enough information to distinguish between webpages. We evaluate our fingerprinting attack using a small closed world dataset with 5 different webpages: facebook.com, twitter.com, google.com, amazon.com, apple.com. For this experiment, we examined two attack setups, one with DDIO and one without. In our experimentation with 1000 trials, Packet Chasing with DDIO detects the correct website with 89.7% accuracy, while disabling DDIO drops the accuracy to 86.5%. The difference between these two attacks comes from increasing the probe time (resulting in more noise) and increased probability of missing large packets if the header-to-payload latency is high.

We use a simple classifier in this experiment, but given the challenges for this

Baseline Processor					
Frequency	3.3 GHz	Icache	32 KB, 8 way		
Fetch width	4 fused uops	Dcache	32 KB, 8 way		
Issue width	6 unfused uops	ROB size	168 entries		
INT/FP Regfile	160/144 regs	IQ	54 entries		
RAS size	8,16, 32 entries	BTB size	256 entries		
LQ/SQ size	64/36 entries	Functional	Int ALU(6), Mult(1)		

Table 1.2. Architecture Detail for Baseline Processor

particular attack, a classifier that is tolerant of noise as well as slight compression or decompression of the vectors would be likely to improve these results. For example, the results in [221] suggests that using only the network packet sizes and their timing information (the exact information that Packet Chasing can obtain) can be enough to build a classifier with up to 95% accuracy.

1.5 Potential Software Mitigation

We consider both long-term (e.g., requiring hardware changes) and short-term (software only) mitigations. In this section, we discuss potential software mechanisms that one could employ to help mitigate the attack before a long-term hardware solution (e.g., our I/O cache isolation) is deployed. These solution each come with some performance impact.

Disabling DDIO/DCA

DDIO enables these attacks because it ensures the header and the payload appear in the cache simultaneously, greatly simplifying the detection of packet size. Without this, however, attacks are still possible. If we can detect the presence of packets (headers are always accessed immediately and will appear in the cache in sequence), we can still establish a covert channel with inter-arrival timing. We could also send types of packets where the reading/processing of the payload is quick and deterministic, again allowing us to distinguish sizes. Therefore, disabling DDIO can not fully mitigate the attack.

Randomizing the Buffers

While Packet Chasing exploits the sequence in which the packets fill the ring buffers to boost the resolution of the side- and covert-channels, we show that attacks are still possible, without knowing the sequence of the buffers (Section 1.3). However, randomization does significantly reduce the channel bandwidth. The cost of randomization could be quite high, as the driver and the network adapter would now need to constantly synchronize on the address of the next buffer. Because our attack setup takes some time, though, it may only be necessary to permute the buffer order at semi-regular intervals, thus limiting the overhead.

Increasing the Size of the Ring

In the absence of sequence information, the required probing of the cache scales with the size of the ring if the attacker wants to catch every packet. Thus a combination of occasional reshuffling of the ring, and a larger ring, may be effective in making the probe set large enough to make the attack difficult to mount cleanly without picking up significant noise.

1.6 Adaptive I/O Cache Partitioning Defense

All the short-term software mitigations that Section 1.5 suggests are either not fully effective (disabling DDIO), or carry a not-insignificant performance cost. In this section, we describe a hardware defense that tackles the root of the vulnerability, i.e., co-location of I/O and CPU blocks in the last-level cache, in such a way that I/O can cause evictions of other processes' lines.

Intel's DDIO technology improves the memory traffic by introducing a last-level cache write allocation for the I/O stream. Upon receiving a write request from an I/O device (e.g., for incoming packets), DDIO allocates cache blocks in the last-level cache and sets those blocks as the DMA destinations for the incoming I/O traffic. While for

performance reasons, the allocator does not allocate more than two blocks in a cache set, these incoming packets can still cause evictions to the CPU's blocks. This makes the incoming packets observable from the perspective of an adversary process running on the CPU.

To circumvent this, we associate a counter for each set (*i*) to hold the size of the I/O partition (*IO_lines_i*). By treating this as a constant (during a single interval) rather than a maximum, we ensure that DDIO-filled lines will only displace other DDIO lines. To adapt to different phases of execution, our partition schemes periodically change the boundary of CPU and I/O partitions by incrementing or decrementing the counter (*IO_lines_i*). To this end, we associate another set of counters for each set to detect the I/O activity on each set (*IO_present_counter_i*). This counter gets incremented if at least one valid I/O line is present in the set, and initialized to zero at every adaptation period cycles (*p*). Note that maintaining these counters does not impose a performance overhead as these are done in parallel to the miss and hit path of the cache.

At every adaptation period, we also re-evaluate the I/O-CPU boundary in the last-level cache. For each set (*i*), if the (*IO_present_counter_i*) is greater than a high threshold, T_{high} (e.g, $T_{high} = 0.5p$), it implies set_i has had significant I/O activity. In such a case, we increment *IO_lines_i* (using a saturating counter), allowing more I/O blocks in the set. Otherwise if the I/O activity is less than a low threshold, T_{low} , we decrement *IO_lines_i* (again, using a saturating counter) to allow more usage for CPU data. If the boundary of the partitions changes, we invalidate the cache blocks that are affected and perform any necessary writebacks to the memory.

Our adaptive partitioning ensures that any process running on the CPU will not see any of its cache lines evicted as the result of an incoming packet or I/O activity. The only exceptional scenario is at each adaptation period when the boundary changes and some CPU blocks get evicted. However, we set the adaptation period to be large enough to prevent the attacker from gleaning any useful information about individual packets. At best, it could receive one bit (high or low network activity) every period.

System Setup for the Defense Performance Evaluation

Table 1.2 shows the architectural configuration of our baseline processor in detail. We model this architecture using the gem5 [28] architectural simulator. We use the full system simulation mode of gem5 which allows us to boot an Ubuntu 18.04 distribution of Linux with a kernel version of 4.8.13. We set a hard limit on the minimum and maximum number of blocks in the I/O partition (i.e., *IO_lines_i*). As such, the size of the I/O partition can be one, two, or three. Also, in these experiments the adaptation period (*p*) is set to 10k cycles. We set the thresholds T_{low} and T_{high} to 2k and 5k, respectively. Furthermore, in order to provide realistic estimates regarding the performance impact of our proposed defense, we select a mix of benchmarks that exhibit considerable amounts of I/O activity. To this end, we include a disk copy (using Linux's *dd* tool) that copies a 100MB file from disk. In addition, we evaluate the defense on a program that constantly receives TCP packets that have 8-byte payloads. Finally, we also evaluate the impact of our defense on the *Nginx* web server [219] using the *wrk2* [90] framework to generate HTTP requests.

Performance Results of the Defense

Figure 1.14 shows the performance of our adaptive cache partitioning scheme by comparing the average throughput of the Nginx web server. On average, we observe less than two percent loss in throughput. This is mainly because the LLC miss rate rises slightly due to the reduced number of lines in the CPU partitions (also see Figure 1.15). The figure also shows the sensitivity of the defense to the last-level cache size. The maximum loss in throughput belongs to the 20 MB case where our approach incurs 2.7% loss. Figure 1.15 further analyzes the performance of the defense by showing the memory traffic and the LLC miss rate of a baseline without any direct cache access (*No DDIO*) vs. DDIO and our adaptive partitioning defense. Both the adaptive partitioning



Figure 1.14. Performance impact of our adaptive partitioning defense on Nginx web server.



Figure 1.15. Memory Traffic and LLC miss rate of our adaptive partitioning defense vs. DDIO.

and DDIO are effective in reducing the memory traffic. The memory traffic of the adaptive partitioning scheme is within 2% of DDIO.

To compare the adaptive cache partitioning with our proposed software-based mitigations (Section 1.5), we devise another experiment using the wrk2 tool. In this experiment we send requests to the Nginx web server on the target host. The wrk2 tool uses eight threads with 1000 open connections and the target throughput is set to 140k requests per second. Figure 1.16 shows the results of this experiment. Besides the adaptive cache partitioning and the vanilla IGB baseline, we examine three other proposed schemes: *Fully Randomized Ring Buffer* scheme that allocates a new buffer in a random memory location for each incoming packet, and two *Partial Randomization* schemes that re-allocate the buffers periodically, after a specified number of packets received – we randomize after either 1k or 10k packets are received. Note that in our setup, the Packet Chasing attack currently requires at least 65,536 packets to fully deconstruct the ring buffers (find cache locations and sequence information) and another 100 packets to mount a reasonable fingerprinting attack. The adaptive



Figure 1.16. Comparison of our defenses in terms of response (tail) latency of HTTP requests to the Nginx web server. Randomization period is the interval (measured in number of packets) that we wait between two ring buffer randomizations.

partitioning method only incurs 3.1% loss in 99th percentile latency while the fully randomized method incurs 41.8%. We use One Gigabit Ethernet for this experiment, but we expect the performance cost of randomization to be exacerbated as the link rate goes higher.

1.7 Conclusions

This paper presents Packet Chasing, a novel deployment of cache side-channel attacks that detects the frequency and size of packets sent over the network, by a spy process that has no access to the network, the kernel, or the process(es) receiving the packets. This attack is not enabled by the DDIO network optimization, but is greatly facilitated by it. This work shows that the inner workings of the network driver are easily deconstructed by the spy process setting up the attack, including the exact location (in the cache) of each buffer used to receive the packets as well as the order in which they are accessed. These two pieces of information dramatically reduce the amount of probing the spy must do to follow the network packet sequence. This information enables several covert channels between a remote sender and a spy anywhere on the network, with varying bandwidth and accuracy tradeoffs. It also enables a side channel leakage attack that detects the web activity of a victim process.

In addition to the covert- and side-channel attacks, this chapter also describes an adaptive cache partitioning scheme that mitigates the attack with very low performance overhead compared to the vulnerable DDIO baseline.

Acknowledgment

We thank the anonymous reviewers for their helpful insights. Chapter 1, in full, is a reprint of the material as it appears in International Symposium on Computer Architecture (ISCA) 2020. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Algorithm 1. Ring Buffer Sequence Recovery

1:	procedure Sequencer	
2:	$samples \leftarrow get_clean_samples(Nsets, Nsamples)$	
3:	$graph \gets build_graph(samples)$	
4:	$sequence \leftarrow make_sequence(graph)$	
5:	return sequence	
6:	<pre>procedure GET_CLEAN_SAMPLES(Nsets, Nsamples)</pre>	
7:	monitor_list $\leftarrow [0Nsets]$	
8:	$samples \leftarrow repeated_probe(Nsamples, monitor_list)$	
9:	for all $x \in monitor_{list} do$	
10:	if activity(samples[x]) > activity_cutoff then	
11:	replace x in monitor_list with the 2 nd block of the page	
12:	goto: 3	
13:	return samples	
14:	procedure BUILD_GRAPH(samples)	
15:	$\operatorname{curr} \leftarrow 0$, $\operatorname{prev} \leftarrow 0$	
16:	for $i \in \{0,, SAMPLES\}$ do	
17:	for all cand \in monitor_list do	
18:	if samples[i][cand] < miss_threshold then	⊳ no activity
19:	continue	
20:	if $curr \neq prev$ then	⊳ no self-loop
21:	$graph[prev][curr][cand] \leftarrow graph[prev][curr][cand]+1$	
22:	$(\text{prev, curr}) \leftarrow (\text{curr, cand})$	
23:	return graph	
24:	procedure MAKE_SEQUENCE(graph)	
25:	$root \leftarrow get_root(graph)$	
26:	sequence \leftarrow [], (prev,curr) \leftarrow root	
27:	repeat	
28:	sequence.push(curr)	
29:	$(next,weight) \leftarrow get_max_weight(graph[prev][curr])$	
30:	if weight < weight_cutoff then	
31:	break	
32:	$graph[prev][curr][next] \leftarrow o$	▷ mark as visited
33:	$(prev, curr) \leftarrow (curr, next)$	
34:	until (prev, curr) \neq root	
35:	return sequence	

Chapter 2 Context Sensitive Decoding

The post-Dennard scaling era has witnessed an upsurge in the adoption of specialized processing elements to improve the execution efficiency of domain-specific workloads. While general-purpose processors continue to gradually add domain-specific instructions every CPU generation, the technical challenges and market risks associated with legacy software have significantly limited innovation in the ISA design space. This work exploits an underutilized feature of modern instruction set decoders to show that even general-purpose processors can be customized, and in fact that customization can be seamlessly configured dynamically at an extremely fine granularity.

The key to this change is the fact that most modern processors employ translated ISAs, as the Intel and AMD x86 processors and many ARM processors typically feature translation from the native instruction set into internal micro-ops that enter the pipeline for execution [19, 126]. These architectures enjoy the dual benefits of a versatile backward-compatible CISC front-end and a simple cost-effective RISC back-end. Moreover, the additional level of indirection enables seamless optimization of the internal micro-op ISA, under the covers, without any change to the programmer interface. However, for those architectures the translation is static, changing once per generation. Instead, we propose that translation be dynamic, potentially changing

frequently within the execution of a single program.

In this chapter, we unlock the full potential of translated ISAs via *context-sensitive decoding* (CSD), a technique that allows native instructions to be decoded/translated into a different set of custom micro-ops based on their current execution context. This presents operating systems, runtime systems, and antivirus programs with the unique opportunity of triggering different custom translation modes, at microsecond or finer granularity, by simply configuring a set of model-specific registers (MSRs). In this way, for example, an insecure executable can instantly become a secure executable, or performance-optimized code can become energy-optimized, without recompilation or binary translation.

By leveraging existing native-to-microcode translation functionality in the decoder and exploiting an already well-established microcode update procedure outlined by Intel [126], we further empower runtime systems and virtual machines (that operate at a certain privilege level) to push custom translation updates written in native x86 code into the processor. At the decode stage of the pipeline, the CSD framework intercepts such custom microcode updates, *auto-translates* and optimizes them into a compact set of micro-ops, and pushes them into the microcode engine. These custom updates could potentially enable instrumentation for profiling and performance monitoring, profile-guided optimizations, and API-hooks for security updates, among other applications.

The CSD framework we describe allows custom translation modes to be triggered by hotspot detection [158], unit-criticality predictors [157], thread-criticality predictors [25], protection-domain crossings [293], interception of a tainted input [112, 199, 245, 277], a watchdog timer event, changes in power or energy availability, or thermal events – all with no significant changes to the pipeline or the ISA. In fact, a major contribution of this work is a set of microarchitectural techniques that enable the seamless integration of the context-sensitive decoding framework into Intel's legacy decode pipeline and micro-op cache design.

Due to its low performance overhead and non-intrusive nature, context-sensitive decoding has potential applications in areas such as malware detection and prevention [24, 38, 224], dynamic information flow tracking (DIFT) [112, 113, 200, 245], runtime profiling and performance programming [105, 311], on-demand type-safety [194, 307], program verification and debugging [95, 96, 281], and runtime phase tracking and code specialization [100, 236]. This chapter showcases two diverse applications of context-sensitive decoding – an obfuscation-based security defense against cache-based side channel attacks, and criticality-aware power gating to improve energy efficiency.

Side-channel attacks have been used to leak secret information by exploiting the micro-architectural and physical characteristics of a cryptosystem. Many types of side-channel attacks have been described in the literature to subvert prominent cryptographic algorithms such as RSA, DES, and AES. These attacks hinge on a spy program running side-by-side with a victim that leaks timing and other execution characteristics via shared micro-architectural structures.

By leveraging custom translation modes offered by context-sensitive decoding, we provide a low-cost, high-performance, and reconfigurable alternative to existing side-channel mitigations [166,217,284]. Owing to its unfettered access to on-chip microarchitectural structures and an array of hardware control signals, context-sensitive decoding allows us to inject *decoy micro-ops* into execution that give the attacker an illusion of a modified architectural state, by obfuscating micro-architectural characteristics alone. These decoy micro-ops are unreadable from both user and kernel modes as they exist within the processor outside any addressable memory. As a result, they remain invulnerable to spyware, rootkits, and other rogue programs, even if they are able to execute with the highest privileges. This chapter shows that by causing micro-architectural perturbations at the decoder level, we can be more performance-efficient than a software-based obfuscation technique and less intrusive than a system that

causes anomalies at the gate level.

Aside from security, we also showcase the potential of CSD in efficiently emulating infrequently used feature sets on alternative functional units. This enables aggressive power gating, even for units that are infrequently but regularly used. In this case, we scalarize vector instructions via micro-op translation onto the scalar units, enabling the system to make more global decisions about when to turn on the vector units, rather than always responding to instruction demand. In summary, the framework we describe in this chapter offers the following unique capabilities:

- Fine-grained dynamic instruction stream customization of legacy binaries without recompilation, and without the full overhead of binary translation.
- Seamless integration into a state-of-the-art Intel processor with no significant changes to the pipeline.
- A flexible *auto-translated* microcode update procedure that allows runtime systems to inject custom translation modes into the microcode engine.
- A firmware-based security defense that completely thwarts instruction and data cache-based side channel attacks on the RSA and AES cryptographic algorithms, while significantly outperforming state-of-the-art software-only solutions.
- An energy-optimization mechanism that scalarizes vector instructions in order to power gate vector units during phases of minimal vector activity to save an average of 12.9% in overall energy.

2.1 Background and Related Work

Translated Instruction Sets. Modern ISAs such as x86 and ARM typically translate complex native instructions into simpler internal micro-ops [19, 126]. While this was originally intended to simplify CPU design and allow complex long latency

47

instructions to be pipelineable, it has been instrumental in enabling several ISA and micro-architectural optimizations [109, 110, 143, 144, 225] that improve the front-end throughput and overall instruction-level parallelism [216]. Speculative decode [143] exploits the macro-op to micro-op translation in order to enable dynamic optimizations such as memory reference combining and silent store squashing. While similar in nature with respect to decode-time instrumentation, CSD offers broader functionalities such as the ability to be programmed via microcode updates, on-demand customization at an extremely fine granularity, as well as seamless integration into the modern x86 front end.

Multi-ISA Architectures. Many modern decoders are equipped with multiple decode units to translate instructions from different feature sets, ISA extensions, and sometimes completely different ISAs. For example, ARM supports three major instruction sets (A₃₂, T₃₂, and A₆₄) and several other feature sets such as Jazelle and NEON. It also allows developers and compilers to take advantage of the ability to switch between the different instruction sets at exception boundaries (A₆₄ to A₃₂) or by simply executing a branch and exchange instruction (A₃₂ to T₃₂). Furthermore, the multi-ISA heterogeneous chip multiprocessor architectures [21,69,274,275] allow applications to migrate back and forth between different ISAs at basic block boundaries. While this work offers similar capabilities in terms of seamlessly switching execution between different custom translations, it does so at a much finer granularity, requires no re-compilation, and no significant changes to the architecture.

Binary Translators and Code-Morphing Machines. In the software world, binary translators have long been used to port/emulate legacy binaries on new architectures [33]. Furthermore, managed runtimes and browsers employ dynamic binary translation to perform profile-guided optimization [111] of hot code regions, program sheperding, and JIT hardening [273]. On the hardware front, several binary-translation-driven processor designs have been proposed. These are typically equipped with a

code-morphing software binary translation layer that feeds translated instructions into the processor's decoder. IBM's DAISY [77], Transmeta's Crusoe and Efficieon [64], and Nvidia's Denver processors [29] have sparked further innovation in this space. Clark, et al. [50] describe a hybrid approach to instruction set customization that involves statically identifying code regions to offload and dynamically replacing jumps to such regions by complex custom instructions that trigger an accelerator.

The most related work to this research is DISE (Dynamic Instruction Stream Editing) [53–55], a macro-engine that exposes the API, allowing programmers to dynamically reconfigure a stream of instructions in order to perform bounds-checking, debugging, and prefetching. However, this work differs in many important ways. First, they require complex patten-matching and user-defined production rules to be integrated into their decoder framework, whereas context-sensitive decoding can be triggered by mere reconfiguration of a set of model-specific registers or even a pipeline event, or a thermal or energy event. Second, while this work is easily integrated, exploiting existing features of modern processors, DISE adds significant new complexity to the pipeline. Third, this work fully explores performance, power, and area implications of incorporating the decoding framework into existing modern designs, including those that sport a wide variety of micro-op optimizations [126]. Finally, our research builds on works like DISE by introducing new applications – better protection against several new attack models that have gotten more sophisticated over the years, and energy-efficient management of vector computation.

Side-channel attacks. Side-channel attacks typically steal secret information from cryptosystems and other sensitive data from a co-located user on the cloud [222, 288]. Numerous spy programs have demonstrated the full/partial reconstruction of a victim's execution behavior by observing its instruction/data cache access patterns [10, 23, 195, 304, 306], branch access patterns [9], differential power consumption characteristics [27], electromagnetic radiation [85], acoustics [88], and fault behav-

ior [108].

Several cache-based side channel attacks have been proposed in the literature [98, 205, 304]. Prominent ones include PRIME+PROBE and FLUSH+RELOAD attacks that can be performed on both a shared private data/instruction cache or on last-level caches. Notable mitigations to these attacks include secure cache partitioning [284], compiler-based obfuscation [166, 217], and run-time software diversity [57]. In this chapter, we leverage context-sensitive decoding to provide stealth-mode custom translations, as a novel security defense that mitigates such side channel attacks. Context-sensitive decoding-based security enhancement has no software performance cost when not in use, minimal software overhead when used, no additional vulnerability, and very minimal hardware/power cost.

Unit-level Power Management. Many power management techniques have been proposed in prior work [73, 80, 175, 186] ranging from unit-level [73, 80, 175] to coarse-grained core-level power management [46, 160], in both cases powering off idle blocks to reduce overall static leakage. *Vector processing units* (VPUs) are promising candidates for power gating since they're typically not in use during most scalar phases, and yet account for a significant portion of the core's peak power. However, phases of intermittent vector activity create small idle intervals that are below the break-even time needed to compensate the power gating overhead.

Dynamic devectorization [150] achieve significant energy savings by using a translation optimization layer to profile and devectorize non-critical vector instructions while the VPUs are power-gated. Similarly, PowerChop [157] proposes a binary translation-driven approach that uses a unit-criticality predictor to assist power-gating of multiple units in the processor (including VPUs). While binary translation can be an effective tool, it is not ideal for adaptive energy optimizations – in many scenarios we can hide the considerable startup cost (in performance and energy) of binary translation; however, when we trigger a new optimization due to an energy event or

emergency, we bear the entire brunt of the startup cost at the worst possible time.

2.2 Context-Sensitive Decoding

In this section, we provide a brief overview of the x86 front-end, describe techniques to enable context-sensitive decoding in the x86 architecture, and discuss potential applications.

2.2.1 Overview of the x86 Front End

The x86 front end in Figure 2.1 has two major components: (a) the legacy decode pipeline that translates native instructions into micro-ops, and (b) a micro-op cache that delivers already translated micro-ops into the instruction queue.

The legacy decode pipeline includes an instruction-length decoder that feeds from a 16-byte fetch buffer and decodes the variable-length x86 instruction byte-by-byte. The decoded instructions are inserted into an 18-entry macro-op queue, 6 macro-ops at a time. These macro-ops then feed into one of the four decoders that translate them into micro-ops. These decoders use a static table-driven approach for the micro-op translation. In fact, only one of the decoders can translate an instruction to more than one micro-op, with the other three performing a simple one-to-one mapping operation. Complex instructions that decompose into more than four micro-ops are microsequenced by a microcode ROM.

The micro-ops translated by the legacy decode pipeline are cached in an 8-way set associative micro-op cache that can hold up to 1536 micro-ops. When the micro-op cache is active, the legacy decode pipeline is disabled to conserve power. The front-end then streams micro-ops from the micro-op cache into the instruction (micro-op) queue until a miss occurs, at which point it switches back to the legacy decode pipeline. The front-end also sports a number of optimization features such as stack-pointer tracking, micro-op fusion, macro-op fusion, and loop stream detection.



Figure 2.1. Intel Front End with CSD Support.

2.2.2 Integration with the x86 Front End

This section describes techniques to integrate our architecture into the x86 front end with the legacy decode pipeline and micro-op cache designs, and further study its synergy/interference with existing front-end optimizations such as micro-op fusion. Figure 2.1 highlights necessary hardware components required to enable context-sensitive decoding in the x86 front-end.

Integration with the Legacy Decode Pipeline. To enable context-sensitive decoding in the x86 front-end, we provision the legacy decode pipeline with one or more custom decoders that perform custom translations. These decoders continue to employ a simple static table-driven translation model, like the four native x86 decoders. However, they can generate more sophisticated micro-op flows by relegating to the microcode ROM. CSD does not require that micro-ops of an instruction be committed atomically. This is consistent with the current implementation of Intel processors which only commit 6 fused micro-ops per cycle while they allow as many as 260 micro-ops

per one instruction (e.g., FBSTP) [83].

Furthermore, when context-sensitive decoding is turned on, we update the macro-op dispatch logic to redirect macro-ops that require custom translation to the custom decoder. In our initial implementation, this logic can be triggered in three different scenarios. First, software programs such as the operating system can trigger this logic by configuring a set of model-specific registers (MSRs) [126]. We leverage the already existing register-tracking optimization in the decoder to track updates to the MSRs and consequently trigger context-sensitive decoding. Second, we allow a translation context switch to be triggered by hardware events such as the interception of a tainted input by information-flow tracking or a power-gating decision by the unit criticality predictor. Finally, we allow a hardware watchdog timer to periodically trigger a translation mode switch.

Interactions with the Micro-Op Cache. The micro-op cache is an important performance and energy optimization that allows certain hot code regions to be completely serviced from the micro-op cache. The Intel Optimization manual recommends software to be carefully optimized since frequent switching between the micro-op cache and the legacy decode pipeline could cause more performance degradation than running without the micro-op cache [125]. This particularly conflicts with one of the major goals of context-sensitive decoding – the ability to frequently switch translation context at a low performance overhead.

Flushing the micro-op cache every translation mode switch could have a major performance impact. We instead choose to extend the tag bits of the micro-op cache with an additional set of context bits (one bit per custom translation mode) that associate a particular micro-op way with the decoder that translated it. While this could potentially create artificial conflict misses, it allows us to improve the micro-op cache utilization by co-locating micro-op translations from different custom decoders.

Finally, customization could involve injecting multiple micro-ops at a time. This

not only clutters the execution stream, but could pollute the micro-op cache. The x86 micro-op cache design has a check that does not allow 32-byte code regions to occupy more than 3 ways (amounting to 18 micro-ops) in the micro-op cache. This is because, unlike a regular cache, the micro-op cache simply allows the front-end engine to stream instructions from it, to avoid expensive indexing and tag comparison. Furthermore, it does not allow instructions longer than six fused micro-ops to be cached. Although we can imagine several options that would allow the architecture to remove that constraint, to be conservative we assume that it still holds in this chapter, which does impact many of our translated micro-op sequences.

2.2.3 Microcode Update and Auto-Translation

CSD exploits the already existing *Microcode update* (MCU) procedure of Intel processors [126] to empower the runtime system with the ability to inject custom translations into the processor's microcode engine, with the API provided to the runtime being the entire x86 instruction set. The CSD framework further *auto-translates* such microcode updates by exploiting Intel's existing front-end translation and optimization infrastructure. While this offers significant flexibility to software agents such as the OS and the runtime system, the chip designer exerts more control over the microcode engine, potentially allowing custom translations that include non user-visible features such as a micro-op that can change the state of the branch predictor or the hardware return address stack. We also note that custom translations injected via microcode updates should not alter architectural register and memory state, unless explicitly specified in the MCU header.

Figure 2.2 shows the MCU procedure in more detail. Since microcode update is performed via a privileged instruction or system call, only trusted entities [18, 244,246] within the OS/runtime system should have the ability to successfully inject microcode updates into the processor. The microcode update system call invokes
Intel's microcode driver [63] that performs sanity and integrity checks, and further invokes the processor's microcode update feature via an MSR update [126]. The MCU itself is provisioned with a descriptive header prepended by data containing custom translations injected by the runtime. When the header contains a reserved field that indicates context-sensitive decoding, the microcode update is assumed to contain only native x86 instructions, and is further marked for *auto-translation*. On the processor end, the MCU header is again verified for sanity and integrity, before extracting the data part. In the event that the MCU is marked for *auto-translation*, the native instructions in the data part of the MCU are further translated into internal micro-ops by leveraging the existing translation capabilities in the decoder. The translated micro-ops are further optimized into more compact micro-ops using existing front-end optimizations such as macro/micro-op fusion, adhering to certain performance guidelines described below. We further note that virtually all of the building blocks we use to provide this feature are already well-established mechanisms that appear in mainstream Debian Linux kernel releases [63].

2.2.4 Performance Guidelines and Optimizations

The micro-op expansion due to customization could potentially have a negative impact on overall performance if the custom translations are not optimized before they are injected into the dynamic execution stream. In this chapter, we take advantage of several existing optimizations in the micro-op engine in order to eliminate bottlenecks at the front-end and potentially, throughout the pipeline.

Micro-op fusion. To take advantage of the micro-op fusion optimization, we use *load+op* and *load+br* combinations as much as we can in our custom micro-op sequences. By doing so, we gain 1.6% in performance and eliminate bottlenecks in the front end stages, and the micro-op cache.



Figure 2.2. Auto-translation Procedure

Micro-loop Specialization. We attempt to create short and tight loops that benefit from the improved micro-op cache utilization, and take advantage of the loop cache when present. The Intel optimization manual [125] recommends loop fission [185] in case of longer loops. If the custom decoder decides to employ loop fission on micro-loops, we recommend that these loops don't occur too close together in the pipeline as they can potentially knock native instructions in a 32-byte region out of the micro-op cache, causing severe degradation in performance.

Register Tracking. Finally, customization may potentially involve reusing microregisters (e.g., as a loop induction variable). By extending the stack pointer tracker to perform full register tracking, more compact instances of custom micro-op translations could be created.

2.2.5 Potential Applications

Later sections of this chapter focus on particular applications of context-sensitive decoding including (1) adding security on-demand and (2) selectively moving vector computation on and off the vector unit to minimize energy. However, other potential applications (subject of future work) abound.

Programming Languages – To reduce costly time spent on finding and fixing bugs, developers are increasingly encouraged to employ software practices that ensure software fault isolation, type safety, and formal verification [159, 194, 307]. While most type checkers and proof assistants rely on static verification, the dynamic nature of JavaScript and other JITed code has made it increasingly hard to statically infer and reason about types. Typed assembly languages [58] and Google's Portable Native Client [307] ensure deep sandboxing and type safety of inherently native and/or JITed code, but at a prohibitively high performance cost for many workloads. With CSD, we can read metadata at the time of a load to identify type, and pass flags between instructions to track computation on registers, thereby increasing the coverage for sensitive code regions where static verification is insufficient.

Debugging – Aside from type checking, breakpoints and watchpoints are indispensible tools that software developers use to find and fix evasive memory errors. Modern ISAs with hardware debugging support reserve a small number of monitor/debug registers to encode breakpoint/watchpoint rules [96, 281]. However, most debuggers and runtime analyses typically run out of debug registers and resort to software breakpoints and watchpoints which are extremely inefficient [95]. In translated ISAs, a context-sensitive decoder can microsequence a performance-efficient watchpoint implementation since it has direct access to microarchitectural structures such as the address translation unit.

Performance Counters - Modern processors implement a variety of perfor-

mance counters, but with several limitations. Only a few can be used at once, and the actual counters typically change from generation to generation, often dependent on where the designer had room for a counter and where they did not. However, we can add many counters in the decoder, with no limit to the number of counters active at once, and providing compatibility across generations despite different layouts and space availability.

Profiling – Modern systems typically rely on instrumentation to profile code. However, intrumentation alters the code, potentially resulting in *heisenbugs*. That is, instruction cache, data cache, and even memory interference behavior is altered by the instrumentation. With CSD, we can add profiling with no change whatsoever to code layout or data layout.

2.3 Case Study I:Side-Channel Defense

In this section, we demonstrate the security potential of context-sensitive decoding. We first lay out our assumptions and threat model, then describe the stealth-mode translation feature of context-sensitive decoding. Finally, we leverage this feature to secure commercial implementations of RSA and AES against the exploitation of the two major data and instruction cache side channel attacks.

2.3.1 Assumptions and Threat Model

Trusted Computing Base. We assume that the micro-op engine – which includes both the legacy decode pipeline and the micro-op cache – is tamper-proof and is a part of the Trusted Computing Base (TCB) [244, 247]. We also further extend the TCB to include all hardware or software mechanisms that can potentially trigger context-sensitive decoding. These include register tracking, dynamic information flow tracking [245], hardware watchdog timers, and anti-virus-driven stealth mode configuration (e.g., an Intel/McAfee security solution). Moreover, we assume that such

hardware security mechanisms, including any microcode that enables security, are formally verified [82]. Note that we only assume that the trigger/injection mechanisms (e.g., watchdog timer, microcode update kernel module, context-sensitive decoder) are a part of the TCB, but the instruction stream itself need not be. Finally, since the API exposed to software only consists of macro-ops, we continue to assume that the translated micro-ops in the micro-op cache (both native and custom-translated) can neither be read by software nor be probed via hardware side-channels.

Attacker Environment. We assume an active attacker who can effortlessly probe, flush, or evict a co-located victim's cache lines, but does not have direct access to the contents in the cache. We also assume that the attacker has the ability to make precise timing measurements and has unlimited access to hardware performance counters. This allows them to make inferences about the software algorithm being run by the victim, by observing the micro-architectural (cache) characteristics alone.

2.3.2 Stealth-Mode Translation

Cache-based side channel attacks typically involve probing one or more cache lines of a co-located victim in order to capture its memory access patterns that could potentially reveal secret information. For example, an attacker who intends to break a cryptographic algorithm could compute one or more bits of a secret key by capturing access patterns of key-dependent loads and branches. The goal of the stealth-mode translation is to provide an illusion of a modified architectural state by obfuscating the micro-architectural characteristics alone. In this specific implementation, we obfuscate a victim's control path and/or access to sensitive data structures in an attacker-oblivious way. In particular, we use *decoy micro-ops* that load data into the caches that would be touched on all data-dependent paths. These include all cache blocks that contain T-tables of AES and the *multiply* functions of RSA. While we intend stealth-mode to be a security feature to be deployed by the chip manufacturer, trusted software



Figure 2.3. Context-Sensitive Decoding with stealth mode



(c) Micro-ops

Figure 2.4. Translation of MOV instruction in stealth mode

entities [244] with the right privileges can achieve similar effects by leveraging the *auto-translated* microcode update feature.

Figure 2.3 shows context-sensitive decoding with stealth-mode translation in action. Stealth-mode translation is primarily triggered by updates to register-tracked *decoy address-range registers*, similar to the already existing Memory Type Range Registers (MTRR) [126] in x86 that allow system software to control cache policies for specific address ranges (e.g., write-back vs write-through). The decoy address range

registers, on the other hand, allow anti-virus and other hardware/software trusted entities to mark specific data and instruction address ranges in a program's address space as sensitive. As soon as the stealth-mode translation is triggered, these decoy address ranges are copied to the context-sensitive decoder's internal registers; after that, the macro-op dispatcher starts redirecting all loads and branches within the PC range for custom translation.

CSD injects *decoy micro-ops* into all instructions that use memory operands and/or attempt control transfer (i.e., all instructions that get translated to load/store/branch micro-ops) during stealth-mode translation. Figure 2.4, as an example, shows stealth-mode translation of the MOV instruction for cache-based side-channel prevention. In this example, CSD injects a micro-loop into the micro-op stream. The micro-loop effectively obfuscates the architectural state by loading all sensitive cache blocks whose addresses have been specified by a software agent (e.g., antivirus) in the MSRRs.

We implement two schemes of translation – one for a software anti-virus-driven stealth-mode configuration where the tainted program counter (PC) values are known a priori with the help of binary analysis and configured in specific MSRs, and one for architectures that implement full information-flow tracking in hardware where the taint-checking is performed dynamically. In both instances, the *decoy micro-ops* execute only for tainted instructions – for the DIFT-enhanced architectures, this decision is made dynamically at run-time.

We do not need to load the decoy structures constantly, since they will stay in the cache for a time, or until the attacker removes them. Thus, stealth-mode translation automatically turns itself off once all the address ranges in the contextsensitive decoder's internal copy of the *decoy address-range* MSRs have been emptied out (all blocks specified by the range registers are loaded) by the *decoy micro-ops*. However, before turning itself off, the context-sensitive decoder starts the hardware watchdog timer in order to periodically trigger stealth-mode translation. It is important to carefully configure the watchdog's timeout period to a value that is smaller than the attacker's best possible probe interval period, but large enough to minimize the performance degradation caused by the *decoy micro-ops* now flowing through the pipeline.

In the next section, we show how stealth-mode translation can be used to secure the instruction and data caches by defeating both the PRIME+PROBE and FLUSH+RELOAD variants of cache-based side-channel attacks. We demonstrate these on two specific instances of known vulnerabilities, but our scheme extends to any application where the secure-data-dependent code and data access ranges can be identified.

2.3.3 Securing the Instruction Cache

Instruction cache-based attacks have been able to successfully break prominent cryptographic algorithms, such as RSA, by being able to capture their key-dependent instruction access patterns. In this section, we show how RSA is inherently vulnerable to the I-cache attack and describe how stealth-mode translation can defeat the attack.

The RSA Cryptographic Algorithm. RSA is a popular public-key cryptographic algorithm that uses modular exponentiation for encrypting messages. Most commercial implementations of the RSA algorithm, including PGP and the open-source version GnuPG, use the *square-and-multiply* algorithm that considerably speeds up modular exponentiation. The algorithm iterates over the binary representation of the exponent in order to selectively perform exponentiation using three major suboperations: *square, multiply*, and *reduce*. While *square* and *reduce* are performed in each loop iteration, *multiply* is invoked only when the exponent bit is 1, which invariably entails a key-dependent branch. I-Cache Side-Channel Attack on RSA. The sub-operations of the *square-and-multiply* algorithm are implemented as fairly large functions that span multiple cache blocks. This implies that the I-cache access patterns captured by a co-located spy process could potentially reveal several bits of the exponent. A PRIME+PROBE attack that exploits this side channel [8] fills up the I-cache set for the *multiply* operation in the prime phase, and probes for it after a carefully chosen interval to check if the victim evicted its block. A FLUSH+RELOAD attack on the other hand, leverages shared libraries and/or page de-duplication in order to flush the cache line that corresponds to the *multiply* routine, and further reloads it after a carefully chosen probe interval – a quick access indicates the victim has accessed the code and brought it into a shared cache.

Effect of Stealth-Mode Translation. As a defense mechanism against I-cache based side-channel attacks, we use stealth-mode translation to periodically (potentially, at every probe interval) inject decoy instruction-load micro-ops into the pipeline. Stealth-mode can seamlessly and instantaneously be enabled for RSA by configuring the instruction *decoy address range* MSRs with the address-range of the *multiply* functions. By periodically loading the right set of cache blocks into the I-cache, stealth-mode successfully obfuscates the instruction access pattern that is perceived by the attacker.

2.3.4 Securing the Data Cache

As with instruction cache-based attacks, attackers have also exploited data-cache side channels to infer secret information by observing a victim's key-dependent data access patterns. In this section, we describe a well-known data cache-based side channel in OpenSSL's implementation of the AES algorithm and discuss the potential of stealth-mode translation to thwart these attacks. The AES Cryptographic Algorithm. AES Algorithm is a substitution-permutation block cipher that performs several rounds of simple substitution and permutation during encryption. Several software implementations including OpenSSL employ lookup tables called T-tables in order to speed up the substitution-permutation rounds, which then consist of several simple table lookup and *xor* operations. The index computation for the T-table lookup involves an *xor* operation between the key bits and the plaintext bits, thereby entailing a key-dependent load.

D-Cache Side-Channel Attack on AES. The OpenSSL implementation of AES employs four 256-entry T-tables, which amounts to sixty-four 64-byte cache blocks in the data cache. A spy process that monitors the access patterns of these blocks during encryption can significantly reduce the possible key space, and potentially reconstruct the entire key, by using a large number of carefully chosen plaintext [23]. As with I-cache attacks, a PRIME+PROBE attack fills up the D-cache sets for one or more of these T-table blocks in the prime phase, and probes for them after a certain interval to check if the victim made an access to any of the primed sets. A FLUSH+RELOAD D-cache attack exploits de-duplication in order to flush one or more of the T-table blocks, and reload them after a carefully chosen probe interval.

Effect of Stealth-Mode Translation. Similar to the use of stealth-mode translation to defend against I-cache attacks, by configuring the data *decoy address range* MSRs with the appropriate address range of the T-tables, we can successfully obfuscate the key-dependent data access patterns. Furthermore, by carefully choosing a watchdog timeout period to enable periodic *decoy load* injection, we can also defend against brute-force key extraction attacks [207].

2.3.5 Securing other Side-Channels.

Although the primary focus of this work is to defend against cache-based sidechannel attacks, we note that the stealth-mode translation feature of CSD could be



Figure 2.5. CSD with Selective Devectorization

exploited in future to defend against other timing and physical attacks. For example, the decoy micro-ops could alter the branch predictor tables and BTB entries to confuse branch prediction analysis attacks, or potentially introduce a random stream of NOPs (and different types of NOPs) to skew timing analysis. Furthermore, owing to its ability to microsequence instructions using the MSROM, it can add additional noise into the sensed power by non-deterministically microsequencing instructions that cause switching activity across different microarchitectural structures.

2.4 Case Study II: Unit-Level Power Gating

In this section we present another use case of context-sensitive decoding, selective devectorization for unit-level power-gating. While similar in functionality to software-based devectorization approaches proposed in prior work [150, 157], we eliminate binary translation costs and related cache effects, add the ability to switch modes at a finer granularity, allow cheaper and more effective monitoring, and provide more direct control over power gating and ungating. Unit-level power gating is one of our primary tools to reduce leakage power by disconnecting the supply voltage of an idle unit. However, upon encountering demand for that unit it must connect back to the supply voltage. Powering a unit on and off uses power and energy cost but also slows execution, typically stalling until the unit is activated. We leverage context-sensitive decoding to enable efficient and more fine-grained power-gating of vector units. CSD alleviates these shortcomings in two ways: 1) for infrequent vector instructions, it avoids powering on the unit by translating vector instruction to equivalent scalar micro-ops, and, 2) it hides the power-on delay by continuing the execution of instructions using scalar mode until the unit is ready.

Figure 2.5 shows our hardware support (beyond the decoder) for dynamic devectorization. We employ nothing more than a simple counter that tracks a window of instructions, counting up one for simple vector instructions and more than one for more complex vector instructions (higher micro-op count). When it goes below a threshold, it turns on devectorization and powers off the entire vector unit, and when it goes above a (higher) threshold, it turns the vector unit back on. It also includes a cycle counter to continue devectorization until the vector unit is fully powered.

When devectorization is enabled, the microcode engine translates the vector instructions to an equivalent set of scalar micro-ops. As an example, Figure 2.6a shows the pseudocode that devectorizes the SSE *PADDB* instruction which performs integer addition on packed bytes. This code is further compiled and optimized into a set of native x86 instructions by a runtime system which performs the actual microcode update. Figure 2.6b shows the equivalent auto-translated micro-op version of such an update. While it is possible to use a simpler translation with a loop, we find that it is more efficient to unroll the loop in this case. This is because, by employing suitable masks, the computation itself can be optimized in a way that allows us to just perform four adds and accumulate the results. While this optimization holds true for this particular example, the decision to use micro-loops and other optimizations purely

```
PADDB(xmm t src, xmm t dst){
                                                          .DEF M1 0x00FF00FF00FF00FF
                                                           .DEF M2 0xFF00FF00FF00FF00
                                                          ;LOW 64-bits , MASK M1
andi t1, src_l, $M1 ;Mask 1st Op
                                                          andi t2, dst_l, $M1 ;Mask 2nd Op
add t3, t1, t2 ;Add Masked Op
                                                          andi t3, t3, $M1 ;Mask output
                                                          ;HIGH 64-bits, MASK M1
                                                          andi t1, src_h, $M1
                                                          andi t2, dst h, $M1
                                                          add t4, t1, t2
                                                          andi t4, t4, $M1
PADDB(xmm_t src, xmm_t dst) {
                                                          ;LOW 64-bits , MASK M2
andi t1, src_l, $M2
     xmm_t masked_src, masked_dst, temp_res;
     xmm_t Mask = 0x0000000000000f;
                                                          andi t2, dst 1, $M2
                                                          add dst_1, t1, t2
     for(int8 t i = 0; i < 16; i++){</pre>
                                                          andi dst_l, dst_l, $M2
          masked_src = src & Mask;
                                                          or dst_l, dst_l, t3
          masked_dst = dst & Mask;
                                                          ;HIGH 64-bits, MASK M2
          temp_res = masked_src + masked_dst;
                                                          andi t1, src_h, $M2
          temp_res = temp_res & Mask;
                                                          andi t2, dst h, $M2
                      = dst | temp_res;
          dst
                                                          add dst_h, t1, t2
          Mask
                       = Mask << 8;
                                                          andi dst_h, dst_h, $M2
     }
                                                          or dst_h, dst_h, t4
                                                      }
}
```

(a) Pseudocode

(b) Optimized micro-ops

Figure 2.6. Devectorization of add byte instruction

depends upon the nature and purpose of the custom translation.

Power Modeling and Power Gating Overheads For powering a unit on and off, power gating uses a header transistor that connects or disconnects the power source of the unit. A sleep signal is applied to the gate of the header device to control its operation. Switching the unit off and on comes at the cost of asserting and deasserting the sleep signal plus switching on and off the header device. These costs are responsible for the non negligible timing and energy overhead of power gating. We use Equation 2.1 from a model proposed by Hu et. al. [114] to account for the energy overhead of power gating.

$$E_{Overhead} \approx 2W_H \frac{E_{cyc}^s}{\alpha}$$
 (2.1)

Where W_H is the ratio of the area of the sleep transistor to the area of the unit and E_{cyc}^s/α is the switching energy of the unit for one cycle when switching factor $\alpha = 1$. We use a conservative value of 0.20 for W_H , as the literature uses an estimated range of 0.05 to 0.20 [114, 133, 157, 170], and for E_{cyc}^s/α we use McPAT estimates [162]. Power gating cycles should be made long enough to compensate for the $E_{Overhead}$. The

Table 2.1. Architecture detail for the baseline x86 core

Baseline Processor				
Frequency	3.3 GHz	I cache	32 KB, 8 way	
Fetch width	4 fused uops	D cache	32 KB, 8 way	
Issue width	6 unfused uops	ROB size	168 entries	
INT/FP Regfile	160/144 regs	IQ	54 entries	
LQ/SQ size	64/36 entries	Functional	Int ALU(6), Mult(1),	
Branch Predictor	LTAGE	Units	FP ALU/Mult(2), SIMD(2)	

break-even time is defined as the number of cycles a unit should stay in power-gated state so that the aggregate energy savings of power gating (E_{saved}) matches the energy of switching the unit on then off ($E_{Overhead}$). We model the leakage current of the header transistor itself, using McPAT. We use Laurenzano et. al.'s [157] estimate of 30 cycles for powering on the VPU.

2.5 Methodology

This section details the evaluation methods we use for our two case study applications of context-sensitive decoding. Since most of the complexity of both is hidden by the decoder, most of the infrastructure, including simulation, is shared between the two. However, because the techniques have very different goals, we do evaluate them in different ways.

2.5.1 Performance Evaluation

We model the x86 pipeline using the gem5 [28] architectural simulator, which already features micro-op translation. We further extend the gem5 front-end to include a micro-op cache and support micro-op fusion as described in the Intel Architectures Optimization Reference Manual [125]. Our baseline processor is based on the Intel Sandybridge microarchitecture [124], adapted to the instruction queue model of gem5. Table 2.1 lists the capabilities of our baseline processor in more detail.

To evaluate the performance of the stealth-mode translation technique we



Figure 2.7. Effect of the cache attacks on AES and RSA with stealth-mode translation enabled.

describe in this chapter, we need an application that exhibits a cache-based sidechannel vulnerability – due to the context-sensitive nature of this technique, we seldom deviate from native performance when not in use and incur no overhead on unaffected code. Therefore, we evaluate the performance of our technique on two commercial implementations of cryptographic algorithms: OpenSSL's AES and GnuPG's RSA. We also include two additional security benchmarks from the MiBench suite [101]: Blowfish and Rijndael cryptographic algorithms that are vulnerable to data cache based side channel attacks. We do not use PGP's IDEA algorithm and the SHA algorithm available in MiBench because we could not find any key-dependent loads or branches that would require our stealth-mode translation mechanism. Each of our included security applications can be run in two modes (encrypt and decrypt) that each perform different computations and therefore exhibit different performance characteristics, giving us 8 performance datapoints.

We use DIFT as a trigger mechanism that detects key-dependent loads and further enables stealth-mode translation. While DIFT can be implemented via CSD, we want to separate the micro-op expansion and related effects of stealth-mode (the primary topic of this chapter) by leveraging a lightweight hardware implementation [245] which incurs an extra 4-cycle L2-tag access latency. Finally, to enable our antivirusdriven trigger mechanism, we set aside five scratchpad registers in the context-sensitive decoder that each contain the addresses of potentially tainted instructions.

To evaluate the selective devectorization mode, we use a wide range of high and

low ILP applications from the SPEC CPU2006 suite. We model power using McPAT with a 32nm technology node [162]. Finally, we use SimPoint [235] and Pinplay [211] to select simulation regions.

2.5.2 Security Evaluation

To evaluate the effectiveness of our security defense, we subject AES and RSA running on our architecture to the FLUSH+RELOAD variant of cache-based sidechannel attacks, modeled after the AES T-Table attacks by Gruss, et. al. [98]. As further demonstration, we try the PRIME+PROBE attack on AES and RSA. Our attack models exploit the I-cache side-channel for RSA and the D-cache side-channel for AES, demonstrating defense against both side channels. Since we model our stealth-mode translation on a cycle-accurate simulator, we allow our attack models to benefit from precise counters and therefore do not require a calibration phase to set thresholds that distinguish between a hit and a miss, and subsequently determine probe intervals.

2.6 Results

In this section, we evaluate each of our instantiations of CSD, starting first with our side-channel defense mechanism and following that with selective devectorization.

2.6.1 Stealth Mode

Security Evaluation. Figure 2.7a shows the results of a well-known PRIME+PROBE attack on AES [263]. The attack repeatedly triggers encryptions with carefully chosen plaintexts while probing 16 different addresses of the AES T-tables. For each probe, only one plaintext has a 100% hit rate (steep dips in the curve), revealing 4 bits of the key. When the stealth-mode translation is not enabled, 64 bits out of the 128 bits of the key get compromised in a matter of 64000 attempts as shown in the figure. These bits are sufficient enough to reverse-engineer the rest of key by well-established



Figure 2.8. The execution time impact of context-sensitive decoding when implementing secure cache obfuscation normalized to insecure execution mode. The watchdog timer is set so that secure mode is re-entered every 500 microseconds.

cryptanalysis techniques. However, when stealth-mode translation is enabled, the attacker-perceived data access patterns are completely obfuscated and always result in a hit for every probe, almost mimicking the behavior of a constant-time defense [23].

Figure 2.7b shows the results of a FLUSH+RELOAD attack on the RSA algorithm [304]. In the absence of stealth-mode translation, the attacker can almost always detect when a *multiply* function has been invoked by measuring hits and misses (shown as dips and spikes in the figure) to the corresponding *reloaded* cache line. However, when stealth-mode translation is in effect, the attacker-perceived instruction access pattern is completely obfuscated resulting in a perceived I-cache hit at the end of every probe interval. The PRIME+PROBE attack on RSA (not shown) is also defeated, recording a miss on the attacker end after every probe interval.

Performance The potential performance overheads of CSD include micro-op expansion and related side effects (micro-op queue pressure, micro-op cache pressure) and possible cache effects due to increased cache pressure from decoy loads. Careful construction of the secure-mode micro-op translation allow us to minimize many of the side effects of micro-op expansion.

Figure 2.8 compares the execution time of our pipeline without any optimization (*NoOpt*) and with both micro-op cache and micro-op fusion enabled (*Opt*). In these results, we see performance loss consistently below 10% and averaging 5.6% when secure mode is enabled. This is to be compared with the current state of the art obfuscation techniques, which rely on the compiler (and consequently don't enjoy a hardware DIFT), that see performance expansion on the order of 20X [217] for applications with large memory footprints.

To break down the performance overhead of the context-sensitive decoding, we first study micro-op expansion relative to unaltered execution. As shown in figure 2.9, context-sensitive decoding causes a micro-op expansion of 8.0% on average. Comparing these results together with Figure 2.8 seems to indicate that the primary cost of context-sensitive decoding is in fact the micro-op expansion. This is somewhat surprising, because we expect additional overheads from the higher incidence of loads and greater memory activity.

We investigate this further with several experiments. First, we measure performance in cycles/micro-op, and find that in fact this figure does not increase (and in some cases decreases), despite the fact that the percentage of load micro-ops has increased. Second, we see in Figure 2.10 that the number of cache misses per kilo instruction (MPKI) stays about the same on average. This indicates the vast majority of additional injected loads are hits. In yet another experiment where we discounted the cost of micro-op expansion, we saw an overall performance increase on average – this was due to a prefetching effect from the added micro-ops. Thus, the cache prefetching effect of the decoy loads is actually muting some of the performance cost of micro-op expansion.

Another negative side effect of context-sensitive decoding is on the micro-op cache. Because we introduce translations not allowed in the micro-op cache, or in other cases expand loops so they no longer fit, we do lose some of the effectiveness of that



Figure 2.9. Micro-op expansion due to CSD.



Figure 2.10. Number of cache misses without and with CSD.



Figure 2.11. Effect of CSD timer on execution time.

cache. However, that effect is small, especially when modeled with micro-op fusion enabled. Without fusion, the micro-op cache hit rate, on average, drops from 44% to 39% when we introduce CSD, but in the presence of micro-op fusion (which shortens some of the code sequences we expanded), it is much more stable dropping only from 43% to 42% with CSD-based stealth mode enabled.

In all above experiments, the watchdog timer is set to 1000 cycles (500 microseconds), so the decoy loads are deployed at the first decoded tainted load or branch



Figure 2.12. Total energy consumption of CSD's devectorizing mode normalized to that of power-gating

encountered, then decoding returns to normal mode until the timer fires again. While re-injecting decoy micro-ops every 1000 cycles provides almost perfect obfuscation against these cache-based side-channel attacks, based on system characteristic (e.g., cache miss/hit delays) and targeted attacks one can tune this parameter to increase the performance of the defense. Figure 2.11 shows the normalized execution time of our defense, sweeping the watchdog timer from 1000 to 10000 cycles. The decrease in execution time is caused by fewer extra micro ops and fewer micro-op cache conflicts.

Overall, we find that obfuscation of secure-data dependent microarchitecture state can be enabled with context-sensitive decoding with almost no performance cost, particularly in comparison with prior techniques.

2.6.2 Selective Devectorization

Figure 2.12 shows the breakdown of energy for regular decoding with (1) conventional power gating and (2) our devectorizing mode using context sensitive decoding. Energy numbers are normalized to the total energy of conventional power gating. On average, dynamic devectorization results in a 12.9% improvement of total energy consumption. This is despite the fact that several of our SPEC benchmarks make little use of vectorization.



Figure 2.13. Execution time for different power gating policies, normalized to always on policy

Figure 2.13 compares the execution time of three different VPU power-gating polices: (1) Always execute on the vector units (*Always On*), (2) Context Sensitive Decoding which scalarizes the instructions based on the unit criticality predictor's decisions (*CSD*), (3) Conventional power gating (*Power Gating*), and (4) Always translate instructions to scalar micro-ops to execute on scalar units (*Always Off*). The always-devectorize policy and power gating both incur considerable performance overheads (12% and 5%, on average respectively) while context-sensitive decoding reduces this to only 1.6% overhead. In general, it keeps performance close to full vectorization even though the vector units are turned off much of the time. The only exception is *namd*, where we are much faster than full devectorization and conventional power gating, but still incur a high cost.

Figure 2.14 shows the number of dynamic micro-ops for the three VPU powergating polices. Here we see that performance scales with micro-op expansion, the primary performance cost of CSD dynamic devectorization.

Figure 2.15 depicts the percentage of time that the VPU is kept power-gated for each benchmark. On average, context sensitive decoding can keep the VPU power-gated more than 70% of the execution time. For benchmarks *astar*, *gcc*, *gobmk*, and *sjeng* with low (but not nonexistent) vector activity, we are able to keep the vector unit



Figure 2.14. Micro-op expansion due to context sensitive decoding normalized to native mode

turned off just about all the time, not having to turn it on for occasional outliers.

Figure 2.16 shows the breakdown of the SSE instructions in each benchmark. We categorize instructions into three categories: 1) instructions that are executed on the VPU (*Powered On*), 2) instructions that are devectorized and executed on scalar units because the VPU was in the process of powering on (*Powering On*), and 3) instructions that are executed on scalar units because the VPU was in power-gated state (*Power-Gated*). We find that *bwaves* and *milc* are frequently forced to execute scalar instructions while waiting for the vector unit to power on. They are still able to slightly come out ahead in energy, though, due to the performance advantage of not having to stall to wait for the VPU to turn on. *Namd* executes the largest number of vector instructions in gated mode, and is in fact gated 20% of the time despite having a large amount of vector activity. This implies that the threshold that performed overall was too aggressive for *namd*, and a more dynamic threshold or usage predictor would work better. Omnetpp, on the other hand, has a reasonable number of scalar operations but executes nearly all of them with the vector unit disabled, resulting in a significant gain in energy. And *gamess* is able to judiciously enable power gating, as it is gated nearly half the time, yet only about 20% of vector instructions are affected.

Overall, for CSD-enabled selective devectorization, we find that we are able



Figure 2.15. Percentage of time that CSD power gates VPUs.



Figure 2.16. Breakdown of vector activity

to power gate the vector unit for longer, unbroken periods, resulting in good energy savings with a small performance cost.

2.7 Conclusion

The paper presents context-sensitive decoding, which enables the decoder to dynamically alter the decoding of programmer-visible ISA instructions. This allows the system to change the functionality of the software without programmer or compiler intervention. We use the technique for stealth mode translation, where the decoder injects instructions which completely obfuscate the effect of secure-data dependent branches and data accesses, defending against multiple variants of cache-based side channel attacks at just 5% performance degradation. This removes the microarchitectural footprint of the secure code from an attacker. Additionally we enable selective devectorization via CSD, saving 12.9% in energy while simultaneously achieving a speedup of 3.4% over conventional power gating.

Acknowledgements

We thank the anonymous reviewers for their helpful insights. Chapter 2, in full, is a reprint of the material as it appears in International Symposium on Computer Architecture (ISCA) 2018. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 3 Context-Sensitive Fencing

Maximizing performance has been a major driving force in the economics of the microprocessor industry. Modern processor architectures feature highly complex and sophisticated performance optimizations. However, scaling performance without considering security implications could have serious negative consequences, as evidenced by the recent pile of lawsuits [295] concerning the Meltdown [165] and Spectre [148] class of microarchitectural attacks. These events have highlighted the need to architect systems that can not only run at high speed, but can also exhibit high resilience against security attacks, not just one or the other. The goal of this work is to secure a particular performance optimization integral to modern processor architectures – speculative execution – against the Spectre class of microarchitectural attacks, while maintaining acceptably high levels of performance.

Modern processors employ branch speculation and out-of-order execution to take advantage of the available instruction-level parallelism beyond control-flow boundaries, thereby improving the overall CPU resource utilization and sustaining high throughput. Spectre attacks exploit speculative execution by leaking secret information along misspeculated paths via cache-based and other timing side channels. Owing to their ability to mistrain the branch predictor to deliberately steer execution to an attacker-intended control-flow path [132, 149], these attacks have notably demonstrated the potential to break all confidentiality and completely bypass important hardware/software security mechanisms such as ASLR, even via remotely accessed covert channels [231].

Mitigating Spectre is a particularly hard problem since it could potentially cause highly intrusive changes to the existing out-of-order processor design, severely limiting performance. Although Intel has announced microcode update patches to mitigate certain variants of the attack, a majority of the high impact vulnerabilities still largely rely on software patching [122, 209]. State-of-the-art software countermeasures take advantage of *fences* that mute specific effects of speculative execution by constraining the order of certain memory operations, or in some cases by completely serializing a portion of the dynamic instruction stream. Liberal fence insertion (e.g., at every bounds check) can mitigate the attacks, but doing so severely hurts performance; however, spraying fences more strategically at appropriate locations in the code requires extensive patching of software via recompilation or binary translation – resulting in significant engineering effort and long delays to deployment. We need hardware architectures that can more seamlessly react to such attacks via unobtrusive field updates.

This work proposes *context-sensitive fencing* (CSF), a novel microcode-level defense against Spectre. The key components of the defense strategy include: (a) a *microcode customization* mechanism that allows processors to surgically insert fences into the dynamic instruction stream to mitigate undesirable side-effects of speculative execution, (b) a *decoder-level information flow tracking* (DLIFT) framework that identifies potentially unsafe execution patterns to trigger microcode customization, and (c) *mistraining mitigations* that secure the branch predictor and the return address stack.

To perform secure microcode customization with minimal impact on performance, this work leverages context-sensitive decoding (CSD) [253], a recently proposed extension to Intel's (and others') micro-op translation mechanism that enables ondemand and context-sensitive customization of the dynamic micro-op instruction stream. In addition, this work also takes advantage of the reconfiguration framework offered by CSD, allowing the operating system and other trusted entities to dynamically control the frequency, type, and behavior of fences that are surgically inserted into the micro-op stream to secure speculative execution.

This work analyzes a significantly expanded suite of fences, considering different possible enforcement stages and enforcement strategies. In particular, we introduce a new fence that prevents speculative updates to cache state with minimal interference in the dynamic scheduling of instructions.

Context-sensitive fencing has the ability to automatically identify fence insertion points via a novel decoder-level information flow tracking-based detection mechanism. Since the processor front-end typically churns instructions at a much higher rate than the rest of the pipeline, information-flow tracking at the decoder-level is prone to frequent overtainting and undertainting scenarios. While overtainting could hurt performance due to the increased frequency of fence insertion, undertainting could undermine the security of the system for a brief window. By solving these challenges through an early and low-overhead mistaint detection and recovery mechanism, this chapter further establishes the viability of decoder-level information flow tracking as an effective attack detection mechanism.

This work further proposes novel micro-op flows that protect the branch predictor, the branch target buffer, and the return address stack against mistraining across different protection domains. While similar in spirit to the proposed Indirect Branch Predictor Barrier (IBPB) instruction by Intel, these micro-op flows apply more generally to a broader class of branch predictors and return address stacks, and offer more fine-grained control.

This research makes the following major contributions:

- It introduces context-sensitive fencing, a mechanism that leverages a dynamic decoding architecture to inject fences between control flow and loads, without recompilation or binary translation.
- It examines several variants of existing fences, and introduces three new fences, which allow aggressive reordering of loads and stores without exposing any microarchitectural evidence, in the cache, of speculative accesses that cross the fence.
- It introduces a decoder-level dynamic information flow tracker, DLIFT, which allows accurate tracking of taints early in the pipeline, giving the pipeline the ability to identify tainted accesses *before* the stages that enable speculative execution.
- It introduces a simple dynamic mechanism that eliminates redundant (per basic block) fences.
- The combination of optimizations introduced in this chapter reduce the cost of a fence-based Spectre mitigation technique from 48% overhead (for a conservative scheme) to less than 8%.
- It also introduces a decode-level branch predictor isolation technique that mitigates branch mis-training variants of Spectre.

3.1 Background and Related Work

Speculative Execution. Dynamic control speculation is a well-known instruction throughput optimization technique used, especially in out-of-order processors, to predict the branch outcome and execute instructions along the predicted path, while waiting for the actual branch outcome to be evaluated at a later pipeline stage. In the event of a misprediction, the processor rolls back execution along the misspeculated path and redirects control to the right branch target. Although speculative execution is largely programmer-invisible in terms of committed architectural register and memory state, in all known instances it leaves some microarchitectural side effects that can be observed through well-established side channels.

Microarchitectural Attacks. Microarchitectural attacks leak secret information of a victim process by observing microarchitectural effects of certain performance/power optimizations such as caches, branch speculation, memory disambiguation, and even dynamic voltage and frequency scaling (DVFS), through sidechannels such as memory bus activity [11], power consumption characteristics [27], branch access patterns [9,79], faults [108,138], acoustics [87,88], electromagnetic effects [85], functional unit timing characteristics [285], and most notably cache access patterns [10,86,167,195,297,304,306].

Cache-based side-channel attacks have been shown to reveal secret information such as cryptographic keys [70, 98, 304], keystrokes [99], and browsing activity [205] by co-locating a spy process alongside a victim in such a way that they share cache memory. The attack unfolds through a pre-attack step in which a spy process fills/flushes specific cache sets, so that the victim leaves observable side effects in terms of its cache access patterns, that can be later inferred by the spy process by timing the access to particular cache blocks.

Exploiting Speculative Execution. This work tackles a highly evasive class of microarchitectural attacks called Spectre [148] that leaks information by exploiting side effects of speculative execution through cache-based side channels. These attacks not only exploit unintended side effects due to speculation, but have the ability to deliberately mislead execution into attacker-intended paths by mistraining the branch predictor and other associated structures. Several variants of the attack have been described (shown in Table 3.1) that can potentially bypass software security/integrity protection mechanisms such as bounds checking and ASLR [233].

Variant	Vulnerability Name
Spectre v1 [132, 148]	Bounds Check Bypass (BCB)
Spectre v2 [132, 148]	Branch Target Injection (BTI)
Spectre v3 [132, 165]	Rogue Data Cache Load (RDCL)
Spectre v3a [19, 182]	Rogue System Register Read (RSRD)
Spectre v4 [182]	Speculative Store Bypass (SSB)
Spectre-NG [243]	Lazy FP State Restore
Spectre v1.1 [146]	Bounds Check Bypass Store (BCBS)
Spectre v1.2 [146]	Read-only Protection Bypass
Spectre v5 [149, 177]	Ret2Spec and SpecRSB
NetSpectre [231]	Remote Bounds Check Bypass
Foreshadow [269,290]	L1 Terminal Fault

Table 3.1. Speculative Attacks Variants

Figure 3.1. Example Spectre Variant-1 Gadget

Figure 3.1 shows a vulnerable code target for the variant-1 attack. The code fragment is composed of a conditional branch that performs a bounds check, and a *Spectre gadget* that results in an observable microarchitectural side effect upon execution. Upon misspeculation, the bounds check is bypassed and the Spectre gadget executes, leaving an observable cache footprint to the attacker, that remains even after the processor detects the misprediction and rolls back execution. In the simplest variant of the attack where the attacker controls both x and *array2*, the attacker can potentially leak the entire address space of the victim.

The variant-2 Spectre attack further allows the attacker to hijack speculative execution by mistraining the branch predictor and associated structures, enabling a ROP-style attack [223] that stitches together Spectre gadgets. To mount such an attack, a co-located adversary process that shares the branch predictor (e.g., a browser process with several user threads) with the victim, first forces an artificial BTB (Branch Target Buffer) entry collision using a carefully chosen indirect branch address. The attacker

1 mov eax, arr1_size	mov eax , arr1_size
² cmp edi, eax	cmp edi,eax
3 jge END_LBL	jge END_LBL
4 mov eax, edi	mov eax,edi
	FENCE
6 mov eax, [eax+arr1]	<pre>mov eax , [eax+arr1]</pre>
7 shl eax, 0x8	shl eax, ox8
8 mov eax, [eax+arr2]	<pre>mov eax ,[eax+arr2]</pre>
mov [v], eax	mov [y],eax
• END_LBL:	END_LBL:
(a) Vulnerable gadget (x86)	(b) Fenced gadget (x86)

Ī.

Figure 3.2. Mitigating Spectre-v1 using a Fence Instruction

next poisons the value of the colliding BTB entry by repeatedly executing its own branch whose target is the address of a Spectre gadget. After successful mistraining of the branch predictor, the processor now predicts the victim's indirect branch that collides in the BTB to be the attacker-chosen Spectre gadget and starts speculatively executing it.

The variant-3 attack (a.k.a. Meltdown) exploits the fact that most out-of-order processors that employ dynamic speculation supress loads with protection violation at instruction retirement (i.e., commit stage), rather than during instruction execution. Therefore, a non-privileged memory access can find its way to the cache and leave a footprint, allowing an attacker to now effectively read arbitrary memory contents of another process or even the kernel or the hypervisor, through careful cache-based side channel analysis.

The literature describes multiple attacks that conform to the above variants, but exploit different attack targets and/or different side channels. These include SgxPectre [42] that bypasses Intel's SGX [121] security mechanisms to steal secrets from SGX enclaves, the MeltdownPrime/SpectrePrime [262] that leverage a PRIME+PROBE [167] cache attack instead of FLUSH+RELOAD [304] by exploiting the side effects of cache line invalidation mechanisms in modern cache coherence protocols, and the NetSpectre attack that leaks information across independent virtual machines on Google Cloud via an AVX-based covert channel [231].

The variant-4 Spectre attack exploits microarchitectural side effects of the memory disambiguation feature employed by most out-of-order processors in order to allow loads to be speculatively ordered and executed before any outstanding store whose effective address has not been calculated yet. Upon misspeculation, i.e, if the load address has a conflict with the outstanding store, the processor flushes the pipeline and triggers the re-execution of the load and all subsequent instructions [126], while the microarchitectural side effects of the misspeculation linger, resulting in memory disclosures similar to variant 1 and 3. Similarly, the NG variant-3 attack [243] leverages the lazy x87 floating-point restore functionality of Intel CPUs to read floating-point registers of a victim process.

The Spectre variants 1.1 and 1.2, dubbed Speculative Buffer Overflows [146], exploit the store-to-load forwarding optimization to stitch together Spectre gadgets. More specifically, these attacks bypass a stack buffer overflow check similar to variant-1 and execute a Spectre gadget that speculatively stores malicious content (the address of the next Spectre gadget) into the return address on the stack. Store-to-load forwarding combined with the fact that most modern processors break down the *return* instruction into micro-ops that *load* the return address from the stack before transferring control, then result in a ROP-style execution of Spectre gadgets that leave a trail of microarchitectural state behind. The variant-5 attack [149] achieves similar effect, by instead mistraining the return address stack employed by most processors to speculatively predict the target of a procedure return.

Spectre Mitigations. The current set of mitigations for Spectre range from simple coding guidelines [39] to proposals that advocate exposing microarchitectural details in the ISA [172]. To mitigate Meltdown, Kernel Page Table Isolation

(KPTI) [52,97] has been proposed and recently patched to the Linux kernel, incurring about 6% in performance. To mitigate Spectre v1, multiple chip manufacturers including Intel [122], ARM [19], and AMD [15], have suggested instrumenting code with serializing instructions or fences to inhibit speculation at specific points in execution.

For example, consider Figure 3.2a that shows the assembly code for the Spectre variant-1 gadget in Figure 3.1. Figure 3.2b shows a software-patched version that employs a serializing instruction or a fence to prevent the speculative execution of the Spectre gadget (i.e., lines 6-9). In most implementations, upon decoding the fence, the processor stops fetching new instructions until the fence gets committed or squashed, thereby serializing execution. In our example, if the attacker calls the vulnerable gadget with out-of-bounds values in the *edi* register, the processor front-end stalls until the fence is committed, thereby disallowing the speculative execution of the Spectre gadget and completely mitigating the attack.

The associated performance overhead with liberal fence insertion, however, could be as high as 10x [203]. While it is possible to perform compiler-directed code instrumentation with fences at a lower performance overhead [209], locating the potential Spectre targets using static analysis is non-trivial and could therefore result in less than full coverage [147]. Speculative Load Hardening [40] and YSNB (You Shall Not Bypass) [203] propose to use predicated execution [136, 176, 210] to alleviate the high overheads for fences by injecting an artificial data dependency between the conditional branch and the Spectre gadget. These proposals incur 36-60% overhead in performance.

To cope with variant-2, both Intel and AMD have announced microcode update patches that introduce new fence instructions [15, 122] that prevent instructions preceding the fence from controlling the indirect branch prediction of branches that follow the fence. The fences also prevent software with lower privileges from influencing the indirect branch prediction of software with higher privileges. Furthermore, these patches restrict indirect branch prediction from being controlled by co-located threads via simultaneous multithreading [265, 266]. Furthermore, the use of *retpolines* [122, 123, 267] have been advocated to replace indirect jumps and calls with an equivalent *push+ret* instruction sequence, in order to bypass the indirect branch predictor. However, they could further expand the attack surface in the wake of v5 [149, 177], v1.1, and v1.2 attacks [146].

On the hardware front, SafeSpec [142] and InvisiSpec [302] propose mitigating side effects of speculative execution by adding new shadow user-invisible structures for caches and TLBs that store transient results from speculative instructions, and committing them to main cache/TLB only if the speculation was deemed correct and the corresponding instructions gracefully retire. Although these techniques make disruptive changes to the processor/memory architecture and consistency models, they make significant strides in secure hardware design with minimal performance impact. Finally, Dong, et al. [71,72] propose leveraging Virtual Ghost [59] to protect applications running on a compromised OS kernel from Spectre.

Secure Instruction Stream Customization. Prior work has established that instruction customization is an effective means to instrument the dynamic execution stream with security checks, and thereby mitigate several attack vectors at a relatively low performance overhead. Instruction stream customization has been proposed and evaluated in various forms and levels, ranging from secure virtual architectures at the ISA and compiler level [60, 272, 274] to full-blown processor binary translation at the microcode level [29, 64, 208, 273]. Corliss, et al. [53–55] propose dynamic instruction stream editing (DISE), a macro-engine that customizes the dynamic instruction stream at the decoder level, by pattern-matching user-defined production rules pushed into the decoder. Taram, et al. [253] propose *context-sensitive decoding* that leverages the CISC to RISC translation feature of modern instruction set decoders to dynamically alter the behavior of programmer-visible instructions without recompilation or binary

translation [69, 275]. While we leverage these approaches in this research, this work targets a different class of emerging attacks, and proposes several additional techniques beyond microcode customization.

Information Flow Tracking. Suh, et al [245] first proposed the idea of dynamic information flow tracking (DIFT) that tags data from untrusted channels as spurious, and further tracks the information flow of spurious data, flagging any violation of an enforced security policy. Multiple hardware information flow tracking techniques have been described at various levels of the hardware [44, 45, 48, 56, 62, 260, 268, 276] as a detection mechanism to circumvent code injection [299], cross-site scripting attacks [62, 279], buffer overflows [62, 299], and SQL injection [62, 102, 103, 154, 308]. This work proposes decoder-level information-flow tracking as a detection mechanism for Spectre, and addresses several associated challenges.

3.2 Assumptions and Threat Model

Among the Spectre variants, the Bounds Check Bypass (variant-1) is the one which has had the highest impact in terms of affected platforms and devices. Therefore, the main focus of this work is to mitigate the variant-1 attack at an acceptable level of performance. However, owing to the flexibility advantages of microcode customization, we also propose new mitigations against other Spectre variants documented in Table 3.1.

We assume that the goal of the attacker is to read arbitrary memory contents by exploiting a Spectre gadget. Also, without loss of generality, in this work we defend against a Spectre attacker that targets the most obvious and the most important victim, i.e., the OS kernel, but our framework allows us to extend the proposed techniques to mitigate different targets such as virtual machines and browsers. Our attack model assumes an adversary with the following capabilities.

89

- **Information Leakage.** We assume that the adversary has the ability and privilege to probe, flush, or evict any cache line including that of the kernel at any particular time. Also, they can make precise timing measurements using instructions such as rdtsc and rdtscp.
- **Co-location.** We assume that the adversary is colocated with the victim, and can not only leak information through cache-based side channels, but can also mistrain shared branch predictor structures including branch target buffers and return address stacks, and can further influence the branch outcome of the victim, as described in the variant-2 attack.
- User-Mode Access. We assume that the adversary has standard user-mode access, and can invoke any system call with arbitrary and carefully chosen arguments, and can further access devices such as the keyboard or network to feed the kernel/driver code with carefully chosen data.

Moreover, we assume that the only communication channel between the attacker and the kernel code is through micro-architectural side channels and the attacker does not have any other channel, direct or indirect, to leak data. Among all the microarchitectural side-channels, data caches are the ones that are predominantly used and are the easiest to exploit. Thus, the mitigation strategies described in this work primarily focus on those variants of the attack that use the data cache as their covert channel to leak information. Again, due to the flexiblity and programmability of the proposed strategy, it is possible to easily extend the approach to mitigate other side-channels such as functional unit contention and AVX timing channels [231].

3.3 Architectural Overview

Figure 3.3 gives the architectural overview of our defense strategy – *context-sensitive fencing*. The central piece of the proposed architecture is an x86 microcode


Figure 3.3. Architectural Overview

engine that has context-sensitive decoding (CSD) capabilities [253], allowing it to optionally translate a native x86 instruction into a customizable, alternate set of microops. In this work, we leverage this capability to perform the surgical insertion of speculation fences (some existing and some newly proposed) at potentially vulnerable Spectre code targets. To this end, we introduce new custom micro-op flows and new configuration mechanisms that trigger such micro-op flows.

The CSD-enabled microcode engine is provisioned with fine-grained reconfiguration capabilities via a set of model-specific registers (MSRs) that can control the frequency, type, and enforcement criteria of speculation fences inserted into the dynamic instruction stream. Such a fine-grained reconfiguration capability is especially important to this work because speculation fences are particularly expensive, allowing us to surgically insert fences that impose varying degrees of restrictions on speculative execution, depending upon the runtime conditions, current level of threat, and the nature of the code being executed.

Moreover, the context-sensitive fencing framework also benefits from a novel decoder-level information flow tracking (DLIFT) engine that has the ability to identify untrusted instructions that are potentially in Spectre gadgets and trigger alternate micro-op flows that insert speculation fences. Owing to its decoder-level and inher-

Туре	Instr.	Desc.
su	INVD	Invalidate Internal Caches
tio	INVEPT	Invalidate Translations from EPT
ruc	INVLPG	Invalidate TLB Entries
ust	INVVPID	Invalidate Translations Based on VPID
19	LIDT	Load Interrupt Descriptor Table Register
ji.	LGDT	Load Global Descriptor Table Register
ziliz	LLDT	Load Local Descriptor Table Register
erië	LTR	Load Task Register
s	MOV	Move to Control Register
Sec	MOV	Move to Debug Register
ileg	WBINVD	Write Back and Invalidate Cache
riv.	WRMSR	Write to Model Specific Register
ivF	CPUID	CPU Identification
Ъ	IRET	Interrupt Return
Von	RSM	Resume from System Management Mode
ing	SFENCE	Store Fence
Drder	LFENCE	Load Fence
em. C	MFENCE	Memory Fence

Table 3.2. List of Intel's Serializing Instructions

ently speculative implementation, DLIFT relies on a mistaint detection and recovery hardware implemented in the execute stage, to avoid overtainting and undertainting scenarios. Finally, the proposed defense framework also includes hardware and microcode-level mechanisms to achieve branch predictor state isolation across protection domains to mitigate the variant-2 and variant-4 attacks.

3.4 Design and Implementation

In this section, we describe in greater detail the architectural techniques and building blocks that together constitute the proposed defense strategy – *context-sensitive fencing*. First, we describe a microcode customization framework that enables the surgical insertion of fences to secure speculative execution with minimal performance impact. Second, we examine the full design space of fences to provide the isolation properties we need without undue obstruction of existing pipeline performance features. Third, we propose a decoder-level information flow tracking (DLIFT) technique to follow potentially malicious execution patterns and trigger secure microcode customization. Finally, we propose protection mechanisms that circumvent the mistraining of the branch predictor and associated structures.



Figure 3.4. Fence Enforcement Points

3.4.1 Microcode Customization

Speculation fences are a processor's primary mechanism to override speculative execution. For Spectre variant-1, context-sensitive fencing works with CSD by providing alternate decodings of all load instructions, with the alternate decoding always including a *fence* micro-op that appears before the load micro-op. The alternate decoding will then be triggered or not based on runtime conditions. The first consideration, then, is what fence instruction to incorporate. Most processors already provide a variety of fences and serializing instructions. We first study the performance and security impact of the existing suite of fences and serializing instructions, and then explore and propose new speculation fences and context-sensitive fencing techniques that manage the impact on performance. For Spectre variant-2, CSF works with CSD by providing alternate decodings of some control flow, and possibly insert fences and/or branch predictor resetting micro-ops – more detail is in Section 3.4.3.

Serializing Instructions and Memory Fences.

Serializing instructions are the strictest amongst all speculation fences and completely override speculative execution. Upon decoding a serializing instruction, the processor stalls fetching any subsequent instruction until all instructions preceding the serializing instruction retire. Due to the high pipeline depth and issue width of modern out-of-order superscalars, the usage of serializing instructions could result in long delays and considerable throughput loss. Memory fences, on the other hand, enforce a memory serialization point in the program instruction stream. More specifically, these fences ensure that memory operations that appear in execution after the fence are stalled until all the outstanding memory requests (including the fence) complete execution.

Table 3.2 shows a complete list of Intel's existing serializing instructions and fences [126]. Most of the serializing instructions need to be run in privileged mode which restricts their usage in defenses for victims that lack sufficient privileges (e.g., browsers). Moreover, a majority of them modify the state or contents of architectural registers, the program counter, cache, TLB, etc. and only serialize execution as a side effect, thereby requiring an additional backup/restore step when used for the sole purpose of serializing execution. An exception to this is the *MOV to debug register* instruction that does not corrupt any architectural state, if not actually in debug mode.

The SFENCE instruction does not allow stores to pass through it, but does not affect loads. That is, it enforces that all stores that precede the SFENCE are executed to completion before any store that succeeds the SFENCE is fetched. The MFENCE instruction restricts all memory operations from passing through it. Unlike SFENCE and MFENCE that are memory ordering operations, LFENCE performs a serializing operation. In particular, LFENCE does not stop the processor from fetching and decoding instructions that appear after the LFENCE, but it restricts the dispatch of such instructions, until the instructions preceding the LFENCE complete execution. Since LFENCE serializes execution and yet makes some progress in the front-end, it has been recommended by Intel as a low-overhead fence that can be inserted at vulnerable points in execution to defend against Spectre [122]. In other words, while serializing instructions such as CPUID are enforced at Fetch, LFENCE is enforced at the instruction queue level. Therefore, in this work, we start with the LFENCE, and propose new fences that come with fewer restrictions, different enforcement policies, and/or sport additional optimizations.

Fence Enforcement Policies

It is important to note that none of the existing instructions that provide fence support were actually created for the purpose we (or the whole industry) need for Spectre mitigation. Thus, this work will examine existing fences, variants of existing fences, and also introduce a new fence primitive. To better understand the design landscape, we examine several possible properties of fences in this section.

Early vs. Late Enforcement. We first categorize fences into *early-enforced* and *late-enforced* fences, based on the pipeline stage at which they are enforced. In particular, we refer to any serializing instruction, such as an LFENCE, that is enforced at the instruction queue or earlier in the pipeline as *early-enforced*. If the fence is enforced at a later stage such as the reservation station, the load/store queue, or the cache controller, we refer to it as a *late-enforced* fence. Late enforcement, in essence, shifts the fence enforcement point towards the leaking structure (e.g., the cache), reducing the impact on instructions that do not access that structure, and allowing for the enforcement of more fine-grained serialization rules (e.g., allow cache hits but not misses). Figure 3.4 shows potential fence enforcement points at various stages in the processor pipeline.

It is important to note that the later a fence is enforced, the fewer the side channels it protects against. For example, Intel's LFENCE prevents information

95

leakage through all microarchitectural structures that appear after the instruction queue. However, to prevent information leakage through the instruction cache side channel, we would have to resort to a regular serializing instruction or an MFENCE, resulting in prohibitively high performance overheads. Similarly, a fence enforced at the data cache controller level would only mitigate data cache-based side channels and will not protect against an FPU-based side-channel. Instead, the FPU-based side-channel may be mitigated using a different fence that is appropriately configured and enforced at the reservation station or the FPU.

Strict vs. Relaxed Enforcement. Depending upon how prohibitive they are, we next classify fences into *strict* and *relaxed*. *Strict* fences are highly prohibitive and do not allow any instructions to pass through them until the fence retires, whereas *relaxed* fences allow certain types of instructions to pass through them. For example, an SFENCE that is enforced at the load/store queue and allows all instructions to pass except stores that have a greater sequence number than itself, is a *late-enforced* and *relaxed* fence. On the other hand, all x86 serializing instructions including LFENCE are *early-enforced* and *strict*. Customizing the micro-op stream with *early-enforced* and *strict* fences typically results in slower execution when compared to customization with *late-enforced* and *relaxed* fences. However, with carefully enforced constraints, the *late-enforced* and *relaxed* fences could offer similar, if not better security guarantees.

Early vs. Late Commit. A fence typically remains effective until it gets committed or squashed. Based on Intel's manual [125], a serializing instruction is only allowed to be committed if there is no preceding outstanding store that is waiting to be written back. While this behavior might be necessary for device synchronization or memory ordering enforcement, for the purpose of securing speculative execution against Spectre attacks, there is no need to wait for stores to be written back. This is because write buffers aren't committed to the cache until the store reaches retirement, and therefore the fact that a store is waiting for an outstanding writeback request

Fence Name	Enforcement Point	Strict/ Relaxed	Instructions not Allowed	Mitigates Variants	Existing/ New?
Intel's SIs (CPUID)	Fetch	Strict	All	All	Existing
LFENCE	IQ	Strict	All	All	Existing
LSQ-LFENCE	LSQ	Relaxed	Ld	V1	New
LSQ-MFENCE	LSQ	Relaxed	Ld&St	V1,V1.1,V1.2	New
CFENCE	CC^*	Relaxed	None	V1	New

Table 3.3. Characteristics of Different Fence Types

* CC: Cache Controller, RS: Reservation Station

to complete is enough evidence that it did not occur along a misspeculated path. Allowing a fence to commit early without waiting for preceding outstanding stores to write back can considerably improve performance because as soon as a fence gets committed, a stream of instructions can advance further in the pipeline. Therefore, in this work, we propose and study the effects of a late-commit version of each fence that does not wait for stores to be written back. However, we note that these versions should only be limited to the security use case we describe and should not be used for synchronization.

Newly Proposed Fences. Table 3.3 summarizes the characteristics of different existing and newly proposed fences. Although the existing set of fences defend against all variants, they incur prohibitively high costs on performance, due to their strict enforcement constraints. To better understand the potential for fences beyond those that already exist, we propose and evaluate three new types of fences. In an attack scenario, we typically insert one of these fences between a branch instruction and a load instruction that potentially leaks sensitive information via a cache side-channel. The proposed fences more effectively and efficiently mitigate the high impact variant-1 attacks, by preventing information leaks along misspeculated paths through cache-based side channels. We describe each of them in greater detail below.

The LSQ-LFENCE and the LSQ-MFENCE are relaxed fences enforced at the

load/store queue. While LSQ-LFENCE fence is in effect, it does not allow any subsequent load instruction to be issued out of the load/store queue, thereby preventing the cache state from being changed by load instructions on misspeculated paths. Thus, the LSQ-LFENCE mitigates the Spectre variant-1 attack. On the other hand, the more restrictive LSQ-MFENCE does not allow any subsequent memory instruction (both loads and stores) to be issued out of the load/store queue, until the fence commits. In addition to mitigating the variant-1 attack that the LSQ-LFENCE protects against, the LSQ-MFENCE mitigates the variant-1.1 and variant-1.2 attacks that exploit store-to-load forwarding between speculative loads and stores.

The CFENCE is a relaxed fence and is enforced at the cache controller level using the following set of rules. First, like any other fence, it allows all preceding instructions to proceed. Second, since store instructions do not commit the contents of the write buffer until the instruction retires, they are unaffected by the CFENCE. Finally, it labels any subsequent load as a *non-modifying load* and allows it to pass through the fence, but the load is restricted from modifying the cache state. In particular, a *non-modifying load* that results in a cache hit is allowed to read the contents of the cache, but is restricted from changing the LRU and other metadata bits. A *non-modifying load* that results in a cache miss is marked as uncacheable, allowing the memory read request to complete without altering the cache state. In this way, we avoid updating the cache state upon encountering a speculative load and don't leave any observable cache footprints along misspeculated paths, thereby mitigating the variant Spectre variant-1 attack. In addition, due to locality of references, the miss rate of a reasonable program is typically very low, and therefore, using CFENCE results in considerably lower performance overhead than other types of fences.

Note that this can be applied recursively at each cache level. For example, an L2 hit (L1 miss) will bypass the L1 cache and not initiate a fill, then read from the L2 cache without altering the LRU bits.

Fence Frequency Optimization

The most naïve yet secure way to insert fences is to liberally instrument every instruction of a vulnerable type (e.g., load instructions in the case of cache sidechannels) in the program and add the fence micro-op to all of them. In fact not every instance of a vulnerable instruction type is necessarily vulnerable; for example, all the loads in the program aren't vulnerable to speculative attacks via cache side-channels. Therefore, we can reduce the number of fences inserted. However, since failing to insert fences, even in one scenario, would enable a Spectre attacker to read the whole victim's memory space, it is of crucial importance that fence frequency optimizations be conducted meticulously. In the following, we introduce two secure optimizations for reducing the number of fences.

Basic Block-Level Fence Insertion. The source of the Spectre attack is dynamic control speculation, which implies that the speculation begins with a branch prediction and the processor starts speculatively executing along the predicted path. To fully mitigate this attack, we want a fence between each branch and subsequent loads; but if one branch is followed by four loads, we only need one fence to protect all four. Thus, in this optimization, we propose to only instrument (provide the alternate decoding for) the first instance of a vulnerable instruction type (e.g., first load) of each basic block, and it is safe to leave the rest of the instructions uninstrumented. This is simply implemented by setting a flag in hardware whenever a branch is decoded, then insert a micro-op in the alternative load decoding that, along with the fence, also resets the flag. When the flag is not set, the original decoding is used.

Taint-Based Fence Insertion. However, even one load per basic block is likely still too conservative. In all known instances of the attacks, the attacker performs some operations (mostly memory read) based on untrusted data that leads to the information leak. For example, in Spectre variant-1 the attacker provides an untrusted out-of-



Figure 3.5. DLIFT integration with a CSD-enabled pipeline

bound index to an array. In this work, we assume any information that comes from the user address space and input devices (e.g., via the x86 IN instruction) as untrusted. In addition, we also consider DMA'd pages as untrusted, for which we rely on the IOMMU [16] to mark DMA'd pages as tainted in the page table. For the taint-based fence insertion optimization, we propose the insertion of fences for only vulnerable/tainted loads that operate on untrusted data, by leveraging a novel decoder-level information flow tracking (DLIFT) strategy described in the following section.

3.4.2 Decoder-Level Information Flow Tracking

As its name indicates, the distinguishing feature that makes Decoder-Level Information Flow Tracking (DLIFT) unique from typical hardware taint tracking systems is that it can provide the taint information at the decoder stage of the pipeline rather than at commit stage. This is a critical distinction for two reasons. First, of course, our solution is decoder-based. Second, and perhaps more important, executeor commit-based taint tracking comes too late in the pipeline for any speculation-based attack, and is of little use.

However, commit is still the only stage where taint information is guaranteed

to be correct, so the design of a decode-based taint tracker is challenging. In particular, since the front-end of the pipeline typically runs far ahead in execution than the rest of the pipeline, reading actual taints from register files at the decode stage will read inaccurate values. Therefore, in the proposed DLIFT framework, we separate the taint information into four taint structures – (a) a decoder-level taint map that tracks and maintains the taint information for architectural registers, (b) the physical register file augmented with taint information that maintains dynamically computed taint information at execute, (c) the TLB and page tables augmented with a taint bitmap to track cache block-level taint information, and (d) a commit-level taint map that maintains verified architectural register taint information. While the latter three structures are typical of any standard DIFT implementation [245], the first structure is a new addition proposed by this work. Instruction translation thus depends on some speculative taint tracking that might not be exact, potentially leaving vulnerable instructions unfenced (i.e., translating to non-secure micro-ops). For this reason, DLIFT relies on a mistaint recovery mechanism that, upon detecting a mistaint at the execution stage, redirects and restarts the execution from the incorrectly tainted instruction.

Figure 3.5 shows the integration of DLIFT with a CSD-enabled pipeline. The DLIFT engine maintains a taint map that stores a speculative taint bit for each architectural register. For each incoming instruction (**①**), the DLIFT engine evaluates the taint of the destination register(s), based on the speculative taint bits of the source registers. Taint evaluation follows the standard DIFT procedure [245]. If a source register is marked as tainted in the speculative taint map, DLIFT propagates this taint to the destination register and further triggers context-sensitive translation (**②**) of the instruction (e.g., with speculative fence insertion), depending upon the configured levels of performance and security.

Tracking taints at the decoder level is not always straightforward. First, during speculative execution, the DLIFT engine continues to propagate taints along the

misspeculated path, potentially leaving the taint map in an inconsistent state upon recovery from a branch misprediction. To this end, the DLIFT engine rolls back the state of the taint map to its commit-level counterpart, as part of the branch misprediction recovery. Second, the effective addresses of load instructions is usually not known at the decoder stage, due to which the DLIFT engine cannot accurately compute and propagate taint information. A conservative approach that marks all loads with unknown effective addresses as tainted could severely degrade performance since overtainting typically results in a high frequency of fence insertion. Therefore, we take a more optimistic approach by assuming loads with unknown effective addresses are untainted, and further rely on our mistaint detection mechanism at the execute stage to validate the predicted taints against the dynamically evaluated taints (**③** and **④**). In the event a mistaint is detected, we update the speculative taint map, flush the pipeline, and restart the translation and execution of the mistainted instruction. Note that we only perform mistaint recovery for the undertainting scenario, since it could potentially leave vulnerable instructions unfenced.

Furthermore, the DLIFT engine is capable of following instruction sequences and execution patterns such as a tainted load followed by a branch, allowing for the seamless detection of different Spectre gadget variants. In this way, although speculative, the DLIFT engine has the potential for fast recovery from misspeculation, and further allows the microcode customization framework to perform a more targeted and surgical insertion of fences for particular vulnerable targets.

3.4.3 Mitigations for Control-flow Mistraining

Two Spectre variants (v2 and v5) rely on the mistraining of the branch predictor and the return stack buffer to influence the victim's branch outcome across different protection domains. To mitigate these attacks, we examine several hardening mechanisms. In the simplest case, where we are protecting kernel branches from getting

Baseline Processor							
Frequency	3.3 GHz	I cache	32 KB, 8 way				
Fetch width	4 fused uops	D cache	32 KB, 8 way				
Issue width	6 unfused uops	ROB size	168 entries				
INT/FP Regfile	160/144 regs	IQ	54 entries				
RAS size	8,16, 32 entries	BTB size	256, 512, 1024 entries				
LQ/SQ size	64/36 entries	Functional	Int ALU(6), Mult(1),				
Branch Predictor	LTAGE	Units	FP ALU/Mult(2), SIMD(2)				

Table 3.4. Architecture Detail for the Baseline x86 Core

influenced, we can instrument syscall instructions to enforce branch predictor state isolation. If we want to protect a wider range of domain crossings, we could rely on a simple hardware mechanism that identifies control transfer to a domain with a higher privilege and then sets a flag. The first control flow instruction that decodes after that flag is set would then trigger an alternate decoding and reset the flag. The alternate decoding would insert a fence (to protect against this branch being mispredicted), then execute micro-ops that enforce branch predictor state isolation, and subsequently resume the original control flow.

This solution, then, is flexible enough that any region of code could be protected in this way by configuring model specific range registers (MSRRs) to indicate a protected region that would always reset the branch predictor upon entry; this, then, even prevents mistraining of sensitive code of different threads within the same process. To prevent the MSRRs from being tampered by the attacker, we only allow the operating system to configure MSRRs, and we use speculative fences to guard x86's *WRMSR* instruction. Furthermore, in the case of remapping via mmap(), the operating system is also responsible for reconfiguring MSRRs.

The easiest and the most heavyweight way to isolate the BP state is to clear all the states by a series of return/branch micro-ops. However, such a hard reset of the BP can be more quickly and efficiently performed using a special micro-op that clears some or all state of the branch predictor, thereby enabling custom solutions to reset different structures within the prediction unit, if such a micro-op is available. In this work, we assume a special micro-op to clear the branch target buffer (BTB) and the return stack buffer (RSB), the primary targets of the Spectre attack.

It should be noted that while resetting the BTB and the RSB will induce more mispredictions (something the attacker wants), it does the attacker no good with respect to these Spectre variants because the attacker can no longer control the target of the control flow misspeculation.

3.5 Methodology

This section details the experimental methodology for the performance and security evaluation of the *context-sensitive fencing* framework.

Our baseline processor is modeled after the Intel Sandybridge microarchitecture [124]. Table 3.4 shows the architectural configuration of our baseline processor in more detail. Note that we evaluate against three different branch predictor configurations (with different BTB and RAS sizes) to measure the performance impact of our micro-op flows that perform branch predictor state isolation across protection domains to defend against variant-2. We model this architecture using the gem5 [28] architectural simulator which already features x86 micro-op translation. Furthermore, gem5 already features a Spectre test infrastructure and a visualization tool that allow us to validate our claims regarding the security guarantees of context-sensitive fencing [171].

One of the primary goals of this work is to protect kernel memory from being leaked along misspeculated paths. Therefore, we use the full system simulation mode of gem5 which allows us to boot an Ubuntu 18.04 distribution of Linux with a kernel version of 4.8.13. Furthermore, in order to provide realistic estimates regarding attack coverage and performance impact of our proposed techniques, we select a good mix of benchmarks that spend different amounts of execution time in kernel mode, touch

Benchmark	Description	Kernel-Time
nginx	HTTP Web Server	66%
ps	Process information query	75%
ping	Sends ICMP ECHO_REQUEST to loopback	95%
ls	Performs two level directory listing	79%
llu	Linked list traversal micro-benchmark	7%
bzip2	Compression	34%
gcc	C Language optimizing compiler	11%
omnetpp	Discrete event simulation	15%
sjeng	Artificial Intelligence (game tree search and pattern recognition)	22%

Table 3.5. Benchmarks Description

different aspects of the operating system, and perform different number of system calls during execution, as illustrated in Table 3.5. To this end, we include four benchmarks, *bzip2, gcc, sjeng, omnet,* from the SPEC CPU2006 suite [106] that exhibit varying degrees of instruction-level parallelism. Furthermore, we use three commonly used Unix tools that target different functionalities of the kernel and use different device drivers – (*ping*) that sends and receives ICMP packets, (*ps aux*) that queries process information, and (*ls* /*/*) that queries the filesystem in order to list multi-level directory information. In addition, we also add a memory-intensive program *llu* that allocates a large amount of memory for a linked list and traverses the list making random memory accesses [316]. Finally, we also evaluate the impact of our fences on the *nginx* web server [219] using the *wrk* [90] framework to generate HTTP requests.

3.6 Evaluation

In this section, we evaluate both security and performance impacts of the proposed strategy, starting first with security assessment and following that with performance evaluation.

3.6.1 Security Discussion

The proposed defense strategy can mitigate five variants of the Spectre attacks. We discuss these in detail below.

Variants 1, 1.1, and 1.2. The goal of these variants is to leak memory contents of a victim along a misspseculated path by bypassing a bounds check and/or further stitch together such Spectre gadgets via store-to-load forwarding. All variants exploit a data cache-based side channel to leak information. Context-Sensitive Fencing mitigates these variants primarily by preventing information from being leaked along misspeculated paths via fences that are surgically inserted into the instruction stream and strategically placed between the conditional branch that performs the bounds check and the first load in the Spectre gadget. This work proposes three new fences – LSQ-LFENCE, LSQ-MFENCE, and CFENCE that each defend against the variant-1 attack by disallowing loads along misspeculated paths to modify the data cache state. LSQ-MFENCE additionally protects against the variants 1.1 and 1.2 by preventing both speculative loads and stores from being issued out of the load/store queue, effectively avoiding store-to-load forwarding between speculative loads and stores.

The surgical insertion of these fences is facilitated by the decoder-level information flow tracker (DLIFT) that can follow instruction sequences that could serve as Spectre gadgets, as described in Section 3.4.2. While the DLIFT itself performs speculative tracking of taint information, it has the ability to detect mistaints and recover within three stages of the pipeline. We further experimentally show later in this section that the speculative nature of the DLIFT engine may sometimes result in overtainting (where trusted operands/instructions get marked as tainted), but never results in an undetected and unrecovered undertainting (where untrusted operands/instructions remain untainted).

Variants 2 and 5. These variants reverse-engineer and mistrain the branch



Figure 3.6. Execution Time of Different Fence Enforcement Levels (normalized to insecure execution)

predictor and the return address stack to influence the branch outcomes of a victim, which in our case is the Linux kernel. In addition to the surgical fence injection and speculative taint tracking, context-sensitive fencing further employs mistraining mitigations that circumvent such attempts. More specifically, we intercept all protection domain crossings via *syscall* and *int ox8o* instructions and perform a full reset of the branch target buffer (BTB) and the return stack buffer (RSB), clearing all previous state. This prevents user code from influencing branch outcomes of the kernel code. We record no instance of BTB collision between user and kernel branches, and we ensure that we always start with a clean BTB and RSB upon entering kernel mode.

To show that our different fences and frequency optimizations close Spectre attacks, we use a proof of concept Spectre implementation and visualization tool [171]. The attacks fail in every case when we use context-sensitive fencing to insert fences, despite our performance optimizations.

3.6.2 Performance

Figure 3.6 measures the performance impact of three different fences – (a) the standard x86 LFENCE, (b) the LSQ-MFENCE, and (c) the CFENCE, all enforced with



Figure 3.7. Execution Time of Early and Late Commit of CFENCE (normalized to insecure execution)

the standard late commit, pessimistically inserted for every kernel load in the program. Clearly, from the figure, the CFENCE incurs the lowest performance overhead, since it is less restrictive than the other two and is enforced at a much later pipeline stage. We further study the effect of late and early commit policies for the CFENCE, as shown in Figure 3.7. The early commit version of the CFENCE consistently performs better than the late commit version, saving about 4% in overall execution time on average.

Overall, the CFENCE reduces the incurred performance overhead due to fencing by 2.3X, bringing down the execution time overhead from 48% to 21%. Furthermore, this performance improvement is consistent across all the benchmarks; the only exception being *llu*, which performs random memory accesses due to the linked list traversal and suffers from a very high cache miss rate.

In Figure 3.8, we study the effect of the CFENCE on cache miss rate. Recall that when a load passes a CFENCE, it gets marked as *non-modifying*, and as a result, if it ends up being a miss in the cache, it is deemed *uncacheable* and therefore the fetched cache block isn't brought into cache. If the miss rate of a program is already high, as in the case of *llu*, the presence of a CFENCE in the instruction stream could potentially



Figure 3.8. Effects of injecting CFENCE on Cache Miss Rate

result in under-utilization of the cache since the missed blocks aren't being filled back into the cache while the CFENCE is begin enforced. For programs that have unusually low hit rates, we suggest using the standard LFENCE instead of the CFENCE. However, in those cases it should be noted that the overhead of the mitigation is low regardless of the fence used.

Furthermore, it is important to note that the number of non-modifying loads in the dynamic instruction stream do not necessarily have a negative impact on performance, while a CFENCE is being enforced. For example, while the *ping* program has the most non-modifying loads, it does not suffer much in terms of overall performance, because the working set fits well into the cache and most of the non-modifying accesses end up being hits. In general, we gain more from employing the CFENCE instead of standard serializing fences when the cache hit rate is low.

Figure 3.9 shows the performance of our proposed fence frequency optimization techniques. In the first optimization, we inject the CFENCE only for tainted loads and branches, as indicated by the DLIFT engine. This reduces the performance overhead of the defense from 21% to 11% on average. We further optimize the number of fences

inserted by performing basic block-level fence insertion, where we only instrument the first instance of a vulnerable load in each basic block. This results in an additional 4% improvement in performance. As Figure 3.9 shows the once per basic block opitimization is successful at improving the performance of all the benchmarks except *llu*. That is because CFENCE changes the cache access pattern, usually at some cost, but occasionally beneficially by bypassing accesses and reducing pressure on the cache. In such a case, like *llu* with its large working set, executing fewer fences would then be slightly less beneficial. Overall, compared to the state-of-the-art fence injection scheme that pessimistically injects an LFENCE for all kernel loads, our DLIFT-Based CFENCE injection reduces the performance overhead from 48% to just 7.7% on average.

We next study the accuracy and coverage of our DLIFT implementation. Figure 3.10 examines overtainting scenarios, where the DLIFT engine conservatively marks instructions as tainted when they're actually not. The results are normalized to the total number of dynamic load instructions executed. Further, we examine two scenarios – the bars to the left represent DLIFT-based fencing without the basic-block level fence insertion, and the bars to the right includes the basic-block level fence insertion optimization. In most cases, the percentage of overtainted loads remains low in both scenarios.

Figure 3.11 examines undertainting scenarios, where the DLIFT engine optimistically forgoes instrumenting certain instructions that are actually vulnerable, but recovers from such scenarios later in the pipeline. We observe that the percentage of undertainted loads is low and consistently below 10%, with the outliers being the kernel-heavy applications, *ping* and *ps*, which have the highest number of tainted instructions (instructions with at least one tainted source register) as shown in Figure 3.12.

Note that our DLIFT engine performs explicit information flow tracking modeled after DIFT [245] that tracks copy, load-address, store-address, and computational



Figure 3.9. Impact of Fence Frequency Optimizations on Execution Time



Figure 3.10. Accuracy of DLIFT: Overtainting Rate of Loads



Figure 3.11. Coverage of DLIFT: Undertainting Rate of Loads

dependencies. However, we also evaluate CSF with a more conservative implicit information flow tracking model in which we taint the program counter when the branch outcome depends upon a tainted value, and further track both sides of the branch. This results in more tainted instructions and incurs 11.8% extra performance overhead on top of our taint-based CFENCE insertion.

To evaluate the effect of our mistraining mitigations, we measure the impact of a full micro-op based BTB and RSB reset at every protection domain crossing. We do this experiment on three separate branch predictor configurations: (a) a small predictor with 256 BTB and 8 RAS entries, (b) a medium predictor with 512 BTB and 16 RAS entries, and (c) a large predictor with 1024 BTB and 32 RAS entries. We measure an average performance degradation of 2.7% on our small predictor, 6.6% on our medium predictor, and 15.2% on our large predictor. Naturally, the overhead is almost completely due to the loss of prediction accuracy, rather than the cost of micro-op expansion.

In summary, the proposed defense strategy introduces a flexible microcode customization framework that perform the surgical insertion of newly introduced



Figure 3.12. Ratio of instructions marked as tainted by DLIFT

speculation fences, that mitigate five variants of the Spectre class of attacks, reducing the fencing overhead of state-of-the-art fence-based Spectre mitigations by a factor of 6.

3.7 Conclusion

In this work, we propose context-sensitive fencing (CSF), a set of architectural techniques that provide high-performance defense against Spectre-style attacks. In particular, we show that we can reduce fencing overhead by a factor of 6 compared to a conservative fence insertion method.

This is done by injecting fence instructions dynamically, in the decoder, with no recompilation and binary translation required. This allows us to employ runtime information to strategically insert fences when needed, using taint information and avoiding redundant fences after branches. This work also introduces new fence primitives which protect sensitive structures from speculation-based microarchitectural effects, with minimal impact on instruction throughput in the pipeline.

Acknowledgments

We thank the anonymous reviewers for their helpful insights. Chapter 3, in full, is a reprint of the material as it appears in Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2019. Taram, Mohammadkazem; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 4 Secure Simultaneous Multithreading

4.1 Introduction

The pursuit of secure computation has always featured a clear tension between performance and security. Security mitigations often come with a high performance [31] impact that can be manifested in serious environmental and economic impacts [134] if they are employed, and in disastrous security and privacy breaches [20,47,187–190] if they are not. In the context of processor architectures, this security-performance tension is only growing as new attacks appear, each exploiting a crucial performance optimization in the processor, threatening to unwind decades of architectural gains.

Microarchitectural attacks exploit different architectural features. Speculative execution [148], shared caches [98], branch prediction [78], execution units [14], and I/O throughput optimizations [256] are examples of the features that are exploited in both well-established attacks and more recent instances [148]. Turning off any of these features could be crippling to performance, so we typically seek ways to continue to enable the optimization but with higher levels of protection.

One performance optimization, however, is often switched off in the name of security, at significant performance cost: Simultaneous Multithreading (SMT) [265, 266]. SMT enables a processor core to issue instructions from multiple threads to the execution units in the same core in the same cycle. With a small investment in

hardware, the processor can greatly increase the throughput/utilization of the pipeline, more effectively hiding latencies and stalls of all types. The substantial benefits of SMT have led to its widespread adoption by virtually all the major players in the high-performance processor market, i.e., Intel, AMD, IBM, and ARM.

SMT achieves its high level of execution efficiency by dynamically allocating resources to threads as they are needed, effectively utilizing resources not needed by one thread to accelerate another. Virtually every part of the pipeline is potentially shared and contended for in some way. This creates a performance coupling between co-resident threads that is an enormous challenge for security. Some have suggested turning off SMT altogether [91]. Google, as a response to MDS attacks, has disabled SMT by default on Chrome OS 74 and later [91]. OpenBSD takes a similar measure by disabling SMT by default on version 6.4 and later [156]. Red Hat has announced [218] kernels with updated controls allowing users to choose whether to disable the feature or not.

Security researchers and architecture researchers are actively working to preserve many of the optimizations recently under attack (speculative execution, caches, branch prediction, store-load forwarding, etc.). This chapter makes the case that it is time to add SMT to the set of features we should preserve even for secure execution. This research seeks to identify the extent of the vulnerability of current SMT processors, and to begin to navigate the middle ground – that is, how much of SMT's performance benefits can we retain while providing vastly greater performance decoupling?

This chapter seeks to provide an exhaustive evaluation of resource contention across the entire pipeline for recent offerings from both Intel and AMD. We find that those two providers already take very different approaches to the security/performance tradeoffs. Both, however, provide a number of high-bandwidth channels of potential leakage, including several never before identified. By focusing on the bandwidth of covert channels utilizing the various resources, we are able to perform

116

a unified, systematic, and exhaustive study of pipeline resource vulnerabilities.

In a similar way, we also seek to examine more secure approaches to multithreading that can be applied in a more holistic and comprehensive manner. We identify three different approaches to partitioning that can be applied to all contended resources with slight variation, and evaluate their ability to restore the performance of a fully dynamically shared SMT processor.

These approaches include full static partitioning (the approach applied already to several pipeline resources) and two new approaches. Adaptive partitioning provides a temporary, hard partition between threads for a particular resource, but that partition can move at regular intervals to adapt to long-term program behavior, preserving some of SMT's ability to adapt to changing execution needs with minimal leakage between threads. Asymmetric SMT enables the system to prevent leakage to an untrusted thread, but without sacrificing the performance of the trusted thread. Our results show that these secure multithreading approaches provide high isolation between threads while retaining most of the performance of a dynamically shared, insecure SMT implementation.

4.2 Background and Related Work

This section provides background on modern x86 processor architectures, SMT architectures, and microarchitectural side-channels, with a focus on SMT-enabled attacks.

4.2.1 The x86 Pipeline Resources/Structures

Figure 4.1a shows a modern x86 processor's frontend. The frontend is responsible for fetching, decoding, and delivering operations to the backend. In x86 processors, this is accomplished using one of the following three methods:

1) The legacy decode pipeline. Each cycle, the frontend reads a 16-byte block from

the instruction cache into the fetch buffer, which then feeds into the predecoder that demarcates individual variable-length x86 instructions (also called macro-ops). These macro-ops are then inserted into a small buffer (the macro-op queue). Instructions in this queue are then distributed to one of the instruction decoders, which translate each instruction into internal RISC (Reduced Instruction Set Computing)-like *micro-ops*. Modern Intel processors feature one complex decoder that is able to translate the instructions into up to four micro-ops, plus simple decoders that can only translate instructions that decompose into a single micro-op. Any instruction that translates to more than four micro-ops is handled via a Micro Sequencing ROM (MSROM). The decoded micro-ops then get queued up in a small structure called the micro-op queue – also called the instruction decode queue – until they get issued into the backend.

2) The micro-op cache. Due to the complexity of the decoding process, the legacy decode pipeline can be a major performance bottleneck. To alleviate this, most modern x86 processors cache decoded micro-ops in a special structure called the *micro-op cache* or the decoded stream buffer. This cache enables the frontend to bypass the slow and power-hungry legacy decode pipeline whenever the translated micro-ops of an address are already available in the micro-op cache. Due to the streaming nature of the micro-op cache, the processors often impose special placement rules [126]. For example, in Intel processors, the micro-ops of a 32-byte region of the code can be placed in the micro-op cache only if the region gets translated into less than 18 micro-ops.

3) *The loop stream detector.* Intel processors feature another optimization called a *loop stream detector* (LSD). The LSD further improves bandwidth and power consumption by bypassing both the legacy decode pipeline and the micro-op cache. The LSD identifies small loops within the micro-op queue and then locks down the micro-ops in the queue. It then delivers the micro-ops from the micro-op queue until an unexpected control flow (i.e., branch misprediction) occurs [126].

Figure 4.1b shows the main components of a modern x86 out-of-order super-

scalar backend. First, the backend renames the architectural registers of the issued micro-ops and allocates an entry in the reorder buffer for each. They are then forwarded to the scheduler, also called reservation stations or the instruction queue, which is responsible for identifying micro-ops ready for execution (i.e., whose operands are ready) and dispatching them into available execution units. Intel groups the execution units into execution ports, and the scheduler can dispatch as many micro-ops per cycle as allowed by the superscalar width of the processor, with the restriction that only one micro-op is dispatched to a free port in any given cycle.

4.2.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) [265, 266] is an architecture that allows for multiple hardware execution contexts in a single out-of-order superscalar pipeline. In an SMT processor, instructions from multiple threads can be dispatched on any cycle. An SMT processor significantly improves the pipeline utilization since it allows continued forward progress if one thread experiences a temporary stall in the pipeline due to long latency (e.g., a cache miss), or even several tightly-dependent short latency operations. Its performance benefit comes from its ability to dynamically assign resources to the thread that needs them each cycle. But this level of sharing provides heavy exposure of one thread's performance to the characteristics of the other.

Contention is possible in various ways throughout the entire pipeline. Consider the Reservation Stations (RS). Ivy Bridge has 54 entries available for instructions to wait for their operands to become available before being eligible for execution. The number of RS entries available to a thread define the window over which the processor can look for out-of-order parallelism, and significantly impacts performance. This resource could be shared between threads in various ways, with very different security implications. RS entries could be *duplicated*, where each thread would have access to exactly 27 entries (assuming two thread contexts). The RS could be *partitioned*, where



Figure 4.1. Simplified Architecture of a Modern X86 Processor.

in SMT mode each thread has access to 27, but when only a single context is running, it has access to 54. In that case, the only effect one thread sees is whether the other thread is running or not. It could be fully *shared* (this was the general assumption in the original SMT literature), where RS entries are allocated to whichever thread asks for them, with the fetch unit guiding instruction entry so that neither thread would fill the structure and starve the other [264, 265]. Full sharing typically, but not always, maximizes performance; however, a thread that put significant pressure on the RS would have its performance constantly vary according to the RS utilization of the other thread, and completely saturating the RS could have a dramatic impact on the other thread's performance. An intermediate solution is *thresholding*, where either thread is allowed to use up to, say 44 entries, ensuring that at least 10 are always available to the other thread even if it's not currently using them. In theory, thresholding can leak just as much information as full sharing, but in practice it is much more difficult to successfully execute an attack because of the difficulty of keeping resource utilization right near the edge of the threshold.

In Figure 4.1, we can see examples of shared structures (e.g., Instruction Cache), duplicated structures (Macro-op Queue), and partitioned structures (Micro-op Queue).

4.2.3 Microarchitectural Covert/ Side Channels

The literature abounds with security attacks that leak information through a shared microarchitectural structure or feature [148, 213, 239, 242]. Shared caches [35, 183, 184, 206, 278, 306], branch predictors [4, 36, 51, 78, 79], store-to-load forwarding [248], Translation Lookaside Buffers (TLB) [94, 118], I/O throughput optimizations [152, 161, 256], and the processor execution ports [7, 14, 26] are a few examples that are exploited as a source of side-channel leakage. Most of these attacks leak information through a timing difference that is caused by contention on a shared resource. The recent transient execution attacks [148, 165, 269–271] also rely on a microarchitectural

side-channel to leak the information to the attacker's domain. Previous work also uses timing measurements to infer different processor's features such as the size of the ROB [296].

Previous work [66, 119] has used information-theoretic and mathematical approaches to study information leakage from microarchitectural channels. For example, Hunger et al. [119] develop an abstract mathematical model for microarchitectural channels, with the end goal of devising an attack detection mechanism. Our characterization study, however, targets a different goal. It aims to compare and contrast resource sharing between threads in different implementations of SMT, to then evaluate the inherent vulnerability of sharing of each resource, and to guide the design of SMT security measures.

Previous work has also shown covert and side channels constructed on GPU resources [76,196,197]. Naghibijouybari et al. [197] exploit contention on GPU resources to infer victim's web browsing activities. Dutta et al. [76] target Intels integrated GPU architectures and present covert channels that put contention on the CPU-GPU bus as well as the shared last-level cache. Similar to the CPU-based channels, these channels also leak information through the timing variation caused by resource contention. However, the list of possible targets for a GPU channel is limited to co-locating kernels from two different applications on a GPU. In contrast, CPU, and in particular SMT-based channels can leak more fine-grained information from a vastly larger set of targets, and thus pose a more imminent threat.

4.2.4 SMT Covert and Side Channels

SMT, with the sheer amount of shared resources, potentially greatly expands the microarchitectural attack surface, as nearly every structure could be contended for at some level. However, while SMT facilitates the exploitation of many side channels, not all of these channels are fundamentally tied to SMT. Cache side-channels [184,207,263],

which include the vast majority of these attacks, for example, are almost as effective in a cross-core setting [227]. Some of the SMT-specific attacks are the result of a design bug [165], not an inherent SMT issue, which can be mitigated without much performance cost in future generations. Nevertheless, the core principles of SMT have already been targeted in some of these side-channel attacks.

The SMT execution port timing channels have a long history [7, 286]. But more recently, PortSmash [14] exploits, in an end-to-end attack, the timing variation caused by the contention of the SMT threads on specific execution ports to leak the private key of a TLS server. SmotherSpectre [26] also exploits the contention on the execution ports but combines it with speculative execution to mount a transient execution attack. SmotherSpectre uses speculation to steer the execution to a gadget that includes a data-dependent branch that accesses a specific port based on the value of a secret.

Other attacks target other shared resources in SMT processors. TLBleed [94] exploits shared TLBs in SMT processors to leak a victim's memory activity. CacheBleed [306] uses contention of the sibling SMT threads on the cache banks as the source of the leakage to break a constant-time RSA implementation. Similarly, MemJam [192] targets the shared memory-dependency detection unit to leak information about memory accesses of a victim. Ren, et al. [220] develop multiple attacks exploiting the microop cache of Intel and AMD processors to leak information across different security domains, including colocated SMT threads. Shared branch predictors are also extensively exploited to leak secret information [5, 6, 78] and also to steer speculation to attacker-desired addresses [148].

There are also efforts to automatically construct covert channels in SMT processors. Covert Shotgun [84] proposes an automated framework to examine possible pairs of instructions in an ISA for building covert channels. Covert Shotgun compares the execution time of the instructions in single-threaded and SMT mode to detect if there is a possible covert channel. More recently, ABSynthe [93] conducts a similar study that expands Covert Shotgun to include all instructions in x86 and ARM ISAs and compares the results for a variety of architectures. While these approaches can discover covert channels in an SMT processor, the exact source of the leakage for the discovered covert channels remains unknown. This work, in contrast, characterizes different covert channels based on the actual source of leakage.

4.2.5 Side-Channel Mitigation

The research community is increasingly active on countermeasures for microarchitectural side-channels. Similar to the microachitectural attacks, the defense research also leans heavily toward mitigations for cache-based side channels [68, 229, 252–254, 301]. A diverse set of strategies has been proposed to mitigate side channels. Partitioning [49, 67, 68, 280, 298], randomization [67, 214, 287, 313], detection [283, 298, 300, 312], oblivious execution [34], and encryption [215] are among frequently suggested highlevel approaches. Many of the more recent defenses focus on a mitigation in the context of speculative execution vulnerabilities [12, 178, 228, 255, 289, 302, 309].

However, much less attention is given to defenses for non-cache microarchitectural side channels. SMT-COP [261] introduces a temporal and spatial partitioning scheme for execution ports in SMT. SMT-COP also introduces a selective approach where it selectively enables and disables functional unit partitioning for some regions of the code. Unlike this work, SMT-COP's partitioning, once enabled, is entirely static, failing to take advantage of the benefits of an SMT architecture. Moreover, our study is not limited to execution ports, and we examine mitigations on a comprehensive set of pipeline resources.

Hyperspace [43] secures SGX execution against SMT channels by creating a trusted shadow SGX thread that runs on the same physical core as the main thread. Hyperspace verifies the co-location of the main and the shadow threads using a cache covert channel. The sole purpose of the shadow thread is to prevent any untrusted thread being scheduled on the same core as the main thread. Therefore, in terms of resource utilization this approach is similar to, and in some cases worse than, turning off SMT.

Xu et al. [298] propose a mitigation strategy for GPU-based covert channels. It relies on a decision tree classifier to detect a potential attack, and then enables temporal and spatial partitioning of GPU resources to mitigate the contention.

4.3 Assumptions and Threat Model

The main focus of this work is on covert channels constructed by contention on the main pipeline structures between the co-located threads on an SMT processor. These channels are the dominant form of leakage introduced by simultaneous multithreading, and until mitigated, will likely dominate other channels. We principally study covert channels, because the goal of this work is to provide, among other things, a comprehensive and systematic analysis of the vulnerability of existing SMT processors. Such an approach would not be possible if we were to try to analyze all specific side-channel attacks customized to each feature, and would become out of date quickly as new attacks are devised. By concentrating on covert channels, we can evaluate the inherent vulnerability of each feature and have some basis for comparing them and understanding where the greatest vulnerabilities lie.

Furthermore, by closing these covert channels we also close any potential side channel that exploits them, including those used in speculative execution attacks. To be successful, most speculative attacks need to effectively perform two tasks: (1) steer the execution to attacker-desired locations, and (2) leak information to the attacker domain using a covert channel. To stop such attacks, it is sufficient to prevent the latter, which we aim to do by denying unauthorized information leakage – speculative or otherwise – between SMT threads. We, then, propose mitigations against the covert channels with the following assumptions. We assume an SMT processor on which two threads, T1 and T2, can share the pipeline. We assume T1 and T2 are running on separate processes and are prohibited from any form of direct communication, but they can run any non-privileged instruction on the SMT core. Any of the threads are allowed to make artificial contention on any of the shared pipeline resources to leak information about the usage of the other thread. We consider multiple scenarios:

- T1 and T2 are mutually untrusted: in such scenarios, any information flow from T1 to T2 and from T2 to T1 should not be allowed.
- The trust is asymmetric, i.e., T1 is untrusted, and T2 is trusted: in such scenarios, the information flow from T1 to T2 is allowed, while the information flow from T2 to T1 should be blocked.
- The threads are mutually trusted: any covert communication between T₁ and T₂ is allowed.

The main focus of this work is on closing timing channels engendered by resource contention between SMT threads. Therefore, this work does not consider power, voltage/frequency, and thermal channels.

4.4 Covert Channel Characterization

The goal of this chapter is to make SMT processors more secure against contention-based side-channel attacks. To that end, we first conduct a rigorous analysis on current SMT processors to assess the degree to which they share each pipeline resource between threads, and to also measure the potential information leakage resulting from sharing each of these pipeline resources. This analysis then guides the design of SMT security measures (Section 4.5). This study also leads to the discovery
		Intel (Skylake)		AMD (Zen2)	
	Resource	Sharing	BW (bps)	Sharing	BW (bps)
Front-End	L1 iCache	S	742K	S	1.27M
	Branch Target Buffer	S	796K	S	478K
	Micro-Op Cache	Р	j24K	S	604K
	Fetch Bandwidth	S	1.64M	S	833K
	Decode/Issue Bandwidth	S	1.15M	S	1.03M
	iTLB	Р	j24K	S	820K
	Micro Sequencing ROM	М	-	S	353K
	Loop Stream Detector	Р	;24K	-	_
Back-End	Reservation Station	Т	_	S	56K
	Reorder Buffer	Р	j24K	Р	_
	Physical Register File	Р	j24K	S	40K
	Execution Port	S	1.22M	S	715K
Memory	dTLB	S	982K	S	964K
	L1 dCache	S	1.13M	S	902K
	L1 dCache Read Bandwidth	S	1.36M	S	1.64M
	Load Queue	Р	j24K	S	36K
	Store Queue	Р	j24K	Р	_

Table 4.1. Sharing Mechanism of Pipeline Resources

S:Shared, P:Partitioned, T:Thresholded, M: Time Multiplexed

j:Limited by the maximum switching frequency between single-threaded and SMT modes

of multiple previously unreported and high-bandwidth covert channels.

4.4.1 Overview

The first step in our analysis is to deconstruct how the processor manages resource sharing between the SMT threads. We go through an exhaustive list of pipeline resources and reverse-engineer the sharing mechanism that the processor uses for each of the pipeline resources. We broadly categorize each pipeline resource into *statically partitioned, dynamically shared*, and *duplicated* resources based on our experimental analysis. We note that partitioned resources can either by spatially shared (half assigned to each thread in SMT mode) or time-multiplexed (one thread uses all resources one cycle, the other thread the next). We consider *thresholding*, where the partition is dynamic, but neither thread can completely exhaust the resource, as a special case of dynamically shared.

To reverse engineer the sharing mechanism of a pipeline resource, we craft a set of assembly instructions that create artificial contention on that resource. This set of assembly instructions needs to have four features: (1) it should saturate the structure-under-test, (2) the amount of saturation should be controllable, (3) it must not create high contention on any other pipeline resources/structures, and (4) it should put the contention on the critical path, so the effect of the contention is exposed via performance. Then we run this test code simultaneously on the sibling threads and measure the effects.

If increasing the saturation in one thread affects the execution time or the usage of the other thread, we conclude that the structure is dynamically shared. For dynamically shared resources, we can typically use the same code to construct a covert channel by selectively saturating or freeing the resource. Finally, we measure the bandwidth and error rate of the constructed covert channel.

We also explore the possibility of constructing covert channels on statically partitioned resources. Table 4.1 shows the list of the pipeline resources that we examine along with their discovered sharing mechanism and the achieved covertchannel bandwidth on two different implementations of SMT: AMD Zen2 and Intel Skylake. The table shows that while AMD allows most of the pipeline resources to be competitively shared, Intel Skylake employs some kind of partitioning or time multiplexing for most key pipeline resources.

In addition to Intel Skylake, we also study the sharing mechanism of pipeline resources on Ivy Bridge, an older Intel microarchitecture. Running our microbenchmarks on these microarchitectures shows that Intel uses similar sharing mechanism across these different microarchitectures. Similarly, we also examine whether any of our channels are impacted by different versions of the microcode. In an exhaustive analysis on the Ivy Bridge processor (because the older processor naturally offers updates spanning a longer time period), we observe no change in any of the discovered sharing mechanisms across all available microcode versions. Details of the methodology are discussed in Section 4.6. The rest of this section examines key pipeline structures, moving from front to back of the pipeline.

4.4.2 Instruction Fetch Bandwidth

To reverse-engineer the sharing policy of the instruction fetch unit in an SMT processor, we develop a microbenchmark that creates artificial contention on the instruction cache read bandwidth, while ensuring that there is no contention for the rest of the pipeline resources. To this end, we take advantage of the fact that NOP instructions have a minimal footprint, as they get eliminated early in the pipeline and consume few backend resources, if any.

However, the most commonly used x86 NOPs are 1-byte instructions. These do not suit our purposes because they saturate the decoders long before they saturate the fetch unit (which can fetch 16 NOPs per cycle). To circumvent this, we leverage a special 15-byte long NOP [126] instruction that allows us to effectively saturate the instruction cache read bandwidth, limiting the throughput of the fetch unit to just one instruction per cycle. Further, we ensure that these NOP15 instructions always miss in the micro-op cache and actually use the instruction cache read bandwidth, by using large loops of static NOP15 instructions in our microbenchmark that exceed the micro-op cache capacity.

Through performance counter measurements, we find that the Intel frontend sustains a throughput of one NOP per cycle when our microbenchmark is run in singlethreaded mode. AMD Zen2 also provides the same throughput, despite enjoying an instruction cache bandwidth of 32 bytes per cycle. This is because Zen2 requires instructions that are larger than 8 bytes to be in the first 16 bytes of the fetch buffer. When our microbenchmark is run in SMT mode along with a competing SMT thread that executes the same code, the frontend throughput is exactly halved, delivering one

SEND_ZERO:	SEND_ONE:	RECEIVER:
MOV RAX, 100	MOV RAX, 100	TIME #Record Time
LO:	L1:	NOP15 #15-Byte NOP
NOP8 #8-Byte	NOP #1-Byte	
#N <lsd< td=""><td></td><td>NOP15 #15-Byte NOP</td></lsd<>		NOP15 #15-Byte NOP
NOP8 #8-Byte	NOP #1-Byte	TIME #Record Time
DEC RAX	DEC RAX	JMP RECEIVER
JNZ LO	JNZ L1	

Listing 1. Fetch Bandwidth Covert Channel on Intel Processors. The total number of microops in the receiver loop is larger than the size of micro-op cache to ensure a zero percent hit rate. NOP8 and NOP15 are aliases for multi-byte NOP instructions [126].

instruction every two cycles, for each thread. However, if the code that runs on the competing thread (T₂) delivers its micro-ops through the micro-op cache or the loop stream detector, T₁ gains back its original one NOP per cycle throughput. This shows that the instruction fetch bandwidth is dynamically shared between the threads as we observe a direct impact on the execution time of its sibling thread when they contend for the fetch unit.

Listing 1 shows the sender and receiver routines for a covert channel implementation that exploits the performance differences that arise due to contention for the instruction fetch bandwidth in Intel. The sender thread transmits 'zero' by executing a set of NOP instructions that are delivered by the loop stream detector or the micro-op cache, creating no fetch contention. The sender thread transmits 'one' by executing a set of regular 1-byte NOP instructions, maximizing fetch contention – note that 1-byte NOP instructions will miss in the micro-op cache as the micro-op cache of Intel processors does not cache the line if there are more than 18 micro-ops in a 32-byte region of the code [126, 220]. The receiver thread then measures the execution time of long NOPs that miss in the micro-op cache to detect that contention. The total number of the micro-ops within the receiver loop is set to be larger than the size of the micro-op cache, ensuring that every instruction uses the fetch bandwidth. This covert channel, as Table 4.1 shows, can achieve a bandwidth as high as 1640 kbps on an Intel Skylake processor and as high as 833 kbps on Zen2.

4.4.3 Decode/Issue Bandwidth

After fetching instructions into the fetch buffer, the x86 frontend translates the instructions into micro-ops (decode) and delivers those micro-ops to the backend (issue).

To contend for these decoders, we choose regular 1-byte NOPs. Not only do they not consume backend resources, but they also do not saturate the fetch bandwidth as described above, putting decode/issue bandwidth on the critical path. Thus, when we run regular NOPs on a single thread, the decode pipeline is able to deliver 4 NOPs (micro-ops) per cycle to the backend of the processor. However, this throughput is reduced to 2 NOPs per cycle if we co-locate this thread (T1) with a sibling thread (T2) that executes the same set of NOPs, thereby contending for the decoder resources.

To construct a covert channel that exploits the decode/issue bandwidth, we need to identify the conditions upon which the processor assigns more decode bandwidth to T1. If we switch T2's instructions to large NOPs (maximum of one instruction every two cycles), T1 still observes half of its single-thread throughput, suggesting that on each cycle the instruction decoders are assigned to one thread as a whole, i.e., the decode bandwidth is time-multiplexed between the threads, rather than being statically partitioned. We note that this is consistent with the details laid out in Intel patents [153].

Further, while one would expect T₁ to gain back its single-thread throughput if T₂ does not use the legacy decode pipeline (because it is using the micro-op cache or the LSD which both bypass the decoder), we observe that even in such cases the throughput of T₁ remains half of its single-threaded throughput, indicating that the decoders remain time-shared between the threads, regardless of whether the threads actually contend for them. On the other hand, if T₂ is stalled due to a bottleneck in the

SEND_ZERO:	SEND_ONE:	RECEIVER:
MOV RAX, 100	MOV RAX, 100	TIME #Record Time
CLFLUSH [RCX]	L1:	NOP
MFENCE	NOP	#N>LSD
LO:	#N>LSD	NOP
#Cache Miss:	NOP	TIME #Record Time
MOV RCX, [RCX]	DEC RAX	JMP RECEIVER
DEC RAX	JNZ L1	
JNZ LO		

Listing 2. Decode/Issue Bandwidth Covert Channel on Intel.

backend (full reservation stations, for example), we observe that T₂ does give up its decode slots and T₁ goes back to its full original 4 NOPs per cycle.

As Listing 2 shows, to slow the backend, we exploit a data cache miss followed by a sequence of instructions that are dependent on the long-latency load. This enables a high-bandwidth covert channel on both the AMD and the Intel architectures with a bandwidth as high as 1150 kbps on Intel and 1030 kbps on AMD. Exploiting the frontend covert channels bolsters the adversary's ability to fingerprint various activities (e.g., cache misses, micro-op cache usage pattern) of a co-located victim thread, without any cache accesses, just by measuring the execution time of NOPs.

4.4.4 Register File

Next, we examine the sharing mechanism of physical register files of the Intel and AMD processors. Our register file characterization microbenchmark, shown in Listing 3, consists of two memory read instructions that always miss in the caches. Between these loads, we have a variable number of instructions that each consume a physical register, i.e., they have a destination register. When a thread's partition fills, no more instructions can be renamed, placing a limit on the window for out-of-order execution. If the second memory read is not renamed, it is then serialized (not renamed until the former commits). If there is sufficient space to rename it, the loads execute in parallel and performance is significantly improved. By increasing the number of CLFLUSH [RDI] CLFLUSH [RSI] MFENCE MOV RAX, [RSI] #Long-Latency Load MOV RBX, 88 #Consumes One Phys. Reg ... #Use N Phys Regs MOV RBX, 88 #Consumes One Phys. Reg MOV RBX, 88 #Consumes One Phys. Reg

Listing 3. Microbenchmark for Making Contention on Register Files. When N is larger than available physical registers the two loads cannot be issued in parallel.

the register-consuming MOVs we can identify the exact point where we exhaust the renaming registers before the second load instruction arrives.

The impact on the execution time is visible in our results of figure 4.2. These figures (one for Intel, one for AMD) each show four lines, representing two threads. This more detailed result is representative of the analysis done for each of the pipeline resources, although those discussions have mostly been condensed for space reasons. Here, T1 is varied in the number of registers that are occupied before the second high-latency load, while T2 is kept constant (at 50). For Intel, when T1 runs in single-threaded mode, we see that it can use about 128 registers before performance plummets, while in SMT mode that happens at 64. Further, we see that T2 in SMT mode (dashed orange line) is relatively unaffected by the size of T1. For AMD, however, we see that T2 in SMT mode (again, dashed orange) is clearly sensitive to the varying register pressure of T1. From this, we conclude that Intel's physical register file is statically partitioned, and AMD's is dynamically shared.

We observe a similar pattern when we change the consuming instructions to instructions that consume vector, vector mask, or floating point registers. That means the same policy is applied to those register files as well. We are able to exploit the contention on physical registers in AMD to construct a covert channel with a bandwidth of 40 kbps with less than 5% error rate.



Figure 4.2. Reverse Engineering the physical register file sharing mechanism.

4.4.5 Reservation Station (RS)/Scheduler

To contend for reservation stations, we use a microbenchmark similar to the previous section (Listing 4); however, we use *cmp* instructions which do not consume a physical register, but are still dependent on the first load instruction, so the instructions will consume an RS entry and cannot be issued until the first load completes execution and makes the result available to dependent instructions, causing them to quickly release RS entries. If we have enough entries in the reservation station available to the thread, the second load can be issued an RS entry and then dispatched to execution in parallel with the first load. Therefore, we see a spike in the execution time when the length of the dependency chain becomes greater than the number of reservation

CLFLUSH [RDI] CLFLUSH [RSI] MFENCE MOV RAX, [RSI] #Long-Latency Load CMP RBX, RAX #Waits in RS ... #Consume N RS entries CMP RBX, RAX MOV RAX, [RDI] #Long-Latency Load

Listing 4. Microbenchmark for Making Contention on Reservation Station. When N is larger than available reservation station entries the two loads cannot be issued in parallel.



Figure 4.3. Reverse Engineering the Reservation Station Sharing Mechanism.

station entries, as the two loads become serialized.

Figure 4.3 shows the results of running Listing 4 in single-thread mode and in SMT mode on Ivy Bridge. In SMT mode, we observe that Ivy Bridge does not let one thread use all of the 54 RS entries and it always limits the maximum number of allocated RS entries to 40, even when the other thread does not consume any RS entries, e.g., only executes NOPs. We refer to this type of sharing as thresholding. While in theory it should be possible to construct a covert channel on the 26 shared entries, it imposes considerable noise and we are not able to construct a reliable channel on the Intel processor. On the other hand, AMD Zen2 uses a shared reservation station with which we can construct a covert channel with a bandwidth of 56 kbps.

4.4.6 Reorder Buffer, Load/Store Queues

We use slightly different variations of the microbenchmark shown in Listing 3 to reverse engineer the sharing mechanism of the ROB (lst:microbench:rob) and the load (lst:microbench:lq) and store queues (lst:microbench:sq). For the ROB, we replace the register-consuming instructions with NOPs. NOPs serve our purpose to isolate the ROB because they consume ROB entries, but not reservation stations or physical registers. For the load and store queues, we leverage load or store instructions that

always hit in the cache to isolate those queues. These instructions each occupy a load/store queue entry and cannot be issued until the long-latency load executes. Therefore, at some point, by increasing the number of load/store instructions, the dispatch cannot progress due to lack of load/store queue entries. This then forces the long-latency loads to be serialized.

Figure 4.4 shows the results of running Listing 5 in both single-thread and SMT modes. This microbenchmark includes a series of stores which are dependent on the first long-latency load. When the number of store instructions exceeds the size of the store queue, the processor cannot issue the store instruction for which we do not have an available SQ entry, nor any of the following instructions. That is because even in an out-of-order Intel processor, the rename and allocation stages happen in order. If you run out of out-of-order resources such as physical registers, IQ, ROB, or SQ entries, the rename and/or allocation will be stalled. It is only *after* these stages that the processor can identify the dependencies between the instructions and dispatch the instructions out of order to the execution units. Thus, if a store instruction cannot proceed due to lack of SQ entries, the store and all younger instructions-independent of their type–will be stalled. As a result, in single-thread mode we see a spike in the execution time at 36 stores, which is precisely the size of the store queue in our Ivy Bridge processor. We also run the same code in SMT mode, along with another thread that does not consume any SQ entries, i.e., it only executes NOPs. In that experiment, we observe that the number of SQ entries available for T₁ is exactly half of the SQ, suggesting a static partitioning scheme for the SQ.

4.4.7 Covert Channel on Partitioned Structures

As shown in Table 4.1, we find that many of the pipeline resources in Intel processors, and some in AMD processors, are statically partitioned between the threads. For the sake of completeness, we investigate the potential information leakage via



Figure 4.4. Reverse Engineering the SQ Sharing Mechanism. In single-thread mode, we see a spike in the execution time at 36, exactly the size of our SQ. In SMT mode, T₂ only executes NOPs, but still causes T₁'s SQ to be halved.

these partitioned resources (even though we know shared resources will be the highest bandwidth channels). We construct a covert channel where the sender goes in and out of SMT mode, allowing the receiver to observe the state of the particular resource in question. The key here is our ability to enter and exit SMT mode as quickly as possible. To this end, we examine several x86 instructions used for busy waiting.

We first look at the *pause* instruction, originally introduced in Intel's Pentium 4 processor to improve the power and performance of the spin-wait loops [126], so that the spinning thread could free resources while waiting. However, our experiments

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
MOV [RBX], RAX #Consumes a SQ entry
... #Consume N SQ entries
MOV [RBX], RAX #Consumes a SQ entry
MOV RAX, [RDI] #Long-Latency Load
```

Listing 5. Microbenchmark for Making Contention on Store Queue. When N is larger than available store queue entries the two loads cannot be issued in parallel.

with the *pause* instruction suggest that the resources remain partitioned even in the presence of a *pause* – that is, *pause* only releases shared resources, not partitioned.

We also consider *mwait*, which is a privileged instruction that provides a hint to the processor so it can enter a specified target low-power state [126]. In fact, this does release partitioned resources, and we are able to successfully create a channel, but we do not consider this the most useful attack vector since there are so many other attacks possible for a user with privileged access.

Finally, we examine Linux's *nanosleep* system call which is a non-privileged call that deschedules the caller thread until the time specified by the user has elapsed. We observe that, on Intel processors, calling nanosleep causes the processor to unpartition resources, allowing the sibling thread to monopolize them. The nanosleep syscall causes the Linux kernel to schedule an idle task on the logical core, which then executes the aforementioned mwait instruction. This enables a covert channel for transmitting information even via statically partitioned resources. The bandwidth of this covert channel, however, is limited to the minimum latency of the nanosleep system call. As shown in Table 4.1, using the nanosleep system call, we can achieve a bandwidth as high as 24 kbps on the Intel processor. On the AMD processor, however, the nanosleep syscall does not cause the resources to become unpartitioned.

In Section 4.5, we examine multiple partitioning schemes that provide greatly

CLFLUSH [RDI] CLFLUSH [RSI] MFENCE MOV RAX, [RSI] #Long-Latency Load MOV RBX, [RAX] #Consumes a LQ entry ... #Consume N LQ entries MOV RBX, [RAX] MOV RAX, [RDI] #Long-Latency Load

Listing 6. Microbenchmark for Making Contention on Load Queue. When N is larger than available load queue entries the two loads cannot be issued in parallel.

increased thread isolation. However, all are still potentially vulnerable to this channel (fast enter/exit of SMT mode) if they release all resources to a single thread. Thus, for the balance of this paper, we assume a simple solution that provides a countdown timer that limits the frequency at which the pipeline can release partitioned resources, even upon exiting SMT mode.

4.4.8 Other Pipeline Resources

For completeness, we also measure most of the other pipeline resources that have been covered extensively in the literature, such as caches [212], TLBs [94], and execution ports [14]. Those results appear in Table 4.1, but without extensive discussion. However, there are other well-studied SMT resources such as Pattern History Table (PHT) [79] that we do not re-measure as we focus more on lesser known channels. Our mitigations, nevertheless, can be readily applied to these structures as well.

Constructing covert channels on most of the cache-like structures requires some information about internal structures of these resources such as the indexing function and associativity. Also, the knowledge of the replacement policy of a cache-like resource can greatly affect the channel bandwidth as the attacker can exploit that to minimize the size of the probe set. It is, however, still quite possible to create a highbandwidth channel on a cache-like structure without access to detailed information on

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
NOP #Consume an ROB entry
... #Consume N ROB entries
NOP
MOV RAX, [RDI] #Long-Latency Load
```

Listing 7. Microbenchmark for Making Contention on Reorder Buffer. When N is larger than available reorder buffer entries the two loads cannot be issued in parallel.

the replacement policy, as the attacker only needs to create enough accesses to cause a single eviction to the other thread's entries. For example, for any LRU-like structure (e.g., tree-PLRU, bit-PLRU) with associativity of *n*, accessing *n* new entries guarantees an eviction to the existing entries of a particular set.

4.5 Mitigations

This section introduces a suite of mitigation schemes we examine to eliminate or reduce the leakage across the SMT pipeline. Prior work has focused on identifying and solving SMT leaks one at a time [67, 261, 287]. However, in keeping with our systematic characterization study of pipeline resources, we will focus on systematic mitigation solutions that can be employed broadly across each individual contended structure.

The solutions described in this section include static partitioning, adaptive partitioning, and asymmetric SMT.

4.5.1 Static Partitioning

The most basic partitioning scheme, already employed heavily, statically divides a resource into equal-sized partitions. Static partitioning can be applied in two forms: spatial or temporal. Spatial partitioning assigns a resource to a thread and that assignment does not change through time. This can only be applied to resources for which the processor has multiple instances, such as ROB entries. If the number of instances of a resource is less than the number of SMT threads (e.g., some functional units), the processor cannot statically assign the resource to a thread. In such cases, the processor employs temporal partitioning (also called time multiplexing). Temporal partitioning assigns a single resource to each thread in a round-robin fashion. These basic partitioning schemes ensure that dynamic contention cannot occur between the threads, eliminating leakage. For example, in a strictly round-robin resource, the time slot assigned to thread T1 does not depend, in any way, on the usage of thread T2. T1 only gets one out of two slots whether or not T2 uses its slot. This completely inhibits a thread from inferring any information from the usage of the other thread.

4.5.2 Adaptive Partitioning

While static partitioning can eliminate dynamic contention between SMT threads, it typically results in underutilization of pipeline resources, sacrificing overall performance – the fundamental benefit of SMT processors is allowing resources to be better utilized by dynamically assigning each resource to the context that needs it. However, we show that it is possible to gain back much of the full performance of SMT if we can adapt to the varied needs of contending threads, but more slowly. Adaptive partitioning is an on-demand partitioning scheme that allows for the dynamic reconfiguration of partition size, once per *adaptation interval*. This not only improves resource utilization and overall performance, but also limits the information leakage to at most one bit per adaptation. In fact, our results show that even a very long adaptation interval (of 100,000 cycles) can be effective in recovering much of the full performance benefits of a fully shared SMT pipeline.

Our adaptive partitioning design augments each resource with a set of counters: (1) the current size of the partitions, (2) counters that track the number of "full" events that each thread encounters, and (3) a countdown timer until the next adaptation interval. Note that in our experiments we assume a single countdown timer for all resources to help simplify presentation of the results. It also takes three parameters that are set at design time: adaptation interval, adaptation step, and adaptation threshold. Figure 4.5a shows an example for adaptive partitioning of the instruction queue. For every adaptation period, we select a thread that has faced more full events in that period. We then increase the size of the partition of the selected thread by the adaptation step size. We increase the partition size only if the new partition size is still smaller than the adaptation threshold.

We can also apply adaptive partitioning for temporally shared resources, deviating from a completely symmetric round-robin assignment. For example, we might adaptively modify the assignment process for a resource that alternates between two threads each cycle, in such a way that the resource is assigned to a more hungry thread two out of every three cycles. If it needs more, we could again alter the policy such that the resource is assigned to the hungry thread three out of four cycles. At each adaptation interval, we examine the number of full events of each thread and increase the share of the thread with the highest full event count. If that thread is more resource hungry, we increase its count (subject to the threshold), otherwise we decrease its count. Figure 4.5b shows an example of adaptive temporal partitioning.

Adaptive partitioning severely restricts the leakage. Now, for each resource, the attacker can only leak at most approximately 1 bit (adaptation without threshold) per adaptation interval (100,000 cycles). For example, an attacker can probe the size of its own reservation station in two consecutive adaptation intervals. If the RS shrunk, the attacker infers that the victim's RS usage exceeds that of the attacker. This is orders of magnitude below known channels across cores on non-SMT processors.

While adaptive partitioning limits leakage to a single bit per interval, the values of the counters could potentially be exposed when a thread crosses protection domain boundaries, and it is therefore critical to reset all such counters at domain crossings. Resetting the adaptation counters stops the current context's behavior from affecting that of the next context. By doing so, we might miss one opportunity to adapt. But we find that in steady-state, partition sizes do not change dramatically, and the performance effect of missing one adaptation opportunity is minimal.

4.5.3 Asymmetric SMT

While the adaptive partitioning scheme can help recover a significant chunk of the performance lost due to partitioning, it is still restrictive as it limits the pipeline

142



Figure 4.5. Adaptive Partitioning Examples.

resource utilization even in scenarios where at least one of the threads running on the processor are trusted or when no sensitive code is running and cross-thread information leakage may not be of concern. In this section, we describe a mitigation called *Asymmetric SMT* that allows two threads with asymmetric trust levels to securely share resources in an SMT processor, while preventing unauthorized information leakage from a higher security domain to a lower security domain.

With Asymmetric SMT, then, assuming active threads T_H and T_L, where T_H is at a higher security level than T_L, we have the ability to block the leakage from T_H to T_L, but allow leakage from T_L to T_H. An example where this is useful is sandboxing in web browsers. While it is not secure to leak information from the browser thread to the sandbox thread, it is safe to leak information from the sandbox to the browser thread. Similarly, it might be safe to leak information from a user process to a kernel process, from a guest virtual machine to the hypervisor, etc. Asymmetric SMT enables lost resource utilization due to partitioning, but the only beneficiary is the trusted thread. The performance of trusted threads is on the critical path for many applications, such as a web browser that runs untrusted Javascript. A study by the

Google v8 team [258] shows that only 20% of Chrome's page processing time is spent in running untrusted Javascript code, while the rest is spent in trusted browser code that performs tasks such as parsing, compilation, and garbage collection. Therefore, by improving the performance of the trusted part of the execution, Asymmetric SMT can significantly impact overall performance.

The key to Asymmetric SMT is that it allows the trusted thread to *borrow* unused pipeline resources from the untrusted user, only in cases where it can *instantly* return the resource when the untrusted thread needs it back. Fortunately, this instantaneous return is possible for many of the pipeline resources as the out-of-order pipeline is already well-equipped with mechanisms to deal with various squash events.

The rest of this section discusses how Asymmetric SMT can be enabled for various pipeline resources. We categorize the resources into stateful (e.g., ROB), stateless (e.g., functional units), and cache-like resources.

Stateful Resources

We refer to the resources that hold the transient execution state of a thread's instructions across multiple cycles and then get released, as stateful resources – this includes physical registers, IQ entries, ROB entries, and load/store queue entries. Asymmetric SMT allows the borrowing thread to use a free unused entry from the other thread's partition. Here, we use Physical Register File (PRF) as an example.

Figure 4.6 depicts two example scenarios for borrowing a physical register. In the initial state, three out of eight entries are assigned to T_H (assuming PRF already uses an adaptive partitioning scheme). Note that Asymmetric SMT is orthogonal to the other two partitioning schemes (static and adaptive) and can be added to either. Once Asymmetric SMT receives a request from the trusted thread, T_H, which has exhausted all of the entries in its partition, it checks if T_H can borrow an unused entry from T_L. Asymmetric SMT permits borrowing only if the number of T_L's free entries

is larger than a threshold (MIN_FREE), i.e., it always leaves some free slack registers. This condition is helpful at reducing the number of expensive squashes which results when T_L runs out of resources and one must be freed immediately by T_H.

Figure 4.6a shows the common-case scenario where we commit one of T_H's instructions before T_L's partition gets full. Note that in this case we are not borrowing a specific register, but rather allowing T_H's count to exceed its threshold. Thus, any T_H instruction that commits and frees a register will restore it to T_L. Figure 4.6b shows a scenario where T_L takes up all of the free entries in its partition before T_H gets the chance to return the borrowed register. In such a scenario, Asymmetric SMT immediately returns the borrowed register to T_L. It selects the youngest T_H instruction that holds a physical register and assigns that register to T_L. T_H then needs to squash that instruction and all of its subsequent instructions and restart execution from there.

Similarly, it is possible to allow a trusted thread to borrow instruction queue and load/store queue entries. During issue, if the trusted partition is full but the untrusted partition has more than MIN_FREE free entries, the trusted thread can borrow unused entries. If the untrusted partition becomes full before the trusted partition returns the borrowed instruction queue entry, the trusted thread should immediately return the borrowed entry.

Not all stateful resources are suitable for borrowing. For example, we can only allow borrowing of a limited number of ROB entries due to the instantaneous return requirement. If we borrow from the ROB more entries than what we can free per cycle (retire bandwidth), then the borrowing becomes visible to the owner thread and that leads to information leakage.

145



Figure 4.6. Borrowing a Physical Register in Asymmetric SMT. It shows the state of the physical register file over time for two scenarios.



Figure 4.7. Partitioning Schemes for Execution Units/Ports. Our adaptive partitioning and Asymmetric SMT architecture can reclaim the unused execution slots caused partitioning.

Stateless Resources

Examples of stateless resources include execution ports, functional units, fetch bandwidth, and commit bandwidth. Asymmetric SMT works well with stateless resources, as borrowing a stateless resource will not ever require an expensive squash. Here we use execution ports to describe Asymmetric SMT in stateless resources.

Figure 4.7 illustrates different temporal partitioning (multiplexing) schemes for execution ports. We can assign the whole dispatch bandwidth to one thread each

cycle, which reverts the pipeline to fine-grain multithreading [240] and sacrifices the benefits of SMT in eliminating horizontal waste [198]. A better approach is to multiplex individual execution ports each cycle, which can either be partitioned evenly or unevenly using our adaptive partitioning methodology. However, even with adaptive partitioning, there are cycles where T_L cannot fully utilize the ports that are assigned to it. In such cases, Asymmetric SMT utilizes the unused ports and borrows them for T_H. The scheduler for Asymmetric SMT, at each cycle, first tries to schedule all the ready instructions of T_L to the execution ports that are assigned to T_L at that cycle, then it assigns any unused T_L slots to T_H and schedules T_H's ready instructions. Note that for brevity, we do not show the non-pipelined functional units in Figure 4.7. Non-pipelined functional units (e.g., Intel's divider unit) are also borrowable, but as soon as we receive a request from the owner thread, we abort the execution of the borrowing thread and immediately start execution of the owner instruction. Most of the functional units in the Intel and AMD processors are pipelined.

Cache-like Structures

Cache-like structures are a special type of stateful resource. In the context of Asymmetric SMT, the major difference between these and other stateful resources is that caches, once warmed-up, do not have empty entries. Structures that fit in this category are the micro-op cache, the private TLBs, and the private data and instruction caches.

For these structures, Asymmetric SMT cannot enable borrowing of blocks without leaving a visible effect. To address this, we introduce a mechanism that invalidates entries in the untrusted (owner) thread's partition. This invalidation mechanism must be deterministic and independent of any requests by the trusted thread. Therefore, it does not leak information.

Similar to some of the methods proposed in architecture literature on cache

147

dead block prediction [75,115,168], we dynamically calculate the average reuse distance of cache blocks of the untrusted partition. Then, we multiply that average with a static parameter (*distance_coefficient* > 1) and use that as a threshold to distinguish between live and dead cache blocks. If the last access to a cache block is greater than the threshold, we invalidate that cache block so that it can be lent to the other thread. Again, only cache accesses of the untrusted thread can influence this invalidation, so it does not leak information about the access pattern of the other thread. We use a simple dead block detection mechanism that is only influenced by accesses, but more sophisticated dead block elimination methods can also be used [75, 115, 168]. Unlike other stateful resources, in caches returning a borrowed cache line does not incur any "squash" cost – it is a simple eviction. As soon as we receive an access from the owner thread, we invalidate one of the borrowing thread's cache lines (based on the cache replacement policy) and return it to the owner partition. For a write-back cache, we only allow a borrowed line to become dirty if the cache features a write-out-buffer (WOB) [117] – a buffer that handles the writes down to the memory hierarchy – so that we can guarantee the process of returning a dirty borrowed line is still instantaneous. In addition, we do not allow the number of dirty borrowed lines to grow larger than the size of the WOB.

Overheads

Overall, we find that borrowing from stateless resources (functional units and fetch bandwidth) does not impose significant overheads on the processor pipeline. In terms of area overhead, we only need to (1) make sure that instructions are tagged with one bit of thread ID across the pipeline (already necessary for other reasons), and (2) add very simple logic that checks if borrowing of a specific resource is allowed in each cycle, i.e., it checks that the current instruction belongs to the borrowing (trusted) thread and also there is no demand for the resource from the untrusted thread. This

simple logic, as shown in Section 4.7.5, does not impose any overhead on the cycle time, and has negligible power and area overheads. For stateless resources, Asymmetric SMT only uses an unused resource which will be lost if not utilized by borrowing. Therefore, the performance effect will always be positive. Borrowing stateful resources, however, may require additional flushing if the untrusted thread requests a borrowed resource. While this flushing imposes performance overhead to the borrowing thread, our results show that the improved utilization brought by borrowing greatly outstrips the flushing costs.

More than Two Threads

While SMT implementations with more than two threads are not common, our asymmetric SMT can be extended to those processors as well. In those cases, we can define a security lattice between trust domains and allow threads with higher trust levels to borrow from threads with lower trust levels. For example, a kernel thread can borrow from both a sandbox thread and a browser thread, while a browser thread can only borrow from the sandbox.

HW/SW Interface

Asymmetric SMT requires knowledge of the trust level of active threads running on the SMT processor. We envision two possible modes of operation for Asymmetric SMT. In the first mode, Asymmetric SMT relies on existing privilege levels. That is, without any software change, Asymmetric SMT enables the kernel to borrow resources from a user thread, or let the hypervisor borrow resources from any of the guest threads. The second mode allows the software to specify more fine-grain trust levels for the SMT threads. Thus, Asymmetric SMT needs to add and maintain new control registers that represent the trust level of an active thread. Privileged software would update the control register via a privileged instruction (e.g., x86's *wrmsr*). The second

Baseline Processor							
Frequency	3.3 GHz	I cache	32 KB, 8 way				
Fetch Width	16 B	D cache	32 KB, 8 way				
Fetch Policy	IQ count	Retire Width	8 uops				
Issue Width	8 uops	Decode Width	5 uops				
INT/FP Regfile	186/144 regs	IQ	97 entries				
LQ/SQ Size	64/36 entries	Functional	Int ALU(4), Mult(1),				
ROB Size	224 entries	Units	FPALU/Mult(2)				

Table 4.2. Architecture Detail for the Baseline x86 Core

mode also does not require extensive software changes as all the modifications required will be contained in the thread/process creation logic.

Quality of Service (QoS)

In addition to the evident security use case, Asymmetric SMT can also be leveraged for better and more versatile performance isolation (i.e., better QoS guarantees). The OS could mark a latency-critical thread as a high security/priority thread. Asymmetric SMT, then, improves QoS by assigning more resources to the latency-sensitive job with guaranteed (performance) protection of other jobs.

4.6 Methodology

This section details the experimental methodology for performance evaluation of the proposed partitioning schemes, including the Asymmetric SMT architecture. We also discuss the methodology we use for our covert-channel characterization framework presented in Section 4.4.

For performance evaluation, we model our partitioning scheme and Asymmetric SMT architecture in the gem5 v20 architectural simulator [173]. We add full support for SMT in gem5's out-of-order CPU model. We choose the parameters of our baseline architectures to resemble an Intel Skylake, except that, for a more intuitive comparison of partitioning schemes, our baseline architecture dynamically shares all the pipeline resources between the threads. The other exception is that the assignment of the functional units to the execution ports are slightly different than Skylake and more

closely resembles AMD processors where floating-point and integer units do not share a single port. Table 4.2 describes the detailed architectural configuration.

To characterize performance, we use the C and C++ benchmarks from the SPEC CPU2017 suite. These benchmarks are compiled at the -O3 optimization level using the LLVM compiler. Following the prevalent methodology for creating accurate and representative simulation points [13, 169, 234], we use PinPlay [211] and Simpoint [235] to select representative regions for simulation. For each of these benchmarks, we select the Simpoint region with the highest weight – the most representative region. We then make one multi-threaded checkpoint for every possible pair of the benchmark programs by combining their selected Simpoints. We run each pair twice: In the first experiment, we simulate until we complete 100 Million instructions from the first thread, then swap the threads and repeat. For example, we will run *lbm* and *perl* together twice. The effect of *perl* on *lbm* performance will be factored into the *lbm* bars in our graphs, and the effect of *lbm* on *perl* will appear in the *perl* bars.

The speedup numbers of different schemes are calculated as the ratio of the combined *instruction per cycle (IPC)* for each pair of the programs over the combined IPC of that pair of programs in the dynamically shared processor. Thus, for the Asymmetric SMT results, it accounts for both the sped-up trusted thread and the unaffected untrusted thread in the overall results. This is also equivalent to weighted speedup [241], a well-established performance metric for multiprogram workloads, where in this case the baseline is that thread's performance in a dynamically shared SMT processor. Weighted speedup more accurately reflects useful performance gains, and avoids over-rewarding speedup of high-IPC threads. For adaptive partitioning we use at least 100,000-cycle adaptation intervals, unless otherwise noted. The performance bars that represent Asymmetric SMT are the results of applying Asymmetric SMT together with adaptive partitioning.

In addition to the SPEC benchmarks, to evaluate Asymmetric SMT in a more

realistic scenario, we model a setting that resembles the computation of web browsers: running untrusted Javascript codes on one thread and sensitive cryptography computations on the other. On the first thread, we run Javascript programs from SunSpider [249] benchmarks on Duktape [74] Javascript engine, and on the second thread, we run *RSA* and *AES GSM* benchmarks from Wolf SSL v4.5.0 [294].

We evaluate the performance of our proposed mitigations by applying them to a wide range of pipeline resources, including the Instruction Queue, the Load Queue, the Store Queue, the integer and vector physical register files, the ROB (Adaptive only), the instruction and data TLBs, the instruction and data caches, Branch Target Buffer, fetch and decode bandwidth, commit bandwidth, and the execution units.

We use Verilog HDL to implement different partitioning schemes on an example structure (Dispatch Unit). To that end, We use Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [1] to synthesize and obtain timing, area, and power information. The results of this analysis are discussed in Section 4.7.5.

For covert-channel characterization experiments we build our microbenchmarks on Agner's test infrastructure [83]. We run our experiments on various processors. Specifically, we use AMD Ryzen Threadripper 396oX (Zen2), Intel Xeon E3-1230 (Ivy Bridge), and Intel Core i7-6770HQ (Skylake). To measure the bandwidth and error rate of the covert channels, we transfer a pseudorandom bit sequence which is generated using a 15-bit wide linear feedback shift register (LFSR). This allows us to identify various errors that might happen during the transmission, including bit loss, multiple insertions of bits, or bit swaps [167]. To estimate the error rate, we use Levenshtein edit distance between the sent and received data for the pseudorandom bit sequence.



Figure 4.8. Covert Channels between a Spy (To) and a Trojan (T1) Thread. In a fully shared pipeline the instructions executed on T1 have a clear effect on To's execution time. *Partitioned, Adaptive* and *Asymmetric* are always constant and share the same straight line.

4.7 Results

This section characterizes our mitigation strategies. We first present the security evaluation, followed by performance.

4.7.1 Security Evaluation

To show the effectiveness of our mitigation strategies in stopping covert channels, we perform a study similar to Covert Shotgun [84]. In this experiment, a spy thread constantly executes one type of instruction and measures its timing. The trojan thread tries to send a signal by executing different instructions, thereby varying the contention on various pipeline resources. Figure 4.8 shows the results of this experiment. In a fully shared pipeline, the timings of the spy process can be clearly influenced by the trojan's instructions. An attacker, therefore, can pick any pair of instructions that



Figure 4.9. Performance of the Proposed Schemes.

show a different effect on the spy process to create a covert channel. When we enable any of our mitigation strategies, the spy thread timings become constant, effectively stopping all identified covert channels. For adaptive partitioning, all measurements are within one adaptation period. For Asymmetric SMT, the trust levels are set so that only the trojan thread can borrow resources. Further examination of the experiments shows that the covert channels in the fully shared pipeline are created by contention on mainly two resources: the fetch bandwidth and the functional units. We observe similar results when the spy uses different instructions.

Note that while this study shows that our mitigation strategies completely stop the covert channels that can be found with this approach, this test does not give complete coverage of all shared resources, particularly structures not documented by Intel or AMD. Also, these are the results of simulation (the only way to evaluate most new hardware mitigation techniques), and a real processor may contain other leaks not simulated.

4.7.2 Performance Evaluation

Figure 4.9 shows the results of our mitigation schemes applied to the pipeline resources mentioned in Section 4.6. Each bar represents the average results of running a benchmark on one thread with each of the other benchmarks on the other SMT thread. Static partitioning of the pipeline resources, as expected, imposes a significant performance cost. On average, it slows the execution by 10% compared to dynamically-



Figure 4.10. Running Trusted Cryptography Computation with Untrusted JavaScript Code.

shared resources. The performance overhead goes as high as 24% for some benchmarks (*mcf*). However, for one program (*lbm*), static partitioning significantly improves the performance. That is because *lbm* frequently exhausts the entire store queue on a dynamically shared pipeline, which causes the other thread to stall due to lack of store queue entries. In this case, *lbm* gets no benefit from more queue entries, and only gets extra interference by causing the other thread to get backed up. Therefore, statically partitioning the store queue achieves better performance for *lbm*, and our schemes further accentuate that advantage. Our adaptive partitioning reduces the performance overhead of partitioning to only 2% on average (5% if we ignore *lbm*), and consistently reduces the performance overhead of partitioning across all the benchmarks.

Asymmetric SMT further improves the performance and even provides a 2% speed-up over the shared pipeline (thanks again to *lbm*). But even excluding *lbm*, Asymmetric SMT almost fully restores the performance of a fully shared pipeline. These results show that opportunistically borrowing resources is highly effective at maintaining high utilization of partitioned resources. Note that, as mentioned in



Figure 4.11. Partitioning Schemes for Functional Units.

Section 4.6, for each pair of the benchmarks, we run the experiment twice. In each run, a different benchmark is considered as the trusted (borrower) thread in the Asymmetric SMT experiments.

Figure 4.10 shows the performance of the Asymmetric SMT architecture in a different, more realistic setting. On one thread, the SMT processor runs the SunSpider Javascript benchmark on Duktape engine, and on the other thread, it runs a trusted cryptography benchmark from the WolfSSL suit. This resembles computation that a web browser might perform. Asymmetric SMT allows the trusted threads (AES and RSA in this case) to borrow resources from the untrusted threads. The combination of our adaptive partitioning and Asymmetric SMT, on average, reduces the overhead of partitioning from 24% to 11% for RSA. For the AES benchmark, Asymmetric SMT not only completely restores the performance overhead of static partitioning, but also outperforms the fully shared baseline by 7% on average. The performance gain mostly comes from the pairs of benchmarks for which static partitioning performs better than the fully shared pipeline, such as *regexp-dna*. These benchmarks exhibit frequent resource full events (e.g., high number of physical register full in *regexp-dna*) that stall



Figure 4.12. Partitioning Schemes for Fetch Bandwidth.

both threads in a shared pipeline. In such cases, partitioning allows one thread to continue execution and thus improves overall performance.

Next, we take a closer look at the performance of the proposed schemes by examining their effects on the individual pipeline resources. Among the resources that we partition, we find that the most significant contributor to the performance cost is the execution ports/functional units. Figure 4.11 shows the results of an isolated experiment where all resources are dynamically shared except the execution ports. We examine four partitioning schemes for the execution ports: two different static partitioning schemes as well as our proposed adaptive and asymmetric SMT.

Multiplexing the dispatch bandwidth refers to the method where we only dispatch instructions from one thread each cycle. It severely impacts performance. On average, it reduces the performance by 12%, compared to dynamically shared execution ports, and is as high as 19% for some benchmarks. The main reason for such poor performance is that, at each cycle, there are not enough instructions from only one thread to fully utilize the execution ports. Another scheme is *multiplexing individual functional units* instead of the dispatch bandwidth as a whole (described in Section 4.5.3). This is also



Figure 4.13. Partitioning Schemes for Caches.

the default static partitioning scheme used for the summary results of Figure 4.9. This improves the utilization of the execution ports over dispatch bandwidth multiplexing. However, the cost of static multiplexing of the functional units is still high (10%, on average) compared to shared execution ports. Adaptive partitioning is able to reduce that overhead to only 6%. Adaptive partitioning, even with extremely long adaptation intervals, is highly effective for the execution ports as different programs naturally exhibit different usage distributions for different functional units. Asymmetric SMT reduces this overhead even further to only 1%.

The next big contributor to the performance cost is the fetch bandwidth. Figure 4.12 shows the results of another isolated experiment where we only partition the fetch bandwidth and keep other resources dynamically shared. The baseline shared architecture uses ICount [265] to dynamically determine the best thread to fetch from. However, as Section 4.4.2 shows, this can be used to construct covert channels. One way to mitigate that is to use a strict round-robin scheme (*partitioned* bars in the figure) where the fetch unit alternates between the threads each cycle. A partitioned fetch policy does not have the ability to choose the fetching thread based on dynamic conditions each cycle; therefore, it loses 7% performance overhead, on average. We get small gains from adapting the partition (and so restore some small amount of the dynamic adaptation), then a bit more from asymmetric SMT, which enables us to retrieve some of the unused fetch bandwidth.

Figure 4.13 shows the effect of our partitioning scheme applied to the cache hierarchy (L1 instruction and data, and L2 caches). On average, the static partitioning of the cache sets into two equally sized partitions imposes 6% performance overhead. This is greatly influenced by one benchmark (*mcf*) that has a large cache working set. Adaptive partitioning reduces the overhead to only 2%. It allocates more cache ways in each set to the thread that shows more misses during the adaptation period. Asymmetric SMT further reduces the overhead to only 1%.

One interesting aspect of these results, compared to Figure 4.9, especially considering results not shown for other individual resources, is that the performance costs incurred overall are far less than the sum of the costs for individual mitigations. This is expected on a well-balanced architecture, as these processors are designed to be. In a well-balanced architecture, restricting one resource but not others will always make that resource a bottleneck. But in a (hypothetically) perfectly balanced architecture, restricting all resources may have no more negative impact than restricting one.

4.7.3 Parameter Search for Adaptive

To select the best parameters for our adaptive partitioning and also to analyze the effects of each parameter on the performance, we conduct an exhaustive parameter search. Due to the combinatorial explosion problem, we cannot perform our parameter search study on all of the SPEC17 benchmarks. Therefore, we select four representative benchmarks: (1) a low ILP program that exhibits frequent memory accesses (*mcf*), (2),(3) two integer-heavy workloads (*deepsjeng*, *perl*), and (4) an FP-heavy workload



Figure 4.14. Parameter Search for Adaptive Partitioning of Three Example Stateful Resources.

(*lbm*). We then use all combinations of these benchmarks to conduct our parameter search. For each stateful pipeline resource, we examine five different adaptive limits (thresholds). For the stateless resources, we introduce another parameter, *Mult*. We increase the share of a thread, only if the full counters of that thread is larger than *Mult* times that of the full counters of the other thread – i.e., if *Mult* is 3, one counter must be more than 3X the other to cause a repartition. A higher *Mult* makes it more difficult to change the share of each thread. It particularly helps the resources where the cost of a wrong adaptation decision is high. In these experiments, we set the adaptation interval to 100,000 cycles.

Figure 4.14a shows the results of our parameter search for the physical register files. One benchmark pair that is sensitive to physical register file partitioning is *mcf-lbm*. For this combination, a limit higher than 0.7 causes a significant performance cost. That is because when the limit is high, one thread can potentially take up most of the physical registers. On the other hand, a limit of 0.6 hinders our ability to fully adapt physical register file allocation to different execution phases. Therefore, we use a limit of 0.7 for the physical register file. Similarly, Figure 4.14b and Figure 4.14c show the results of our parameter search for two other example stateful structures–Store Queue and Caches. An adaptation limit of 1.0 for the SQ severely impacts performance,



Figure 4.15. Parameter Search for Adaptive Partitioning of Two Example Stateless Resources. We only increase the share of a thread, if the full counter of that thread is larger than *Mult* times of the counter of the other thread.

as it allows one thread to potentially take up all the SQ. This performance impact is reduced as we reduce the limit. Therefore, we choose the smallest adaptation limit (0.6) for Store Queue. For caches, however, the threads can fully take advantage of a larger adaptation limit, and the performance would improve as we allow the partitions to grow larger. Thus, we choose the adaptation limit of 1.0 for caches.

Figure 4.15 shows the results of the parameter search for two example stateless resources–fetch bandwidth and functional units. The adaptation limit for stateless resources determines the maximum number of consecutive cycles that one thread can hold the resource before it is assigned to the other thread. The results suggest that, for both of these resources, the best performance is achieved when Mult is set to 1.2, and adaptation limit to 8.

4.7.4 Partitioning Effects on Other Resources

This section discusses the effects of our schemes on different stateful pipeline resources. Figure 4.16a shows the number of full register events for static and adaptive partitioning of physical register files (both integer and vector). That is the number of times the rename unit could not rename an instruction due to lack of physical registers. Partitioning, in general, significantly increases the number of full register events. However, the results show that adaptive partitioning can be highly effective at reducing the number of full register events. On average, it reduces the number of full register events by 63%.

Figure 4.16b shows the number of SQ full events. Dynamically sharing SQ results in a significant number of full events for all benchmarks. *lbm*, in particular, exhibits an exceptionally high pressure on SQ, resulting in $5\times$ more full SQ events than the average. Static partitioning of SQ reduces the number of SQ full events to almost half compared to shared, while our adaptive partitioning reduces that even further to 37%, on average.


Figure 4.16. Impact of Partitioning Schemes on Individual Resource

Table 4.3. Delay, Area, and Power Results for Different Implementation of the Dispatch Unit.

Module	Delay (ns)	Area (μm^2)	Static Power (mW)	Dyn. Power (mW)
Shared Dispatch	0.955	7567	0.168	4.118
Partitioned Dispatch	0.951	7660	0.170	4.900
Asymmetric Dispatch	1.037	12147	0.284	6.508
Mult 32×32	1.318	7597	0.163	6.835

Figure 4.16c shows the effects of adaptive partitioning on the data TLB miss rates. On average, our adaptive partitioning scheme reduces the number of data TLB misses by 14% compared to a static partitioning baseline.

Similarly, Figure 4.16d compares the number of full IQ events for different partitioning schemes. Static partitioning, in general, significantly increases the number of the IQ full events (by $3\times$). However, our adaptive partitioning exhibits 11% fewer IQ full events than static partitioning.

4.7.5 RTL Model of Asymmetric SMT

To fully evaluate the effects of resource borrowing on cycle time, power, and area (and to supplement our simulation-based performance results), we implement different partitioning schemes on an example resource in Verilog HDL. To that end, we choose the dispatch unit, the biggest contributor to the performance loss of partitioning and likely the most latency-critical, for which we implement three different sharing schemes: fully shared dispatch, partitioned dispatch, and Asymmetric dispatch. Fully shared dispatch assigns the functional units to the instructions marked "ready" in the queue, in a simple first-in-first-out fashion. The partitioned dispatch is similar to the shared, but it only dispatches instructions from a single thread each cycle. The Asymmetric dispatch is similar to the partitioned dispatch, but it assigns any unused functional units to the trusted thread. We then use the Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [1] to synthesize and obtain timing, area, and power information.

Table 4.3 shows the post-synthesize analysis of different implementations of the dispatch unit. Our Asymmetric scheme increases the delay of the dispatch unit from 0.955 *ns* to 1.037 *ns*. However, this 8.6% extra overhead does not affect the processor's cycle time, as it is not enough to put the dispatch unit on the critical path of the whole processor core. As an example, we show that the (pipelined) integer multiplication unit has a longer delay. To see this, we implement a three-cycle multiplication module [17], imitating Skylake's three-cycle integer multiplication design. Our results show that the delay of our Asymmetric dispatch is still significantly smaller than the cycle time determined by the multiplier (the longest of the three stages); thus, our Asymmetric dispatch will not affect the cycle time.

Asymmetric dispatch covers a 16% larger area compared to a shared dispatch unit. However, this is also not a matter of concern as the dispatch unit constitutes only a tiny fraction of a modern processor's die area. The Skylake core, for example, has an area of 8.73 mm^2 [292]. The asymmetric dispatch overhead, thus, will be only 0.051% of the core area (calculated conservatively, not accounting for the technology node differences). Similarly, the power overhead is also not considerable compared to the total power consumption of an out-of-order core, which could be in the order of tens of Watts.

4.8 Conclusion

This chaoter provides the first comprehensive and exhaustive analysis of sharingbased security vulnerabilities in modern, high-performance SMT processors. This analysis shows that despite the fact that many resources are statically partitioned, there still remain many resources that are dynamically shared and present high bandwidth leakage channels. Among the channels identified are some previously unknown, including fetch bandwidth dynamic sharing and dynamically shared issue bandwidth, each enabling channels of over 500 Kbps.

This work also examines some novel, unified approaches to mitigation that can be applied throughout the pipeline. These provide high isolation between threads (allowing collectively a few bits of leakage over, for example, 100,000 cycles) while retaining most of the performance of a fully dynamically shared, insecure SMT implementation. Adaptive partitioning gets within 5% of shared SMT, and asymmetric SMT, which further enables unfettered performance of a trusted thread in the presence of an untrusted, all but eliminates the loss.

It is common for SMT execution to be disabled in security-critical code, or in the presence of frequent untrusted execution streams. This work shows that SMT contention-based vulnerabilities can be reduced below the level of other known vulnerabilities, making SMT execution a viable alternative for secure execution. We do so while still preserving the bulk of the performance benefit of SMT.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. Chapter 4, in full, is a reprint of the material as it appears in USENIX Security Symposium (USENIX Security) 2022. Taram, Mohammadkazem; Ren, Xida; Venkat, Ashish; Tullsen, Dean. The dissertation author was the primary investigator and author of this paper.

Chapter 5 Conclusion and Future Directions

In modern processor architectures, the ever-present tension between security and performance has been considerably growing in recent years as new attacks appear, each exploiting a crucial performance optimization in the processor, threatening to unwind decades of architectural gains. Speculative execution, shared caches, branch prediction, shared execution units, etc. are exploited in well-established classic attacks, as well as recent instances. Turning off any of these features would be crippling to performance, so we seek to design new architectures to continue to enable the optimization but with higher levels of protection.

To that end, in this dissertation we first look at the new practical ways that microarchitecure can be exploited in real attacks, deepening our understanding of how microarchitectural optimizations can leak important information. Then we turn our attention to defusing this tension with secure high-performance microarchitectures that can preserve critical optimizations, such as speculative execution and simultaneous multithreading, but at the same time provide protections against those vulnerabilities. We also demonstrate how microarchitectural techniques can be utilized to provide more efficient security solutions.

This dissertation addresses the question of how we can disentangle the inherent tension between security and performance. This dissertation shows that, at the mi-

croarchitecture level, where the sources of these vulnerabilities reside, it is possible to ease this tension with secure microarchitectural techniques. This dissertation proposes techniques that allow us to flexibly reach desirable points in the security-performance space.

5.1 Future Directions

This dissertation has unlocked several new secure high-performance architectures, including new secure multithreading architectures. These secure architectures have the potentials to change the landscape of microarchitectural optimizations. Consider multithreading and aggressive speculation, which are the two major alternatives to achieve higher utilization of the pipeline. Security concerns may have led to a more heavy reliance on speculation than higher levels of multithreading; however, this resulted in the opposite effect, as it opened a Pandora's box of new vulnerabilities. This dissertation, however, has shown that multithreading's vulnerabilities are much easier to cap. As a result, it behooves us to completely rethink the balance of multithreading and speculation, and consequently, the whole suite of tools for parallelism.

While the attacks we present in this dissertation represent a substantial leap in understanding how microarchitectural optimizations can be exploited to leak information, it is only one step in that direction. In the future, to battle the continued slowing of Moore's Law, architects will likely begin to more aggressively introduce and adopt new paradigms such as more complex optimizations, heterogeneity, specialization, and new memory technologies. Any of these, if not designed carefully, will have the potential to bring about devastating security vulnerabilities. Following on from the offensive microarchitecture research presented in this dissertation, we need to explore the security implications of these emerging architectures. To that end, we need to investigate tools that architects can use to reason about the security implications of their proposed optimizations. Such tools need to be integrated into architects' everyday simulation methodology (e.g., gem5 architectural simulator) for encouraging widespread adoption. However, for emerging architectures that are past the prototyping stage and are making their ways into productions systems, we need to take a different approach. We need to understand them (if necessary, by reverse engineering) and unveil their vulnerabilities before they become rampant. We need to continue offensive security research on other new technologies so we can avoid a new silent issue, such as Spectre, to get into our large-scale production systems.

Bibliography

- [1] Nangate open cell library.
- [2] Christopher Abad. Ip checksum covert channels and selected hash collision, 2001.
- [3] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.
- [4] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, 2007.
- [5] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.
- [6] Onur Acuiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2007.
- [7] O. Aciicmez and J. Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2007.
- [8] Onur Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture,* 2007.
- [9] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers Track at the RSA Conference*, 2007.
- [10] Onur Acuiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the aes. In *Cryptographers Track at the RSA Conference*, 2007.
- [11] Shaizeen Aga and Satish Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 94–106, New York, NY, USA, 2017. ACM.

- [12] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture (ISCA)*, 2020.
- [13] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019.
- [14] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garca, and N. Tuveri. Port contention for fun and profit. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [15] AMD. White paper: Software techniques for managing speculation on amd processors. Technical Report REVISION 1.24.18, January 2018. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.
- [16] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: Efficient iommu emulation. In *USENIX Annual Technical Conference*, 2011.
- [17] R Aneesh and Sarin K Mohan. Design and analysis of high speed, area optimized 32x32-bit multiply accumulate unit based on vedic mathematics. *International Journal of Engineering Research and Technology*, 3(4), 2014.
- [18] ARM. Arm security technology building a secure system using trustzone technology. 2009.
- [19] ARM. Whitepaper: Cache speculation side-channels. Technical Report Version 2.2, July 2018. https://developer.arm.com/-/media/developer/pdf/ Security%20update%2010%20July%2018/Cache_Speculation_Side-channels-v2. 2.pdf?revision=7de26366-a49f-4c23-85f0-34e8f5e38881.
- [20] Taylor Armerding. The 18 biggest data breaches of the 21st century, 2018.
- [21] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In Proceedings of the 10th European Conference on Computer Systems, April 2015.
- [22] Harsha Basavaraj. A case for effective utilization of direct cache access for big data workloads. Master's thesis, UC San Diego, 2017.
- [23] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.
- [24] Sandeep Bhatkar and R Sekar. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2008.

- [25] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.
- [26] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.
- [27] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, 1997.
- [28] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The M5 Simulator: Modeling Networked Systems. *Micro*, IEEE, 2006.
- [29] Darrell Boggs, Gary Brown, Nathan Tuck, and K. S. Venkatraman. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro*, 2015.
- [30] James E.J. Bottomley. Linux kernel dma api, 2019. Online: https://www.kernel. org/doc/Documentation/DMA-API.txt, Accessed on Aug 2019.
- [31] Lucy Bowen and Chris Lupo. The performance cost of software-based security mitigations. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020.
- [32] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In USENIX Annual Technical Conference (USENIX ATC), 2010.
- [33] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. Performance analysis and predictability of the software layer in dynamic binary translators/optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers*, 2013.
- [34] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr.sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.
- [35] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.
- [36] Yuriy Bulygin. Cpu side-channels vs. virtualization malware: the good, the bad or the ugly. *ToorCon: Seattle, Seattle, WA, US,* 2008.

- [37] Serdar Cabuk, Carla E Brodley, and Clay Shields. Ip covert timing channels: design and detection. In *11th ACM conference on Computer and communications security* (*CCS*), 2004.
- [38] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report MSR-TR-2008-120, Microsoft Research, 2008.
- [39] Chandler Carruth. Mitigating Speculative Attacks in Crypto. https://github.com/HACS-workshop/spectre-mitigations/blob/master/ crypto_guidelines.md, 2018. Online; accessed Jul 2018.
- [40] Chandler Carruth. RFC: Speculative Load Hardening (a Spectre variant1 mitigation). https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_ 61e_K03TmoCS3uXLcJRo/edit, 2018. Online; accessed Jul 2018.
- [41] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 147160, USA, 2006. USENIX Association.
- [42] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.
- [43] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [44] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In 2008 International Symposium on Computer Architecture, pages 401– 412, June 2008.
- [45] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, and Todd C. Mowry. Logbased architectures: Using multicore to help software behave correctly. SIGOPS Oper. Syst. Rev., 45(1):84–91, February 2011.
- [46] Tao Chen, Alexander Rucker, and G. Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.
- [47] Long Cheng, Fang Liu, and Danfeng (Daphne) Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *WIREs Data Mining and Knowledge Discovery*, 7(5):e1211, 2017.

- [48] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Software-based gate-level information flow security for iot systems. In *Proceedings* of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50 '17, pages 328–340, New York, NY, USA, 2017. ACM.
- [49] Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. Smokebomb: Effective mitigation against cache side-channel attacks on the arm architecture. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.
- [50] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.
- [51] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on sel4. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.
- [52] Jonathan Corbet. The current state of kernel page-table isolation. https://lwn. net/Articles/741878/, 2017. Online; accessed Jan 2018.
- [53] Marc L Corliss, E Christopher Lewis, and Amir Roth. Dise: A programmable macro engine for customizing applications. In *Computer Architecture*, 2003. *Proceedings.* 30th Annual International Symposium on, 2003.
- [54] Marc L Corliss, E Christopher Lewis, and Amir Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *11th International Symposium on High-Performance Computer Architecture*, 2005.
- [55] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using dise to protect return addresses from attack. *SIGARCH Comput. Archit. News*, March 2005.
- [56] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.
- [57] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.
- [58] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. Talx86: A realistic typed assembly language. In 1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA, 1999.

- [59] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM.
- [60] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM.
- [61] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 171–191, 2018.
- [62] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.
- [63] Debian. Debian microcode update. https://wiki.debian.org/Microcode, 2017.
- [64] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003.
- [65] Dell. Poweredge t620 technical guide, 2013. Online: http://i.dell.com/sites/ doccontent/shared-content/data-sheets/en/Documents/dell-poweredge-t620technical-guide.pdf, Accessed on Aug 2019.
- [66] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Sidechannel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, 2012.
- [67] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure tlbs. In *International Symposium on Computer Architecture (ISCA)*, 2019.
- [68] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In USENIX Security Symposium (USENIX Security), 2020.
- [69] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

- [70] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision 13 cache attack using intel TSX. In *26th USENIX Security Symposium (USENIX Security)*, 2017.
- [71] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, virtual ghosts, and hardware support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, pages 5:1–5:9, New York, NY, USA, 2018. ACM.
- [72] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In 27th USENIX Security Symposium (USENIX Security 18), pages 1441–1458, Baltimore, MD, 2018. USENIX Association.
- [73] Steven Dropsho, Volkan Kursun, David H. Albonesi, Sandhya Dwarkadas, and Eby G. Friedman. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002.
- [74] Duktape. Duktape javascript engine, 2020.
- [75] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *International Symposium on Microarchitecture (MICRO)*, 2012.
- [76] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *International Symposium on Computer Architecture* (ISCA), 2021.
- [77] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 2001.
- [78] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [79] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [80] H. Farrokhbakht, M. Taram, B. Khaleghi, and S. Hessabi. Toot: an efficient and scalable power-gating method for noc routers. In 2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS), 2016.

- [81] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, Jr., and Dejan Kostić. Make the most out of last level cache in intel processors. In *14th EuroSys Conference* (*EuroSys 19*), 2019.
- [82] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.
- [83] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and microoperation breakdowns for intel, amd and via cpus. http://www.agner.org/ optimize/instruction_tables.pdf.
- [84] Anders Fogh. Covert shotgun, 2016.
- [85] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2001.
- [86] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.
- [87] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via lowbandwidth acoustic cryptanalysis. In *International Cryptology Conference*, 2014.
- [88] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *Journal of Cryptology*, 2016.
- [89] C. Gray Girling. Covert channels in lan's. IEEE Transactions on software engineering, 13(2):292, 1987.
- [90] Will Glozer. wrk a HTTP benchmarking tool. https://github.com/wg/wrk. Online; accessed Jun 2018.
- [91] Google. Product status: Microarchitectural data sampling (mds), 2019.
- [92] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.
- [93] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *Network and Distributed Systems Security Symposium (NDSS)*, 2020.
- [94] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leakaside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In USENIX Security Symposium (USENIX Security), 2018.

- [95] Joseph L Greathouse, Zhiqiang Ma, Matthew I Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In ACM SIGARCH Computer Architecture News, 2011.
- [96] Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A case for unlimited watchpoints. In *ACM SIGARCH Computer Architecture News*, 2012.
- [97] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.
- [98] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2016.
- [99] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In 24th USENIX Security Symposium (USENIX Security), 2015.
- [100] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. Reliability-aware data placement for heterogeneous memory architecture. In *High Performance Computer Architecture* (HPCA), 2018 IEEE International Symposium on, 2018.
- [101] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization*, 2001.
- [102] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In 21st Annual Computer Security Applications Conference (ACSAC'05), pages 9 pp.–311, Dec 2005.
- [103] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, Jan 2008.
- [104] Theodore G Handel and Maxwell T Sandford. Hiding data in the osi network model. In *International Workshop on Information Hiding*, 1996.
- [105] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In Proceedings of the 2Nd International Symposium on Memory Management, 2000.
- [106] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, September 2006.
- [107] A Hintz. Covert channels in tcp and ip headers, 2002. Presentation at DEFCON.

- [108] Jonathan J Hoch and Adi Shamir. Fault analysis of stream ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2004.
- [109] Derek L. Howard and Mikko H. Lipasti. The effect of program optimization on trace cache efficiency. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.
- [110] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar cisc processors. In *The Twelfth International Symposium on High-Performance Computer Architecture*, 2006., 2006.
- [111] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.
- [112] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.
- [113] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 2012.
- [114] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004.
- [115] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *International Symposium on Computer Architecture (ISCA)*, 2002.
- [116] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *32nd International Symposium on Computer Architecture* (*ISCA*), 2005.
- [117] Herbert HJ Hum. Dirty line cache, June 20 2000.
- [118] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [119] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

- [120] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.
- [121] Intel. Software guard extensions programming reference. 2014.
- [122] Intel. White paper: Intel analysis of speculative execution side channels. Technical Report 336983-001, Revision 1.0, January 2018. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.
- [123] Intel. White paper: Retpoline: A branch target injection mitigation. Technical Report 337131-003, Revision 003, June 2018. https: //software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.
- [124] Intel Corporation. 2nd Generation Intel Core vPro Processor Family, 2008. Available at http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf.
- [125] Intel Corporation. *Intel*[®] 64 and IA-32 Architectures Optimization Reference Manual. March 2009.
- [126] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, August 2011.
- [127] Intel Corporation. Intel[®] Data Direct I/O Technology (Intel[®] DDIO): A Primer, 2012.
- [128] Intel Corporation. Intel[®] ethernet controller i350 datasheet, 2017. https://www.intel.com/content/dam/www/public/us/en/documents/ datasheets/ethernet-controller-i350-datasheet.pdf.
- [129] Intel Corporation. Intel gigabit etherenet driver, 2019. Online: https://downloadcenter.intel.com/download/13663/Intel-Network-Adapter-Driver-for-82575-6-82580-I350-and-I210-211-Based-Gigabit-Network-Connections-for-Linux-, Accessed on June 2019.
- [130] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Euromicro Conference on Digital System Design*, 2015.
- [131] Van Jacobson, Craig Leres, and Steven McCanne. Tcpdump/libpcap. *Accessed: Aug 2019*, 23:2016, 1987. https://www.tcpdump.org.
- [132] Project Zero Jann Horn. Reading privileged memory with a sidechannel. https://googleprojectzero.blogspot.com/2018/01/reading-privilegedmemory-with-side.html, 2018. Online; accessed Jan 2018.

- [133] Hailin Jiang, M. Marek-Sadowska, and S. R. Nassif. Benefits and costs of powergating technique. In 2005 International Conference on Computer Design, pages 559–566, 2005.
- [134] Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–167, 2018.
- [135] Daniel Jurafsky and James H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall PTR, 1st edition, 2000.
- [136] Morteza Mohajjel Kafshdooz, Mohammadkazem Taram, Sepehr Assadi, and Alireza Ejlali. A compile-time optimization method for wcet reduction in realtime embedded systems through block formation. ACM Trans. Archit. Code Optim., 12(4):66:1–66:25, January 2016.
- [137] David Kanter. Sandy bridge-ep launches, 2012. https://www.realworldtech. com/sandy-bridge-ep/2/.
- [138] R. Karri, K. Wu, P. Mishra, and Yongkook Kim. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1509–1517, Dec 2002.
- [139] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In 54th Design Automation Conference (DAC), 2017.
- [140] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *53rd Design Automation Conference (DAC)*, 2016.
- [141] Michael Kerrisk. Linux programmer's manual. Accessed Aug 2019 http://man7. org/linux/man-pages/man7/raw.7.html.
- [142] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. arXiv preprint arXiv:1806.05179, 2018.
- [143] Ilhyun Kim and Mikko H. Lipasti. Implementing optimizations at decode time. In Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002.
- [144] Ilhyun Kim and Mikko H. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003.

- [145] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In 51st IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018.
- [146] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.
- [147] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler . https://www. paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. Online; accessed Jul 2018.
- [148] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In 40th IEEE Symposium on Security and Privacy (S&P), 2019.
- [149] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT 18), Baltimore, MD, 2018. USENIX Association.
- [150] Rakesh Kumar, Alejandro Martínez, and Antonio González. Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment. *ACM Trans. Archit. Code Optim.*, 2014.
- [151] Deepa Kundur and Kamran Ahsan. Practical internet steganography: data hiding in ip. *Multimedia and Security Workshop*, 2003.
- [152] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In 41st IEEE Symposium on Security and Privacy (S&P), May 2020.
- [153] Patrick P Lai, Ethan Schuchman, David Keppel, Denis M Khartikov, Polychronis Xekalakis, Joshua B Fryman, Allan D Knies, Naveen Neelakantam, Gregor Stellpflug, John H Kelm, et al. Apparatus and method for efficiently implementing a processor pipeline, September 10 2019.
- [154] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '08, pages 3–12, New York, NY, USA, 2008. ACM.
- [155] Butler Lampson. Perspectives on protection and security. In *SOSP History Day* 2015, SOSP '15, New York, NY, USA, 2015. Association for Computing Machinery.
- [156] Michael Larabel. Openbsd disabling smt / hyper threading due to security concerns, 2018.

- [157] Michael A. Laurenzano, Yunqi Zhang, Jiang Chen, Lingjia Tang, and Jason Mars. Powerchop: Identifying and managing non-critical units in hybrid processor architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.
- [158] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Microarchitecture (MICRO)*, 2014 47th Annual IEEE/ACM International Symposium on, pages 558–570. IEEE, 2014.
- [159] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. In ACM SIGPLAN Notices, volume 47, pages 383–394. ACM, 2012.
- [160] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, 2009.
- [161] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *USENIX Security Symposium (USENIX Security)*, 2019.
- [162] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
- [163] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *42nd International Symposium on Computer Architecture (ISCA)*, 2015.
- [164] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.
- [165] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security), 2018.
- [166] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

- [167] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In 36th IEEE Symposium on Security and Privacy (S&P), 2015.
- [168] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *International Symposium on Microarchitecture (MICRO)*, 2008.
- [169] Gabriel H. Loh. 3d-stacked memory architectures for multi-core processors. In *International Symposium on Computer Architecture (ISCA)*, 2008.
- [170] Changbo Long and Lei He. Distributed sleep transistors network for power reduction. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, 2003.
- [171] Jason Lowe-Power. Visualizing Spectre with gem5. http://www.lowepower.com/ jason/visualizing-spectre-with-gem5.html, 2018. Online; accessed Jun 2018.
- [172] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. Position paper: A case for exposing extra-architectural state in the isa. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '18, pages 8:1–8:6, New York, NY, USA, 2018. ACM.
- [173] Jason Lowe-Power et al. The gem5 simulator: Version 20.0+, 2020. [arXiv preprint arXiv:2007.03152].
- [174] Norka B Lucena, Grzegorz Lewandowski, and Steve J Chapin. Covert channels in ipv6. In *International Workshop on Privacy Enhancing Technologies*, 2005.
- [175] Niti Madan, Alper Buyuktosunoglu, Pradip Bose, and Murali Annavaram. A case for guarded power gating for multi-core processors. In *Proceedings of the* 2011 IEEE 17th International Symposium on High Performance Computer Architecture, 2011.
- [176] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO*, 1992.
- [177] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2109–2122, New York, NY, USA, 2018. ACM.
- [178] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: A tool to analyze speculative execution attacks and mitigations. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

- [179] Baolei Mao, Wei Hu, Alric Althoff, Janarbek Matai, Yu Tai, Dejun Mu, Timothy Sherwood, and Ryan Kastner. Quantitative analysis of timing channel security in cryptographic hardware design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1719–1732, 2017.
- [180] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *12th ACM Workshop on Hot Topics in Networks* (*HotNets*), 2013.
- [181] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Network and Distributed Systems Security Symposium* (NDSS), 2019.
- [182] Microsoft Security Response Center (MSRC) Matt Miller. Analysis and mitigation of speculative store bypass (CVE-2018-3639). https: //blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigationof-speculative-store-bypass-cve-2018-3639/, 2018. Online; accessed Jul 2018.
- [183] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, 2015.
- [184] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.
- [185] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, July 1996.
- [186] FatemehSadat Mireshghallah, Mohammad Bakhshalipour, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. Energy-efficient permanent fault tolerance in hard real-time systems. *IEEE Transactions on Computers*, 68(10):1539–1545, 2019.
- [187] Fatemehsadat Mireshghallah, Kartik Goyal, Archit Uniyal, Taylor Berg-Kirkpatrick, and Reza Shokri. Quantifying privacy risks of masked language models using membership inference attacks, 2022.
- [188] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Ali Jalali, Ahmed Taha Elthakeb, Dean Tullsen, and Hadi Esmaeilzadeh. Not all features are equal: Discovering essential features for preserving prediction privacy. In *Proceedings of The Web Conference*, WWW '21, April 2021.
- [189] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. Shredder: Learning noise

distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 318, New York, NY, USA, 2020. Association for Computing Machinery.

- [190] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. Privacy in deep learning: A survey, 2020.
- [191] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, 2017.
- [192] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.*, 47(4):538570, August 2019.
- [193] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In USENIX Security Symposium (USENIX Security), 2022.
- [194] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger sfi for the x86. In ACM SIGPLAN Notices, 2012.
- [195] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are aes x86 cache timing attacks still feasible? In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, 2012.
- [196] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *International Symposium on Microarchitecture (MICRO)*, 2017.
- [197] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.
- [198] Mario Nemirovsky and Dean M Tullsen. Multithreading architecture. *Synthesis Lectures on Computer Architecture*, 8(1):1–109, 2013.
- [199] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of the 12th Annual Network and Distributed System Security Symposium, 2005.
- [200] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner. Leveraging gate-level properties to identify hardware timing channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2014.

- [201] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. A practical testing framework for isolating hardware timing channels. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013.
- [202] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In USENIX Annual Technical Conference (USENIX ATC), 2018.
- [203] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *CoRR*, abs/1805.08506, 2018.
- [204] Cisco Security Research & Operations. Bandwidth, packets per second, and other network performance metrics. Accessed Aug 2019 https://www.cisco.com/c/ en/us/about/security-center/network-performance-metrics.html.
- [205] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In 22nd ACM Conference on Computer and Communications Security (CCS), 2015.
- [206] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2015.
- [207] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology (CT-RSA)*, 2006.
- [208] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the intel® architecture. In *Proceedings* of the 8th ACM International Conference on Computing Frontiers, CF '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.
- [209] Andrew Pardoe. Spectre mitigations in MSVC . https://blogs.msdn.microsoft. com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018. Online; accessed Jul 2018.
- [210] Joseph CH Park and Mike Schlansker. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California, 1991.
- [211] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2010.

- [212] Colin Percival. Cache missing for fun and profit, 2005.
- [213] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan apkun. Frontal attack: Leaking control-flow in sgx via the cpu frontend. In USENIX Security Symposium (USENIX Security), 2021.
- [214] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.
- [215] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *International Symposium on Computer Architecture (ISCA)*, 2019.
- [216] Ravi Rajwar, Martin Dixon, and Ronak Singhal. Specialized evolution of the general purpose cpu. In *CIDR*, 2015.
- [217] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital sidechannels through obfuscated execution. In 24th USENIX Security Symposium (USENIX Security), 2015.
- [218] Red Hat. Simultaneous multithreading in red hat enterprise linux, 2019.
- [219] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, (173), September 2008.
- [220] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead ops: Leaking secrets via intel/amd micro-op caches. In *International Symposium on Computer Architecture (ISCA)*, 2021.
- [221] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In 25th Network and Distributed System Security Symposium (NDSS), 2018.
- [222] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.
- [223] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Returnoriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 2012.
- [224] Chris Rohlf and Yan Ivnitskiy. Attacking clientside JIT compilers. *Black Hat,* USA, 2011.
- [225] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, 1997.

- [226] Craig H Rowland. Covert channels in the tcp/ip protocol suite. *First Monday*, 2(5), 1997.
- [227] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin K. Qureshi. Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the Twenty-sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.
- [228] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *International Symposium on Microarchitecture* (*MICRO*), 2019.
- [229] Gururaj Saileshwar and Moinuddin K. Qureshi. Lookout for zombies: Mitigating flush+reload attack on shared caches by monitoring invalidated lines. *CoRR*, abs/1906.02362, 2019.
- [230] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019.
- [231] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535*, 2018.
- [232] Sergio D Servetto and Martin Vetterli. Communication using phantoms: covert channels in the internet. In *IEEE International Symposium on Information Theory (ISIT)*, 2001.
- [233] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.
- [234] Rasool Sharifi and Ashish Venkat. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *International Symposium on Computer Architecture (ISCA)*, 2020.
- [235] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
- [236] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, 2003.
- [237] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In 28th USENIX Security Symposium (USENIX Security), 2019.

- [238] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC/Information Sciences Institute, 2007. Available online: http://netweb.usc.edu/ %7ersinha/pkt-sizes/.
- [239] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *International Symposium on Computer Architecture (ISCA)*, 2020.
- [240] Burton J Smith. Architecture and applications of the hep multiprocessor computer system. *Real-Time signal processing IV*, 298:241–248, 1982.
- [241] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2000.
- [242] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20:465488, 2018.
- [243] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.
- [244] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.
- [245] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.
- [246] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. Aegis: A single-chip secure processor. *Information Security Technical Report*, 2005.
- [247] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *Proceedings of the 32Nd Annual International Symposium* on Computer Architecture, 2005.
- [248] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [249] SunSpider. Sunspider javascript benchmarks, 2020.

- [250] Lin Tan, Ellick M. Chan, Reza Farivar, Nevedita Mallick, Jeffrey C. Carlyle, Francis M. David, and Roy H. Campbell. ikernel: Isolating buggy and malicious device drivers using hardware virtualization support. In 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC), 2007.
- [251] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. Dma cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance. In *16th International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.
- [252] Mohammadkazem Taram, Dean Tullsen, Ashish Venkat, Hossein Sayadi, Han Wang, Sai Manoj, and Houman Homayoun. Fast and efficient deployment of security defenses via context sensitive decoding. In *Government Microcircuit Applications and Critical Technology Conference (GOMACTech)*, 2019.
- [253] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In 45th Annual International Symposium on Computer Architecture (ISCA), 2018.
- [254] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive decoding: On-demand microcode customization for security and energy management. *IEEE Micro*, 39(3):75–83, 2019.
- [255] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
- [256] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet chasing: Spying on network packets over a cache side-channel. In *International Symposium on Computer Architecture (ISCA)*, 2020.
- [257] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Mitigating speculative execution attacks via context-sensitive fencing. *IEEE Design & Test*, 39(4):49–57, 2022.
- [258] Google V8 team. How v8 measures real-world performance, 2016.
- [259] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *38th International Symposium on Computer Architecture (ISCA)*, 2011.
- [260] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the

gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems,* ASPLOS XIV, pages 109–120, New York, NY, USA, 2009. ACM.

- [261] D. Townley and D. Ponomarev. Smt-cop: Defeating side-channel attacks on execution units in smt processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.
- [262] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018.
- [263] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 2010.
- [264] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *International Symposium on Microarchitecture* (*MICRO*), 2001.
- [265] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture (ISCA)*, 1996.
- [266] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *International Symposium on Computer Architecture (ISCA)*, 1995.
- [267] Paul Turner. Retpoline: a software construct for preventing branch-targetinjection. https://support.google.com/faqs/answer/7625886, 2018. Online; accessed Jul 2018.
- [268] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture*, 2004. MICRO-37 2004. 37th International Symposium on, pages 243–254, Dec 2004.
- [269] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-oforder execution. In USENIX Security Symposium (USENIX Security), 2018.
- [270] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

- [271] Stephan van Schaik, Alyssa Milburn, Sebastian sterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [272] Ashish Venkat. *Breaking the ISA Barrier in Modern Computing*. PhD thesis, UC San Diego, 2018.
- [273] Ashish Venkat, Arvind Krishnaswamy, Koichi Yamada, and Rajan Palanivel. Binary Translation driven Program State Relocation. In United States Patent Grant US009135435B2, 2015.
- [274] Ashish Venkat, S Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems, 2016.
- [275] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, 2014.
- [276] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In 2008 IEEE 14th International Symposium on High Performance Computer Architecture, pages 173–184, Feb 2008.
- [277] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture*, 2008. HPCA 2008. IEEE 14th International Symposium on, 2008.
- [278] Pepe Vila, Andreas Abel, Marco Guarnieri, Boris Kpf, and Jan Reineke. Flushgeist: Cache leaks from beyond the flush, 2020. [arXiv preprint arXiv:1409.0876].
- [279] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In NDSS, volume 2007, page 12, 2007.
- [280] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett. Brb: Mitigating branch predictor side-channels. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.
- [281] Robert Wahbe. Efficient data breakpoints. In ACM SIGPLAN Notices, 1992.
- [282] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B. Abu-Ghazaleh, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

- [283] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In 26th USENIX Security Symposium (USENIX Security), 2017.
- [284] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, 2016.
- [285] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference*, 2006. ACSAC'06. 22nd Annual, pages 473–482. IEEE, 2006.
- [286] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference (ACSAC)*, 2006.
- [287] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.
- [288] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.
- [289] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *International Symposium on Microarchitecture (MICRO)*, 2019.
- [290] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient outof-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [269].
- [291] Eric W. Weisstein. Cross-correlation. from mathworld–a wolfram web resource. lessons in digital estimation theory, 2019. Accessed Aug 2019 http://mathworld. wolfram.com/Cross-Correlation.html.
- [292] WikiChip. Skylake (client), 2020.
- [293] Emmett Witchel, Josh Cates, and Krste Asanović. *Mondrian memory protection*. ACM, 2002.
- [294] wolfSSL. wolfssl cryptography librar, 2020.
- [295] Troy Wolverton. Spectre and Meltdown are now a legal pain for Intel the chip maker faces 35 lawsuits over the attacks. https://www.businessinsider.com/35lawsuits-have-been-filed-against-intel-over-spectre-and-meltdown-2018-2, 2018. Online; accessed Aug 2018.

- [296] Henry Wong. Measuring reorder buffer capacity, may 2013.
- [297] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: Highspeed covert channel attacks in the cloud. In *21st USENIX Security Symposium* (*USENIX Security*), 2012.
- [298] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.
- [299] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In USENIX Security Symposium, pages 121–136, 2006.
- [300] M. Yan, Y. Shalabi, and J. Torrellas. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [301] M. Yan, J. Wen, C. W. Fletcher, and J. Torrellas. Secdir: A secure directory to defeat directory side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2019.
- [302] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. 2018.
- [303] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016. Online: https://cs.adelaide.edu.au/~yval/Mastik/, Accessed on Aug 2019.
- [304] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security)*, 2014.
- [305] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.
- [306] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.
- [307] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009.

- [308] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS* 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.
- [309] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *International Symposium on Microarchitecture (MICRO)*, 2019.
- [310] Sebastian Zander, Grenville Armitage, and Philip Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, 2007.
- [311] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using hardware features for increased debugging transparency. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.
- [312] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time sidechannel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses,* 2016.
- [313] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors, 2020. [arXiv preprint arXiv:2005.08183].
- [314] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *35th IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [315] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete gpus. In *Proceedings of the General Purpose GPUs (GPGPU)*, 2017.
- [316] Craig Zilles. Linked List Traversal Micro-Benchmark. http://zilles.cs.illinois. edu/llubenchmark.html, 2001. Online; accessed Jun 2018.