TECHNIQUES FOR SECURE AND EFFICIENT COMPUTING

A Thesis

by

LANG FENG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jiang Hu
Committee Members,	Jeff Huang
	Peng Li
	Jose Silva-Martinez
	Duncan Henry M. Walker
Head of Department,	Miroslav M. Begovic

August 2020

Major Subject: Computer Engineering

Copyright 2020 Lang Feng

ABSTRACT

Security and computing efficiency are two important aspects in the field of computer engineering. With the increasing complexity of modern computers, various security problems are exposed. Moreover, applications continuously present need for further computing efficiency. In this dissertation, techniques for both security and computing efficiency are studied. For hardware security, split fabrication is recognized as a promising approach to defense against attacks by untrusted foundries. The existing split fabrication methods mostly neglect manufacturability, which is an unavoidable challenge in nanometer technologies. Observing that security and manufacturability can be addressed in a synergistic manner, our research introduces routing techniques that can simultaneously improve both security and manufacturability. The effectiveness of these techniques is confirmed by experiments on benchmark circuits. For software security, Control-Flow Integrity (CFI) and Data-Flow Integrity (DFI) are effective defense techniques against a variety of memory-based cyber attacks. CFI and DFI are usually enforced through software methods, which entail considerable performance overhead. Hardware-based CFI techniques can largely avoid performance overhead, but typically rely on code instrumentation, which forms a non-trivial hurdle to the application of CFI. DFI often leads to even larger performance overhead comparing to CFI, and its real-world application has been quite limited. The overhead is intrinsically difficult to reduce unless the DFI verification criterion is lowered. We propose the hardware-based solutions for CFI and DFI verification, where FPGA and Processing-In-Memory (PIM) are leveraged, respectively. Experiments on popular benchmarks confirm that our designs can detect fine-grained CFI violations over unmodified binaries, and completely enforce the DFI defined in the original seminal work. The measurement results show an average of 0.36% performance overhead for CFI and an average $4\times$ performance overhead reduction for DFI on SPEC 2006 benchmarks. For computing efficiency, serverless or functions as a service runtimes offer an efficient and cost-effective mechanism for event-driven cloud applications. Training deep neural networks can be both compute and memory intensive. We investigate the use of serverless runtimes for neural network training

while leveraging data parallelism for large neural network models, show the challenges and limitations due to the data communication bottleneck, and propose modifications to the underlying runtime implementations that would mitigate them. For hyperparameter optimization of smaller deep learning models, we show that serverless runtimes can provide significant benefit.

DEDICATION

To my mother, my father, my grandfather, and my grandmother.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Jiang Hu, for his guidance and advice during my Ph.D. study. He shows me how to be an independent researcher, and I have learned a lot from his attitude toward work and research. I'm lucky to have him as my advisor, and it's almost impossible for me to finish the Ph.D. study without his guidance. I would like to thank Professor Jeff Huang, for his advice and help on my research projects. With his suggestions, I successfully addressed the hard research problems. I would like to thank my committee members, Professor Peng Li, Professor Jose Silva-Martinez and Professor Duncan Henry M. Walker of Texas A&M University, and the cooperators of my researches, Professor Jeyavijayan Rajendran, Professor Dilma Da Silva and Doctor Prabhakar Kudva, for their helpful suggestions on my researches. Thanks to my mates in Texas A&M University: Hao He, Jiafan Wang, Yujie Wang, Chaofan Li, Wenbin Xu, He Zhou, Justin Sun, Yanxiang Yang, Hongxin Kong, Nithyashankari Jayasan, Erick Carvajal Barboza, Yaguang Li, Rongjian Liang, Yishuang Lin, Kyungrak Choi, Jianfeng Song and Hailiang Hu. Thanks to Jiayi Huang and Peiming Liu for their kind help on my research projects. Finally, I would like to thank my parents and my girlfriend for their support during my Ph.D. study.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jiang Hu, advisor, and Professor Peng Li, Professor Jose Silva-Martinez of the Department of Electrical and Computer Engineering, and Professor Jeff Huang and Professor Duncan Henry M. Walker of the Department of Computer Science and Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was partially supported by NSF (CNS-1618824), NSF (CCF-1525749), SRC (2016-TS-2688), MOST (104-2628-E-007-003-MY3) and MOST (106-2918-I-007-008).

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGMENTS v

CONTRIBUTORS AND FUNDING SOURCES vi

TABLE OF CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xiv

1. INTRODUCTION 1

2. SECURE AND EFFICIENT CIRCUIT LAYOUT DESIGN 6

2.1 Previous Works 6
2.2 Preliminaries 7
2.2.1 Definitions 7
2.2.2 Attack Method 8
2.2.3 Framework of Routing for Security and Manufacturability 9
2.3 CMP-Friendly Routing Defense 9
2.3.1 Background on CMP 9
2.3.2 Layer Elevation 10
2.3.3 Motivation and Wire Selection 11
2.3.4 Wire Rerouting Method 13
2.4 SADP-Compliant Routing Defense 15
2.4.1 Background on SADP 15
2.4.2 Security Enhancement under SADP 16
2.5 Experiments 19
2.5.1 Results of CMP Friendly Defense 21
2.5.2 Results of SADP Compliant Defense 22
2.6 Conclusions 23

3. HARDWARE-ASSISTED SOFTWARE SECURITY 25

3.1 Previous Works 25
3.1.1 Control-Flow Integrity 25

3.1.1.1 Software-based Control-Flow Integrity ... 25
3.1.1.2 Hardware-based Control-Flow Integrity ... 25
3.1.2 Data-Flow Integrity ... 26
3.2 Hardware-assisted Control-Flow Integrity Verification ... 28
3.2.1 CFI and Control-Flow Graph ... 28
3.2.2 The Proposed System Design ... 29
3.2.2.1 System Platform ... 29
3.2.2.2 System Design Overview ... 29
3.2.2.3 Offline CFG Checker Generator ... 30
3.2.2.4 Trace Decoder ... 31
3.2.2.5 CFI Verification Module ... 32
3.2.2.5.1 CFG Checker ... 33
3.2.2.5.2 Verification Controller ... 35
3.2.2.6 CFG Compression ... 38
3.2.3 Hardware Implementation ... 44
3.2.3.1 Automatic Verilog Generation for CFG Checker ... 44
3.2.3.2 Pipelined Trace Decoder ... 50
3.3 Hardware-assisted Data-Flow Integrity Verification ... 52
3.3.1 Background ... 52
3.3.1.1 Data-Flow Integrity (DFI) ... 52
3.3.1.2 Processing-In-Memory (PIM) ... 54
3.3.2 Overview of the Proposed Approach ... 54
3.3.3 Software Program Instrumentation ... 57
3.3.3.1 Instrumentation for DFI Verification ... 58
3.3.3.2 Handling Library Functions ... 60
3.3.3.3 Function Return Protection ... 61
3.3.4 Hardware Design ... 62
3.3.4.1 DFI Packet Generation ... 62
3.3.4.2 Packet Transfer to PIM ... 64
3.3.4.3 Lossless Data Compression ... 65
3.3.4.4 Runtime Optimization ... 67
3.3.4.5 Implementation of the Optimizations ... 68
3.3.4.5.1 Circuit Design for Optimization C ... 68
3.3.4.5.2 Circuit Design for Optimization E ... 70
3.3.5 DFI Verification Program ... 70
3.3.6 Discussion ... 71
3.3.6.1 Static Analysis ... 71
3.3.6.2 The Role and Effect of Cache ... 72
3.3.6.3 Multithreading ... 73
3.4 Experiments and Results ... 74
3.4.1 Experiment Setup ... 74
3.4.2 Experiments and Results of Hardware-assisted CFI Verification ... 74
3.4.2.1 CFI without Code Instrumentation ... 74
3.4.2.2 Security ... 75
3.4.2.2.1 Fine-grained, stateful attacks. ... 76

3.4.2.3 Performance Overhead 78
3.4.2.4 Latency 78
3.4.2.5 Circuit Resource Use and Compilation Time 79
3.4.3 Experiments and Results of Hardware-assisted DFI Verification 83
3.4.3.1 Security 83
3.4.3.1.1 Comparison with Hardware-assisted Data-flow Isolation 83
3.4.3.1.2 RIPE Benchmark 85
3.4.3.1.3 Heartbleed 86
3.4.3.1.4 Nullhttpd 86
3.4.3.2 Performance and Circuit Overhead 87
3.4.3.3 Analysis of Optimizations 90
3.5 Conclusions and Future Research 90

4. EXPLORING SERVERLESS COMPUTING FOR MACHINE LEARNING MODEL TRAINING 92

4.1 Previous Works 92
4.2 Training Large Neural Network Models with Serverless 93
4.2.1 Data Transfer and Parallelism with Serverless 93
4.2.2 Data Parallelism for Neural Network Training 94
4.2.3 Optimizing Parallelism Structure for Serverless Training 95
4.2.4 Cost and Performance-Cost Ratio Optimization 99
4.3 Parallel Hyperparameter Tuning of Neural Network Models with Serverless 101
4.4 Experiments 102
4.4.1 Experiment Setup 102
4.4.2 Latency Variation 102
4.4.3 Structure Optimization 103
4.4.4 Training Accuracy and Convergence Rate 104
4.4.5 Result of Cost and Performance-Cost Ratio Optimization 105
4.4.6 Results on Hyperparameter Tuning 106
4.5 Opportunities in Serverless Runtime Design 106
4.6 Conclusion 108

5. SUMMARY AND CONCLUSIONS 109

REFERENCES 111

LIST OF FIGURES

FIGURE Page

2.1 Illustration for definitions. 8

2.2 Uneven surface after CMP due to non-uniform wire density [1]. The shaded rectangles indicate cross sections of metal wires. 10

2.3 (a) The original routing; (b) security and CMP-driven rerouting. (c) dangling stub for security. The blue squares indicate the decoy region for the net driven by gate A. 12

2.4 Routing graph. 13

2.5 SADP starts with dense lines generation as in (a). By cutting, the desired patterns are the purple parts in (b). 15

2.6 (a) Original layout with SADP violations. (b) Wire extension for both SADP compliance and security. 16

2.7 Wire extension enhances security in (a), but degrades security in (b). 17

2.8 Scenarios of different security implications by wire extension. 18

2.9 The source wire in the middle line and the four sink wires on the other lines belong to the same net. 19

2.10 Rerouting overhead. 22

2.11 Security versus % selected wires being rerouted. 22

2.12 Wire density variance change versus % selected wires being rerouted. 23

2.13 Delay and wirelength overhead versus % selected wires being rerouted. 23

3.1 Example of control-flow graph. 28

3.2 System platform for the proposed CFI. 30

3.3 System design overview: (a) offline CFG checker generator; (b) online CFI verifier. . 30

3.4 Architecture of CFI verification module. 32

3.5 State transition diagram of the controller. 36

3.6 An example of CFG optimization. 40

3.7 The Verilog template for CFG checker block with direct branch. 46

3.8 The Verilog template for CFG checker block with indirect branch with constant target. 47

3.9 The Verilog template for CFG checker block with indirect branch with unspecified target. 47

3.10 The Verilog template for CFG checker block with indirect branch with constant target for coalesced CFG node. 48

3.11 The Verilog template for CFG checker block with indirect branch with unspecified target for coalesced CFG node. 49

3.12 An example of grouping blocks in CFG checker. 50

3.13 Structure of the pipelined decoder. 50

3.14 An example of vulnerable code. 53

3.15 A code example for illustrating DFI. 53

3.16 (a) The architecture for software-based DFI enforcement. (b) The architecture for our hardware-based DFI enforcement. 56

3.17 The flow for DFI verification. 57

3.18 An example of code instrumentation. 59

3.19 The instrumentation for library functions. 60

3.20 Instrumentation for function return. 61

3.21 Operations of info-collector. 63

3.22 Operations of FIFO memory. 64

3.23 Examples of address locality. 66

3.24 Circuit for implementing optimization C. 69

3.25 Bitonic network for sorting 8 packets. 70

3.26 An example of multithreaded program with locks. 73

3.27 Code illustrating the stateful SP1 attack. 76

3.28 Code illustrating the fine-grained SP2 attack. 76

3.29 The runtime overhead on SPEC 2006 benchmarks. 78

3.30 The latency for FPGA to identify attacks. 79

3.31 An CFG compression example. 81

3.32 The number of ALMs per block of different benchmarks. 83

3.33 The relationship between the number of ALMs, compilation time and maximum number of blocks in one block group of 483.xalancbmk. 83

3.34 An example of vulnerability that HDFI cannot detect. 85

3.35 Tradeoff between performance overhead and transmission buffer size. 88

3.36 Impact of sampling rate on performance overhead. 89

3.37 The effectiveness of different optimization techniques. 89

3.38 The effect of transmission buffer size on data reduction. 90

4.1 Data transfer gateway between two serverless instances 94

4.2 Data parallelism by serverless computing 95

4.3 The relationship between data transfer latency and the number of gradients sets received by one parameter server. (experiment environment: the transferred data contains 42601 gradients, and parameter server is a 512MB serverless instance) 96

4.4 Different structures for merging gradients by parameter servers. 97

4.5 General structure of parameter servers 97

4.6 Reuse worker serverless instances as parameter servers. 99

4.7 Training accuracy vs. training time for Case A. 105

4.8 Training accuracy vs. training time for Case B. 105

4.9 Cost per iteration under different Lambda instance memory sizes and optimization results. 106

4.10 Performance-cost ratio per iteration under different Lambda instance memory sizes and optimization results. 106

4.11 Computing latency versus the number of searched hyperparameter sets $|\mathcal{H}|$ 107

4.12 Serverless affinity in runtimes 107

LIST OF TABLES

TABLE Page

2.1 Results of the original design, security only routing [2] and our CMP friendly security routing. ... 21

2.2 Results of SADP violations and SADP compliant security routing... 23

3.1 Security performance for different attack methods. ... 75

3.2 Resource use on SPEC 2006 benchmarks... 80

3.3 The performance overhead of DFI. †Computation time of optimizations and compression is neglected. ‡Computation time of optimizations and compression is considered... 84

3.4 Scenarios for the case of Figure 3.34 where HDFI fails... 85

4.1 AWS Lambda price vs. memory use. ... 99

4.2 Testcases ... 102

4.3 Latency of different structures ... 104

1. INTRODUCTION[1]

With the development of modern computers, computer hardware designs and software applications become rather complex. As a result, researchers and engineers pay more attention on computer security and efficiency. Due to the importance and the potential, our work focus on both computing security and efficiency, covering a wide range of fields.

For security, there are two kinds of security fields in computer engineering: hardware security and software security. For hardware security, our work targets at the potential new semiconductor fabrication technology called split fabrication. Semiconductor fabrication is drastically complex and expensive, and its business is increasingly concentrated to a few high-end foundries, which are often offshore. Offshore fabrication leads to various attacks such as piracy and Trojan insertion. Recently, the concept of split fabrication is proposed for security against untrusted foundries. By separating the fabrications of Front-End-Of-Line (FEOL) and Back-End-Of-Line (BEOL) at different foundries, the difficulty for attacks at a single foundry is considerably increased [3,4]. Even with the increased difficulty, however, a foundry may still be able to reverse engineer an entire design according to design conventions and therefore continue to launch security attacks [3,5,6]. Cell placement [5] and routing perturbation [2] techniques have been developed to further enhance security for split fabrication. However, almost none of the previous works on split fabrication considers the manufacturability issue, which is fundamentally important and cannot be ignored in practice. Usually, the security for split fabrication is improved by modifying circuit layout. However, layout modification affects manufacturability as semiconductor manufacturing process is usually sensitive to layout patterns. It may inadvertently degrade lithography printability, de-

[1]©2017 IEEE. Reprinted, with permission, from Lang Feng, Yujie Wang, Jiang Hu, Wai-Kei Mak and Jeyavijayan Rajendran, "Making Split Fabrication Synergistically Secure and Manufacturable", IEEE/ACM International Conference on Computer-Aided Design, 11/2017. ©2018 IEEE. Reprinted, with permission, from Lang Feng, Prabhakar Kudva, Dilma Da Silva and Jiang Hu, "Exploring Serverless Computing for Neural Network Training", IEEE International Conference on Cloud Computing, 07/2018. Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature, International Conference on Computer-Aided Design, "FastCFI: Real-Time Control Flow Integrity using FPGA without Code Instrumentation", Lang Feng, Jeff Huang, Jiang Hu and Abhijith Reddy, ©2019.

crease manufacturing yield, increase manufacturing cost or even cause unsolvable manufacturing hot-spots [7]. On the one hand, all the previous techniques that ensure the security of split manufacturing lead to DFM violations, making them not manufacturable. On the other hand, designs that do not have DFM violations need not be secure, because attackers can use design conventions as hints to retrieve the missing parts [3, 5, 6]. Thus, it is imperative to consider manufacturability in conjunction with security-driven layout modification.

On the other hand, software security is the same important as hardware security. Two popular regulations in the software security fields are Control-Flow Integrity (CFI) and Data-Flow Integrity (DFI). CFI [8] is to regulate instruction flow transitions, such as `branch`, toward target addresses conforming to the original design intention. Such regulation can prevent software execution from being redirected to erroneous address or malicious code. It is widely recognized as an effective approach to defend against a variety of security attacks including Return-Oriented Programming (ROP) [9] and Jump-Oriented Programming (JOP) [10]. DFI is a regulation to ensure that data to be accessed are written by legitimate instructions. As such, DFI verification can identify unwanted data modifications that are not consistent with programmer's intention. Therefore, it can detect various security attacks including control data attacks such as JOP and ROP, and non-control data attacks such as heartbleed [11] and nullhttpd [12]. Normally, most implementations of CFI and D-FI verification are based on software. Software-based CFI usually competes for the same processor resource as the software application being protected [8, 13–15], and therefore it tends to incur large performance overhead unless its resolution is very coarse-grained. Meanwhile, the concept of DFI was introduced in the seminal work of [16], and has received a lot of attention thereafter due to its potential of being a powerful security measure. However, a complete DFI enforcement by using software as in [16] incurs more than 100% performance overhead even though several optimization techniques have been applied. Indeed, a large overhead seems inevitable as every data access needs to be examined. Because of this intrinsic difficulty, there have been few follow-up works on DFI despite its widely recognized importance. This is in contrast to CFI, which has much more published studies. Alternatively, CFI and DFI can be realized through hardware-based enforce-

ment. Indeed, hardware-based CFI has attracted significant research attention recently [17–20]. Apart from relatively low overhead, some other issues of hardware CFI are worth a close look. For example, some hardware CFI systems are stateless, have low granularity, or require code instrumentation. The first two issues are for security in term of CFI coverage. The last one issue affects practical applications. To the best of our knowledge, there is no previous work that well addresses all of these issues along with low overhead. In our research, we propose a hardware-based CFI verification approach implemented on Field Programmable Gate Array (FPGA) named FastCFI. We also propose a hardware approach to realize the general and complete DFI as defined in the seminal work [16]. The main idea is to perform DFI verification in memory using the Processing-In-Memory (PIM) technology. The data transferring to memory for the verification is much less than those transferring to processor cores from memory required by the CPU-based software verification [16]. The considerable reduction of data transfer contributes to a large decrease of performance overhead. PIM was originally suggested for improving microprocessor performance [21], and recently becomes a very active research subject [22–24] as the progress of manufacturing process technology is making it close to practical applications. Besides the hardware and software designs for DFI verification in memory, lossless compression and runtime optimization techniques are developed to further reduce overhead. Library functions and function return addresses are also protected in our work. Moreover, circuit design techniques are proposed to make the hardware overhead to be at a reasonable level.

For computing efficiency, we combine two popular topics, which are cloud computing and machine learning. The former one has the ability to provide computing efficiency, while the computing efficiency is largely needed by latter one. As cloud computing increasingly becomes the platform of choice for commercial and scientific computing, serverless computing (also known as Functions as a Service or FaaS) [25], has emerged in recent years. Serverless applications can be either a set functions as code, triggered by some external event [26], or a larger application composed of multiple functions. A key reason for their success has been the cost efficiency that such runtimes provide in event-driven environments, where sporadic events may trigger computations,

and the users only pay for the compute time they consume [27], rather than have long running servers implemented on virtual machines for these event processing, which will cause idle waiting for those sporadic events and result in unwanted monetary cost. Serverless computing has been applied to the area of machine learning [28, 29] with mixed results. The technology has been shown to be particularly useful for inference and prediction in cloud environments. For training models, especially deep learning models, which are compute and memory intensive and tightly coupled, serverless has not yet shown promise. The use of distributed computing for deep learning with accelerators is a well understood area [30–33]. While solutions such as MxNet [34] and Distributed TensorFlow have increased the performance of distributed computing with GPU acceleration to speed up deep learning training, there are limited studies on the investigation of parallelism in serverless runtimes. In order to facilitate the development of new serverless runtimes, and add features to the implementation backend (like compute and memory affinities based on cold and warm start) and others, it is important to understand strengths and limitations of deploying deep learning models in existing technologies.

In our research, we focus on computing security and efficiency. Specifically, we solve problems in the following three different fields:

- For secure and efficient circuit layout design, split fabrication is selected as our target. We propose two algorithms to improve the FEOL security and manufacturability synergistically. The algorithms can be applied to either Chemical Mechanical Planarization (CMP) uniformity or Self-Aligned Double Patterning (SADP) compliance.

- For hardware-assisted software security, in CFI verification, to improve the performance, decrease the latency, and increase the security simultaneously, the hardware-based CFI verification system called FastCFI is proposed with the hardware circuits implemented on FPGA. In DFI verification, our work propose a DFI solution based on PIM, with data compression and runtime optimization techniques proposed. Besides, circuit design are also developed to restrain the hardware overhead.

- For exploring serverless computing for machine learning model training, approaches are proposed for large neural networks training under serverless environment and hyperparameter tuning for small neural networks. Optimizations are also proposed for further improve the performance of training and the monetary cost.

2. SECURE AND EFFICIENT CIRCUIT LAYOUT DESIGN[1]

In this chapter, we propose routing techniques in split fabrication that can simultaneously improve both security and manufacturability in terms of either Chemical Mechanical Planarization (CMP) uniformity or Self-Aligned Double Patterning (SADP) compliance. We assume the FEOL is fabricated at a high-end untrusted foundry, and the BEOL is fabricated at a trusted foundry. The attack by untrusted foundry is the state-of-the-art network flow method [5] as it is one of the most effective attack techniques.

2.1 Previous Works

The vulnerability of split fabrication alone to attacks was first demonstrated in [3]. This work considered hierarchical designs, where the BEOL wires that connect different blocks can be easily guessed by attackers according to the proximity of the corresponding pins. Later, a set of obfuscation techniques [35] are suggested to improve the security for split fabrication. A placement perturbation technique is proposed in [5] to confuse attackers by moderately violating common design conventions. This work also describes an advanced network flow based attack method for flattened designs. Along the same direction, routing perturbation techniques are introduced in [2,6] to enhance the security for split fabrication. Improvement to the proximity attack [3] is also discussed in [6]. The partitioning between FEOL and BEOL layers for security is studied in [36]. Security of memory and analog IP blocks in split fabrication is investigated in [37].

It is noticed in [38] that 3D IC can play a similar role as split fabrication for hardware security. The concept of K-security and accordingly the wire lifting technique are introduced in [39] for 3D IC chips; this work has a different threat model compared to others—the attacker has access to the golden netlist. The work of [40] studies layer partitioning for 3D ICs such that the difficulty of attacks is increased. It further suggests a placement technique to enhance the security.

[1]©2017 IEEE. Reprinted, with permission, from Lang Feng, Yujie Wang, Jiang Hu, Wai-Kei Mak and Jeyavijayan Rajendran, "Making Split Fabrication Synergistically Secure and Manufacturable", IEEE/ACM International Conference on Computer-Aided Design, 11/2017.

Note that none of these security solutions addresses the manufacturability of their solutions. In fact, our results indicate that many of these solutions are not manufacturable, as they have DFM violations. Hence, we focus on developing solutions that are both manufacturable and secure.

Design For Manufacturability (DFM) has been an active research area for almost two decades. This section summarizes several works on layout design considering CMP and SADP, which are relevant to our methods. An early work on CMP-aware routing is [41], which shows that optimizing wire density for CMP is not equivalent to minimizing wire congestion in conventional routing. It modifies a conventional global router by incorporating wire density and the impact on timing into consideration. Another CMP routing work is [42]. Its important contribution is the use of Voronoi diagram for accurately estimate wire density for CMP-driven global routing. In [43], an SADP-aware detailed routing is developed in conjunction with layout decomposition. The work of [44] is an SADP-driven routing method considering cut process. SADP routing for 1D gridded designs is studied in [45, 46].

2.2 Preliminaries

2.2.1 Definitions

Some technical terms used in this paper are defined as follows.

Dangling wire: A wire on the topmost FEOL layer, with one end connected to driver/sink through a via to lower layers and the other end connected to via to BEOL layers.

Sink wire: A dangling wire that is connected to sink through lower metal layers. Its connection to driver is through BEOL layers. An example is the wire from b to B in Figure 2.1.

Source wire: A dangling wire that is connected with the driver through lower metal layers. Its connection with sinks is through BEOL layers. An example is the wire from A to a in Figure 2.1.

Complete wire: A wire at the topmost FEOL layer and connected with the driver/sink through lower metal layers, i.e., without using BEOL layers, such as the wire from c to d in Figure 2.1.

Up-via: A via connecting the topmost FEOL layer and the bottom BEOL layer. In Figure 2.1, a and b are two up-vias.

Down-via: A via connecting the topmost FEOL layer with its lower metal layers.

2.2.2 Attack Method

In our work, we assume the FEOL is fabricated at a high-end untrusted foundry, and the BEOL is fabricated at a trusted foundry. The attack by untrusted foundry is the state-of-the-art network flow method [5], which combined and optimized more factors of design conventions comparing with other attack models. In the network model, there is an edge between a source wire and a sink wire. The edge cost is defined according to hints from common design conventions. For example, if the up-vias of the two wires are close to each other like a and e in Figure 2.1, the corresponding edge cost tends to be small. Thus, they are more likely to be connected by the attack. We improved this method by considering preferred routing direction on related layers. Suppose the horizontal solid lines in Figure 2.1 indicate wires on the topmost FEOL layer. In other words, the preferred routing direction of the topmost FEOL layer is horizontal, in this example. In such scenario,

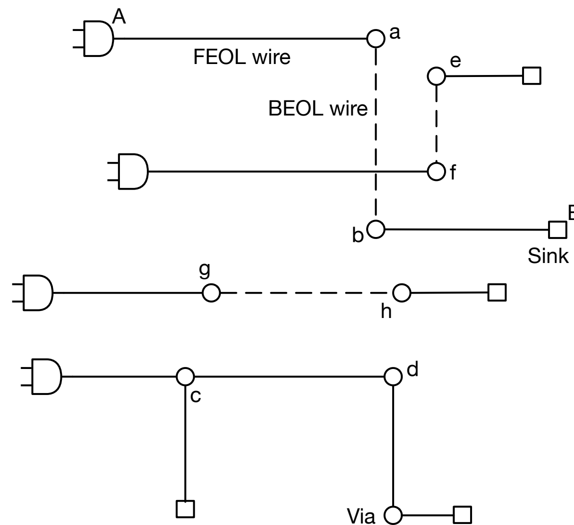

Figure 2.1: Illustration for definitions.

vertical alignment between two up-vias (like a and b in Figure 2.1) is much more common than

horizontal alignments like g and h. In chip layout, the distance between to points (x_1, y_1) and (x_2, y_2) is in Manhattan space as $|x_1 - x_2| + |y_1 - y_2|$. To improve the attack, we changed the distance to be a weighted form $w_x \cdot |x_1 - x_2| + w_y \cdot |y_1 - y_2|$, where w_x and w_y are the horizontal and vertical weights, respectively. For the scenario of Figure 2.1, where the preferred routing direction on the topmost FEOL layer is horizontal, we make $w_x > w_y$ such that the horizontal distance between two up-vias is emphasized. Since the preferred direction of the lowest BEOL layer is usually set different from that of the topmost FEOL layer, in this case, the preferred direction of the lowest BEOL layer is vertical, which means it's likely to connect two up-vias vertically. By making $w_x > w_y$, the cost of vertical distance is much smaller than horizontal, which implies that vertical alignment is preferred for this layer. Similarly, the horizontal alignment for a layer can be preferred by altering the weights.

2.2.3 Framework of Routing for Security and Manufacturability

Two routing-based defense methods are proposed for split fabrication. One considers the CMP friendliness and the other addresses SADP compliance. Both methods share the same framework, although they have significant differences. Taking a fully placed and routed circuit as the input, the framework consists of two steps.

- Step 1: Layer elevation. This is to selectively move some FEOL wires to BEOL layers such that they become invisible to attackers and manufacturability is benefited.
- Step 2: Rerouting. Some FEOL wires are rerouted to improve both security and manufacturability.

The details of these steps are elaborated in Section 2.3 and 2.4.

2.3 CMP-Friendly Routing Defense

2.3.1 Background on CMP

Chemical Mechanical Planarization (CMP) is an important semiconductor manufacturing step. After one layer of metal and Inter-Layer Dielectric (ILD) is finished, CMP is performed so that the surface is flat enough for fabricating another layer. The effect of planarization depends on metal

wire density as illustrated in Figure 2.2. If wire density is not uniform, the surface corresponding to the sparse region is lower than that in the dense region. The unevenness causes not only manufacturing difficult for upper metal layers but also ILD thickness variation, which worsens circuit timing variability.

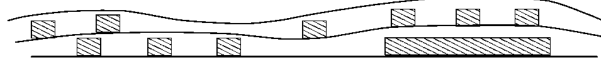

Figure 2.2: Uneven surface after CMP due to non-uniform wire density [1]. The shaded rectangles indicate cross sections of metal wires.

If the oxide density before CMP at location (x, y) is $\rho_0(x, y)$, the ILD thickness z at this location can be estimated by [1, 47, 48]

$$\begin{cases} z = z_0 - [K_i t/\rho_0(x, y)] & t < (\rho_0 z_1/K_i) \\ z = z_0 - z_1 - K_i t + \rho_0(x, y) z_1 & t > (\rho_0 z_1/K_i) \end{cases}$$

where t is the polish time, and K_i, z_0 and z_1 are constant parameters. Since the oxide density $\rho_0(x, y)$ is directly determined by wire density, the surface uniformity or the variability of z also depends on the variability of wire density. Wire density also affects metal thickness t_i [41, 42] as

$$t_i = \alpha(1 - \frac{m_i^2}{\beta})$$

where m_i is the wire density at region i, and α and β are constant parameters. It is shown in [41,42] that wire density must be explicitly considered in routing algorithms in order to reduce the CMP related variations.

2.3.2 Layer Elevation

Given a routed circuit, the first step of the CMP-friendly routing defense is layer elevation, where some wire segments are moved from an FEOL layer to a BEOL layer. Such move makes

the elevated wires from visible to invisible to attackers and hence improves security. At the same time, it affects wire density and CMP as well.

The wire segments to be elevated are selected according to several principles.

1. The wire segment has a significant logic difference from its neighboring wires. As such, an incorrect connection in attacking this wire may lead to more signal differences.

2. This wire segment has large observability so that an attack error can easily affect the circuit primary output signals.

3. This wire segment is originally at a wire-dense region. The wire density of this region would be reduced by the layer elevation and makes the corresponding FEOL layer have more uniform wire density.

4. The BEOL region where the wire segment is elevated to has low wire density so that the density of the corresponding BEOL layer is more uniform.

Items 1 and 2 are for security enhancement like in [2]. Items 3 and 4 are the new constraints, which intend to improve the uniformity of wire density and facilitate improved CMP.

2.3.3 Motivation and Wire Selection

After layer elevation, a set of wire segments is selected for rerouting. The rerouting has two purposes: CMP-friendliness and security improvement. For CMP-friendliness, one wishes to s-elect wires in dense regions so that they can be rerouted into sparse regions. When rerouting a segment, the mechanism for security enhancement is two-fold. This is illustrated by an example in Figure 2.3, where we consider to reroute wire segment from driver B. The original layout is given in (a), and the rerouting result is in (b). The FEOL and BEOL wires are represented by solid and dashed lines, respectively. By the rerouting, the wire detour makes via point b is closer to c' than a. Such proximity can mislead an attacker to think b should be connected with c'. Then, via c' serves as a decoy to the net driven by A. The blue region in Figure 2.3 is designated as a target-decoy region for the net driven by A. The additional consideration of CMP also helps

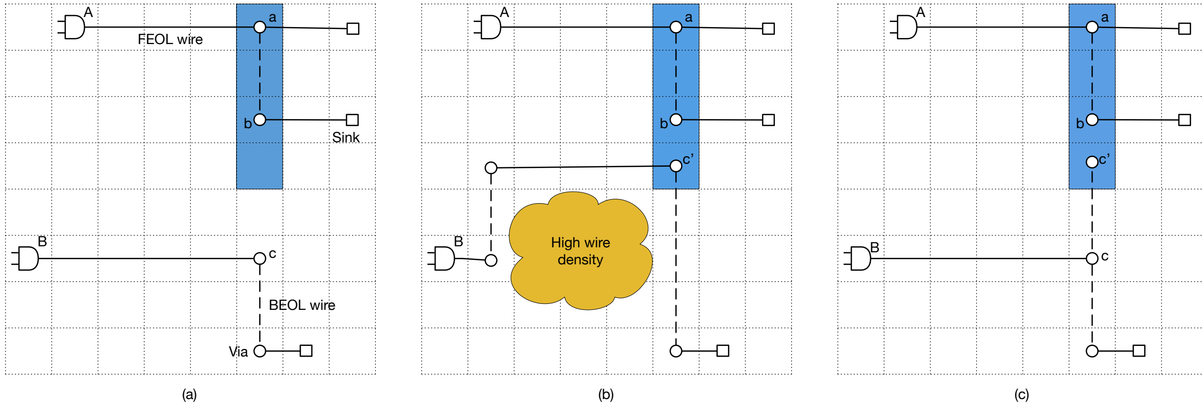

Figure 2.3: (a) The original routing; (b) security and CMP-driven rerouting. (c) dangling stub for security. The blue squares indicate the decoy region for the net driven by gate A.

security. The rerouting in Figure 2.3(b) is a detour to avoid the high wire density region. As such, an attacker would regard such detour as CMP-driven, and the defense purpose is disguised. In contrast, if a security-driven detour is around a sparse region, which is allowed in [2], an attacker would feel suspicious and may realize that the detour is a defense measure. Besides detour, we consider another defense approach, which is the dangling stub in Figure 2.3(c) and not used in previous routing defense works [2, 6]. The dangling via point c' in Figure 2.3(c) may also mislead an attacker to connect b with c'. Although such layout is against common design convention, its wire and timing overhead may be less than that of a detour. According to the mechanism in Figure 2.3, the security aspects of the wire selection is to see if a wire near the decoy region of another net. Also, the logic difference between the net to be selected (net driven by B in Figure 2.3) and the net to be decoyed (net driven by A in Figure 2.3) should be large, and so is the observability of the net to be decoyed.

Please note we use a different strategy from the K-security in [39]. The concept of K-security is to provide $K-1$ additional connection options, which are *equally good* as the original connection, to attackers. Then, it is very difficult for an attacker to make the correct connection among K options, especially when K is large. In contrast, our defense just provides one additional connection option (decoy), which looks *better* than the original connection. For example, in Figure 2.3(b), c'

is closer to b than a and therefore looks better than the original connection between a and b. Since rerouting usually comes with wirelength and delay overhead; our strategy requires less rerouting and lower overhead.

2.3.4 Wire Rerouting Method

For the wire segments selected according to Section 2.3.3, we reroute them one at a time. Our approach has a key difference from the routing perturbation in [2]. As CMP is not considered, the rerouting of wire segment in [2] can be solely focused on security. For example, in rerouting the wire segment connected to driver B in Figure 2.3, only nets driven by A and B are considered. By contrast, our method needs to consider wire density in addition. Therefore, we divide layout area into an array of tiles like in Figure 2.3 and perform a coarse rerouting on the tiles, which is further refined as like detailed routing. Moreover, we consider the application of dangling stub like Figure 2.3(c) while [2] does not.

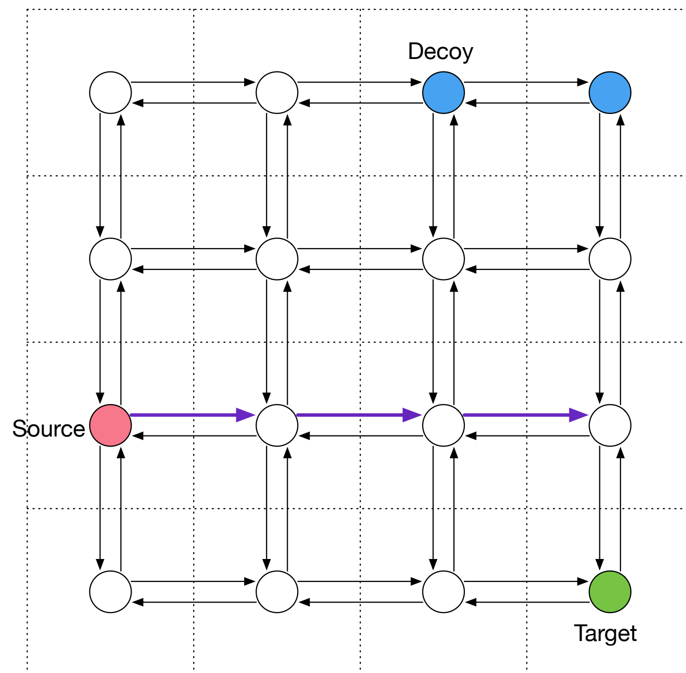

Figure 2.4: Routing graph.

In our methods, a routing graph is built according to the routing tiles as in Figure 2.4. In this graph, each node corresponds to one tile and there is a pair of directed edges between two adjacent tile nodes. Please note this is different from conventional global routing where there is only one undirected edge between two adjacent tiles. This difference is due to the fact that wire congestion in conventional global routing is evaluated across a boundary between two neighboring tiles while the effective oxide density [1] is evaluated within each tile. In order to capture the preference to wires from high-density tiles to low-density tiles, such directed edges are needed.

Considering an edge $(a \rightarrow b)$ from tile a to b, where the effective oxide densities are d_a and d_b, respectively. The edge is associated with a density weight $w_d(a \rightarrow b)$ defined as

$$w_d(a \rightarrow b) = \begin{cases} \frac{1}{d_a - d_b + \delta}, & \text{if } d_a \geq d_b \\ K \cdot (d_b - d_a) + \frac{1}{\delta}, & \text{if } d_a < d_b \end{cases} \tag{2.1}$$

where δ and K are two constant parameters. This weight definition implicitly addresses routing congestion and wirelength as well. If tile b is very congested, its oxide density and edge weight $w_d(a \rightarrow b)$ are both high. As a result, this weight definition resists connection through tile b. As a density weight is always non-trivially positive, a long wire detour or wirelength is also penalized.

The rerouting is to find a new wire connection among the source node, like B, the target node like c and decoy nodes like the blue tiles in Figure 2.3. This is equivalent to constructing a Steiner tree on the graph shown in Figure 2.4, which is a well-known NP-hard problem. Therefore, we design a heuristic based on the Dijkstra's shortest path algorithm.

If the Dijkstra's algorithm is performed using the density weight w_d for edges, we can obtain the minimum weight paths from the source to the target and decoy nodes, respectively. However, such approach neglects the benefit of sharing the two paths, which means the dangling stub approach. In order to encourage sharing between the two paths, we augment edge weight with sharing weight, which is defined as follows. First, we draw two bounding boxes, one is between the source and the target, and the other is between the source and decoy nodes. If there is no edge overlap between

the two boxes, no change is made to edges. If an edge is on the shared boundary between the two boxes and along the direction from the source to the target and decoy nodes, this edge is called a *sharing edge*, which is indicated as the purple thickened edges in Figure 2.4. If an edge is k hops away from any sharing edges, its weight is incremented by $k \cdot \epsilon$, where ϵ is a small constant. With the augmented weight, the Dijkstra's algorithm is performed on the routing graph to obtain paths from the source to the target and decoy nodes as the rerouting results.

2.4 SADP-Compliant Routing Defense

2.4.1 Background on SADP

For sub-16nm technology nodes, Self-Aligned Double Patterning (SADP) is an excellent option for fabricating the lower metal layers of a design to achieve the required fine metal pitches. In addition, unidirectional routing is usually advocated in these layers for its higher manufacturing yield and uniformity compared to 2D routing. In such case, the SADP process will first print a sea of parallel tracks as Figure 2.5(a), and then wire-end cuts are printed using a cut mask to cut up the tracks into the target wire segments and some dummy wire segments. Due to the constraints on the cut mask of self-aligned double patterning, two cuts cannot be too close to each other, i.e., there is the minimum spacing rule between cuts. Some wire-end of the target wire segments is extended after initial routing in order to avoid violation of this rule, or make a layout SADP compliant [45, 46].

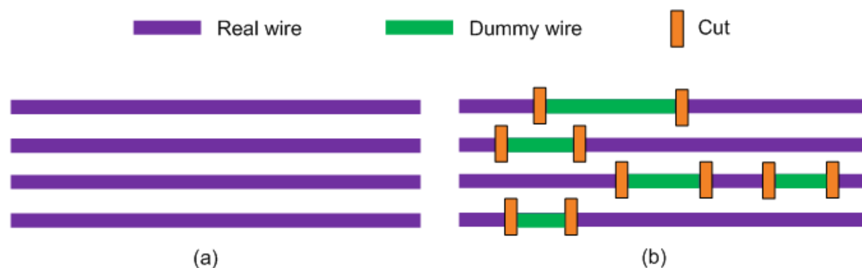

Figure 2.5: SADP starts with dense lines generation as in (a). By cutting, the desired patterns are the purple parts in (b).

2.4.2 Security Enhancement under SADP

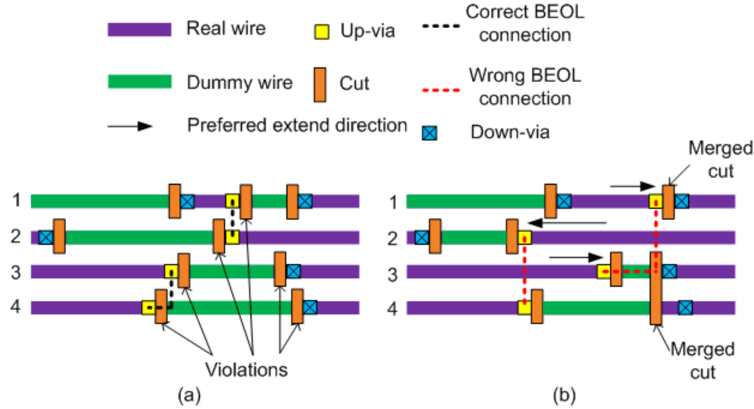

Figure 2.6: (a) Original layout with SADP violations. (b) Wire extension for both SADP compliance and security.

We introduce an approach to modify a 1-D layout on the topmost FEOL layer to facilitate SADP compliance and enhance security at the same time. The context is that FEOL foundry uses high-end process technology including SADP while BEOL foundry still operates with the conventional manufacturing process without SADP. Our approach follows the same 2-step framework of Section 2.2.3. The first step is layer elevation, which is very similar to the one in CMP routing (Section 2.3.2) except that wire density is not considered, as SADP layout with wire-end cuts intrinsically has near uniform wire density.

The second step of rerouting is actually wire extension of FEOL wires as in [45]. Please note the wire extension of FEOL wires inevitably causes rerouting of connected BEOL wires. We use the example in Figure 2.6 to illustrate how such wire extension can simultaneously help SADP compliance and security. Figure 2.6(a) shows a part of the original topmost FEOL layer. Recall that an up-via is a via connecting the topmost FEOL layer to the bottom BEOL layer. A down-via is a via connecting the topmost FEOL layer to the layer below it. In Figure 2.6(a), there are three pairs of cuts too close to each others and each causes an SADP rule violation. Wire extension where four cuts are relocated is shown in Figure 2.6(b). After the wire extension, there is no

SADP rule violation. Moreover, the proximity of the up-vias is changed by the wire extension. As a result, an attacker is misled to the wrong (red) connection in (b) while the original design is the black BEOL connections in (a). It is worthy to note that when a wire-end with an up-via is extended, the corresponding up-via is moved accordingly. On the other hand, a wire-end with a down-via can be extended, but the corresponding down-via will not be moved as shown in the bottom right corner of Figure 2.6(b). Finally, we allow two cuts to be merged into a single one as the two cuts in the top line and the two cuts near the bottom right corner in Figure 2.6.

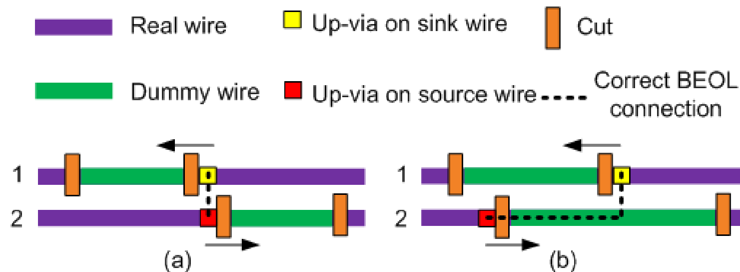

Figure 2.7: Wire extension enhances security in (a), but degrades security in (b).

The wire extension for simultaneous SADP compliance and security is realized using Integer Linear Programming (ILP) like [45], but the ILP here is quite different from [45] and needs to handle more complicated situations. The ILP in [45] attempts to build constrains for eliminating the SADP violations and minimize total wire extension (or overhead) subject to SADP manufacturability constraints and maximum wire extension constraint for each wire. When security is considered, the problem is more complex as wire extension can degrade security as well. In Figure 2.7(a), the extension enhances security as it moves the two up-vias apart. By contrast, the extension in Figure 2.7(b) makes the two up-vias closer and then the correct connection is easier to be figured out by an attacker. Therefore, whether or not to maximize or minimize an extension in our ILP depends on layout scenarios. In Figure 2.8, we summarize if wire extension is good or bad for security in eight different scenarios.

When security is considered, another complicated situation is multi-pin net. In Figure 2.9, the

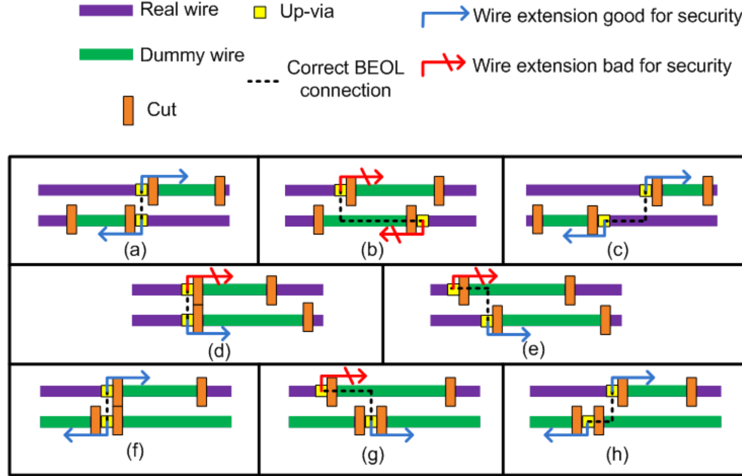

Figure 2.8: Scenarios of different security implications by wire extension.

source wire in the middle (associated with the red via) and the four sink wires on the other lines belong to the same net. For the sink wires, whether or not line extension is good for security is indicated by the blue and red arrows. For the source wire, wire extension toward right makes a connection to the sink wire on line 5 easier, i.e., the security of the source and the 5th line sink connection is weakened. However, the extension of the source wire increases the security for connections with the other sink wires. In this situation, we decide if increasing a source wire extension is preferred or not according to majority vote. More specifically, we prefer to increase (decrease) the extension of a source wire if the number of sink wire connections with security improvement is more (less) than the number of sink wire connections with security degradation.

As a result, our ILP to determine the wire extension of each wire for simultaneous SADP compliance and security enhancement is constructed as follows. We incorporate the constraints similar to [45] to enforce (i) two wire-end cuts in the same track or nearby track have to maintain a minimum spacing min_s or have to be merged (as in Figure 2.6), and (ii) enforce a maximum allowed wire extension δ_i for each wire i due to timing consideration. Our objective function is to maximize the difference in the total security improvement and the total wire extension. When an extension of a wire-end with an up-via is good for security, a positive security improvement proportional to the extension length is accumulated. On the other hand, when an extension of a

wire-end with an up-via is bad for security, a negative security improvement proportional to the

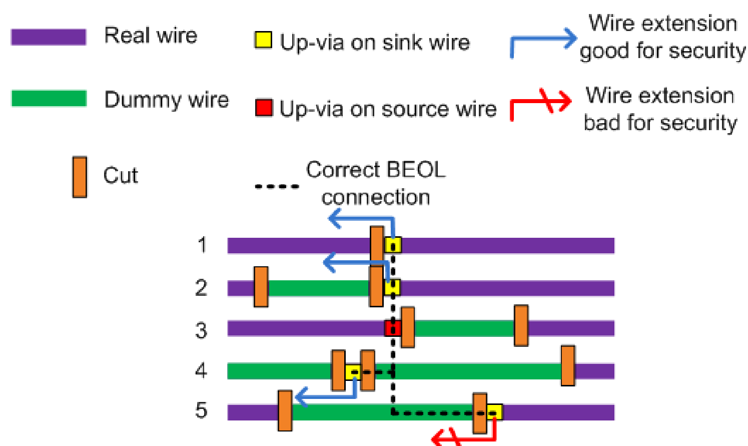

Figure 2.9: The source wire in the middle line and the four sink wires on the other lines belong to the same net.

extension length is incurred. Note that any non-dangling end of a wire (i.e., any end with a down-via as in Figure 2.6) can be extended, but the corresponding security improvement is always zero. We note that any wire-end cut conflict that cannot be resolved in the final layout is handled by an e-beam shot in [45], but we simply report it as an SADP violation in this paper.

2.5 Experiments

The experiments are conducted on the five largest circuits in ISCAS'85 benchmark suites and the seven largest circuits in ITC'99 benchmark suites. These circuits are synthesized by Synopsys Design Compiler using 45nm standard cell library. The initial layout is generated by Cadence SoC Encounter. The timing analysis is obtained through Synopsys PrimeTime. The defense and attack algorithms are implemented and run on a PC with Intel 3.4GHz CPU with 16GB memory. The ILPs are solved by solver Gurobi 7.0.2 [49]. The defense results are complete layouts with detailed routing and design rule checking are performed by Cadence SoC Encounter.

Comparisons are made to the following layout results.

- Original: The layout generated by Cadence SoC Encounter without routing defense or manufacturability improvement.
- Security-only: The latest previous work on routing based security for split fabrication [2], which is performed on the original layout.
- Security+CMP: Our CMP-friendly routing defense (Section 2.3) performed on the original layout.
- Security+SADP: Our SADP-compliant routing defense (Section 2.4) performed on the original layout.

The layout results from all the above methods are attacked using the network flow method [5] with our improvement (Section 2.2.2) to evaluate their security.

The results are evaluated with the following metrics.

1. Security
   - Connection error: the percentage of wrong connections by the attack (Section 2.2.2) among all missing wires, which are on BEOL layers.
   - Hamming distance: the Hamming distance between output vectors of the original complete design and the reverse engineered design by the attack (Section 2.2.2). Security is strong when Hamming distance is near 50%. The result is obtained by 5K runs of Monte Carlo simulation. We found that the 5K-run results are usually very close to 50K-run results, typically with less than 1% difference.
2. Manufacturability
   - $\Delta Var(den)$: the change of wire density variance compared to the original layout. A negative change means variation decrease and is preferred.
   - # SADP violations: the number of SADP rule violations. The violations can be solved by using electron-beam lithography at a certain expense. Thus, such violation is permitted but not preferred.

3. Overhead

    - Delay overhead: the critical path delay change compared to the original layout according to Synopsys PrimeTime.
    - Wirelength overhead: the total wirelength change compared to the original layout.

Table 2.1: Results of the original design, security only routing [2] and our CMP friendly security routing.

Circuits	# nets	Original Design				Security Only [2]				Security + CMP			
		Attack [5]		Improved Attack (Sec. 2.2.2)		Improved Attack (Sec. 2.2.2)		FEOL	BEOL	Improved Attack (Sec. 2.2.2)		FEOL	BEOL
		Connection error (%)	Hamming distance (%)	Connection error (%)	Hamming distance (%)	Connection error (%)	Hamming distance (%)	$\Delta Var(den)$ (%)	$\Delta Var(den)$ (%)	Connection error (%)	Hamming distance (%)	$\Delta Var(den)$ (%)	$\Delta Var(den)$ (%)
c2670	607	51.9	14.5	38.0	11.9	68.6	20.3	-17.2	24.3	66.7	20.5	-9.2	-17.7
c3540	638	55.3	16.9	22.4	13.5	71.2	38.5	-7.4	68.9	88.5	35.0	-44.7	-1.4
c5315	997	54.7	15.9	41.2	13.3	57.9	24.1	-21.8	5.7	85.1	23.6	-59.7	-16.1
c6288	1921	6.1	2.4	0.0	0.0	72.3	44.3	3.8	-27.7	66.9	40.6	-0.4	-20.9
c7552	1041	59.5	18.6	45.2	15.5	75.7	30.7	-19.3	6.8	78.7	24.7	-59.4	-28.3
b14_1	3018	66.8	10.6	15.9	4.3	83.7	25.2	-17.7	98.9	68.5	21.7	-30.3	-28.1
b15	6018	73.0	10.2	48.6	9.5	75.7	19.7	-13.3	-21.8	80.1	20.3	-22.2	-8.6
b17	18613	78.3	17.3	54.8	13.8	93.8	33.7	-44.1	-71.0	82.3	22.4	-70.1	-71.4
b18	55029	87.9	21.6	67.8	18.1	94.5	34.2	-30.7	105.8	89.3	27.6	-48.0	38.2
b20	8109	84.6	30.7	44.3	26.5	89.9	37.3	-11.4	-14.2	77.0	32.2	-28.3	-46.1
b21	8153	84.3	33.1	47.8	26.1	67.5	36.3	-18.8	-11.7	79.3	36.5	10.4	-48.6
b22	12065	50.1	13.9	15.4	9.8	87.5	32.6	-13.9	55.8	93.2	29.8	-77.6	-45.6
Ave		62.7	17.1	36.8	13.5	78.2	31.4	-17.7	18.3	79.6	27.9	-36.6	-24.5

2.5.1 Results of CMP Friendly Defense

The main results of CMP friendly security routing are summarized in Table 2.1 along with those from the original layout and the previous work [2]. On average, the security routing of [2] can increase attack connection errors and Hamming distance from 37% to 78% and from 14% to 31%, respectively. However, it increases BEOL wire density variance by 18%, which implies a significant degradation of manufacturability. By contrast, our security+CMP approach can reduce wire density variance for FEOL and BEOL by 37% and 25%, respectively. At the same time, the security of our approach is similar to that of the previous work [2]. Table 2.1 also compares the attack of [5] and that with our improvement (Section 2.2.2) in columns 3-6. The results show that our improvement can reduce connection errors and Hamming distance from 63% to 37% and from 17% to 14%, respectively. The delay and wirelength overhead of both methods are shown in Figure 2.10. One can see that the overhead from our approach is equally small as that of [2]. The CPU runtime of our method is typically several seconds for each case.

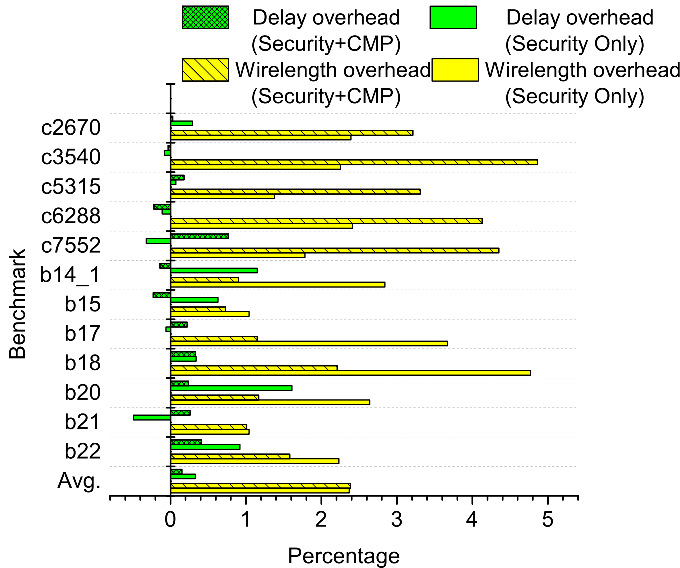

Figure 2.10: Rerouting overhead.

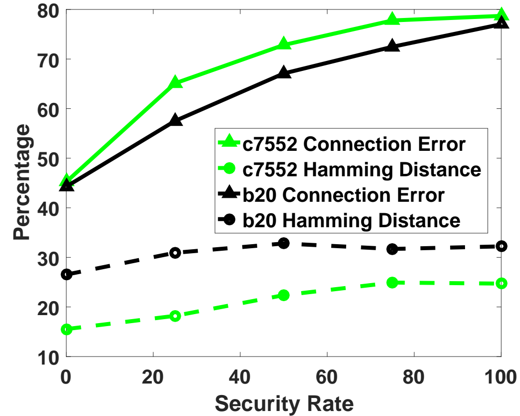

Figure 2.11: Security versus $\%$ selected wires being rerouted.

We further study the impact of wire selection for two circuits c7552 and b20. The wires selected to be elevated (Section 2.3.2) and rerouted (Section 2.3.3) are sorted according to a weighted combination of potential wire detour distance and benefit to CMP, with wires of small distance and large CMP benefit at top. Then, top $\alpha\%$ of these selected wires are elevated and rerouted. We vary the value of α to see the impact on security, CMP uniformity and overhead. The results are plotted in Figure 2.11, 2.12 and 2.13. In general, increasing the number of elevated and rerouted wires improves security and CMP uniformity, and causes more overhead. There are a couple of non-monotone changes of $\Delta Var(den)$ in Figure 2.12. These are due to the heuristic nature of our approach.

2.5.2 Results of SADP Compliant Defense

The SADP results are shown in Table 2.2. The previous work of security-only routing [2] increases the number of SADP violations by 44%. In contrast, our SADP compliant security routing can reduce the violations by 97%. The security of our method in terms of connection error and Hamming distance is about the same or even better than the previous work [2]. The delay and wirelength overhead by our method are increased, but still quite small. The ILP runtime is always

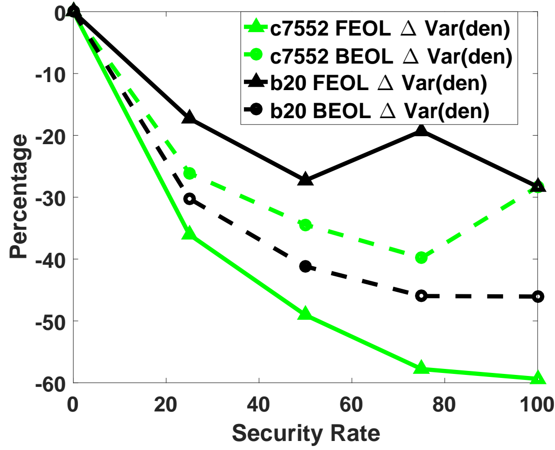

Figure 2.12: Wire density variance change versus % selected wires being rerouted.

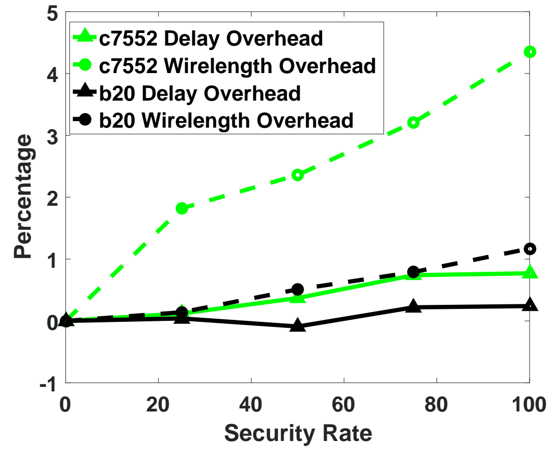

Figure 2.13: Delay and wirelength overhead versus % selected wires being rerouted.

Table 2.2: Results of SADP violations and SADP compliant security routing.

Circuit	Original	Security Only [2]	Security + SADP				
			Improved Attack (Sec. 2.2.2)			Delay	Wirelength
	#SADP violations	#SADP violations	Connection error (%)	Hamming distance (%)	#SADP violations	overhead (%)	overhead (%)
c2670	20	33	93.6	24.4	0	0.05	7.49
c3540	12	28	85.0	36.9	0	0.14	2.41
c5315	39	48	96.3	29.7	3	0.02	3.74
c6288	40	49	65.7	33.3	0	-0.01	0.65
c7552	27	64	93.0	33.4	1	0.08	4.53
b14_1	16	25	98.1	26.8	0	0.53	1.49
b15	42	50	96.2	22.4	1	5.45	3.07
b17	203	385	95.9	29.6	3	1.06	4.82
b18	740	928	97.3	29.7	27	0.55	4.64
b20	70	131	90.5	39.4	3	0.43	4.55
b21	46	62	89.9	40.2	2	0.95	4.89
b22	50	76	85.9	30.2	0	0.74	1.93
Ave	109	157	90.6	31.3	3	0.83	3.68

within 2 minutes for each circuit.

2.6 Conclusions

Although the security risk associated with untrusted foundries partially arises from the advanced process technology and related manufacturability challenge, existing works on split fabrication almost always focus on security while neglect manufacturability issues. In our work, we show that manufacturability and security in split fabrication can actually be addressed in a synergistic manner. In particular, two routing-based security methods are developed, one is friendly

with chemical mechanical planarization, and the other improves compliance with self-aligned double patterning. Comparison with the latest previous work indicates that the proposed methods can achieve the same security with significantly improved manufacturability.

3. HARDWARE-ASSISTED SOFTWARE SECURITY[1]

In this chapter, we propose the system design for Control-Flow Integrity (CFI) and Data-Flow Integrity (DFI) verification. For CFI verification, our FastCFI system contains offline part and online part. The online part is implemented by hardware circuit on FPGA. For DFI verification, Processing-In-Memory (PIM) is used for DFI checking. With great amount of data transferring reduced by using PIM, the performance overhead is largely reduced without sacrificing security.

3.1 Previous Works

3.1.1 Control-Flow Integrity

3.1.1.1 Software-based Control-Flow Integrity

Early work on CFI was mostly realized by software implementation. The seminal work by Abadi et al. [8] proposes two code instrumentation approaches, which have average overhead of 16% and 21%, respectively, on the SPEC 2000 benchmark. Later work targeted CFI at specific application scenarios. For example, the method of Davi et al. [14] is designed for smartphones, and the work by Zhang and Sekar [13] addresses how to handle COTS binary codes. Several works [15, 50, 51] attempt to reduce performance overhead or avoid code instrumentation by sacrificing granularity or security coverage. For example, kBouncer [15] has very low overhead and code instrumentation is avoided in [50]. However, both methods handle ROP attacks only.

FastCFI is hardware-based CFI, which avoids some disadvantages in software-base CFI, such as high performance overhead [8, 14], coarse-grained CFI policy [15, 50, 51], and requiring code instrumentation [8, 13, 14].

3.1.1.2 Hardware-based Control-Flow Integrity

Recently, several hardware-based CFI approaches [17–20, 52–68] have been proposed, based on Intel Processor Trace [17, 19, 55, 62], performance counters [20], FPGA [18, 59, 68], and oth-

[1]Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature, International Conference on Computer-Aided Design, "FastCFI: Real-Time Control Flow Integrity using FPGA without Code Instrumentation", Lang Feng, Jeff Huang, Jiang Hu and Abhijith Reddy, ©2019.

ers [61]. Intel also proposed the Control-flow Enforcement Technology (CET) [69]. However, processors that support CET are still not available. Meanwhile, CET only implements the weakest form of CFI in that there's only a single class of valid targets and is too weak to protect against the larger class of code reuse attacks.

Besides these approaches, great amount of the hardware-based CFI approaches require hardware modification [52–54,56–60,63–67]. Modifying hardware structure such as adding additional modules inside the processor's pipeline is not practical, since one will need to repeat the whole design flow, which is a tedious task.

Compared to previous hardware-based CFI, FastCFI has novelties in multiple directions. Firstly, FastCFI does not depend on code instrumentation. Previous hardware-based approaches leverage code instrumentation for getting more information [18, 54, 56, 63, 64, 67, 68]. However, this results in large overhead, and code instrumentation itself is also not secure and sometimes even impossible. Secondly, FastCFI has a low overhead compared to some previous works [17–19,65,68]. High overhead is unacceptable in some real-time applications. Also, not all the hardware-based CFI are fine-grained and stateful [18, 20, 59, 61, 68]. They may miss some attacks. In operating system, false positive may delay all the processes, but some techniques used in previous works lead to this [20, 61]. Through results obtained in FastCFI we show that such cases are avoided in our CFI solution. Hardware-based CFI is harder to be implemented than software-based CFI due to the cost, difficulties in manufacturing, resources, etc. A few previous works prefer using simulator for implementation [56, 58, 59, 63, 64], but this will not guarantee the functionality because there are differences between simulation and real world conditions. We use FPGA to implement the hardware design. By taking advantage of existing devices in the processor, we avoid changing the structure of the processor and are able to build a real system for CFI verification.

3.1.2 Data-Flow Integrity

The concept of Data-Flow Integrity (DFI) was proposed in the seminal work [16] in 2006. This work also provides a software implementation technique and optimization techniques for overhead reduction. It received a lot of attention as it can detect a significantly wider range of security

attacks than an earlier well-known concept of Control-Flow Integrity (CFI) [8]. Although the DFI verification procedure is simple, its performance overhead is intrinsically large as it involves a huge volume of data. Compared to CFI, which has many followup research activities [14, 17, 18, 20, 55, 61], the study on DFI has been much less due to the challenge.

The few previous works [70–73] on DFI after [16] achieved much lower overhead by focusing on reduced versions of DFI. The work of [70] is restricted to only certain selected data for kernel software. One of its main contributions is the techniques on how to select data to be protected. Although its performance overhead is only $7 - 15\%$, its application is restrictive and misses many attacks at user programs. For example, nullhttpd [12], heartbleed [11] and data-oriented programming [74] are conducted at user level. By contrast, our approach covers both kernel and user level programs.

While DFI involves both `load` and `store` instructions, the work of Write Integrity Testing (WIT) [71] is focused on `store/write`. It requires that each `store` instruction can only write to certain data objects, and each indirect call can only call certain functions. Detailed specifications are obtained according to static analysis. Although its overhead is at most 25%, it does not consider the integrity of `load` instructions. Therefore, an unsafe `load` instruction may read more bytes than the programmer's intention, and consequently information leak may occur. Heartbleed [11] is an attack that WIT would fail to detect since all its `store` instructions behave normally, and only some `load` instructions read more data than allowed. The illegally obtained data are then sent to the attacker.

Data isolation is another approach to protecting data with relatively low overhead. A hardware solution for data-flow isolation is proposed in [72]. It designates two data regions, a sensitive one and a non-sensitive one. A 1-bit tag is employed to tell the region that a data belongs to. Instruction set is modified such that the tags can be read and set. Moreover, processor hardware, operating system and compiler also need changes. If data in one region, it cannot be written by an instruction for the other region. Although the isolation between two regions helps security, it cannot handle the case where `load/store` instructions for different data of the same region are mingled. Although

its overhead is less than 2%, its security resolution is very coarse. To certain degree, the original DFI [16] can be regarded as data isolation among individual instructions. If 16 bits are used for each instruction identifier, it is equivalent to isolation among up to 2^{16} regions. Compared to the only 2 regions of HDFI [72], the resolution of the original DFI is $2^{15} = 32768$ times higher.

In addition to the work of [72], there are other tag-based isolation techniques. The work of [75] uses 1-bit tag for each word of data to indicate its integrity level in Biba's low-water-mark integrity policy [76], which requires that an instruction can only modify data with integrity level no higher than that the instruction. In [75], processor hardware is modified to enforce this policy for control data protection. In [73], a 1-bit tag is also employed to specify if each data can be referred by certain instructions. Overall, the tag-based techniques [72, 73, 75] provide only coarse-grained isolation as different data/instructions with the same tag cannot be isolated from each other.

3.2 Hardware-assisted Control-Flow Integrity Verification

3.2.1 CFI and Control-Flow Graph

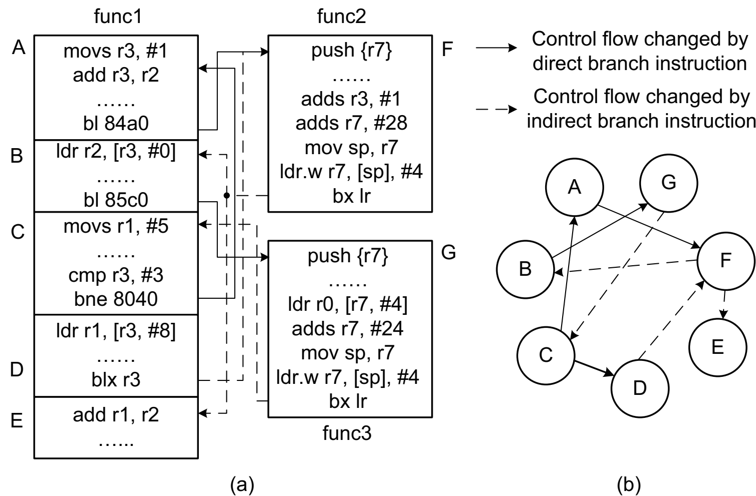

Figure 3.1: Example of control-flow graph.

The specification of CFI is a Control-Flow Graph (CFG) of the target program, in which each node corresponds to one segment or block of instructions and each directed edge indicates a legal

transition between instruction segments. In the example of Fig. 3.1(a), the instructions are divided into seven segments, each of which corresponds to a node in Fig. 3.1(b). The solid and dashed edges in Fig. 3.1(b) indicate transitions by direct and indirect branch instructions, respectively. For example, the edge from node A to F implies that the branch instruction `bl 84a0` in A is taken and the software execution switches from A to F.

Once a CFG is constructed for a software, CFI of this software execution is enforced by verifying if an execution trace conforms to the CFG. For instance, transition $A \to B$ is illegal as there is no edge from A to B in the CFG. CFI for function returns can be stateful. For example, there are edges from F to both B and E. However, if F is invoked by function call from A, the last instruction in F should only return to the instruction right after A, which is in B. Therefore, function return $F \to B$ is legal while transition $F \to E$ is illegal.

3.2.2 The Proposed System Design

3.2.2.1 System Platform

FastCFI is developed on a platform depicted in Fig. 3.2. It is composed of an ARM Cortex-A9 processor and an FPGA. The CFI of a software execution on the ARM core is verified by the FPGA. Program Trace Macrocell (PTM) generates compressed control-flow traces according to instructions processed by the ARM core. The CoreSight Debug module in the ARM core can obtain traces from PTM and send the traces to FPGA through the Trace Port Interface Unit (TPIU), which acts as a bridge between the trace data and a data stream. The key ideas of FastCFI can be applied to other platforms such as x86 architecture.

3.2.2.2 System Design Overview

The system design of FastCFI consists of an offline CFG checker generator and an online CFI verifier, as depicted in Fig. 3.17. The CFG checker generator is a software that takes application software binary as input and generates CFG checker design in Verilog. During online software execution, a trace captured through ARM CoreSight is first decoded in order to understand its semantics. The decoded trace data is then fed to the CFI verification module, which is composed

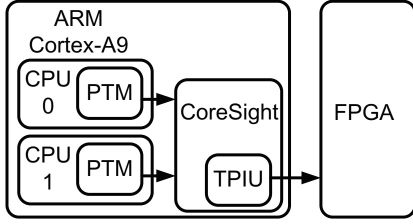

Figure 3.2: System platform for the proposed CFI.

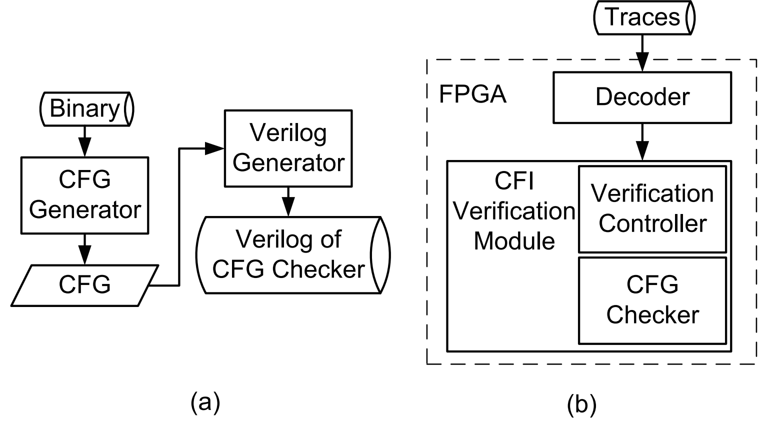

Figure 3.3: System design overview: (a) offline CFG checker generator; (b) online CFI verifier.

of a verification controller and a CFG checker. Both the decoder and the verification module are implemented on FPGA.

3.2.2.3 Offline CFG Checker Generator

To give the hardware verification circuits the correct execution information that can be represented by CFG, target software binary has to be analyzed, and CFG should be extracted. Since we implement the CFG as a hardware circuit called CFG checker in the verification module for higher speed, the output of the CFG checker generator is the CFG checker's Verilog HDL file.

Given software binary, the generator first converts it to assembly code. It extracts CFG from the assembly code and generates the Verilog design of CFG checker circuit. Then, the CFG checker is mapped on FPGA. The generator is able to help the fast implementation of CFI verification given a system to be protected, and only the target vulnerable binary is required.

We denote a sequence of assembly instructions as $I_1, I_2, ...I_{m1}, B_1, I_{m1+1}, I_{m1+2}, ..., I_{m2}, B_2, ...B_n...$, where $B_1, B_2, ...B_n$ are branch instructions (e.g., jmp, call, ret, etc.) and the others are non-branch instructions. Then, the instruction sequence is partitioned into multiple segments $\{I_1, I_2, ...I_{m1}, B_1\}$, $\{I_{m1+1}, I_{m1+2}, ..., I_{m2}, B_2\}$,..., each of which has a single branch instruction at its end. Each instruction segment forms a node in the CFG. In the sequel, we use CFG node and instruction segment interchangeably when the context is clear.

By examining the source node and target node of each branch instruction, the generator can establish edges of the CFG. Recognizing the source node is trivial, but finding target node can be quite difficult. The target address of a direct branch instruction is hardcoded in the binary and can be easily found. Indirect branch is a tricky case, as its target address is stored in a register. Such address can be a constant hardcoded somewhere in the binary, and can be recovered through tracing instructions. The more difficult case is where the target address depends on software input data at runtime. As such, it is almost impossible to find the address with an offline static analysis. Despite this difficulty, we find how to perform partial CFI check for unspecified target address and this technique will be described in Section 3.2.2.5.

We developed a software program to automatically construct CFG from binary code. The generator further creates Verilog description for the CFG checker circuit. Meanwhile, our framework is general and can accommodate other tools such as IDA [77].

3.2.2.4 Trace Decoder

The decoder takes software execution trace from TPIU as input, interprets its semantic and extracts information that is relevant to CFI. A trace consists of many packets, each of which is usually a few bytes. Two types of packets are of particular relevance to CFI, *Atom* and *Branch address* [78], which is simply called *Branch* subsequently. An *Atom* tells if a direct branch is taken or not, and indicates the case that an indirect branch is not taken. If an indirect branch is taken, its target address is contained in *Branch*. Some other types of packets, such as *I-sync* [78], can periodically indicate the current instruction address.

The decoder extracts the following required information:

- Context ID that identifies the current program.
- The current program state.
- The current packet type: *Atom*, *Branch*, or *I-sync*, etc.
- The current instruction address, which is obtained from *Branch*, or *I-sync*. Note that this

information is not always available and the scenarios of its availability are complex. The starting address of a program is available at *I-sync*, which continues to provide current address periodically.

- T/N from *Atom*, where T indicates that a branch is taken and N means an indirect branch is not taken.

- Program exception and PTM buffer overflow information.

The TPIU channel in the ARM core has 32-bit bitwidth, which means 4 bytes of packets can be sent to FPGA in every clock cycle. When implementing, we design a 3-phase pipeline decoder to increase the throughput and match the speed of the TPIU.

3.2.2.5 CFI Verification Module

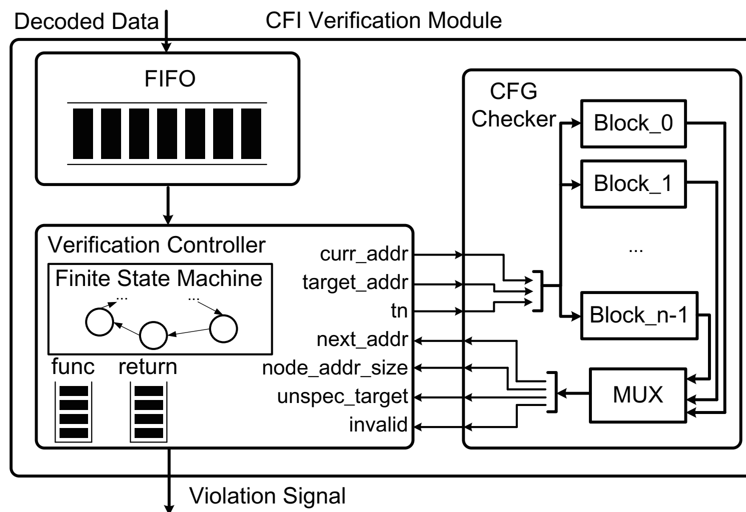

Figure 3.4: Architecture of CFI verification module.

The CFI verification module is to examine if flow transitions in a software execution trace are consistent with transitions specified in CFG, which is embedded in the CFG checker. In order to do so, we need to obtain the source node and target node of a branch instruction from the execution

trace. The source node of a branch instruction, which is equivalent to the current instruction address of the branch, is often unavailable in trace packets. In [18], it is acquired through code instrumentation. Without code instrumentation, identifying the source/current node is much more difficult. We solve this difficulty by using the periodically available instruction address information and tracking the other addresses by following the CFG.

Consider the example in Fig. 3.1. Suppose we know the address of the first instruction of node A. The last instruction of A, `bl 84a0`, is a branch to node F, whose execution results in an *Atom* with T indicating that the branch is taken. Note that every direct branch has only one deterministic target when it is taken. When Write$\oplus$Execute [79] technique is applied in an operating system, an attacker is not able to change the code and the target of each direct branch. Therefore, by observing T from trace decoder and examining the CFG in the CFG checker, we know that the software execution now moves to node F even if the current instruction address is not available at trace packets. Since the transition from A to F changes the current function from `func1` to `func2`, `bl 84a0` is inferred as a function call. Therefore, `func2` should return to the next instruction of `bl 84a0` of `func1`, which is the first instruction of B. The last instruction of node F is function return, which is an indirect branch. Its execution leads to a *Branch* in decoded trace packet. By receiving this *Branch*, we can be aware of the occurrence of a transition from F. The target address is contained in *Branch* and we can examine if it is consistent with the target node B in the CFG.

The architecture of the CFI verification module is shown in Fig. 3.4. Its key components, CFG checker and verification controller, are described as follows.

3.2.2.5.1 CFG Checker The CFG checker is an FPGA circuit that contains CFG information and outputs specific CFG details for given execution trace information. It has n blocks, as shown in Fig. 3.4, each of which corresponds to a node in CFG. Assigning each CFG node in one block makes the CFG node search run in parallel, and this greatly increases the performance of FastCFI.

In detail, there are three main inputs to the checker circuit, all of which are from the decoded trace packets or earlier computations.

- *curr_addr*: current instruction address from trace or earlier calculation.
- *target_addr*: indirect branch target address decoded from *Branch*.
- *tn*: T/N information decoded from *Atom*.

The four main outputs are:

- *next_addr*: the next program counter address after executing the branch of current node according to CFG.
- *node_addr_size*: the start address and size of current node, and function size if the current node is the first node of a function, where the size is equivalent to difference between end and start addresses of a node/function.
- *invalid*: a binary signal whose assertion indicates that the *target_addr* does not conform to the *next_addr*.
- *unspec_target*: a binary signal whose assertion indicates that an indirect branch target depends on application input and is not specified in CFG.

Each block first checks if an input *curr_addr* is within the node corresponding to this block. If so, the block is activated and always generates its *node_addr_size* output. The other outputs vary depending on three different types of blocks. Since each node in CFG contains only one branch instruction at its end, the categorization of blocks is based on their branch instructions.

1. **Direct branch**. An activated block with direct branch generates *next_addr* according to input *tn*. If *tn* is T, indicating that the branch is taken, the *next_addr* can be found in CFG and is hardcoded in the FPGA. Otherwise, the *next_addr* is the address of the next instruction.
2. **Indirect branch with constant target**. When a block with indirect branch is activated, the *target_addr* is compared with the possible *next_addr* from CFG. If they are the same, the *next_addr* is sent to output. Otherwise, signal *invalid* asserts.

3. **Indirect branch with unspecified target**. In this case, *next_addr* is not specified in CFG as the target address depends on software application input and cannot be identified in the offline analysis. Then, *next_addr* is output as *target_addr* and at the same time signal *unspec_target* asserts.

Note that at most one block can be activated in the checker circuit. The checker outputs cover the following scenarios.

C1 **No output**: Current address is not in any CFG nodes.

C2 **There is output**: Current address is in one CFG node, whose start address is found. The start address of the next node is also found.

C3 **Output contains function size**: The current node is at the beginning of a function. The address range of this function is found.

C4 **No *invalid* or *unspec_target* assertion**: The actual control-flow is valid after executing the branch instruction in the current node. The next address after the current node is found so that the actual software execution position is located. Meanwhile, the current node has a direct branch or has an indirect branch with constant target, which the actual execution target address.

C5 ***invalid* asserts but no *unspec_target* assertion**: The current node has an indirect branch with constant target, which is different from the target address of the actual software execution.

C6 ***unspec_target* asserts but no *invalid* assertion**: The current node has an indirect branch with unspecified target in CFG and the verification module is to perform other checks for CFI that will be discussed later in this section.

3.2.2.5.2 Verification Controller The verification controller takes the decoded trace packets as input, feeds input to the CFG checker, and analyzes the checker results to locate current instruction

address, if not available from the trace packets, and performs CFI verification. It is mainly a finite state machine with state transition diagram provided in Fig. 3.5. It also has a function stack, which stores information about the current function, and a return stack that stores function return addresses. These two stacks are the critical parts for realizing the stateful attribute of the proposed system.

The controller operations start from the WAIT state, which attempts to capture executing instruction address from decoded trace packets. This address provides a reference for the verification module to track the software execution location, and can be obtained from *Branch* or *I-sync*.

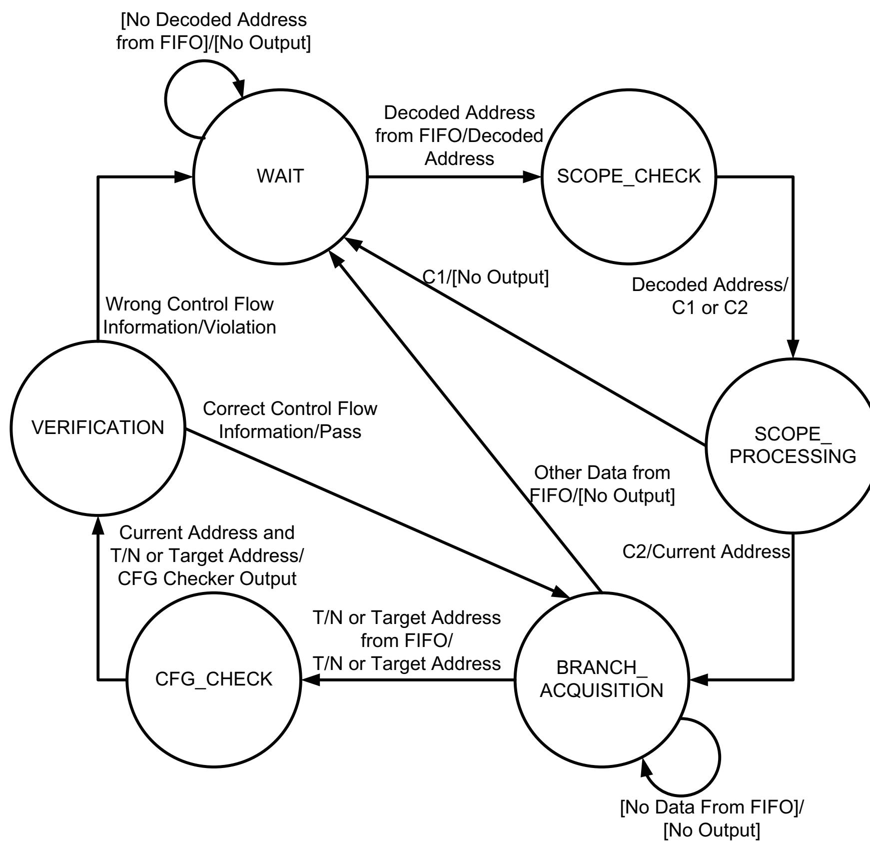

Figure 3.5: State transition diagram of the controller.

Once an executing instruction address is acquired, the controller enters SCOPE_CHECK state, where the instruction address is sent to the CFG checker as *curr_addr* to tell if it is in the scope

of CFG. After the scope checking is finished, SCOPE_PROCESSING state is entered where the controller analyzes the checking result and decides what to do next. If the result is C1, the instruction address is not in the CFG and the next state is WAIT. If the result is C2, the controller records the context ID, which identifies the software execution to be verified, and then moves to state BRANCH_ACQUISITION.

At BRANCH_ACQUISITION, the controller attempts to capture decoded *Atom* or *Branch*, and feeds *tn* or *target_addr* to the CFG checker. If the received trace packet is *I-sync* with current instruction address, the controller switches to the WAIT so as to update the reference instruction address. If branch information, *Atom* or *Branch*, is received, it enters the CFG_CHECK state, where the CFG checker processes the *Atom* or *Branch* information, along with *curr_addr*.

When the CFG checking is finished, the controller switches to VERIFICATION. This state is to analyze the checking results and keep track of instruction execution location. If condition C3 occurs, the function address range is pushed into the function stack. The function stack top always stores the address range of the current function. C3 also implies that the previous node made a function call, and notifies the controller to push the return address onto the return stack.

If the target address of a branch instruction is specified in the CFG, either C4 or C5 holds when the corresponding block is activated. Condition C4 indicates that CFI verification is passed without seeing any violations. Then, the controller updates the current address with the next address and the state goes back to BRANCH_ACQUISITION. Condition C5 shows CFI violation, then the controller outputs a violation signal and goes back to the WAIT state.

Otherwise, if the target address of an indirect branch is not specified, condition C6 holds, which is a very difficult case for CFI verification as the CFG alone does not immediately tell if the actual target address is legal or not. Despite the difficulty, our controller continues to evaluate three sub-cases and detect as many CFI violations as possible. The first case is function return. The controller compares the actual target address from a trace packet with the return address at the top of the return stack. If they are same, the current indirect branch is confirmed to be a function return, which is legal. Note that this check is stateful as it relies on historical information stored in the

return stack. The second one is Branch within current function. The controller checks if the actual target address is within the range of function address at the top of the function stack. If the check result is yes, no violation signal is triggered. The third one is Branch as a new function call. If the actual target address is not in current function, the only legal scenario is that a new function call is made. To verify if a new function call is indeed made, the controller updates the current address with the next address and waits for the next BRANCH_ACQUISITION and CFG_CHECK result. If the next result indicates C3 and the current address is the same as the new function entry address, a new function call is confirmed. Evidently, this is also a stateful check. Any other scenario beyond the above three is illegal and then a CFI violation signal is triggered.

The verification is not only stateful, but also fine-grained as its resolution is on each individual edge in the CFG. We also re-emphasize that our work is general and flexible enough to be applied with other code static analysis tools.

3.2.2.6 CFG Compression

According to Section 3.2.2.5, the direct branches are safe when the Write$\oplus$Execute [79] technique is applied. The blocks corresponding to CFG nodes with direct branches are only used by verification controller for tracking the current node and instruction address. In this case, we found one scenario where the current node does not have to be precisely tracked by the verification controller, and therefore the CFG nodes and CFG checker blocks can be trimmed to reduce hardware expense.

Let us consider a simple case. If there is a set $\mathcal{S}$ of CFG nodes with direct branches, every CFG node in $\mathcal{S}$ has no path to other CFG nodes outside this set, then once the current instruction address is at the address range of one node in $\mathcal{S}$, all the subsequent control-flow transitions can be ignored. This is because all the later control-flow transitions must be from direct branches and they do not have to be checked. Then the CFG checker blocks corresponding to the nodes in $\mathcal{S}$ can be removed and thus, the hardware expense is saved.

However, this simple case rarely exists in a realistic program. In a realistic program, there are often indirect branches (such as function return) that have to be checked. A set $\mathcal{S}$ of CFG

nodes with direct branches is likely to have at least one path to CFG nodes with indirect branches. Therefore, we allow one additional CFG node k with an indirect branch to be included in $\mathcal{S}$, and the possible outgoing edges from $\mathcal{S}$ are only from node k. In this way, when the current instruction address is in the address range of one node in $\mathcal{S}$, all the later direct branch control-flow transitions can be ignored, and only indirect branch control-flow transitions have to be checked by CFI verification module. The indirect branch transitions are only from node k that contains the only indirect branch in $\mathcal{S}$. All the nodes in $\mathcal{S}$ can be replaced by one new node N named *coalesced node*, of which the address range is the union of all the address ranges of the nodes in $\mathcal{S}$. The edges to and from $\mathcal{S}$ are also replaced by the edges to and from N. The CFG checker blocks corresponding to the nodes in $\mathcal{S}$ are replaced by only one block corresponding to N, thus, the hardware consumption is reduced. Once the instruction address is in the address range of N, CFI verification module can ignore all the decoded trace packets from direct branches, until there is a decoded trace packet from an indirect branch, which indicates the indirect branch k is taken. Then, the control-flow from the indirect branch k is checked.

Meanwhile, the additional indirect branch k has to be unconditional, because if it is conditional and is not taken, PTM would generate an *Atom* trace packet for this control-flow transition, which cannot be distinguished from the *Atom* trace packets of the direct branches.

Nevertheless, if there are two or more additional CFG nodes with indirect branches in $\mathcal{S}$, when the current instruction address is in the address range of one node in $\mathcal{S}$ and there is an indirect branch control-flow transition, there is no clue about which indirect branch in $\mathcal{S}$ is the source of this control-flow transition. Then, there is no way to check if the control-flow transition is legal or not. Therefore, only one additional CFG node with the indirect branch is allowed to be included in $\mathcal{S}$.

Based on this idea, an algorithm is developed to further compress CFG and CFG checker by replacing multiple CFG nodes by one coalesced node, and each coalesced CFG node still corresponds to one block in CFG checker. CFG checker can spend less hardware resource without losing any precision.

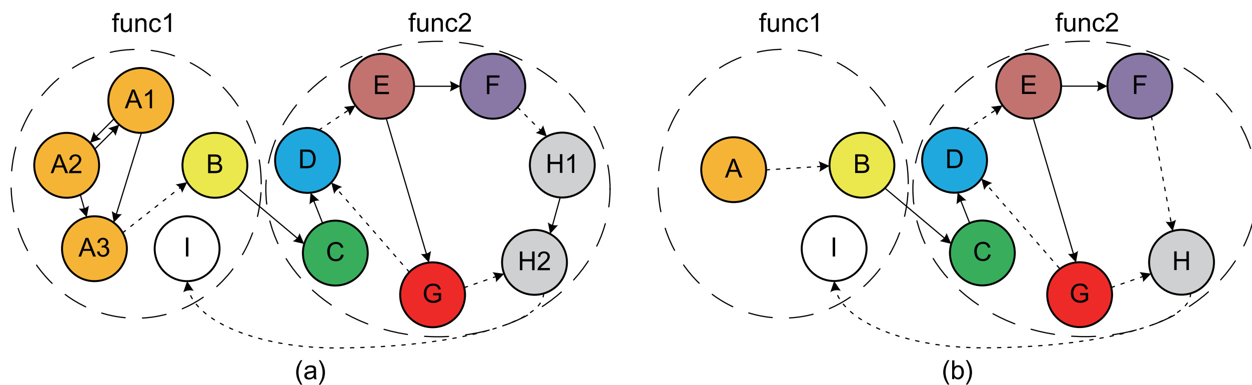

Figure 3.6: An example of CFG optimization.

Consider the example in Fig. 3.6, where Fig. 3.6(a) represents the CFG and Fig. 3.6(b) represents the compressed CFG. In Fig. 3.6(a), the CFG nodes are from two functions $func1$ and $func2$, which are marked by the dashed circles. The legal control-flow transitions by direct branches and indirect branches are illustrated by the solid and dotted arrows, respectively. One can easily figure out when the control-flow transits between node $A1$, $A2$, and $A3$, there are only direct branch control-flow transitions, thus, PTM will not generate any *Branch* but only *Atom* trace packets until the current control-flow is $A3 \rightarrow B$. Since the direct branches do not have to be checked and only $A3$ contains an indirect branch, node $A1$, $A2$, and $A3$ can be replaced by one node A shown in Fig. 3.6(b). The CFG checker block corresponding to A is basically generated by the same way described in Section 3.2.2.5, except:

SP1 The block is activated if an input *curr_addr* is within the range of instructions' addresses of one of the corresponding nodes before replacement.

SP2 Only if there is *target_addr* decoded from *Branch*, the block begins checking. Otherwise, the block outputs *curr_addr* to *next_addr* port. The output port *node_addr_size* is the start address and instruction size (in bytes) of the node with the indirect branch.

In this way, if the current instruction address is at $A1$, $A2$, or $A3$, all the T/N information from *Atom* trace packets are ignored until $A3 \rightarrow B$, which lets decoder decode a *target_addr* by a *Branch* trace packet.

Assuming that there is a set of CFG nodes $\mathcal{S}$, $\mathcal{S}$ can be replaced by one coalesced CFG node if and only if $\mathcal{S}$ satisfies the following requirements:

R1 $\mathcal{S}$ contains one and only one CFG node k with an indirect branch, and this indirect branch should be unconditional.

R2 Except node k, there is no path from each node in $\mathcal{S}$ to another node, which is not in $\mathcal{S}$.

R3 There is no node in $\mathcal{S}$ of which the instructions are at the beginning of one function.

Using Fig. 3.6(b) as an example for illustrating the requirements, $\{H1, H2\}$ can be replaced by H. Note that after the replacement, although $H2$ no longer exists, illegal transition $F \to H2$ can still be identified because the indirect branch target is checked by the block corresponding to F but not $H2$. Set $\{E, F\}$ does not satisfy R2 and cannot be replaced. If they are replaced, once the current control-flow is $E \to G$, it is ignored by CFG checker because this control-flow is from a direct branch. Then, the current instruction address kept by the verification controller is not updated to G, and the legal control-flow $G \to D$ is wrongly regarded as the control-flow from set $\{E, F\}$ to D, which is illegal. Thus, false alarm is raised. Meanwhile, if C is at the beginning of the function *func2*, then set $\{C, D\}$ does not satisfy R3 and cannot be replaced by an coalesced node, since verification controller pushes the function address range to the function stack when the current instruction address is in the address range of C. If $\{C, D\}$ is replaced, verification controller cannot tell if the current instruction address is at C or D, and thus, cannot tell if the function stack should be pushed or not.

Given a CFG graph G, the algorithm to find the compressed graph G' is shown as Algorithm 1. The first part (lines 1-11) of the algorithm is to find if a CFG node can be a candidate to be included in a set to be compressed. Line 1 construct a new graph G^d without edges representing indirect control-flow transitions. This new graph can be used to find the set $\mathcal{P}$ of all the nodes that each node n can reach through the direct control-flow transitions. Lines 4-5 correspond to this part. Each node n is a candidate only if n is likely to be included into a set to be compressed without violating any of the requirements (R1 to R3). Firstly, if a node n can reach multiple nodes k_1, k_2..

with indirect branches, n cannot be included into any set $\mathcal{S}$ to be compressed. This is because $\mathcal{S}$ can only contain one node k_i with an indirect branch according to R1. If $n \in \mathcal{S}$, there must be paths from n to another node k_j that is outside $\mathcal{S}$, and this violates R2. Secondly, n should have no path to a node m that is at the beginning of one function. Otherwise, if $n \in \mathcal{S}$ and $m \in \mathcal{S}$, R3 is violated, and if $m \notin \mathcal{S}$, R2 is violated. In summary, only if node n passed the two checks mentioned above (lines 6 and 8), n is chosen as candidate and appended into candidate set $\mathcal{C}$ (line 11).

Algorithm 1: The algorithm to construct compressed CFG.

Input: *G:- CFG graph*

Output: *G':- Compressed CFG graph*

1 *Construct $G^d = G$, and remove all the edges $n \rightarrow m$ in G^d if node n contains an indirect branch;*

2 *Construct a candidate set $\mathcal{C} = \varnothing$;*

3 **for** *each node n in G^d* **do**

4   *Do depth first search (DFS) in G^d from n;*

5   *Construct a set $\mathcal{P}$ containing all the searched nodes during the DFS;*

6   **if** *there exists a node $m \in P$ that is at the beginning of a function* **then**

7   *Continue;*

8   **else if** *!(there is only one $k \in P$ with an indirect branch, and this indirect branch is unconditional)* **then**

9   *Continue;*

10   **else**

11         $\mathcal{C} \leftarrow \mathcal{C} \cup \{n\}$;

12 *Construct a graph G^{dr}, where the nodes are the same as G^d, with all the edges reversed;*

13 **for** *each node n in G^{dr}* **do**

14   **if** *n contains an unconditional indirect branch* **then**

15   *Do DFS in G^{dr} from n; Stop searching the branch $m_i \rightarrow m_j$ if $m_j \notin C$;*

16   *Construct a set $\mathcal{S}$, which contains all the searched nodes during the DFS this time;*

17   *Add a node N to G;*

18   *Add an edge $l \rightarrow N$ if there exists $l \rightarrow m$ in G where $m \in \mathcal{S}$;*

19   *Add an edge $N \rightarrow l$ if there exists $m \rightarrow l$ in G where $m \in \mathcal{S}$;*

20   *Remove all $m \in \mathcal{S}$ from G;*

21 *Output $G' = G$;*

The second part (lines 12-21) of the algorithm is to compress the CFG. According to R1, we can construct the set $\mathcal{S}$ to be compressed starting from each node n with an unconditional indirect branch (lines 13-14). By constructing a new graph G^{dr} that is the same as G^d, with all the edges reversed, we can search all the nodes at n's predecessor side by depth first search (lines 15-16). However, we should stop searching a branch $m_i \rightarrow m_j$ once m_j is not the candidate, which means m_j cannot be included into $\mathcal{S}$. After we get the set $\mathcal{S}$ that can be compressed, new node N is added, which is used to replace the nodes in set $\mathcal{S}$ (lines 17-20). After all the possible replacement, the compressed graph is returned to the output (line 21).

After G' is created, the blocks in CFG checker are automatically generated the same way as the CFG checker generated from G in Section 3.2.2.5. The behavior of each block in CFG checker is still the same, except the additional behaviors SP1 and SP2 for the blocks corresponding to the node that is generated during compression.

3.2.3 Hardware Implementation

All the hardware modules are described in Verilog HDL and synthesized to FPGA implementation. This section is focused on CFG checker and pipelined trace decoder, which are the two relatively sophisticated modules compared to the others. Most of the hardware modules are applicable for general software applications, except the CFG checker design, which is specific to each individual software application. In order to mitigate the design overhead of the CFI system for each application due to different CFG checker requirements, we develop an automated Verilog generation technique for CFG checkers. As such, there is no need to manually write Verilog code for each application, which can be quite time consuming. Although Verilog code can also be obtained through High Level Synthesis (HLS) of C code, manually writing C code for the CFG check of each application is not efficient either.

3.2.3.1 Automatic Verilog Generation for CFG Checker

The CFG checker is a Verilog module where each CFG node is implemented as a block. The block is a hardware circuit that can check if the control-flow from its corresponding CFG node is

valid or not. Different application softwares have different CFGs and thus require different CFG check implementations. However, in CFG checker, there are only 3 types of blocks corresponding to the CFG nodes with 3 kinds of branch instructions, which are direct branches, indirect branches with constant targets, and indirect branches of which the targets are unspecified. Different CFGs only need to be implemented by using these 3 types of blocks, and blocks of the same type share the same structure with the only difference on parameters. Therefore we can prepare 3 Verilog description templates for all the three types of blocks. Given a CFG, each of its nodes can be mapped to one of the templates and thereby the Verilog code of an entire checker can be automatically generated. The software program is designed by us and written in Python, and it is different from general High Level Synthesis (HLS) programs that can synthesis a general level language program into hardware circuit. The software program designed by us is specifically designed only for generating the CFG checker, with the use of 3 Verilog templates. The input of our software program is CFG but not high level language such as C.

The pseudo codes of the 3 Verilog templates are shown in Fig. 3.7, 3.8 and 3.9. The parentheses of the `function size` in Fig. 3.7, 3.8 and 3.9 mean that only when the corresponding CFG node is the entry of a function, function size is included. For input `tn`, '0' indicates *N* (the branch is not taken) and 1 implies *T* (the direct branch is taken). For `invalid` and `unspec_target`, '0' and '1' means false and true, respectively.

When the current address is in the range of a CFG node corresponding to a block, the condition of the *if* statement (line 1 in Fig. 3.7, 3.8 and 3.9) holds, and the start address of the node and the instruction size (in bytes) of the node are outputted to port `node_addr_size`. If this CFG node is a function entry, `node_addr_size` of the corresponding block also contains the instruction size of the function (line 2 in Fig. 3.7, 3.8 and 3.9).

If a CFG node contains a direct branch, the corresponding block is realized as shown in Fig. 3.7. When the direct branch is taken, PTM sends an *Atom* trace packet that can be decoded as *T*, and the verification controller inputs 1 to `tn`. Then, the next instruction address `next_addr` is updated as the target address of the direct branch (line 6). Otherwise, if the direct branch is not taken, *N*

is decoded from the trace decoder and the verification controller inputs 0 to `tn`, and this makes the block output the address of the instruction right after this direct branch as the next instruction address (line 4). Since direct branch is assumed to be unable to be attacked, `invalid` is outputted as 0. The target of the direct branch is not unspecified, so `unspec_target` is outputted as 0.

```
1  if(curr_addr in the range of this CFG node){
2      node_addr_size=node start address, node size, (function size);
3      if(tn==0){
4          next_addr=the address of the closest instruction after curr_addr;
5      }else if(tn==1){
6          next_addr=target address of this direct branch;
7      }
8      invalid=0;
9      unspec_target=0;
10 }
```

Figure 3.7: The Verilog template for CFG checker block with direct branch.

If a CFG node contains an indirect branch with constant targets, the corresponding block is realized as shown in Fig. 3.8. When the indirect branch is taken, PTM sends an *Branch* trace packet that can be decoded as a target address, and the verification controller inputs this target address to `target_addr`. The block begins to check if the target address is a valid target or not. If it is, the next instruction address is updated as this target address, and `invalid` is 0 since this control-flow is valid (lines 7-8). Otherwise, this control-flow is invalid (line 10). If the indirect branch is not taken, an *Atom* trace packet is generated from PTM and *N* is decoded from the trace decoder. The verification controller inputs 0 to `tn`, and this makes the block output the address of the instruction right after this direct branch as the next instruction address (line 4). The target of the indirect branch is not unspecified, so `unspec_target` is outputted as 0.

```
1  if(curr_addr in the range of this CFG node){
2      node_addr_size=node start address, node size, (function size);
3      if(tn==0){
4          next_addr=the address of the closest instruction after curr_addr;
5          invalid=0;
6      }else if(target_addr is one of the valid addresses){
7          next_addr=target_addr;
8          invalid=0;
9      }else{
10         invalid=1;
11     }
12     unspec_target=0;
13 }
```

Figure 3.8: The Verilog template for CFG checker block with indirect branch with constant target.

```
1  if(curr_addr in the range of this CFG node){
2      node_addr_size=node start address, node size, (function size);
3      if(tn==0){
4          next_addr=the address of the closest instruction after curr_addr;
5      }else{
6          next_addr=target_addr;
7      }
8      invalid=0;
9      unspec_target=1;
10 }
```

Figure 3.9: The Verilog template for CFG checker block with indirect branch with unspecified target.

If a CFG node contains an indirect branch with unspecified target, the corresponding block is realized as shown in Fig. 3.9. Fig. 3.9 is similar to Fig. 3.8 except that the target address input is not checked by this block, but directly outputted as the next instruction address. The target address is checked by the verification controller described in Section 3.2.2.5.2. Meanwhile, `unspec_target` is set to 1 due to the unspecified target (line 9).

```
1  if(curr_addr in the range of this CFG node){
2      node_addr_size=node start address, node size;
3      if(target_addr==0){
4          next_addr=curr_addr;
5          invalid=0;
6      }else if(target_addr is one of the valid addresses){
7          next_addr=target_addr;
8          invalid=0;
9      }else{
10         invalid=1;
11     }
12     unspec_target=0;
13 }
```

Figure 3.10: The Verilog template for CFG checker block with indirect branch with constant target for coalesced CFG node.

For the coalesced CFG node, according to Section 3.2.2.6, the outgoing edges are from the unconditional indirect branch in the coalesced node, so the Verilog template for the CFG checker block of coalesced CFG node is based on the block with the indirect branch, which is Fig. 3.8 or 3.9, depending on the indirect branch has constant targets or not. The Verilog templates for the coalesced node are shown in Fig. 3.10 and 3.11. When there are only direct branch control-flow transitions, only *Atom* trace packets are generated by PTM and only *T/N* information is decoded.

The `target_addr` is 0 in this case (line 3 in Fig. 3.10 and 3.11), which represents there is no *Branch* trace packet decoded and thus, there is no indirect branch that is taken, and the direct branch control-flow transitions are ignored by output `curr_addr` as `next_addr` (line 4 in Fig. 3.10 and 3.11). These two Verilog templates ignore all the direct branch control-flow transitions until there is an indirect branch taken (lines 6-11 in Fig. 3.10 and lines 6-9 in Fig. 3.11), according to Section 3.2.2.6.

```
1  if(curr_addr in the range of this CFG node){
2      node_addr_size=node start address, node size;
3      if(target_addr==0){
4          next_addr=curr_addr;
5          unspec_target=0;
6      }else{
7          next_addr=target_addr;
8          unspec_target=1;
9      }
10     invalid=0;
11 }
```

Figure 3.11: The Verilog template for CFG checker block with indirect branch with unspecified target for coalesced CFG node.

Each block has its own output, and the MUX is implemented by bitwise OR to select the checker output among all blocks, as shown in Fig. 3.4. For each block, if the current address is not in the range of CFG node corresponding to this block, all the output bits of this block are 0, indicating no output from this block. Because the current address belongs to at most one block, all the outputs of the other blocks are 0, then the checker output is always from the block where the current address belongs to. If the checker output is 0, the current address is out of the ranges of CFG nodes represented by all blocks (condition C1).

The CFG checker design is written in Verilog and compiled by Quartus Prime 17.1 to generate the circuits on FPGA. In general, the compilation tool strives to achieve optimized logic circuits with high performance and low resource use. However, optimizing a large number of blocks is more difficult than optimizing fewer blocks. Therefore, we develop a hierarchical approach that groups blocks into small Verilog modules. Note that the group is a different concept from the CFG set to be replaced during CFG compression in Section 3.2.2.6. The CFG checker is never trimmed during the hierarchical approach. For example, in Fig. 3.12, the CFG checker has 6 blocks, $B0$ to $B5$, which are grouped into two small modules. Each small module takes the checker input to all of its internal blocks, and selects an output among all of its internal blocks. In this way, the compiling optimization is directed to perform in a hierarchical manner to reach different resource use and compiling time tradeoffs.

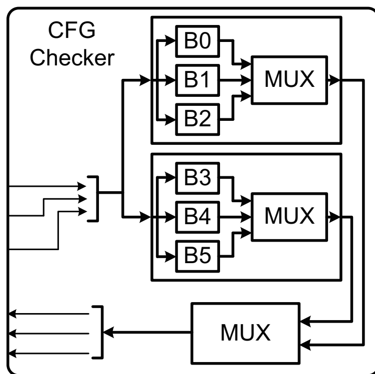

Figure 3.12: An example of grouping blocks in CFG checker.

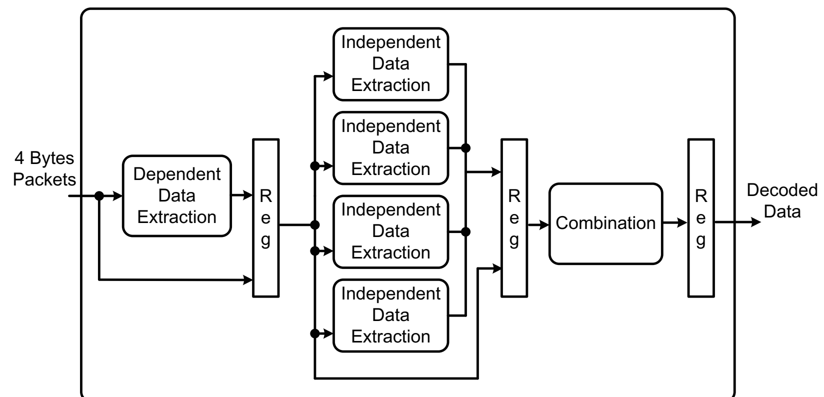

Figure 3.13: Structure of the pipelined decoder.

3.2.3.2 Pipelined Trace Decoder

Implementing a fast decoder for the traces generated by ARM CoreSight is challenging. The TPIU channel in the ARM core has 32-bit bitwidth, which means 4 bytes of packets can be sent to FPGA in every clock cycle. To match this speed, we must design a decoder that can decode 4 bytes in each clock cycle. A naïve design is to have the same 4 units of decoding modules, and

each module can decode 1 byte. However, each packet may contain multiple consecutive bytes, and the decoding of the latter byte may depend on the decoded information from the former byte. For example, the highest bit of the first byte in *Branch* trace packet indicates if there is the second byte in this *Branch* packet. For the naïve design, if the decoding module unit A is decoding the first byte of *Branch* trace packet, decoding module unit B for decoding the next byte needs to wait the result of A to identify if the byte for B is the second byte of this *Branch* packet or the first byte of a new trace packet. Moreover, not only each byte in one *Branch* packet are dependent, but also the latter *Branch* packet depends on the former *Branch* packet, because each *Branch* packet only carries the value of the changed bits comparing to the instruction address decoded from the most recent *Branch*. Due to the dependency, when using the naïve design, the 4 units need to be designed in a serial chain. However, such design is vulnerable to timing errors when implementing under our experiment setup, since the the serial chain structure has a long critical path. Meanwhile, for each byte of a trace packet, not all the bits have dependency relationships. For example, the decoding of the byte next to the first byte of *Branch* trace packet only depends on the highest bit of the first byte of *Branch* trace packet. Based on this idea, we design a pipelined decoder in order for robust decoding operations without timing errors.

The structure of the pipelined decoder is illustrated in Fig. 3.13. The pipeline has 3 stages: dependent data extraction, independent data extraction, and combination. The first stage decodes all the dependent information in each byte, so the four bytes are decoded serially in this stage. However, there is no timing error in our experiments since this stage only decode the necessary bits in each byte that have the dependency, but not all 8 bits like the naïve design, and the serial circuit in this stage is simpler than the serial structure of the naïve design. Then, the second stage extracts the independent data of every byte (such as certain bits of the branch target address), based on the decoded information of the first stage (such as the trace packet type of each byte). For example, we may extract bit 13 to bit 7 of target address from the second byte of *Branch*. The second stage can be performed in parallel since only the independent part of the 4 bytes are extracted. After that, the third stage combines all the information we have extracted from the 4

bytes of packets. In order to accommodate *Branch* trace packets, which may depend on previous *Branch* trace packets, this stage is also serial. Since this stage only does information combination serially but not the complete decoding procedure serially, the serial circuit is also simpler than the serial structure of the naïve design, and there is also no timing error in our experiments.

Note that the pipeline registers also carry the raw packet information and transmit it through different stages. For example, for the second stage, not only the output of the first stage, but also the raw packets are needed.

3.3 Hardware-assisted Data-Flow Integrity Verification

3.3.1 Background

3.3.1.1 Data-Flow Integrity (DFI)

Data-flow integrity requires that data to be loaded from memory can only be stored by legitimate instructions that are consistent with the programmer's original intention. DFI is a superset of Control-Flow Integrity (CFI), which only regulates instruction flow transitions toward target addresses conforming to the original design intention. The attackers have to modify the control data to modify the control-flow, such as the data used as the control-flow transition target of an indirect call. By protecting all the data, DFI can also prevent all the control-flow attacks. Besides control data, DFI can even protect non-control data that cannot be protected by CFI. In the program of Figure 3.14, line 4 determines whether the user has administrator permission or not. Then, line 5 reads user's input data and stores them into `buffer`. However, there can be buffer overflow in function `read_data` and consequently more than 32 bytes are written into `buffer`. If the address of `admin` is larger than `buffer`, `admin` may be modified by this buffer overflow. This vulnerability can be exploited by an attacker to illegally obtain administrator permission. Please note that such attack cannot be detected by CFI, because line 7 is a legal target of the branch instruction compiled from line 6. DFI requires that when `admin` is accessed in line 6, it should be most recently stored by only line 4. If it is stored by line 5, DFI is violated and therefore can be detected.

```
1  char buffer[32];
2  int admin;
3  ...
4  admin=identify_user();
5  read_data(buffer, user_input);
6  if(admin){
7    process_data_under_admin(buffer);
8  }
```

Figure 3.14: An example of vulnerable code.

```
1  store x1 addr1
2  store x1 addr2
3  jump label
4  store x2 addr1
5  load x3 addr1
6  label:
7  load x4 addr1
```

Figure 3.15: A code example for illustrating DFI.

Every instruction in a given program has a numerical **identifier**. The **reaching definition** of an instruction `A` is the latest instruction `B` that stores the data loaded by `A` and is represented by the identifier of `B`. Each instruction that can load data from the memory has its own **reaching definition set (RDS)**, which consists of all the possible reaching definitions of this instruction. A static analysis can be performed for a program to obtain the RDSs for all the instructions that can load data from the memory. In the example of Figure 3.15, "`store x y`" means storing variable `x` at address `y`, "`load x y`" is to load the data at address `y` to variable `x`, and "`jump label`" implies an unconditional branch to the location marked by `label`. In this example, if the identifier of each instruction is the same as its line number, the RDS of line 7's instruction is

{1}. DFI requires that all the instructions that can load data from the memory are consistent with their RDSs, i.e., when executing an instruction `A` that loads data from the memory, the data should be indeed most recently stored by one of the instructions in the RDS of `A`. Hence, the identifier of the latest instruction that stores a data needs to be tracked for the data. Such identifiers for all data form a **reaching definition table (RDT)**. Since a large number of software attacks need to change some data in the memory, DFI enforcement can detect a wide range of attacks.

3.3.1.2 Processing-In-Memory (PIM)

Usually, memory (DRAM) is a stand-alone chip separated from processors, and hence accessing data in memory takes a long time, which is a well-known bottleneck to processor performance. The idea of PIM is to perform some computations near or at memory so that the bottleneck is largely circumvented. This idea was already proposed [21] in 1994 and has been extensively studied [80–89] later on. Recently, along with the progress of 3D-stacking technology, PIM is practically implemented as Hybrid Memory Cube (HMC) [22] and High Bandwidth Memory (HBM) [23,24] in a Near Data Processing (NDP) manner [90–96]. HBM was used in AMD GPU from 2015 [97], which has 512GB/s bandwidth. Nvidia also adopts HBM in their GPU products such as Tesla P100 [98] with memory bandwidth 732GB/s. Micron developed its HMC Gen2 [99] with bandwidth 160GB/s. In our work, the PIM structure for NDP is composed by memory dice as DRAM and a logic die that contains the PIM processor. The PIM processor is a light weighted processor with computing capability similar to ARM Cortex A5 and typical power consumption less than 100mW [86,93,94,100].

3.3.2 Overview of the Proposed Approach

We first summarize the information required for DFI verification as follows.

1. RDSs (Reaching Definition Sets) for all `load` instructions in the program. This information does not change throughout the program execution and can be loaded into the PIM processor once at the beginning.

2. RDT (Reaching Definition Table). This information changes dynamically during the program

execution. It is established and maintained by the PIM processor, and therefore does not need to be transferred from the processor core.

3. Target instruction information. A target instruction is an instruction in a user program to be verified for DFI. Two types of instructions are involved: `load` instructions for which DFI verification is performed, and `store` instructions that affect RDT. These informations change at runtime and need to be transferred from the processor core to the memory. It consists of the following components:

   - Instruction type: either `load` or `store`.
   - Instruction identifier.
   - Target address of `load` or `store`.

The hardware architecture, where the proposed DFI enforcement is carried out, is shown in Figure 3.16(b). Except that the memory contains a PIM processor, it is similar to an ordinary processor system in Figure 3.16(a), which is used by software DFI [16].

Generally speaking, RDSs have the smallest amount of data and RDT has the largest amount of data among the three types of information. The size of RDT is roughly 50% of overall data size used by the user program, and its entries are used repeatedly during DFI verification. The amount of data of target instruction information is roughly in the same order as storage size of all instructions in the program, with one inaccuracy that an instruction can be executed multiple times. In the software DFI [16], target instruction information is obtained and consumed locally without being moved around, while RDSs and RDT reside in memory and need to be used at the processor, as the red arrows shown in Figure 3.16(a). Although cache can reduce some transfer of RDS/RDT data, the reduction is limited as the RDT size is usually much larger than cache. In our approach, RDSs and RDT also reside in memory, and are consumed locally at memory, as the green arrows shown in Figure 3.16(b). The data transfer in our approach is dominated by moving target instruction information from the main processor to memory, as the red arrows shown in Figure 3.16(b). As RDT size is significantly larger than data size of target instruction information,

our approach can significantly reduce data transfer between main processor and memory compared to the software DFI [16].

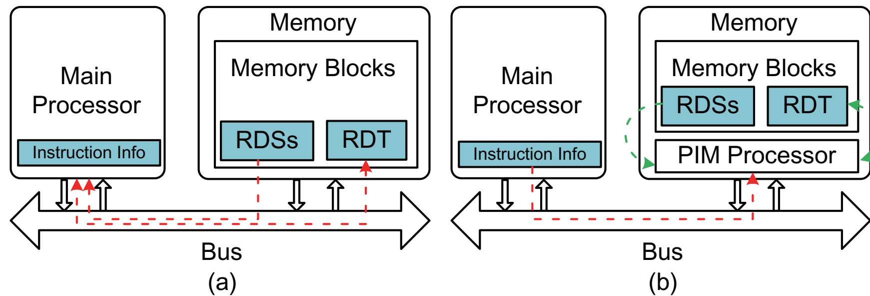

Figure 3.16: (a) The architecture for software-based DFI enforcement. (b) The architecture for our hardware-based DFI enforcement.

The overall flow of the proposed DFI enforcement is depicted in Figure 3.17, where the green numbers indicate the order of steps:

1. Static analysis is performed for the program.
2. RDSs are obtained from the static analysis.
3. The codes are instrumented. The main instrumentation is to add `store` instruction after each target instruction so that its information is sent to the PIM processor. The instrumentation is illustrated by the red font instructions in Figure 3.17. These instrumentation `store` instructions are called **DFI `store`**. For example, in Figure 3.17, line 2 is an instrumentation instruction `store "load, id=12"`, which tells the instruction type and identifier of the target instruction in line 1. Please note the target address is not included in the DFI `store`, but can be easily inferred from the prior target instruction.
4. The DFI verification program and RDS are loaded onto the PIM processor before the user program execution starts on the main processor.
5. During the program execution, each DFI `store` sends the PIM processor the instruction type and identifier of the target instruction. A hardware module, named *info-collector* in Figure 3.17,

is added in the main processor to extract the target address from the instruction prior to each DFI `store`. Then the info-collector forms the target instruction information including instruction type, identifier and target address as a **DFI packet**. For an ordinary instruction, the module relays its input `addr/data` to its output `addr'/data'` without change. For a DFI `store`, the `data'` produced by the module is a DFI packet and `addr'` is the address in the packet **FIFO memory** allocated in the memory. The dotted box in Figure 3.17 also keeps track of the target address of the latest `load` and `store` executions. At the same time, The DFI verification is performed at the PIM processor.

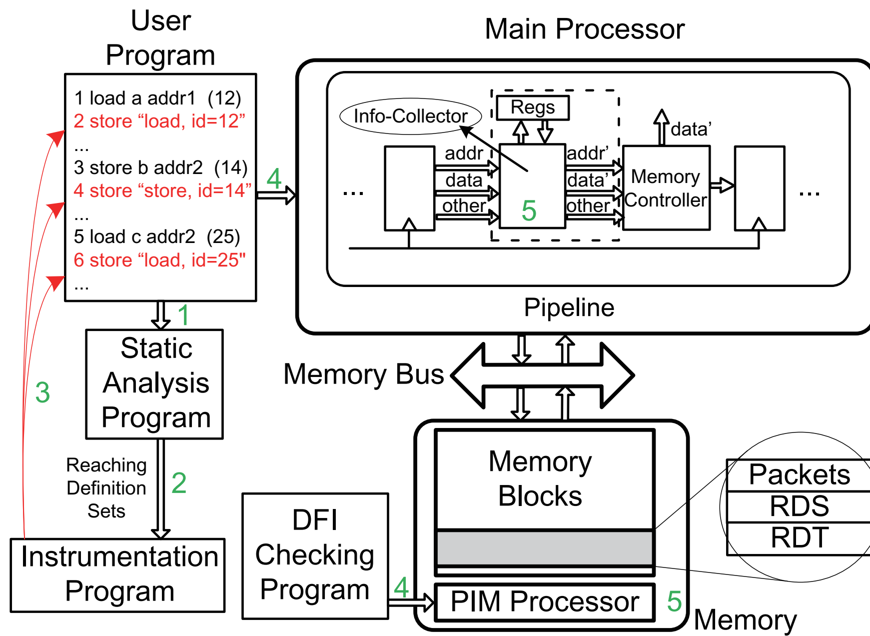

Figure 3.17: The flow for DFI verification.

3.3.3 Software Program Instrumentation

The software instrumentation in our approach helps not only extract the necessary information but also avoid changing the instruction set. The description here is based on C/C++ programs compiled by LLVM [101] and the static analysis is performed by SVF [102]. However, our techniques are general and directly applicable to other software languages, compilers and static analysis tools.

3.3.3.1 Instrumentation for DFI Verification

Given a program's LLVM IR (Intermediate Representation), static analysis is performed to obtain its RDS, which will be sent to the PIM processor once at the beginning of code execution. The instrumentation is performed onto the IR automatically. Then, the instrumented IR is further compiled into binary code.

The instrumentation is mainly to extract the runtime information including instruction type and identifier, either `store` or `load`, and send to the PIM processor. This is realized by an overload use of the `store` instruction, which is DFI `store`. Therefore, DFI `store` is a special use of `store` instruction with different underlying semantics from the ordinary use of `store`. A key part of our instrumentation technique is on how to differentiate between ordinary `store` and DFI `store` without any instruction change.

The basic syntax of the DFI `store` is

```
store runtime_info dfi_global
```

where `dfi_global` is a global variable declared at the beginning of the program and serves as a signature to indicate a DFI `store`. The address of this global variable is set by writing a dummy value at the beginning of the program as:

```
store dfi_dummy dfi_global
```

The info-collector (dotted box in Figure 3.17) checks if a `store` instruction has a target address the same as that of `dfi_global`. If yes, then the instruction is a DFI `store`.

Every `store` and `load` instruction in the original user program, which is called target instruction, is followed by a DFI `store`. The `runtime_info` contains the instruction type and identifier of the proceeding target instruction. When the info-collector recognizes a DFI `store`, it extracts the target address of the proceeding target instruction. The target address and the `runtime_info` form a DFI packet to be sent to the PIM processor. At the beginning of code execution, a memory space is allocated at the PIM processor for DFI verification. This includes the memory space for storing an incoming packet, which is called packet FIFO memory. The starting address of packet

FIFO memory is packet_mem_addr. It is specified by adding the following instruction at the beginning of the original program:

```
store packet_dummy packet_mem_addr
```

The packet_dummy is a dummy packet to obtain the destination address for future DFI packets. Later during code execution, all DFI packets are sent to FIFO memory based on packet_mem_addr. Please note that dfi_global and packet_mem_addr are generated by the automatic code instrumentation, and not visible to security attackers.

An example of the instrumentation is shown in Figure 3.18.

```
1   =====beginning of the program======
2   (instructions for allocating memory regions)
3   (instructions for storing RDS to memory)
4   store dfi_dummy dfi_glabal
5   store packet_dummy packet_mem_addr
6   ...
7   store x1 addr1      //(12)
8   store (0<<16)+12 dfi_global
9   ...
10  load x2 addr2       //(25)
11  store (1<<16)+25 dfi_global
```

Figure 3.18: An example of code instrumentation.

In Figure 3.18, lines 7 and 10 are the original instructions in the user program, while lines 2, 3, 4, 5, 8 and 11 are our instrumentations. The identifier of the instructions at lines 7 and 10 are in the parentheses (12 and 25). According to [16], 16 bits are sufficient for representing instruction identifiers in a large program. We use an additional bit to indicate instruction type, where 0 means write and 1 means read. Putting together, the data of a DFI store (lines 8 and 11 in Figure 3.18) has bit 16 for instruction type and bits 15-0 for storing an instruction identifier.

3.3.3.2 *Handling Library Functions*

A software program often calls library functions, whose source code or IR is not directly accessible. However, instrumentation can still be performed to obtain the target instruction information, which is a library function call. This is similar to the wrapper approach [16] in spirit, but the realization of our approach is quite different. As a library function call may involve a multi-byte data block in general, the instrumentation needs to keep track of data-length besides data address. We describe our approach through the example of Figure 3.19.

```
1   store (1<<20)+(1<<19)+(0<<18)+(1<<17)+7 dfi_global
2   store (y1's addr) dfi_global
3   store (x1's addr) dfi_global
4   store 40 dfi_signal
5   memcpy(x1, y1, 40)     //(7)
6   ...
7   store (1<<20)+(0<<19)+(1<<18)+(1<<17)+15 dfi_global
8   store (x2's addr) dfi_global
9   store 12 dfi_global
10  store 9 dfi_global
11  memset(x2, 3, (9<<32)+12)     //(15)
```

Figure 3.19: The instrumentation for library functions.

In this example, the target instructions are the function calls in lines 5 and 11, with their identifiers in parentheses. The instrumentation for each library function call is multiple DFI `store` instructions like lines 1-4 for the target instruction of line 5. The first DFI `store` keeps the corresponding identifier in its lower 16 bits. Its bits 17-20 are four binary indicators telling if the target instruction is a library function call or not, if the data-length needs 64 bits to represent or not, and if the function loads/stores data or not.

The info-collector parses these indicators and then takes corresponding actions. Additional verification `store` instructions are added to send other information to the PIM processor. For

example, lines 2 and 3 send load and store addresses. Depending on if the data-length is represented in 32 or 64 bits, the data-length is needed to be sent through a single or two DFI `store` instructions. For example, line 4 sends the data-length in a single DFI `store` while lines 9 and 10 send in two DFI `store` instructions.

Compared to the work of [16], our approach considerably reduces the data transfer between the main processor and memory. Consider an example that a `memcpy` loads 128 words and stores them to another address. As the work of [16] needs to transfer the identifiers from 128 entries of the RDT in memory to the main processor, it needs to store another 128 entries of the RDT back to the memory. By contrast, our approach only needs to send 5 words from the main processor to the memory, as the RDT is maintained at PIM.

3.3.3.3 Function Return Protection

```
1 =====beginning of the function======
2 p_ret_addr = instruction_getting_ret_addr_pointer
3 store (1<<21)+(max_id+thread_id) dfi_global
4 store p_ret_addr dfi_global
5 ...
6 store (1<<21)+(1<<16)+(max_id+thread_id) dfi_global
7 store p_ret_addr dfi_global
8 return
9 =====end of the function======
```

Figure 3.20: Instrumentation for function return.

Function return addresses are stored in stack and vulnerable to security attacks such as Return-Oriented Programming (ROP) [9]. We treat their access as implicit `load/store` instructions and perform DFI check accordingly. When a parent function `parent_func()` calls a child function `child_func()`, the return address is stored in the stack by an instruction `parent_inst`. When function `child_func()` returns, the return address is loaded by another instruction

child_inst. DFI is to ensure that the return address by child_inst should have the latest store only by parent_inst. However, function return is not covered by some static analysis tools like SVF [102]. Hence, we develop a dedicated instrumentation technique different from that for ordinary load/store instructions. Similar idea was also proposed in the work of [16], but the DFI of the return addresses is directly verified by the instrumentation in [16]. By contrast, because our approach verifies it on the PIM processor, the performance overhead is reduced.

The instrumentation for function return is illustrated in Figure 3.20. At the beginning of a function (line 2), the pointer to return address p_ret_addr is obtained. For a C/C++ program, this can be realized by calling the builtin function __builtin_frame_address(0) and adding 4 to the returned result.We designate the identifier of the implicit store instruction parent_inst as the maximum identifier from the static analysis plus the thread ID (lines 3 and 6). This can ensure that the identifier of parent_inst is unique. Bit 21 of the data in the DFI store in line 3 is set to be 1, to inform the info-collector that this is a special instrumentation for function return. Then, the info-collector would expect a subsequent DFI store for the pointer to the return address. The info-collector combines instruction type (implicit load/store), identifier and the pointer to form a DFI packet to be sent to the PIM processor. At the end of the child function (lines 6 and 7), similar instrumentation instructions are added for the implicit load. For each load whose identifier is larger than the maximum identifier of static analysis, DFI enforces the identifier of the latest store to be the same as the identifier of this load.

3.3.4 Hardware Design

3.3.4.1 DFI Packet Generation

Info-collector is the key hardware component to be added at the main processor. It detects DFI store instructions, collects runtime information of a target instruction, generates DFI packets and sends the packets to the PIM processor. Its basic operations are depicted in the flow chart of Figure 3.21.

The info-collector acts only when there is a store instruction being executed. In step B of

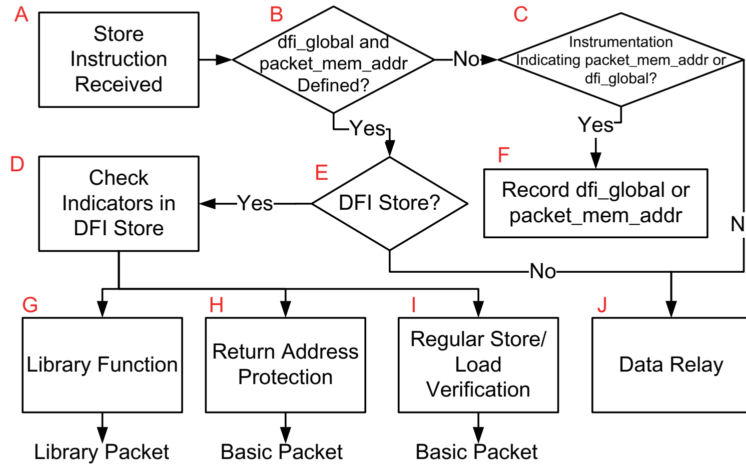

Figure 3.21: Operations of info-collector.

Figure 3.21, it checks if `dfi_global` and `packet_mem_addr` have already been defined. If not, it proceeds to step C to capture `dfi_global` or `packet_mem_addr`. Please note "`store dfi_dummy dfi_glabal`" and "`store packet_dummy packet_mem_addr`" are instrumented at the beginning of a program. Moreover, both `dfi_dummy` and `packet_dummy` have signature values that can be recognized by the info-collector. If they have already been defined, the info-collector further checks if the `store` is a DFI `store` from instrumentation. This is by examining if the target address is the same as that of `dfi_global`.

If this `store` is a DFI `store`, the info-collector parses the indicators in the data part of the DFI `store` and tells if this is to verify `load/store`, function return or a library function call. If this instrumentation is for a `load/store` instruction, info-collector collects instruction type and identifier from this DFI `store` instruction, and the target address from the previous instruction. These information forms a **basic packet** `data'` to be sent to PIM.

If this DFI `store` is for a return address protection (step H in Figure 3.21), the info-collector takes the identifier and instruction type from this DFI `store`, and extracts the pointer to the return address from the next DFI `store`. These information also forms a **basic packet** to be sent to PIM. If this DFI `store` is for a library function (step G), the indicators of this `store` tell if the library function is to load data, store data or not, and if the data-length needs to be encoded by 64 bits or

not. Next, the info-collector continues to collect additional information from the subsequent DFI `store` instructions and generates a **library packet** to be sent to PIM.

If the `store` instruction is a part of the original program (step J), its `data` is relayed to memory without any change and its target address is stored in a local register for future use. The info-collector with the aforementioned operations can be straightforwardly implemented with a combinational circuit through synthesizing Verilog description.

3.3.4.2 Packet Transfer to PIM

A memory space is allocated to store DFI packets sent from the main processor. Although it is physically a memory design, we use it as a FIFO as the packets are processed at the PIM processor in a first-come-first-serve manner. This is not a usual use of memory and memory design is different from conventional FIFO queue. We develop techniques to solve this mismatch.

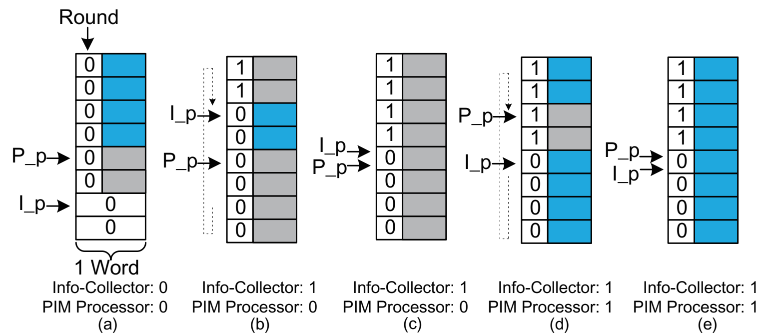

Figure 3.22: Operations of FIFO memory.

The info-collector specifies the starting memory address for a DFI packet as `packet_mem_addr`, and maintains an offset to `packet_mem_addr` as the tail pointer for the FIFO. The PIM processor reads the FIFO and maintains the head pointer. A key difference from the conventional FIFO is that the head and tail pointers here are at two different locations - one at the memory and the other at the main processor.

The operations of the FIFO memory is illustrated by an 8-word example in Figure 3.22, where

a white rectangle indicates an empty slot, a grey rectangle means a word newly written by the info-collector but has not be read by the PIM processor and a blue rectangle shows a word that has been read by the PIM processor while the slot has not been written by a new word yet. There are two pointers: `I_p` by the info-collector or tail pointer, and `P_p` by the PIM processor or head pointer. Although the two pointers are drawn in the same figure, they are physically at different places. Each of the info-collector and the PIM processor maintains a *round* bit, which is initialized to 0 and flips when the write/read wraps around. Detecting if the FIFO memory is empty or full depends on comparing the round bits of the info-collector and the PIM processor. However, they are different locations and the comparison would require additional communication from the info-collector to PIM. To avoid the performance penalty due to the communication, we reserve 1 bit in each word as a copy of the round bit at the info-collector. The round bits of the words are in the left columns in Figure 3.22.

Figure 3.22(a) shows a state in the first round, where all round bits are 0. When the write wraps around as in (b), the round bit of the info-collector and the words newly written after wrap around flip to 1. In (c), when the round bit at `P_p - 1` is different from that of the PIM processor, FIFO full is asserted. When the read wraps around in (d), the round bit of the PIM processor flips to 1. In (e), when the round bit at `P_p` is different from that of the PIM processor, FIFO empty is detected. As such, FIFO empty/full can be detected without communication from the info-collector to the PIM processor.

3.3.4.3 Lossless Data Compression

A main reason for performance overhead is the data transfer (DFI packets) to the memory. We propose to compress target addresses and identifiers by exploiting locality. The compression is realized in the info-collector hardware.

Consider the two C program examples in Figure 3.23. For example A, assume the starting memory address of `aa` is 0x8000, then the program stores the data at 0x8000, 0x8004, 0x8008, and so on. Starting from `i=1`, each target address increases by 4 compared to the previous one. Therefore, we only needs to send the increment in 4 bits, which include 1 sign bit, instead of an

entire address of 32 bits. Example B in Figure 3.23 is similar, but has an address pattern of 0x8000, 0x8400, 0x8800, etc. Although the address increment here 0x400 is relatively large and needs 11 bits to represent, the lower bits of the increment are all 0s. Therefore, we use a format similar to floating point number representation to further reduce the bitwidth of the address increment. This format consists of a sign bit, significand and exponent of 16. To represent 0x400, the sign bit is 0, there are 3 bits for significand to represent 4 and the exponent is 2. Overall, the bitwidth is 6, which is shorter than the 11-bit binary encoding.

```
1  ========Example A==========
2  int aa[1024];
3  for(int i=0;i<1024;i++)
4    aa[i]=i;
5  ========Example B==========
6  int bb[1024][1024];
7  for(int i=0;i<1024;i++)
8    for(int j=0;j<1024;j++)
9      bb[j][i]=i+j;
10 ===========================
```

Figure 3.23: Examples of address locality.

The increment of a target address is represented by an 8-bit floating number, with 1 sign bit, 4 bits of significand and 3 bits of exponents (the power of 16). This representation can cover the range from -15×2^{28} to 15×2^{28}. The info-collector calculates the difference between two target addresses. If the difference is within this range and the significand is within -15 to 15, then the difference is represented by the 8-bit floating number and sent to PIM. Identifiers can also be compressed based on their value locality. However, they rarely have the patterns like example B, where the increment is at the middle bits of an address. Thus, the difference between two identifiers is represented by a binary number. Overall, a DFI packet can be compressed to 15 bits. Therefore, we can pack two **compressed packet**s into a single word. As each word sent to the PIM processor

needs to have 1 bit tag for detecting PIM memory full at the PIM processor (Section 3.3.4.1), 15-bit is the largest bitwidth for accommodating two compressed packets in one word.

3.3.4.4 Runtime Optimization

We develop packet pruning techniques and a technique for increasing the opportunity of locality for data compression. These optimization techniques help reduce the amount of data to be sent to the PIM processor and thereby decrease performance overhead. Some pruning techniques described here are similar to those in [16]. However, the pruning techniques in [16] are offline while our hardware approach allows pruning at runtime. As more information, such as target address, is available at runtime, the opportunity of pruning is increased.

Similar to data transfer between memory and cache in cache lines, we pack multiple DFI packets into a block of hundreds of bytes before sending them to the PIM processor. The packets of blocks are organized in a **transmission buffer**, which is implemented as a register file. The proposed optimizations are performed for the packets in the transmission buffer before they are sent out. Please note that waiting other packets to form a block only increases the DFI verification latency while does not increase the performance overhead.

Consider two pairs of basic packets in the transmission buffer, (P_1, P_2) and (Q_1, Q_2). Each basic packet is for instruction `load`, `store`, or function return, which is implicit `load/store`, but not for library functions. Packet P_1 (Q_1) precedes P_2 (Q_2). The packets of each pair share the same target address and there is no other DFI packet recording `store` of the same target address between them. There are five optimization techniques described using the packet pairs:

A: If P_1 and P_2 are for `store` instruction, and there is no other DFI packet for a `load` with the same target address between them, then packet P_1 is redundant and can be pruned out without being sent to PIM.

B: If P_1 and P_2 are both for `store` instruction, and their identifiers are also the same, then P_2 is redundant and can be pruned out.

C: If P_1 and P_2 are both for `load` instruction, and their identifiers are also the same, then P_2 is

redundant and can be pruned out.

D: If P_1/P_2 records `store`/`load` for the same target address. After P_1 and P_2, if packets Q_1 and Q_2 record `store` and `load` for another same target address, and Q_1/Q_2 have the same identifiers as P_1/P_2, respectively, then Q_1 and Q_2 are redundant. This is to make sure that the same `store/load` pair appears only once in the transmission buffer.

E: All basic packets in the transmission buffer are sorted according their target addresses. If two packets have the same target address, their relative order keeps unchanged. If there is a library packet, the basic packets before and after this library packet are sorted separately. After the sorting, the target address difference between two adjacent packets is examined to find if data compression can be performed. The sorting helps find opportunities of data compression. On the other hand, `load/store` of different target addresses are not relevant to DFI and hence the sorting does not affect the DFI verification.

Among the above optimization techniques, A, B and C are similar to those in [16] except they are performed at runtime while those in [16] are performed offline. Techniques D and E are newly developed in our work. After the optimizations, a packet is compressed if possible.

3.3.4.5 Implementation of the Optimizations

For the 5 optimizations, C and E are the most effective ones and their circuit implementations are described as follows.

3.3.4.5.1 Circuit Design for Optimization C The schematic of combinational circuit implementation of optimization C is shown in Figure 3.24. Assume there are n basic packets in the transmission buffer, Pi represents the i-th packet, and R_i indicates if the i-th packet is redundant or not. Each square in Figure 3.24 is a Processing Element (PE) that computes if a packet is redundant or not. In each column of Figure 3.24, a packet Pi is compared with all later packets $Pj, j > i$ and attempts to find a redundant Pj to be pruned. If there are multiple packets that are redundant with respect to Pi, only the topmost one (with the smallest $|j - i|$) is asserted for pruning and the others can be pruned later in other columns to the right. The R signals in a row are ORed such

that the packet in a row can potentially be pruned by any proceeding packets organized in columns. For example, $P3$ in row 3 of Figure 3.24 can be potentially pruned by $P0$, $P1$ or $P2$ in the left three columns. Like illustrated in the dotted box, a PE compares two input packets Pa and Pb. A necessary but insufficient condition for asserting $R = TRUE$ is that Pa and Pb are both for `load` with the same target address and identifier. The final result of R also depends on Din, which is a disable signal for the pruning. The value of $R = TRUE$ when $Din == 0$ and the necessary condition holds. There are two scenarios where the disable signal asserts: (1) there is a `store` at the same target address between the two `load` instructions of Pa and Pb, and thus the conditions for optimization C is not completely satisfied; (2) a redundant packet has already been found and no further pruning is needed in a column. For scenario (1), $Dout = 1$ when Pa is for `load` while Pb is for `store`. For scenario (2), $Dout = 1$ if $R = TRUE$ for the same PE.

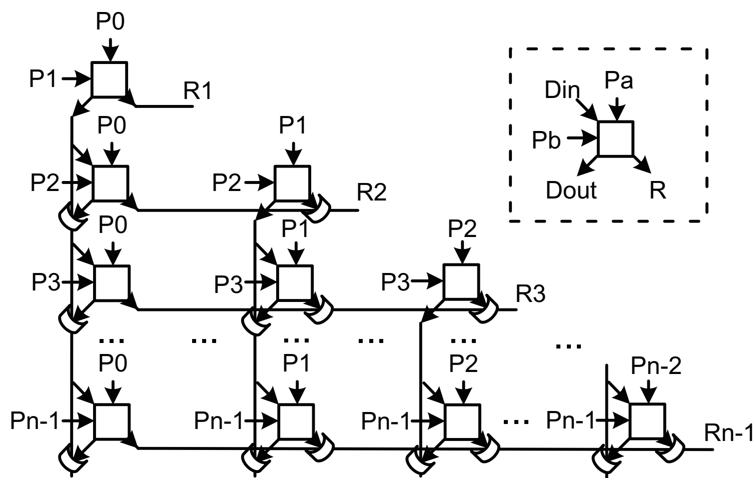

Figure 3.24: Circuit for implementing optimization C.

The combinational circuit requires $\frac{n \cdot (n-1)}{2}$ PEs. To reduce circuit area, it can be transformed into a sequential circuit with $m \cdot (n-1)$ PEs, which identifies all the redundant packets in $\lceil \frac{n-1}{m} \rceil$ clock cycles. The value of m determines the tradeoff between circuit area and the number of clock cycles. The sequential circuit requires additional flip-flops.

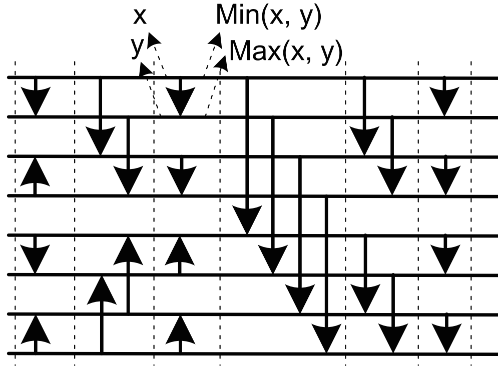

Figure 3.25: Bitonic network for sorting 8 packets.

3.3.4.5.2 Circuit Design for Optimization E Optimization E is for sorting the packets in the transmission buffer according to their target addresses. For combinational circuit implementation, we use Bitonic sorting network [103], which sorts n elements in $O(log^2(n))$ time. Figure 3.25 illustrates one such network for sorting 8 elements. Each component in the network, which is shown as an arrow, takes two inputs along the two horizontal lines from left and then moves the greater element along the arrow direction to obtain the two sorted outputs along the lines at the right. The circuit area can be reduced by transforming into a sequential circuit. In Figure 3.25, the circuit can be partitioned into 6 stages separated by dashed lines and the sequential circuit has 4 comparators to compute each stage in one clock cycle. In general, $\frac{n}{2}$ comparators are needed to compute one stage for sorting n packets.

3.3.5 DFI Verification Program

The DFI verification program is written in C language, and its executable code is loaded onto the PIM processor at the beginning of user program execution.

At PIM, RDT consumes a large memory space, which is allocated by the instrumentation code at the beginning of user program execution. Same as in [16], all program data are organized in words, each of which requires one RDT entry to record the identifier of the latest instruction writing into this word. If the data memory for user program has N bytes, there are $N/4$ entries in the RDT. Since each identifier has 16 bits = 2 bytes, the RDT uses $\frac{N\times2}{4} = N/2$ bytes of memory.

The verification program at the PIM processor continuously reads DFI packets from the FIFO memory, and either performs DFI verification for a packet or updates RDT according to a packet. There are three kinds of DFI packets to be processed by the verification program.

- **Packet for `store` or `load` without compression (basic packet):** The verification program extracts instruction type, identifier α and target address β from the packet. If the instruction type is `store`, identifier α is stored at entry $\beta >> 2$ of RDT. The right shift is performed because RDT is organized in words. If the instruction type is `load`, the verification program reads identifier γ from entry $\beta >> 2$ of RDT, and loads the RDS of identifier α. Then, the program checks if γ is in the RDS of α or not. If not, a DFI violation is reported. Finally, identifier α and target address β are saved in registers for future decompression of compressed packets.
- **Packet for `store` or `load` with compression (compressed packet):** The process is similar to handling basic packets except that decompression is performed. This is to add the increment of target address (or identifier) to the most recently saved target address (or identifier).
- **Packet for library function (library packet):** The verification program extracts target address α if there is `load` in the library function call, and target address β if there is `store`. Then, data-length γ (in words) of the `load` and/or `store` and identifier δ of this function are also extracted. If there is address α, the verification program loads the identifiers ϵ_0, ϵ_1... $\epsilon_{\gamma-1}$ from entries $\alpha >> 2$, $(\alpha >> 2) + 1$, ... $(\alpha >> 2) + \gamma - 1$ in the RDT, and checks if every ϵ_i is in the RDS of δ. If there is address β, the program stores identifier δ to all the entries from $\beta >> 2$ to $(\beta >> 2) + \gamma - 1$ in the RDT.

3.3.6 Discussion

3.3.6.1 Static Analysis

Through static analysis, RDSs of a software program are obtained. An RDS consists of all *possible* reaching definitions for an instruction. However, the *possibility* in a program is sometimes not very clear and it is difficult for a static analysis to precisely find an RDS. If a reaching definition is likely to be possible yet actually never occurs in program execution, it may be includ-

ed in the RDS by an inaccurate analysis. Consequently, this inaccuracy may be exploited by an attack to bypass DFI enforcement. Therefore, the DFI security highly depends on the quality of static analysis. In [16], two kinds of static analysis are discussed: intra-procedural analysis and inter-procedural analysis. Intra-procedural analysis is flow-sensitive, but difficult to scale for large programs. Inter-procedural analysis based on Andersen's points-to analysis [104] is more practical for handling large programs. In our experiments, inter-procedural analysis [102] is performed for both the software DFI method [16] and our approach for a fair comparison, although other kinds of static analysis can be applied with our work as well.

3.3.6.2 The Role and Effect of Cache

Instructions in a user program sometimes accesses data in cache instead of directly dealing with memory. However, such instructions and data are still verified by our DFI as they are always instrumented by DFI `store` instructions. On the other hand, cache may affect the latency (but not performance overhead) of our DFI verification. If the cache writing policy is *write-back*, DFI packets may stay at cache for certain amount of time before it is sent to the FIFO memory. For *write-through* policy, DFI packets can be directly sent to memory without waiting in cache.

One may wonder that cache can reduce data transfer from memory for software DFI [16] and thus our DFI PIM may not be very necessary. In reality, the RDT size is huge, about 50% of all data size of a program, and much larger than ordinary cache size. Therefore, the help from cache on reducing data transfer is limited. In fact, the software DFI [16] is performed on a processor where the benefit of cache has already been enjoyed. In our approach, RDT also resides in memory but is used by PIM processor. Thus, it does not need to be transferred between memory and main processor at all. Compared to software DFI [16], we need to send runtime information to memory. However, the amount of runtime information is significantly less than sending RDT entries repeatedly from memory to processor in [16].

3.3.6.3 Multithreading

```
1  =====thread 1======
2  lock
3  store x1 addr1       // (12)
4  store (0<<16)+12 dfi_global
5  load x2 addr1        // (25)
6  store (1<<16)+25 dfi_global
7  unlock
8  ==================
9  =====thread 2======
10 lock
11 store x1 addr1       // (36)
12 store (0<<16)+36 dfi_global
13 unlock
14 ...
15 ==================
```

Figure 3.26: An example of multithreaded program with locks.

The proposed DFI enforcement approach works for multithreaded programs under the condition that the program is completely data race free and race condition free. Further, during code instrumentation, we need to ensure that a target instruction to be verified and its DFI `store` instructions from instrumentation need to be locked together to form an atomic operation. This is illustrated in Figure 3.26.

3.4 Experiments and Results

3.4.1 Experiment Setup

For CFI verification, all our experiments were run and measured on an Altera DE1-SoC board, containing a Cyclone V FPGA working at 50MHz and an ARM Cortex-A9 dual core processor working at 1GHz on which we loaded a Linux kernel. In addition, we use Quartus Prime 17.1 [105] for Verilog compilation and FPGA layout synthesis, and Signal Tap Logic Analyzer for FPGA signal monitoring. The Verilog compilation is done on a desktop with an Intel 3.8Ghz CPU and 16GB RAM.

For DFI verification, as PIM is used and there is no commercially available CPU processors using PIM yet, the proposed techniques are evaluated by architecture simulations through SMC-sim [100, 106], which is an extension to the gem5 simulator [107] for accommodating PIM. As PIM simulation is a relatively young technology, it is supported for only a couple of host processor platforms. The simulated system in our experiment is an ARM SoC, with an ARM Cortex-A15 processor working under 2GHz frequency and 512 MB memory for user programs. The PIM processor operates with the typical 1GHz frequency [93, 94] and 64MB for RDT. Although the RDT memory size is less than the ideal 256MB, it is sufficient for application testcases in the experiment.

3.4.2 Experiments and Results of Hardware-assisted CFI Verification

3.4.2.1 CFI without Code Instrumentation

All the experiments are performed with no modifications of the target program, and with no code instrumentation. In the proposed hardware-based CFI approach, the information required for CFI violation identification is gathered from the output of TPIU and computed by using CFG. The results show that even without any code instrumentation, it is feasible to realize fine-grained and stateful hardware-based CFI with negligible performance overhead. Avoiding code instrumentation, not only improves performance but more importantly, enables a practical solution eliminating implementation difficulties and the related potential security issues.

3.4.2.2 Security

We use RIPE [108, 109] to evaluate the effectiveness of FastCFI. RIPE is a popular benchmark that has been used frequently in previous works [17, 19] for evaluating control-flow defenses. However, RIPE is designed for Intel processors, and does not directly run on our ARM platform. There are numerous processor architecture specific assembly and shell codes in RIPE, which we had to modify for the ARM processor.

Table 3.1: Security performance for different attack methods.

No.	Overflow Technique	Attack Code	Target Code Pointer	Location	Identify?
1	direct	createfile	ret	stack	√
2	direct	createfile	funcptrstackvar	stack	√
3	direct	createfile	structfuncptrstack	stack	√
4	direct	createfile	funcptrheap	heap	√
5	direct	createfile	structfuncptrheap	heap	√
6	direct	createfile	structfuncptrbss	bss	√
7	direct	createfile	funcptrdata	data	√
8	direct	createfile	structfuncptrdata	data	√
9	indirect	createfile	ret	stack	√
10	indirect	createfile	funcptrstackvar	stack	√
11	indirect	createfile	funcptrstackparam	stack	√
12	indirect	createfile	funcptrheap	stack	√
13	indirect	createfile	funcptrbss	stack	√
14	indirect	createfile	funcptrdata	stack	√
15	indirect	createfile	ret	heap	√
16	indirect	createfile	funcptrstackvar	heap	√
17	indirect	createfile	funcptrstackparam	heap	√
18	indirect	createfile	funcptrheap	heap	√
19	indirect	createfile	funcptrbss	heap	√
20	indirect	createfile	funcptrdata	heap	√
21	indirect	createfile	ret	bss	√
22	indirect	createfile	funcptrstackvar	bss	√
23	indirect	createfile	funcptrstackparam	bss	√
24	indirect	createfile	funcptrheap	bss	√
25	indirect	createfile	funcptrbss	bss	√
26	indirect	createfile	funcptrdata	bss	√
27	indirect	createfile	ret	data	√
28	indirect	createfile	funcptrstackvar	data	√
29	indirect	createfile	funcptrstackparam	data	√
30	indirect	createfile	funcptrheap	data	√
31	indirect	createfile	funcptrbss	data	√
32	indirect	createfile	funcptrdata	data	√
33	direct	returnintolibc	ret	stack	√
34	direct	returnintolibc	funcptrstackvar	stack	√
35	direct	returnintolibc	structfuncptrstack	stack	√
36	direct	returnintolibc	funcptrheap	heap	√
37	direct	returnintolibc	structfuncptrheap	heap	√
38	direct	returnintolibc	structfuncptrbss	bss	√
39	direct	returnintolibc	funcptrdata	data	√
40	direct	returnintolibc	structfuncptrdata	data	√
41	direct	rop	ret	stack	√
42	-	-	-	-	No False Alarm
SP1	-	-	-	-	√
SP2	-	-	-	-	√

Due to the engineering difficulties, it is hard to port all RIPE functions to ARM. Most attacks in RIPE are based on buffer overflow related to 10 methods: *memcpy()*, *strcpy()*, *strncpy()*, *sprintf()*, *snprintf()*, *strcat()*, *strncat()*, *sscanf()*, *fscanf()* and *homebrew()*. All these methods can be performed on Intel processors, but only *memcpy()* and *homebrew()* work for our ARM-based Linux system. However, the way how a buffer is overflowed does not affect how an attack is performed. The difference is which API is used to copy data to the buffer. Therefore, we focused on only

the attacks related to *memcpy()* and *homebrew()*. In total, we recovered 41 attacks (which can run successfully on ARM), including both Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) attacks, as shown in Table 3.1 (Row #1-41). To assess the precision (i.e., no false positive), we also added a new function (Row #42) in RIPE and let it run without attack.

The results in Table 3.1 show that all these attacks can be identified by FastCFI. In addition, FastCFI does not report any false positive (for the newly introduced function with no attack). The results are the same with and without CFG compression.

```
int func1(char *payload)        void vuln(char *payload){
{                                  char buf[256];
   vuln(payload);                  ...
   return 0;                       long new_addr=0x000084af;
}                                  memcpy(payload+length,
                                      &new_addr,sizeof(long));
int func2(char *payload)           memcpy(buf,payload,
{                                     length+sizeof(long));
   vuln(payload);               }
   return 0;
}
                    (a)
   00008488 <func1>:
     ...
     8492:    f000 f817   bl     84c4 <vuln>
     8496:    2300        movs r3, #0
     ...
   000084a0 <func2>:
     ...
     84aa:    f000 f80b   bl     84c4 <vuln>
     84ae:    f248 50e4   movwr0, #34276
     ...
                    (b)
```

Figure 3.27: Code illustrating the stateful SP1 attack.

```
typedef struct {
   char buffer[32];
   void (*func)();                int main(int argc, char* argv[]){
} vuln_struct;                       ...
                                     vuln_struct struct_attack;
void func_wrong(){                   struct_attack.func =
   ...                                  func_correct;
}                                    memcpy(struct_attack.buffer,
                                        data, 64);
void func_correct(){                 struct_attack.func();
   ...                               return 0;
}                                 }
...                                  (a)

...
8490:  f248 4371   movw   r3, #33905  ; 0x8471
8494:  f2c0 0300   movt   r3, #0
8498:  613b        str    r3, [r7, #16]
...
84be:  693b        ldr    r3, [r7, #16]
84c0:  4798        blx    r3
...
                                     (b)
```

Figure 3.28: Code illustrating the fine-grained SP2 attack.

3.4.2.2.1 Fine-grained, stateful attacks. We also designed two special attacks not included in RIPE, as shown in the last two rows of Table 3.1.

SP1 is a stateful attack that cannot be detected by stateless CFI techniques. As shown in Fig. 3.27(a), in SP1, there is a function *vuln* that may be called by function *func1* or *func2*. So in the CFG, the node with function return of *vuln* has edges to nodes in both *func1* and *func2*. However, only one of them is valid each time *vuln* is called. If *func1* calls *vuln*, then *vuln* can only

return to *func1*. In our test, we use buffer overflow to change the return address of *vuln* to *func2*, even if it is called by *func1*. Our experiment shows that FastCFI can easily identify this attack with and without CFG compression. However, stateless CFI such as [20, 61] and the coarse-grained approach in [17], would not be able to identify this attack.

SP2 is a fine-grained attack. In SP2, the attack changes a function call, making it call another unintended function in the program's binary. The C code is shown in Fig. 3.28(a). In *main*, there is a structure *struct_attack*, which contains a buffer and a function pointer. Usually, the function pointer in the memory is right after the buffer. The user data, which can be controlled by the attacker, is copied to the buffer through *memcpy*. An attacker can input the data with a larger size than the buffer, and put the address of the function *func_wrong* right after the 32-byte's data. In this way, when the *struct_attack.func()* is called, function *func_wrong* is executed rather than the correct function *func_correct*.

For our fine-grained CFI, FastCFI can easily identify this attack. Fig. 3.28(b) shows part of the assembly in *main*. The instruction at *84c0* is the function call *struct_attack.func()*. The program would jump to the address stored in *r3*. By backtracking the value in r3, we can find that it should be *0x8471*, where there is the entry of *func_correct*. This is a typical example of indirect branch with constant target address that we discussed before. We create the CFG with a node containing the instruction at *84c0*, and the only outgoing edge of this node is to the node containing the entry of *func_correct*. If the buffer overflow is performed by an attacker, then the control-flow does not go through the correct edge in CFG. This can be detected by FastCFI, both with and without CFG compression.

However, this attack cannot be identified by coarse-grained CFI techniques such as Lee et al. [18]. In [18], it only checks if the indirect branch instruction performed as a function call is at the function's entry. For the example above, the attacked target address of the indirect branch instruction at *84c0* is still the function entry. This would be ignored by [18].

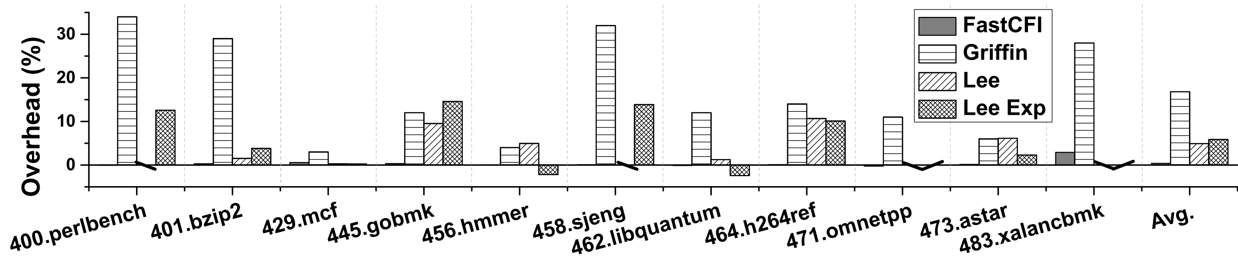

Figure 3.29: The runtime overhead on SPEC 2006 benchmarks.

3.4.2.3 Performance Overhead

We used the SPEC CPU2006 benchmarks [110] to evaluate the runtime overhead of FastCFI. We successfully ran all the benchmarks, except *403.gcc*, which could not be cross compiled by the *arm-linux-gnueabihf-gcc(g++)* compiler.

The results are reported in Fig. 3.29, including a comparison with the results from two recent works: *Griffin* [17] and *Lee* [18]. Both results of *Lee* and *Griffin* are copied from the original papers [17, 18]. For *Lee* [18], some benchmarks are marked with "\", because they were not evaluated in *Lee*'s work. Besides, we also did the code instrumentation and repeated the overhead experiments in [18], the results are shown as *Lee Exp*. The benchmarks not evaluated in *Lee Exp* (marked with "/") are also not evaluated by *Lee*'s original work [18]. Moreover, *400.perlbench* and *458.sjeng* are not evaluated by *Lee* but evaluated by our repeated experiment *Lee Exp*. There are some benchmarks, such as *471.omnetpp*, which have overhead less than 0. This is likely due to cache effects or the noise of the measurement , since the actual overhead is negligible.

Overall, FastCFI has the lowest performance overhead, only 0.36% on average. The reason is that we do not add or modify anything on the software side, and there is no code instrumentation or running of other programs. The only overhead is caused by enabling the PTM device.

3.4.2.4 Latency

We also evaluated the latency introduced by FPGA to detect CFI violations, since it relies on TPIU to communicate the trace between the ARM core and FPGA. The latency is the clock cycles needed by FPGA to identify the attacks after receiving the trace packet containing the CFI

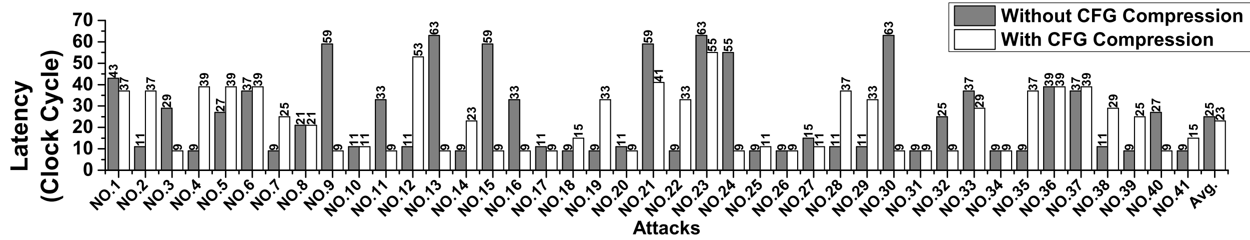

Figure 3.30: The latency for FPGA to identify attacks.

violation information. The results are shown in Fig. 3.30. Overall, FastCFI has a latency within dozens of clock cycles only. We note that some other hardware-based techniques such as [111] incur a latency of tens of thousands of clock cycles, due to a more complex architectural design. Meanwhile, the CFG compression does not have an obvious affect to the latency.

The latency varies between different attacks. This depends on the quantity of data in the FIFO when the wrong control-flow information comes. The data in the FIFO must be processed sequentially by the CFI verification module. The more data, the longer latency. In general, this can be affected by many factors, such as the target program itself, the input, or the other programs running on the same processor.

3.4.2.5 Circuit Resource Use and Compilation Time

Resource use is important for hardware design. Due to the resource limitation of our FPGA, for some benchmarks the system may not fully verify the whole CFG, but a partial CFG, and ignores the instruction flow transitions that happen outside the partial CFG. In our experiments, we always create the complete CFG first, apply CFG compression next, and then select as many CFG nodes or coalesced CFG nodes as our FPGA can contain for the partial CFG. In practice, the partial CFG can be specified by the user or developer, who may choose the most security sensitive parts of the code to protect against CFI attacks.

The resource use results are reported in Table 3.2. The results are separated into two parts in terms of the columns, one without CFG compression and another with CFG compression. The results are also separated into two parts in terms of the rows, one with partial CFG implement-

Table 3.2: Resource use on SPEC 2006 benchmarks.

	Benchmark	Without CFG Compression			With CFG Compression					Total CFG Nodes	False Alarm?
		CFG Nodes	# of ALMs	Compile Time	Compressed CFG Nodes	CFG Nodes	CFG Coverage Increase	# of ALMs	Compile Time		
Partial CFG	400.perlbench	4563	32070	18m55s	4611	5816	27.46%	31559	21m00s	65083	None
	445.gobmk	4585	31604	19m11s	4547	6133	33.76%	31308	20m36s	37019	None
	456.hmmer	4602	31449	19m10s	4597	5269	14.49%	30590	20m23s	12286	None
	458.sjeng	4591	18738	15m08s	4719	5405	17.73%	31048	20m57s	6458	None
	464.h264ref	4513	32070	19m15s	4448	5641	24.99%	30757	19m02s	15195	None
	471.omnetpp	4763	30250	18m07s	4719	4837	1.55%	28507	20m26s	31811	None
	483.xalancbmk	4807	31576	18m39s	4605	5243	9.07%	29608	20m37s	173204	None
	Benchmark	Without CFG Compression			With CFG Compression					Total CFG Nodes	False Alarm?
		CFG Nodes	# of ALMs	Compile Time	Compressed CFG Nodes	CFG Nodes	ALM Use Decrease	# of ALMs	Compile Time		
Full CFG	401.bzip2	2247	22840	15m32s	1796	2247	19.51%	20518	15m57s	2247	None
	429.mcf	471	16171	13m48s	262	471	13.20%	15480	14m32s	471	None
	462.libquantum	1300	18738	15m08s	1223	1300	4.76%	18367	16m33s	1300	None
	473.astar	1345	18995	15m07s	1116	1345	10.21%	18169	16m11s	1345	None

ed and another with full CFG implemented. An example in Figure 3.31 is used to illustrate the metrics used for evaluating the effectiveness of CFG compression. The example CFG and its corresponding compressed CFG are shown in Figure 3.31(a) and (b), respectively. In Table 3.2, for the results without CFG compression, “CFG Nodes" represents the number of CFG nodes in the non-compressed CFG included in the CFG checker, such as the nodes in the set of the red circle in Figure 3.31(a), which is $\{A1, A2, B1, B2, E\}$. The set of these CFG nodes is named S. For the results with CFG compression, “Compressed CFG Nodes" represents the number of CFG nodes in the compressed CFG included in the CFG checker, such as the nodes in the set of the green circle in Figure 3.31(b), which is $\{A, B, C, D, E\}$. The set of these CFG nodes is named S'. For the results with CFG compression, “CFG Nodes" represents the number of the CFG nodes in the non-compressed CFG, which the CFG nodes in the compressed CFG included in the CFG checker correspond to, such as the nodes in the set of the blue circle in Figure 3.31(a), which is $\{A1, A2, B1, B2, C, D, E\}$. The set of these CFG nodes is named S'', where the nodes are the nodes in S' before CFG compression. “CFG Coverage Increase" indicates percentage increase of the number of nodes in S'' comparing to the number of the nodes in S. For the example in Figure 3.31, the CFG coverage increase is $\frac{7-5}{5} = 40\%$. The ALM means adaptive logic module in Altera FPGA, which is the basic element of FPGA and similar to LUT (Lookup Tables). The decoder and CFI verification module (without CFG checker) uses 2755 and 4015 ALMs, and all the

parts (such as the modules communicating with ARM processor and FPGA) excluding CFG checker use 10938 ALMs. “ALM use decrease" indicates the percentage of the number of ALMs needed decreased by applying CFG compression, when the CFG can be fully implemented. For these experiments, we group 100 blocks in one small Verilog module as discussed in Section 3.2.2.5.1. Overall, without CFG compression, our current FPGA can support around 4600 CFG nodes. When

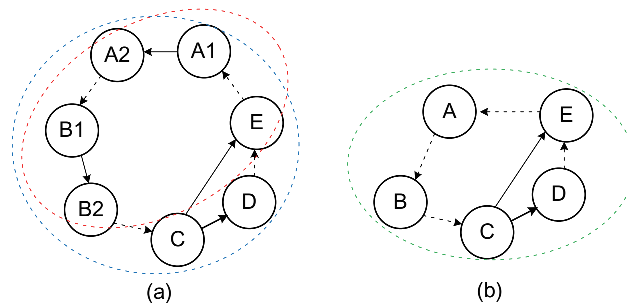

Figure 3.31: An CFG compression example.

the CFG compression is applied, for the benchmarks where partial CFG is implemented, the FPGA can still support around 4600 CFG nodes in the compressed CFG, which corresponds 18.44% more non-compressed CFG nodes (the nodes in S'') comparing to the implementation without CFG compression (the nodes in S). For the small benchmarks where full CFG is implemented, CFG compression can decrease the number of ALM needed, and the number of ALMs needed by the CFG checker is decreased 11.93% on average. Both partial and full CFG implementation benefits are from the fact that the compressed CFG has less hardware expense when the compressed CFG is implemented in the CFG checker, because some CFG nodes coalesced into fewer nodes. Note that even though with only the partial CFGs, FastCFI does not report any false alarms on the studied benchmarks. As also reported in Table 3.2, the Verilog compilation time, including FPGA layout synthesis, in our experiments is less than around 20 minutes for each benchmark.

The number of ALMs used by each block in CFG checker is shown in Fig. 3.32. It does not vary much between different benchmarks, except *429.mcf* and *458.sjeng*. The result is the ratio

between the number of ALMs needed for the CFG checker and the number of CFG checker blocks. The number of CFG checker blocks is defined in the following: The CFG checker blocks in the black bar results correspond to the CFG nodes in S. The CFG checker blocks in the gray bar results correspond to the CFG nodes in S'. The CFG checker blocks in the white bar results, of which the legend is marked by “*", correspond to the CFG nodes in S''. The average number of ALMs per block is about 5. The block corresponding to the CFG node in the compressed CFG (grey bars) spends more ALMs than the block corresponding to CFG node without compression (black bars), because the address range of the coalesced node is the union of the address ranges of all the nodes before CFG compression, and this may require more comparisons in *if* statement of the block if the address ranges of the nodes before CFG compression are not continuous. However, by CFG compression, lots of CFG nodes are compressed to coalesced nodes, and the total number of CFG nodes in the compressed CFG is decreased. This overwhelms the disadvantage that the average number of ALMs needed by each block of the compressed CFG is increased, which is proved by the results of the white bars. The results of the white bars indicate that each block corresponding to the node in S'' spends fewer ALMs than the block corresponding to the node in S. Since node in S and node in S'' are both the nodes in the non-compressed CFG, the results indicate that it costs less hardware expense (fewer ALMs) to implement each node in the non-compressed CFG by using CFG compression than not using CFG compression.

We also evaluated the performance improvement (as described in Section 3.2.3.1) by using small Verilog modules of block groups instead of putting Verilog codes of all the blocks in the top CFG checker Verilog module. We use a typical benchmark *483.xalancbmk* to find out the relationship between the number of blocks in one group and the resource use. The experiments are done without CFG compression, and results are shown in Fig. 3.33.

Overall, using more small Verilog modules reduces the time for compilation. When more blocks are grouped into one small Verilog module, it increases the optimization workload of each small module for the compilation tool, and the time complexity of Verilog compilation is also not linear. This results in the large time cost when the number of blocks is about 2000 in one group.

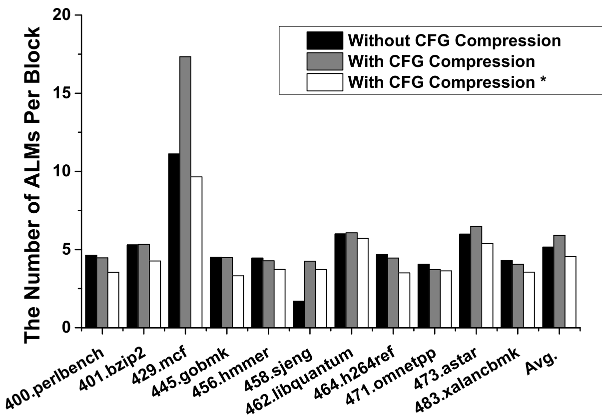

Figure 3.32: The number of ALMs per block of different benchmarks.

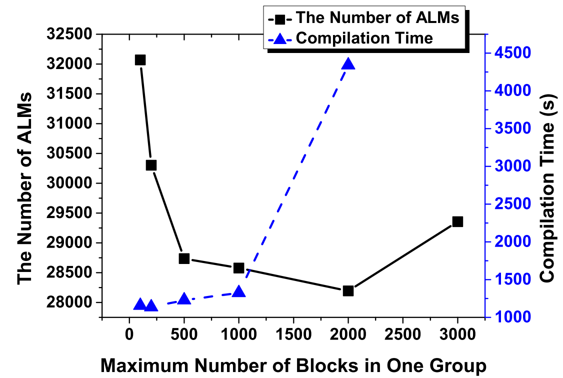

Figure 3.33: The relationship between the number of ALMs, compilation time and maximum number of blocks in one block group of 483.xalancbmk.

For the number of ALMs, the results show an odder behavior. In theory, if we put more blocks in one group, the number of ALMs should decrease because of the better optimization. However, in Fig. 3.33 after the number of blocks in one group is more than 2000, the number of ALMs increases a lot. The reason may be that when there are too many blocks in one small Verilog module, it is hard for the compilation tool to find the optimized solution, even though there may exist a solution that is more optimized. When the number of blocks is not large, the compilation tool is competent to find almost the best solution, which makes the curve in Fig. 3.33 as we expect.

3.4.3 Experiments and Results of Hardware-assisted DFI Verification

3.4.3.1 Security

Our approach realizes the same DFI as defined in [16] and therefore can achieve the same security as [16]. The effectiveness of our approach is further confirmed by a comparison with the method of HDFI [72], control flow attack benchmark RIPE [108, 109], and some non-control attacks.

3.4.3.1.1 Comparison with Hardware-assisted Data-flow Isolation Hardware-assisted Data-Flow Isolation (HDFI) [72] only enforces partial DFI at a very coarse granularity to reduce the overhead. It uses a 1-bit tag to differentiate a sensitive region and a non-sensitive data region, and only en-

Table 3.3: The performance overhead of DFI. †Computation time of optimizations and compression is neglected. ‡Computation time of optimizations and compression is considered.

		Software [16]	Hardware (no cmpr or opt)			Hardware (512B Buffer)			Hardware (1KB Buffer)			Hardware (2KB Buffer)		
Column ID		1	2	3	4	5	6	7	8	9	10	11	12	13
Compression		×	×	√	×	√	√	√	√	√	√	√	√	√
Transmit Buf Size		-	-	-	2KB	512B	512B	512B	1KB	1KB	1KB	2KB	2KB	2KB
Runtime Optimize		×	×	×	All	All	C, E	C, E	All	C, E	C, E	All	C, E	C, E
#Gates of Info-Collector		-	<2908	2908	†	†	†	116,769‡	†	†	252,021‡	†	†	753,666‡
Bench-marks	401	221.88%	72.42%	72.08%	57.25%	46.73%	47.71%	49.50%	45.10%	46.33%	47.79%	44.05%	45.59%	46.80%
	429	102.06%	40.64%	37.31%	34.21%	28.53%	29.53%	30.58%	28.52%	28.82%	29.67%	28.02%	28.43%	29.14%
	433	87.23%	40.95%	37.18%	27.36%	28.70%	29.69%	30.53%	27.12%	28.74%	29.41%	26.36%	27.90%	28.46%
	445	169.95%	78.05%	74.45%	66.24%	62.13%	62.88%	64.32%	61.08%	62.18%	63.35%	60.24%	61.78%	62.76%
	456	228.49%	105.43%	103.43%	69.14%	62.87%	62.90%	65.83%	57.81%	57.91%	60.29%	55.38%	55.50%	57.48%
	458	372.68%	37.42%	35.50%	30.49%	29.53%	29.94%	30.63%	28.52%	29.15%	29.75%	27.84%	28.65%	29.18%
	462	60.36%	37.46%	29.47%	23.39%	22.58%	22.62%	23.49%	22.44%	22.47%	23.20%	22.39%	22.42%	23.05%
	464	218.79%	80.60%	75.39%	61.31%	59.47%	60.68%	62.74%	57.73%	59.36%	61.37%	56.17%	58.53%	60.80%
	473	115.39%	60.50%	57.77%	44.33%	40.44%	41.01%	42.43%	38.85%	39.65%	40.80%	37.70%	38.64%	39.61%
	482	36.86%	24.62%	23.96%	21.51%	21.74%	21.65%	21.92%	21.50%	21.60%	21.82%	20.88%	21.01%	21.19%
	Avg.	161.37%	57.81%	54.65%	43.52%	40.27%	40.86%	42.20%	38.87%	39.62%	40.74%	37.90%	38.85%	39.85%

sures that data in one region are not lastly written by an instruction for the other region. In other words, it reports a violation only when data intends to be in one region but is actually written by an instruction for the other region. Although its overhead is very small, the verification granularity is very coarse and may miss attacks that mingle different data within the same region. As such, it cannot detect attacks that mingles data within the same region. Consider the example in Figure 3.34, where input data are first written into `u0` and `u1` in lines 10 and 11. Later, the data are copied to buffers in lines 13-15. If there is buffer overflow when executing line 10, i.e., the input data size exceeds 256, then offset `u0->off` is modified unintentionally. If this happens, when the program should copy the users' data to their own buffers by lines 13-15, line 13 may copy user0's data to other users' buffers through the modified `u0->off`. Meanwhile, user1 can write to user2's buffer in line 14 in the same way. As HDFI partitions data into only two regions. Then, one of the user pairs - (user0, user1), (user0, user2) or (user1, user2) must share the same region. As such, the former user in a pair can attack the latter in the pair without being detected by HDFI. For the example of Figure 3.34, we tested different tag schemes of HDFI, which are listed in the left three columns of Table 3.4. For each of those tag scheme, there is some overflow that cannot be detected by HDFI as shown in the 4th column, where $u0 \Rightarrow u1$ means some data of user0 is written into user1's data through overflow. By contrast, our approach can successfully detect all these overflows.

```
1  struct vuln{
2    char data[256];
3    int off=0;
4    int size=0;
5  }*u0, *u1, *u2;
6  ===============
7  char user0_buffer[256];
8  char user1_buffer[256];
9  char user2_buffer[256];
10 read_user_input(u0, user0_input);
11 read_user_input(u1, user1_input);
12 ...
13 memcpy(user0_buffer+u0->off, u0->data, u0->size);
14 memcpy(user1_buffer+u1->off, u1->data, u1->size);
15 memcpy(user2_buffer+u2->off, u2->data, u2->size);
```

Figure 3.34: An example of vulnerability that HDFI cannot detect.

Table 3.4: Scenarios for the case of Figure 3.34 where HDFI fails.

HDFI				Our approach detect?
$u0$	$u1$	$u2$	Missed overflow	
Tag 0	Tag 0	Tag 0	$u0 \Rightarrow u1$, $u0 \Rightarrow u2, u1 \Rightarrow u2$	Yes
Tag 0	Tag 0	Tag 1	$u0 \Rightarrow u1$	Yes
Tag 0	Tag 1	Tag 0	$u0 \Rightarrow u2$	Yes
Tag 0	Tag 1	Tag 1	$u1 \Rightarrow u2$	Yes
Tag 1	Tag 0	Tag 0	$u1 \Rightarrow u2$	Yes
Tag 1	Tag 0	Tag 1	$u0 \Rightarrow u2$	Yes
Tag 1	Tag 1	Tag 0	$u0 \Rightarrow u1$	Yes
Tag 1	Tag 1	Tag 1	$u0 \Rightarrow u1$, $u0 \Rightarrow u2, u1 \Rightarrow u2$	Yes

3.4.3.1.2 RIPE Benchmark RIPE [108, 109] is a well-known benchmark containing various control-flow attacks, and all control-flow attacks can also be identified by DFI. RIPE is originally designed for X86 architecture and modification is required for executions on an ARM processor. We totally implemented 156 attacks of the benchmark for our system, including Return-Oriented Programming (ROP) [9] attacks and Jump-Oriented Programming (JOP) [10] attacks. In addition,

we also prepared a RIPE program without any attack. It is observed that our DFI system successfully identifies all the 156 attacks and does not make false alarm for the case without attack.

3.4.3.1.3 Heartbleed Heartbleed (CVE-2014-0160) [11] is a vulnerability in OpenSSL cryptography library. When a message, including the payload and the length of the payload, is sent to a server, the server echoes back the message with the claimed length. However, it is not checked if the actual payload length is the same as the claimed one. As such, an attacker may send a message with the actual payload length smaller than the claimed one. Then, the server sends back not only the original payload but also some additional data to fulfill the claimed length. It is likely the additional data is some private sensitive data on the server. Consequently, sensitive data is stolen by the attacker. This is a non-control data attack and thus cannot be detected by Control-Flow Integrity (CFI). We make use of the source code in [112] to simulate such attack. This attack is successfully detected by our DFI system as the data to be loaded for sending back cannot be most recently written by an instruction not from the sender. An attack-free transaction, where the actual payload length conforms to the claimed one, is also tested and no false alarm is made by our DFI system.

3.4.3.1.4 Nullhttpd Nullhttpd is a HTTP server that has heap overflow vulnerability (CVE-2002-1496) [12]. If the server receives a POST request with negative content length L, it should not process the request. However, the server continues to process and allocates a buffer of $L+1024$ bytes, which is less than 1024 bytes. Later, the server writes data of 1024 bytes into the buffer, and therefore buffer overflow occurs. The experiment shows that our DFI system successfully detects such buffer overflow. When some `load` instruction attempts to access the data written by overflow, it is found that the data is not written by any instructions in the RDS of the `load` instruction. An experiment is also conducted to confirm that our DFI system does not produce false alarm in this context.

3.4.3.2 Performance and Circuit Overhead

The performance overhead of our approach as well as the original software DFI [16] are evaluated through simulations on the SPEC CPU 2006 benchmark [110]. As architecture simulation is orders of magnitude slower than running on an actual system, only 4 billion clock cycles are simulated for each application of the benchmark. Nevertheless, 4 billion clock cycle time is long enough to pass the warm up phase and get deep into the region of interest of an application. To ensure a fair comparison, each benchmark application was simulated to the same end point for our approach, the previous work [16] and original benchmark without DFI enforcement. The experiments for the previous work [16] and original benchmark without DFI enforcement use exactly the same system as the experiments for our approach. The results are summarized in Table 3.3. As the static analysis tool failed in some applications, results only from those with successful static analysis are shown in this table.

On average, the overhead of software DFI [16] is 161% while our PIM approach without any compression or optimization can reduce the overhead to about 58%. The compression alone can further reduce another 3% and the runtime optimization alone can trim the overhead by another 14%. More overhead reduction is achieved when the compression and runtime optimizations are jointly applied as optimization E allows more opportunities for data compression. This is confirmed by the results in columns 5-13. The three groups on the right are from results of different transmission buffer sizes. One can see that the overhead decreases with buffer size increase. In each group, “All" means all of the 5 optimization techniques are applied and “C, E" corresponds to the results where only the two most effective optimizations are employed. The best result is from column 11, where all optimizations are applied, and the average overhead is 37.9%. The computation time of compression and optimization contributes to about 2% overhead. Note that even when using 2KB transmission buffer, the average latency introduced by the transmission buffer of all the testcases is 19264 clock cycles of the main processor, which is reasonably small. When using 2KB transmission buffer, the performance overhead is largely reduced to less than 40%. Although there are still needs for further reducing the performance overhead, our work has already made a huge

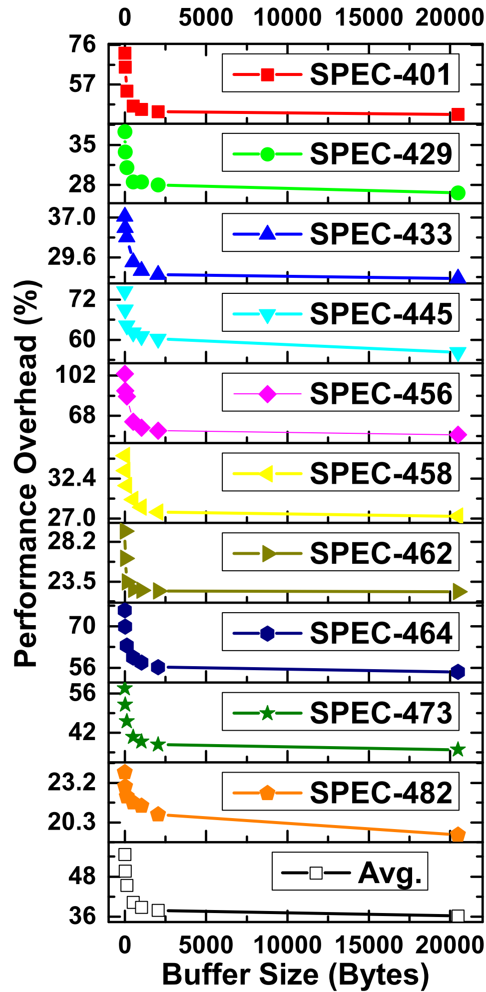

Figure 3.35: Tradeoff between performance overhead and transmission buffer size.

leap on DFI, which is the next frontier in security field.

The info-collector circuit is implemented by synthesizing Verilog using Synopsys Design Compiler and ASAP 7nm cell library [113]. The info-collector with basic operation and compression costs only 2908 gates and less than 30ps circuit delay. Hence, its area and delay are negligible. We also implemented the circuit for optimization C/E. The results with these implementations are in columns 7, 10 and 13 of Table 3.3, where the gate counts of the info-collector with optimization are listed. In our experience, the circuit overhead is mainly due to the optimization part. The gate count of 754K is not trivial, but still a tiny fraction of a modern microprocessor that often has hundreds of millions of gates.

The effect of transmission buffer size on performance overhead is further investigated and the results are plotted in Figure 3.35. It shows that an increase of buffer size from 0 quickly brings down the overhead. However, the overhead reduction soon diminishes as buffer size reaches 2K bytes and this is why we limit our buffer size to be no more than 2K in the main experiments.

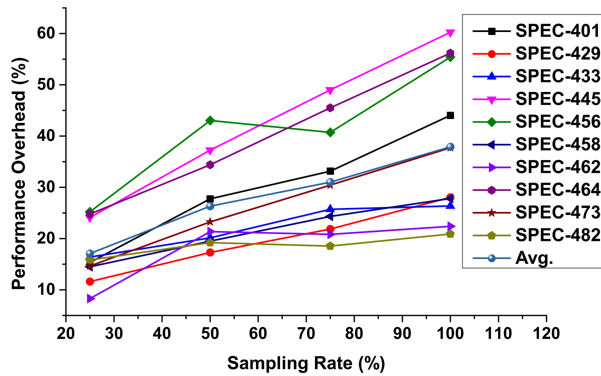

Figure 3.36: Impact of sampling rate on performance overhead.

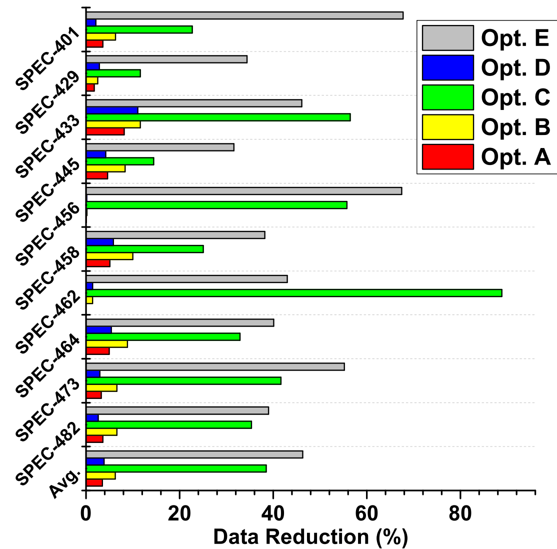

Figure 3.37: The effectiveness of different optimization techniques.

We also evaluated a random sampling technique based on our approach. That is, a certain percent of randomly selected `load/store` instructions of a program are verified for DFI with the setting same as that for column 11 of Table 3.3. Please note this is not a complete DFI. The results are plotted in Figure 3.36. In general, a low sampling rate incurs relatively low overhead as expected. However, the overhead change versus sampling rate is not always monotone. For SPEC-456, the overhead decreases when the sampling rate increases from 50% to 75%. This phenomenon indicates that not all `load/store` instructions contribute equally to the overhead. The overhead for verifying an instruction also depends on if it is on the critical path of program execution.

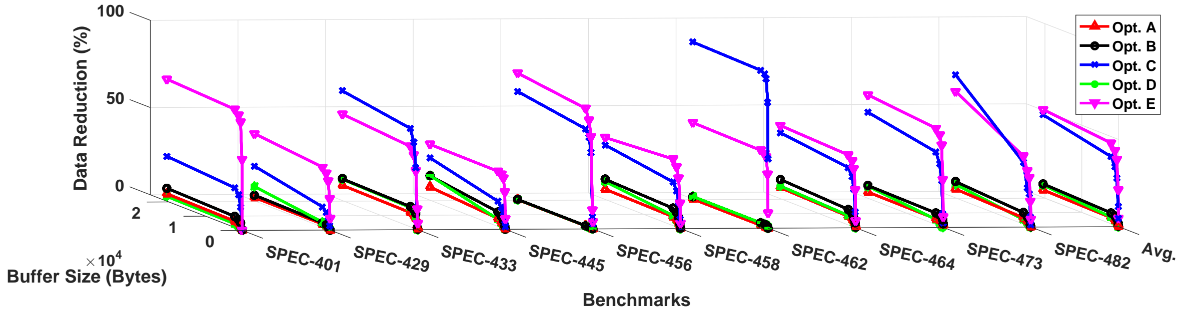

Figure 3.38: The effect of transmission buffer size on data reduction.

3.4.3.3 Analysis of Optimizations

The effects of the 5 optimization techniques described in Section 3.3.4.4 on data reduction are evaluated separately and the results are depicted in Figure 3.37. One can see that optimizations C and E always lead to more data reduction than the other techniques. For SPEC-462, optimization C can reduce data by over 80% while optimization E reduces data by more than 60% for both SPEC-401 and SPEC-456. The effect of transmission buffer size on the data reduction is studied as well and the results are plotted in Figure 3.38. When the buffer size increases at the beginning, more opportunities are found for pruning and compression, and thereby more data reduction is obtained. However, the benefit quickly diminishes when the buffer size exceeds 2K bytes.

3.5 Conclusions and Future Research

We have presented an FPGA-based CFI system named FastCFI. To the best of our knowledge, it is the first to simultaneously achieves low overhead, fine-grained and stateful verification and independence of code instrumentation. It does not produce false alarms and has low detection latency. It successfully detects all CFI violations in major benchmarks and incurs an average overhead of 0.36%. While it offers the computing efficiency of FPGAs, its deployment is nearly as convenient as software due to our automated Verilog generation technique. These advantages make FastCFI be feasible to be applied to the systems having high real-time and security requirements.

Besides, We proposed a new DFI approach based on Processing-in-Memory (PIM), which can largely reduce the data transfer between main processor and memory. Data compression and

runtime optimization techniques are further developed. The PIM-based approach can reduce the overhead by $4\times$ compared to the original software DFI. At the same time, a complete DFI enforcement as defined in the original seminal work is achieved. To the best of our knowledge, this is the only major progress on complete DFI over the past 10 years. As the reduced overhead is still significant, we will study a collaborative software-PIM approach for further overhead reduction in future research.

4. EXPLORING SERVERLESS COMPUTING FOR MACHINE LEARNING MODEL TRAINING[1]

In this chapter, we introduce the way to train large neural networks under serverless environment, with the use of a specific data parallelism scheme. In addition, optimizations are proposed. We also introduce the approach for doing hyperparameter tuning for small neural networks with serverless computing.

4.1 Previous Works

Previous works on serverless runtimes mainly fall into two areas. The first is the area with more benefits on cost or performance over other runtimes like virtual machine or distributed computing. Examples of applications in this area are shown in [25], where tasks are mostly event-driven, stateless and have short run time.

The second is the area which aims at broadening the use of serverless runtimes, whereas most of the works in this area do not focus on machine learning, and there is no work towards deploying neural network training on serverless runtimes as our work. In this area, the works about scientific computing have been demonstrated successfully on serverless platforms. These efforts demonstrate the feasibility and promise of deploying scientific workload on serverless infrastructures. With Pywren [114] for example, one can deploy python-based workloads on multiple AWS Lambda services. The work of [115] proposed a performance evaluation framework with the use of a scientific workflow system: HyperFlow. The results show different behaviors between different serverless providers, such as AWS, Google, IBM and Microsoft. Another work, [116], mainly focuses on the deployment of scientific workflows on serverless environment, and proposes many feasible models for implementation. Those investigations are pioneer efforts on supercomputing with serverless architectures.

[1]©2018 IEEE. Reprinted, with permission, from Lang Feng, Prabhakar Kudva, Dilma Da Silva and Jiang Hu, "Exploring Serverless Computing for Neural Network Training", IEEE International Conference on Cloud Computing, 07/2018.

There are also works in the second area towards machine learning on serverless architectures, but the machine learning in these works has mostly been used for inference [28]. For example, in [117], the latency impact of the use of serverless for deep neural networks is investigated. The experiment in [117] shows the difference of the inference latency between the warm and cold execution, and the latency difference between different memory sizes. However, this work does not focus on deploying neural network for training, while our work proposes an optimized way for taking advantage of parallelism for training deep neural networks.

4.2 Training Large Neural Network Models with Serverless

We define large neural network models as those whose training cannot be completed within one serverless instance either due to constraints in runtime or memory requirements, such as the model used in TensorFlow Tutorials [118]. In this section we review cases that require workflows of several serverless instances with data transfer among them.

4.2.1 Data Transfer and Parallelism with Serverless

A key difference in the application of parallelism is the different nature between serverless instances and normal distributed computing: serverless instances are inherently time-limited and stateless, unlike parallel threads previously studied with deep neural networks. At present, there is no way to transfer data between two serverless instances directly, or to assign serverless instances affinity to compute resources close to the shared data. Therefore, intermediate storage such as databases are used for holding states that are to be shared between subsequent serverless instances. The data transfer between instances is shown in Figure 4.1. Since two serverless instances cannot communicate directly, the parallelized data or models have additional costs, including:

- data transfer latency from source instance to database,
- data transfer latency from database to destination instance,
- warm up latency for loading data.

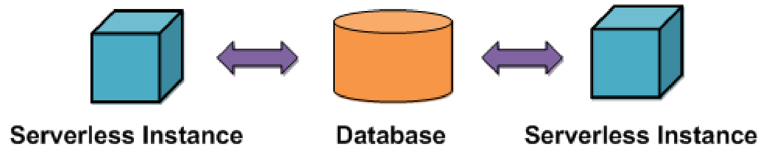

Figure 4.1: Data transfer gateway between two serverless instances

Distributed deep learning platforms have been well studied over the years [33], with more recent platforms such as distributed TensorFlow [119] and MxNet [34], as well as variants of them [120] showing dramatic improvements on performance and scalability. Significant improvements are noted when deep learning models take advantage of parallelism. For our work, we adapt such well known approaches to training neural networks in a distributed fashion and investigate their suitability to serverless. To coordinate the component instances in a training workflow, we use the graph-based notation of step functions [121]. In the rest of this paper, the words *graph* and *structure* will refer to the interconnected structure of serverless instances interleaved by writes and reads to storage.

4.2.2 Data Parallelism for Neural Network Training

Given a dataset, the training of a neural network is to iteratively modify network parameters, such as edge weights and biases, such that the network inference results match the dataset. Many common training algorithms, such as stochastic gradient [122], compute gradients according to the training data and then the gradients are applied to update the parameters. In data parallelism, a given dataset is partitioned into multiple subsets, each of which is applied to train a complete network model on a machine, called *worker*. All workers share the same network model. When a subset of training data is applied to a worker, corresponding gradients are computed there. Then, gradients from all workers need to be collected by a machine, called *parameter server*, where the network parameters are updated. The data parallelism by serverless computing is illustrated in Figure 4.2, where each gray rectangle indicates one serverless instance. In serverless environments, all the machines are serverless instances. The dataset is partitioned into n workers, which compute gradients and send the gradients to the parameter server.

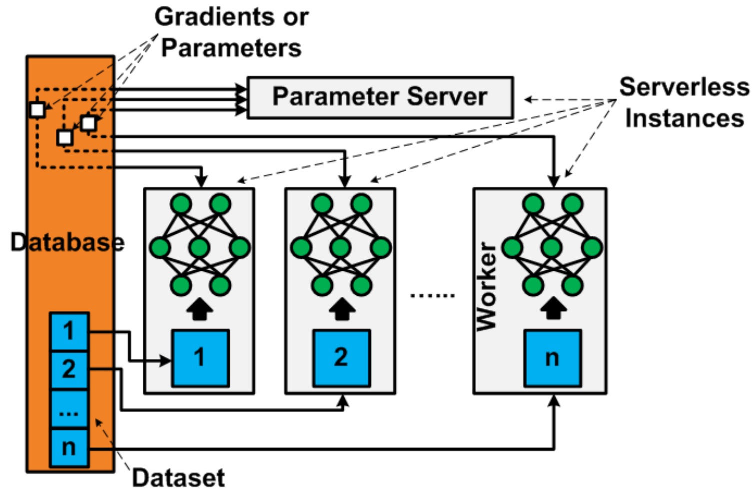

Figure 4.2: Data parallelism by serverless computing

There are two approaches for updating parameters in the data parallelism: Synchronous and asynchronous update.

In a synchronous update, the parameter server waits till gradients from all workers are received and then updates the parameters. In an asynchronous update, the parameter server updates the parameters each time it receives one set of gradients from one worker.

In our work, we focus on the synchronous update.

4.2.3 Optimizing Parallelism Structure for Serverless Training

For data parallelism, data transfers occur between workers and the parameter server. There are two kinds of transfers:

- The parameter server transfers parameters to all workers.
- Workers transfer gradients to the parameter server.

For the transfer from the parameter server to workers, the latency is a constant as the number of parameters is fixed for a given neural network model and all workers can read the same parameters in parallel.

For the other transfer type, if there are n workers, the parameter server needs to receive n sets of gradients, each of which corresponds to one set of parameters on one specific worker. If

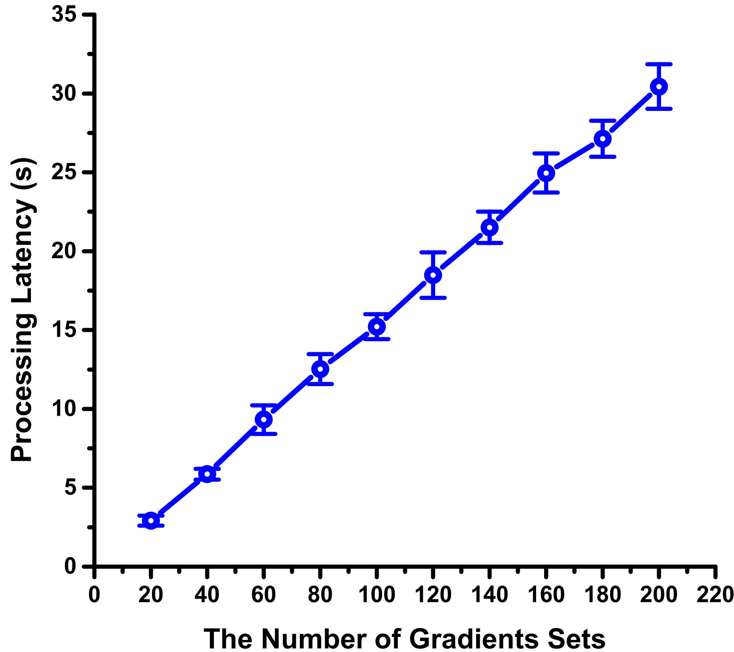

Figure 4.3: The relationship between data transfer latency and the number of gradients sets received by one parameter server. (experiment environment: the transferred data contains 42601 gradients, and parameter server is a 512MB serverless instance)

it takes time t_m for each set of gradients to be transferred to the parameter server, the latency of transferring all gradients in one iteration is $n \cdot t_m$. The linear dependence on the number of gradient sets is confirmed by the measurement results shown in Figure 4.3, where the bars indicate plus/minus standard deviation.

Since the data transfer is the main performance bottleneck for serverless training of neural networks, we propose a multi-layer parameter server structure to reduce the transfer latency. Please note that the focus here is to reduce the latency of transferring gradients, as the latency of transferring parameters is constant. In Figure 4.4, we use blue nodes to represent workers and yellow nodes to indicate parameter servers. Usually, multiple workers send their gradients to the parameter server as in Figure 4.4(a). The parameter server is also called merging node in the graphs in Figure 4.4.

We propose a multi-layer merging structure as in Figure 4.4(b), which contains 6 workers and 3 parameter servers distributed in 2 layers. In such structure, a parameter server or merging node

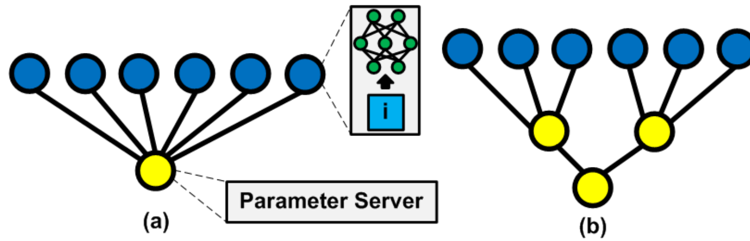

Figure 4.4: Different structures for merging gradients by parameter servers.

can receive gradient data from other merging nodes. In the upper merging layer of Figure 4.4(b), each parameter server merges gradients from 3 workers and the merging takes $3t_m$. Since the two merging nodes work in parallel, the merging latency of this layer is also $3t_m$. In the lower merging layer, there is one parameter server, which merges gradients from the two parameter servers of upper layer and the merging latency is $2t_m$. Therefore, the total gradient data transfer latency in Figure 4.4(b) is $5t_m$. By contrast, the naïve merging structure in Figure 4.4(a) costs $6t_m$ transfer latency.

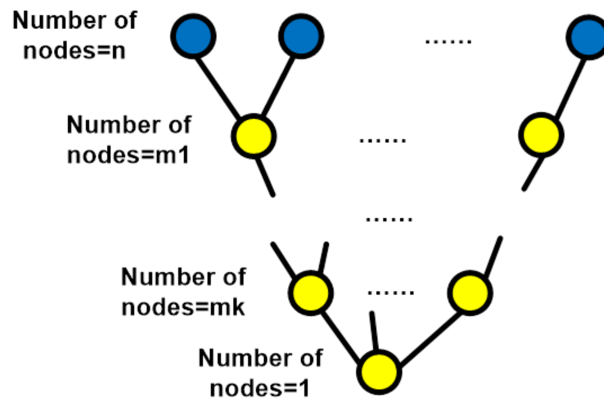

Figure 4.5: General structure of parameter servers

The example of Figure 4.4 indicates that the proposed multi-layer merging structure can reduce merging (gradient data transfer) latency. A general problem is how to decide the number of merging layers and number of parameter severs in each merging layer such that the gradient

data transfer latency is minimized. To solve this problem, we first introduce the latency model for transferring gradients by workers to the parameter servers.

Consider a general data parallel structure as Figure 4.5. Assume there are n workers and k intermediate merging layers, and for the i_{th} merging layer, there are m_i parameter servers. For the bottom merging layer, there is only 1 parameter server to do the final data processing. Then, the total latency can be described as

$$t = t_m \frac{n}{m_1} + t_m \frac{m_1}{m_2} + ... t_m \frac{m_{k-1}}{m_k} + t_m m_k \tag{4.1}$$

To minimize t, we first take partial derivatives with respect to each m_i as below.

$$\begin{cases} \frac{\partial t}{\partial m_1} = -\frac{n}{m_1^2} t_m + \frac{1}{m_2} t_m \\ \frac{\partial t}{\partial m_i} = -\frac{m_{i-1}}{m_i^2} t_m + \frac{1}{m_{i+1}} t_m \quad 1 < i < k \\ \frac{\partial t}{\partial m_k} = -\frac{m_{k-1}}{m_k^2} t_m + t_m \end{cases} \tag{4.2}$$

One can tell that the second order derivatives are all positive, then the function t versus m_i is convex. By letting all first order derivatives be 0, the values of all m_i minimizing t are given by

$$\begin{cases} m_k^{k+1} = n \\ m_i = m_k^{k-i+1} \quad 1 \leq i \leq k-1 \end{cases} \tag{4.3}$$

Thus, the minimum t is found to be

$$t_{min} = t_m (k+1) n^{\frac{1}{k+1}} \tag{4.4}$$

We find the k that minimizes t_{min} by letting $\frac{dt_{min}}{dk} = 0$, which gives $k = ln(n) - 1$. However, a large k means many hops of data transfer, which increase the chance of packet loss and the costly data retransmission. Therefore, we bound the value of k to be no greater than 2 in practice.

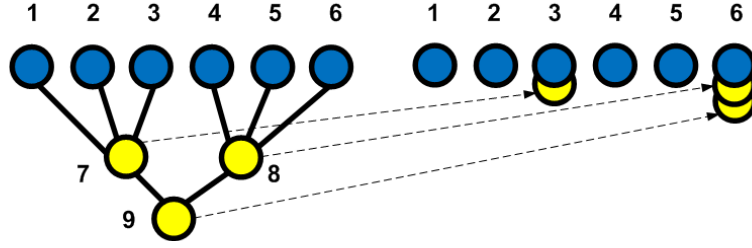

Figure 4.6: Reuse worker serverless instances as parameter servers.

Since creating new serverless instances is associated with latency overhead, the actual data transfer latency can be further reduced by reusing worker serverless instances as parameter servers as shown by Figure 4.6. After completing their neural network training work, workers 3 and 6 continue to collect gradient data from 1, 2, 3, and 4, 5, 6, respectively. In other words, worker 3 (6) plays the role of parameter server 7 (8) now. After merging data from workers 4, 5, and 6, node 6 further collects the gradient data from parameter server 3, and is actually doing the merging formerly done by node 9. Please note such reuse is possible only for the synchronous data update.

4.2.4 Cost and Performance-Cost Ratio Optimization

A key motivation for serverless computing is its economic advantage over the conventional cloud services. Hence, we study how to minimize monetary cost and maximize performance-cost ratio in using serverless computing. In contrast to the offline structure optimization, the cost and performance-cost ratio optimizations are online techniques.

Table 4.1: AWS Lambda price vs. memory use.

Memory (MB)	Price per 100ms ($)
128	0.000000208
192	0.000000313
256	0.000000417
...	...
1408	0.000002292
1472	0.000002396
1536	0.000002501

We derive a model of monetary cost with respect to the memory allocated to a serverless in-

stance. Please note memory size z of a serverless instance should be no less than the minimum memory required to run the application there. In addition, a large memory size z implies shorter latency [123]. Hence, latency is a function of memory size as $t(z)$. For AWS Lambda [27], the monetary price per unit runtime for different memory sizes [124] is shown in Table 4.1. By such pricing, the monetary cost has linear dependence on memory size of the Lambda instance and latency, and can be defined as

$$C(z) = p \cdot n \cdot z \cdot t(z), \tag{4.5}$$

where $p = 1.63 \times 10^{-8} \$/(MB \cdot s)$ and n is the number of Lambda instances.

We propose an online gradient descent method for finding memory size z for each serverless instance such that the monetary cost $C(z)$ is minimized. The online optimization starts with a random memory size z_1, which is sufficiently large for the neural network training and satisfies serverless instance specification. The training with z_1 memory is continued with q iterations and the average cost $\bar{C}(z_1)$ over the q iterations is estimated according to Equation (4.5). Then, memory size is changed to another random and feasible value z_2 for another q iterations of training to obtain an average estimation $\bar{C}(z_2)$. After the sampling of two random sizes, we find an optimized memory size as

$$z_3^* = z_1 - \alpha \frac{\bar{C}(z_1) - \bar{C}(z_2)}{z_1 - z_2} \tag{4.6}$$

where α is the step size for the gradient decent. Since there are lower bound z_{min} and upper bound z_{max} for the actual memory size due to serverless instance restrictions and application requirement, the actual memory size to be used next is

$$z_3 = \max(z_{min}, \min(z_{max}, z_3^*)) \tag{4.7}$$

and this procedure can be repeated such that the memory size is continuously optimized.

We propose another online gradient decent method for maximizing performance-cost ratio. Given a computing task that requires f floating point operations, the performance can be characterized by FLOPS (floating point operations per second), which can be estimated by $\frac{f}{t(z)}$. Then,

the performance-cost ratio is defined by

$$R(z) = \frac{f}{p \cdot n \cdot z \cdot t^2(z)} \tag{4.8}$$

which is a function depending on memory size z. Like minimizing the cost, one can sample a size for q iterations. If the two consecutive sample sizes are z_{j-1} and z_j, then the optimized memory size can be obtained as

$$z_{j+1} = \max(z_{min}, \min(z_{max}, z_j + \alpha \frac{\bar{R}(z_j) - \bar{R}(z_{j-1})}{z_j - z_{j-1}})) \tag{4.9}$$

where $\bar{R}$ indicates the average ratio over q iterations.

4.3 Parallel Hyperparameter Tuning of Neural Network Models with Serverless

The effectiveness of a neural network model and its training efficiency highly depend on hyperparameters, such as the number of hidden layers, activation function and training rate. The hyperparameters can be decided either manually or through automated search such as random search, grid search and Bayesian optimization [125].

Since the evaluations of different hyperparameters can be independently carried out, serverless computing is a particularly appealing choice for the tuning. Suppose $H = \{h_1, h_2, ...\}$ is a set of hyperparameters for a specific neural network model. All sets hyperparameters to be explored are $\mathcal{H} = \{H_1, H_2, ...\}$. One can request n_i serverless instances for training the model specified by $H_i \in \mathcal{H}$. Since the total number of serverless instances one can request is bounded by N. We need to make sure that

$$\sum_{i=1}^{|\mathcal{H}|} n_i \leq N. \tag{4.10}$$

Due to this restriction, serverless hyperparameter tuning is mostly for small network models.

4.4 Experiments

4.4.1 Experiment Setup

These experiments are conducted on a randomly generated dataset, CIFAR-10 dataset [126] and MNIST dataset [127]. The random dataset contains 1 million samples, each of which is composed by 20 binary features and 1 binary label. The random dataset is applied with a fully connected neural network with 5 hidden layers, 500 hidden nodes and 42601 parameters. The CIFAR-10 dataset is to be trained by a convolution neural network, which has 2 convolution layers, 2 pooling layers, 2 normalization layers, 2 fully connected layers and 1 softmax output layer. This structure is the same as the structure used in the code of TensorFlow Tutorials [118]. The model for MNIST is a fully connected neural network, whose structure is investigated through the hyperparameter tuning. The characteristics of the 3 testcases are summarized in Table 4.2. The training of using the datasets on the models is by TensorFlow [128]. The serverless computing experiments are conducted through AWS Lambda [27], where latency, memory use and monetary cost are measured. The training experiment is also performed on a desktop computer with a Intel 3.4GHz CPU with 16GB memory.

Table 4.2: Testcases

	Case A	Case B	Case C
Dataset	Random dataset	CIFAR-10 [126]	MNIST [127]
Network type	Fully connected neural network	Convolution neural network	Fully connected neural network
Network structure	5 hidden layers, 500 hidden nodes and 42601 parameters.	Same as in the code of TensorFlow Tutorial [118].	Structure investigated through the hyperparameter tuning.

4.4.2 Latency Variation

When evaluating serverless computing latency, one faces the challenge of its variations. Serverless runtimes are instantiated on infrastructure via resource scheduling by the service provider in

a manner invisible to the end user. Similarly, the location and the latency response of database for reads and writes may vary depending on the resource allocation on the cloud provider side. There are no guarantees on latency and performance of such serverless instantiations beyond the requested parameters such as memory size (which are priced). Likewise, the read and write latency between serverless instance and database may vary depending on a variety of factors, like the actual location of the database relative to the instance, traffic on networks, multi-tenancy, to name a few. The end-user does not have control on these latencies and performance metric, and expectations are that they vary within a certain known range (based on the provider) from a statistical perspective. *Therefore, all latency and performance measurements reported in the paper are representative, and a few percent variation or improvement is considered normal statistical variation.*

4.4.3 Structure Optimization

This part of experiment is to evaluate the effectiveness of the proposed multi-layer merging structure and its optimization, which are introduced in Section 4.2.3. It is performed on Case A and Case B. For Case A, the number of training iterations is 50. The training is done by 100 workers, each of which has 512MB memory. For Case B, the number of training iterations is 20. The training is done by 100 workers, each of which has 1536MB memory. The results are summarized in Table 4.3. Each structure is indicated by a vector, where each element specifies the number of nodes in a layer and the elements are in bottom-up order of the tree structure depicted in Figure 4.5. For example, $[1, 5, 100]$ means the gradients from 100 workers are transferred to 5 parameter servers, and finally merged at a single parameter server. The result in the first row, which is labeled with '*', is the optimal solution according to our optimization. For Case A, One can see this is the second to the minimal latency result according to the measurement. Its actual latency 569.55 is close to the minimal latency 534.28. The discrepancy between our optimal solution and the actual minimal is due to the latency variation, which is discussed in Section 4.4.2. For Case B, our optimal solution has the least latency and therefore the effectiveness of our optimization is confirmed. One should also note that according to our discussion in Section 4.2.3, the optimal

solution does not depend on the neural network model used but only depends on the number of workers.

Table 4.3: Latency of different structures

Structure	Latency for Case A (s)	Latency for Case B (s)
*[1,5,22,100]	569.55	789.20
[1,100]	1216.97	1848.05
[1,2,100]	878.78	1195.97
[1,5,100]	650.66	866.29
[1,25,100]	616.67	920.22
[1,2,10,100]	570.66	823.25
[1,5,50,100]	604.90	890.91
[1,10,50,100]	534.28	838.89
[1,2,10,50,100]	585.25	883.96
[1,5,20,50,100]	578.22	868.78

4.4.4 Training Accuracy and Convergence Rate

We evaluate the training accuracy and convergence rate of the proposed serverless computing and sequential computing on desktop PC on Case A and Case B. The serverless structures used here are the same as in Section 4.4.3. The accuracy of a neural network is estimated by comparing its inference results on training dataset labels. The accuracy versus training time results for Case A are shown in Figure 4.7. The serverless computing converges slower than desktop PC, but reaches a better accuracy. The results for Case B are plotted in Figure 4.8, where the serverless computing leads to worse accuracy and convergence rate than the desktop PC.

The accuracy difference between the desktop PC and serverless results arises from the parameter update difference between sequential and parallel training. In serverless computing, the parameter update is based on the average of gradients obtained from multiple workers. In the sequential training on desktop PC, by contrast, each parameter update is according to a single set of gradients from a single process.

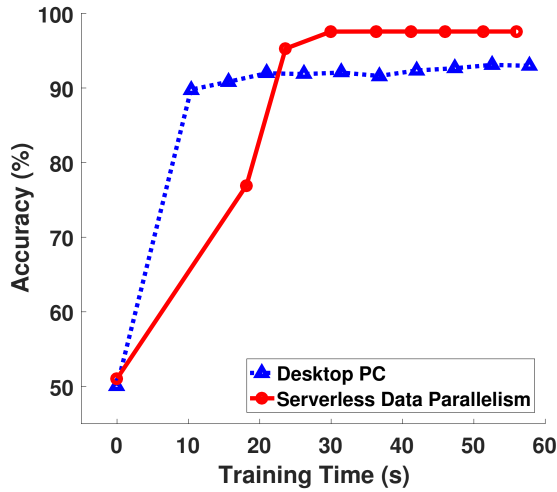

Figure 4.7: Training accuracy vs. training time for Case A.

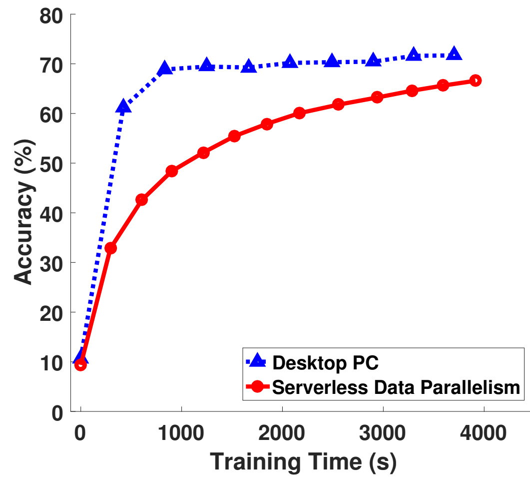

Figure 4.8: Training accuracy vs. training time for Case B.

4.4.5 Result of Cost and Performance-Cost Ratio Optimization

The proposed online cost minimization method is evaluated on Case A. In this experiment, $q = 10$, which means we modify the memory size every 10 iterations. In addition, the gradient decent is performed at most five times. The lower and upper bounds of memory size are set as $z_{min} = 256MB$ and $z_{max} = 1536MB$, respectively. The results are shown in Figure 4.9, where the red circles indicate our optimization results. The experiment is repeated 10 times. Due to the latency $t(z)$ variations, two different results (red circles) are obtained. For 9 times, the optimization result is $256MB$ and $512MB$ is obtained once. The blue triangles and bars are the measurement results of monetary cost at different memory sizes without optimization. For each memory size, the experiment is repeated 100 times. Each blue triangle represents the average cost and the bars indicate $\pm\sigma$, which is the standard deviation. One can see that the cost variation can be very large due to the latency uncertainty. Moreover, the cost vs. z change is not monotone. The average cost of $640MB$ is less than that for $512MB$ memory. Most importantly, our optimization indeed reaches the minimum or near minimum cost memory size.

The performance-cost ratio optimization results are plotted in Figure 4.10. Here, we attempt to

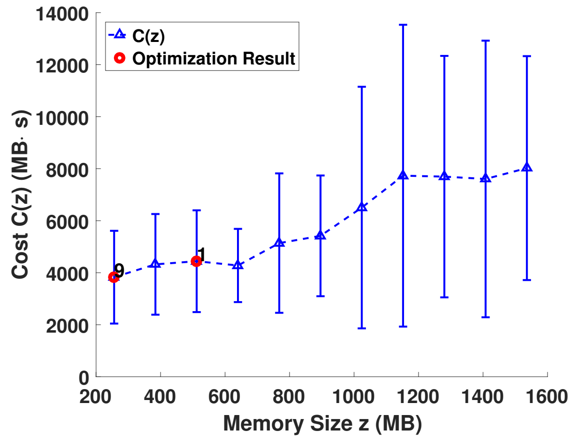

Figure 4.9: Cost per iteration under different Lambda instance memory sizes and optimization results.

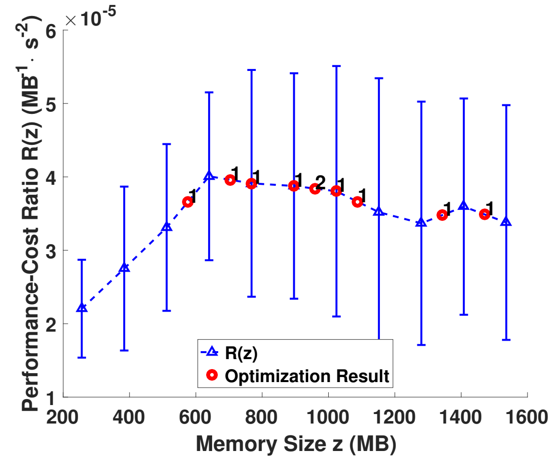

Figure 4.10: Performance-cost ratio per iteration under different Lambda instance memory sizes and optimization results.

maximize the ratio. Indeed, the red circle results from our optimization are generally at memory sizes where the ratio is at least near the maximum.

4.4.6 Results on Hyperparameter Tuning

The experiment on hyperparameter tuning is performed on Case C. In Figure 4.11, the computing latency results versus the number of searched hyperparameter sets $|\mathcal{H}|$ for desktop PC and AWS Lambda are plotted. Each dot in the figure is the average of 10 different experiments with the same number of searched hyperparameter sets. One can see that the latency of desktop PC grows linearly when more hyperparameters are evaluated because of its sequential computing nature. The hyperparameter tuning on AWS Lambda is carried out in parallel. Thus, its latency does not change when the hyperparameter search is expanded. This result clearly demonstrates the advantage of serverless computing for hyperparameter tuning in neural network model construction and training.

4.5 Opportunities in Serverless Runtime Design

Serverless runtimes have been used for inference with good results. Our exploration of using serverless for training large deep learning models has identified some disadvantages compared to

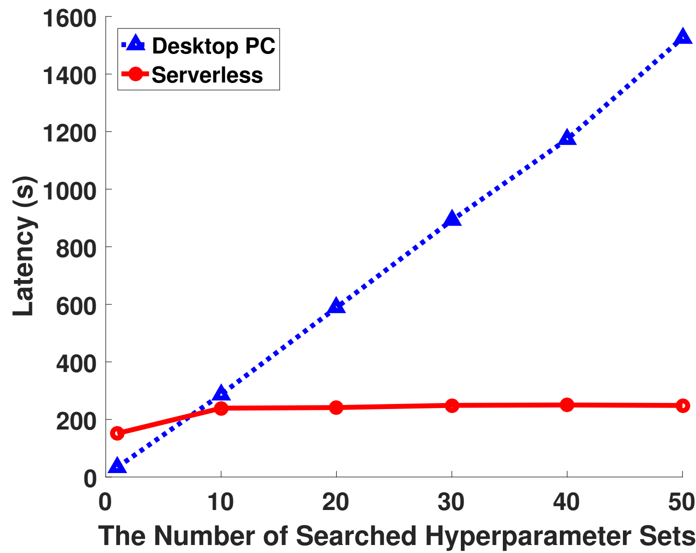

Figure 4.11: Computing latency versus the number of searched hyperparameter sets $|\mathcal{H}|$.

other distributed computing runtimes where data transfer between compute instances are not as frequent (such as with GPUs). In order to improve serverless performance for the task of training deep learning models, it is necessary to minimize the frequency and quantity of data transfer between subsequent serverless instances. We illustrate opportunities for improved data transfer latencies, while at the same time maintaining the benefits of serverless such as the ability to pay for a compute instance used only when and just long enough for needed computation.

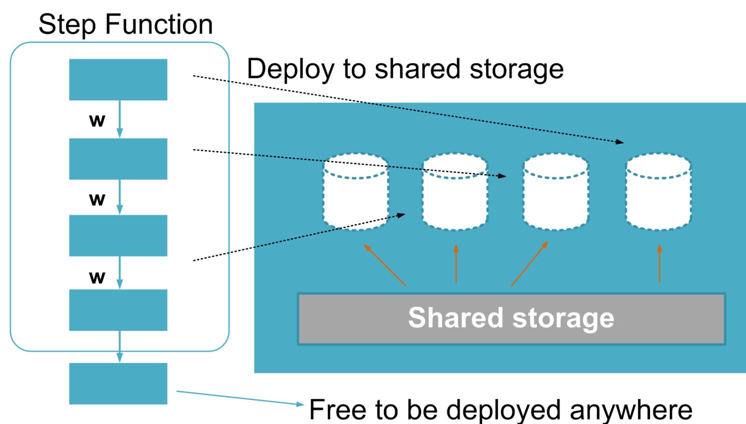

Figure 4.12: Serverless affinity in runtimes

Consider a step function where edges between serverless instances can be assigned higher affinities indicating sharing of data between them, then the instances can be mapped by the infrastructure manager in a manner where the latencies for data transfer between the instances is minimized. In Figure 4.12, a generic approach to such a solution is given, where a portion of the step function has some serverless instances with weight w on the edges indicating shared data, while the last instance has no weight assigned, indicating no such affinity. Such a specification can be mapped in several ways as described below:

- Given affinities between serverless instances in a step function, the infrastructure maps these instances to a common host where storage is persistent across serverless instance invocations. In a runtime where each serverless instance is implemented as a Linux container, the storage on the host is mounted onto the container during boot up, thus enabling sharing of data.
- Implementations based on processor in memory (PIM) may also be considered for this purpose. A processor associated with a memory device such as a Linux on ARM associated with SSD or Memory, can also be used to support the affinity for especially large models.

4.6 Conclusion

Training deep learning models with serverless runtimes is challenging and provides several opportunities. We have investigated both large and small models. For large models, various structures for composition of serverless instances provide the best performance and cost to train deep learning models, while taking advantage of data parallelism are explored. The challenges posed by the ephemeral, stateless and warm up latency of serverless runtimes are studied. Potential innovations in runtime design for future serverless runtimes with containers are proposed to mitigate the challenges and strengthen the opportunities. For smaller models, it is shown that serverless runtimes showed benefit for hyperparameter tuning that could be performed in a truly distributed manner.

5. SUMMARY AND CONCLUSIONS[1]

This dissertation research is focused on security and efficiency across different levels of a computing system, from hardware to software, from manufacturing to application, and from circuit level to system level.

For secure and efficient circuit layout design, new techniques are developed to enhance the split fabrication technology. Besides, DFM is considered in our security enhancement and thus, the practicality is increased. The results prove the effectiveness of our proposed techniques. For CMP, the uniformity of the wire density can be improved and each layer of the design can be more flatten after CMP process. For SADP, almost all the violations are eliminated and thus, the design can be manufactured by SADP technology without using many high cost ebeams.

For hardware-assisted software security, we designed hardware for CFI and DFI verification. For hardware-based CFI verification, instead of software, FPGA is used to do the verification, which decreases performance overhead to almost 0. Meanwhile, due to the performance advantage of hardware circuits, the verification latency is as low as only more than 20 FPGA clock cycles. The CFG compression algorithm is proposed and can make CFG checker accommodate 18% more CFG nodes or further reduce the hardware resource consumption of implementing the CFG by 11%. Besides, automatic CFG checker generator is designed to provide users the ability to deploy the CFI verification FPGA design in 20 minutes given a software to be checked. For hardware-based DFI verification, processing-in-memory is used to mitigate the most critical problem existing in the software-based data-flow integrity verification, which is the intensive data transferring. By implementing the DFI checking inside processor in the memory, the data trans-

[1]©2017 IEEE. Reprinted, with permission, from Lang Feng, Yujie Wang, Jiang Hu, Wai-Kei Mak and Jeyavijayan Rajendran, "Making Split Fabrication Synergistically Secure and Manufacturable", IEEE/ACM International Conference on Computer-Aided Design, 11/2017. ©2018 IEEE. Reprinted, with permission, from Lang Feng, Prabhakar Kudva, Dilma Da Silva and Jiang Hu, "Exploring Serverless Computing for Neural Network Training", IEEE International Conference on Cloud Computing, 07/2018. Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature, International Conference on Computer-Aided Design, "FastCFI: Real-Time Control Flow Integrity using FPGA without Code Instrumentation", Lang Feng, Jeff Huang, Jiang Hu and Abhijith Reddy, ©2019.

ferring rate between the main processor and memory is largely reduced and the performance is improved by $4\times$ comparing to software-based DFI verification, without sacrificing verification precision.

For exploring serverless computing for machine learning model training, large neural network training is deployed in serverless environment. Besides, optimization approaches are proposed to optimize the performance, cost and performance-cost ratio. For small neural network, parallelized hyperparameter tuning by leveraging serverless computing is proposed. It shows serverless computing has huge advantage to do hyperparameter tuning due to its high parallel capacity. The bottleneck of the serverless computing, which is the data transferring can be mitigated by sharing the storages between multiple serverless instances.

In the conclusion, this dissertation successfully explored different fields related to computing efficiency and security, and proposed solutions for various of problems. Experiment results proved the effectiveness of the solutions.

REFERENCES

[1] R. Tian, D. F. Wong, and R. Boone, “Model-Based Dummy Feature Placement for Oxide Chemical-Mechanical Polishing Manufacturability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 7, pp. 902–910, 2001.

[2] Y. Wang, P. Chen, J. Hu, and J. Rajendran, “Routing Perturbation for Enhanced Security in Split Manufacturing,” *Asia and South Pacific Design Automation Conference*, pp. 605–510, 2017.

[3] J. Rajendran, O. Sinanoglu, and R. Karri, “Is Split Manufacturing Secure,” *Design, Automation Test in Europe Conference*, pp. 1259–1264, 2013.

[4] Intelligence Advanced Research Projects Activity, “Trusted Integrated Circuits Program.” `https://www.fbo.gov/utils/view?id=b8be3d2c5d5babbdffc6975c370247a6`, 2011.

[5] Y. Wang, P. Chen, J. Hu, and J. Rajendran, “The Cat and Mouse in Split Manufacturing,” *ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, 2016.

[6] J. Magaña, D. Shi, and A. Davoodi, “Are Proximity Attacks a Threat to the Security of Split Manufacturing of Integrated Circuits?,” *IEEE/ACM International Conference on Computer-Aided Design*, pp. 90:1–90:7, 2016.

[7] D. Z. Pan, B. Yu, and J.-R. Gao, “Design for Manufacturing With Emerging Nanolithography,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1453–1472, 2013.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow Integrity,” *ACM Conference on Computer and Communications Security*, pp. 340–353, 2005.

[9] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86),” *ACM Conference on Computer and Communications Security*, pp. 552–561, 2007.

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented Programming: A New Class of Code-reuse Attack,” *ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, 2011.

[11] The Heartbleed Bug. `http://heartbleed.com/`.

[12] Null HTTPd Remote Heap Overflow Vulnerability. `https://www.securityfocus.com/bid/5774`.

[13] M. Zhang and R. Sekar, “Control Flow Integrity for COTS Binaries,” *USENIX Security Symposium*, pp. 337–352, 2013.

[14] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. NÃijrnberger, and A. reza Sadeghi, “MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones,” *Symposium on Network and Distributed System Security*, 2012.

[15] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent ROP Exploit Mitigation Using Indirect Branch Tracing,” *USENIX Security Symposium*, pp. 447–462, 2013.

[16] M. Castro, M. Costa, and T. Harris, “Securing Software by Enforcing Data-Flow Integrity,” *Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006.

[17] X. Ge, W. Cui, and T. Jaeger, “GRIFFIN: Guarding Control Flows Using Intel Processor Trace,” *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 585–598, 2017.

[18] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, “Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 3, pp. 52:1–52:25, 2017.

[19] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, “Efficient Protection of Path-Sensitive Control Security,” *USENIX Security Symposium*, pp. 131–148, 2017.

[20] Y. Xia, Y. Liu, H. Chen, and B. Zang, “CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters,” *International Conference on Dependable Systems and Networks*, pp. 1–12, 2012.

[21] P. M. Kogge, “EXECUBE-A New Architecture for Scaleable MPPs,” *International Conference on Parallel Processing*, pp. 77–84, 1994.

[22] J. Jeddeloh and B. Keeth, “Hybrid Memory Cube New DRAM Architecture Increases Density and Performance,” *Symposium on VLSI Technology*, pp. 87–88, 2012.

[23] D. U. Lee, K. W. Kim, K. W. Kim, K. S. Lee, S. J. Byeon, J. H. Kim, J. H. Cho, J. Lee, and J. H. Chun, “A 1.2 V 8 Gb 8-Channel 128 GB/s High-Bandwidth Memory (HBM) Stacked DRAM With Effective I/O Test Circuits,” *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 191–203, 2015.

[24] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, “HBM (High Bandwidth Memory) DRAM Technology and Architecture,” *IEEE International Memory Workshop*, pp. 1–4, 2017.

[25] Awesome serverless Git. `https://github.com/anaibol/awesome-serverless`.

[26] G. McGrath and P. R. Brenner, “Serverless Computing: Design, Implementation, and Performance,” *IEEE International Conference on Distributed Computing Systems Workshops*, pp. 405–410, 2017.

[27] AWS Lambda. `https://aws.amazon.com/lambda/`.

[28] Google Cloud, "Building a Serverless ML Model." https://cloud.google.com/solutions/building-a-serverless-ml-model.

[29] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," *Computing Research Repository*, 2017.

[30] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized Stochastic Gradient Descent," *International Conference on Neural Information Processing Systems*, pp. 2595–2603, 2010.

[31] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," *International Conference on Neural Information Processing Systems*, pp. 693–701, 2011.

[32] J. Keuper and F.-J. Pfreundt, "Asynchronous Parallel Stochastic Gradient Descent: A Numeric Core for Scalable Distributed Machine Learning Algorithms," *Workshop on Machine Learning in High-Performance Computing Environments*, pp. 1:1–1:11, 2015.

[33] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," *International Conference on Neural Information Processing Systems*, pp. 1223–1231, 2012.

[34] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *CoRR*, 2015.

[35] M. Jagasivamani, P. Gadfort, M. Sika, M. Bajura, and M. Fritze, "Split-Fabrication Obfuscation: Metrics and Techniques," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 7–12, 2014.

[36] K. Vaidyanathan, P. B. Das, E. Sumbul, R. Liu, and L. Pileggi, "Building Trusted ICs using split fabrication," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 1–6, 2014.

[37] K. Vaidyanathan, R. Liu, E. Sumbul, Q. Zhu, F. Franchetti, and L. Pileggi, "Efficient and Secure Intellectual Property (IP) Design with Split Fabrication," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 13–18, 2014.

[38] J. Valamehr, T. Sherwood, R. Kastner, D. Marangoni-Simonsen, T. Huffmire, C. Irvine, and T. Levin, "A 3-D Split Manufacturing Approach to Trustworthy System Development," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 4, pp. 611–615, 2013.

[39] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara, "Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation," *USENIX Security Symposium*, pp. 495–510, 2013.

[40] Y. Xie, C. Bao, and A. Srivastava, "Security-Aware Design Flow for 2.5D IC Technology," *International Workshop on Trustworthy Embedded Devices*, pp. 31–38, 2015.

[41] M. Cho, D. Z. Pan, H. Xiang, and R. Puri, "Wire Density Driven Global Routing for CMP Variation and Timing," *IEEE/ACM International Conference on Computer Aided Design*, pp. 487–492, 2006.

[42] H.-Y. Chen, S.-J. Chou, S.-L. Wang, and Y.-W. Chang, "A Novel Wire-Density-Driven Full-Chip Routing System for CMP Variation Control," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 193–206, 2009.

[43] J. R. Gao and D. Z. Pan, "Flexible Self-aligned Double Patterning Aware Detailed Routing with Prescribed Layout Planning," *ACM International Symposium on Physical Design*, pp. 25–32, 2012.

[44] I.-J. Liu, S.-Y. Fang, and Y.-W. Chang, "Overlay-Aware Detailed Routing for Self-Aligned Double Patterning Lithography Using the Cut Process," *ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, 2014.

[45] Y. Ding, C. Chu, and W.-K. Mak, "Throughput Optimization for SADP and E-beam based Manufacturing of 1D Layout," *ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, 2014.

[46] J. Kuang, E. F. Y. Young, and B. Yu, "Incorporating Cut Redistribution with Mask Assignment to Enable 1D Gridded Design," *IEEE/ACM International Conference on Computer-Aided Design*, pp. 48:1–48:8, 2016.

[47] D. Ouma, D. Boning, J. Chung, G. Shin, L. Olsen, and J. Clark, "An Integrated Characterization and Modeling Methodology for CMP Dielectric Planarization," *IEEE International Interconnect Technology Conference*, pp. 67–69, 1998.

[48] N. Dhumane and S. Kundu, "Critical Area Driven Dummy Fill Insertion to Improve Manufacturing Yield," *International Symposium on Quality Electronic Design*, pp. 334–341, 2012.

[49] Gurobi Optimization, "Gurobi-7.0.2." `http://www.gurobi.com`.

[50] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and H. R. Deng, "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack," *Symposium on Network and Distributed System Security*, 2014.

[51] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection," *USENIX Security Symposium*, pp. 401–416, 2014.

[52] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, "Hardware-Assisted Detection of Malicious Software in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94–97, 2012.

[53] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1271–1285, 2006.

[54] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy Without Tactics: Policy-Agnostic Hardware-Enhanced Control-Flow Integrity," *ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, 2016.

[55] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and Efficient CFI Enforcement with Intel Processor Trace," *IEEE International Symposium on High Performance Computer Architecture*, pp. 529–540, 2017.

[56] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch Regulation: Low-Overhead Protection from Code Reuse Attacks," *International Symposium on Computer Architecture*, pp. 94–105, 2012.

[57] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for Signature-based Protection from Code Reuse Attacks," *IEEE International Symposium on High Performance Computer Architecture*, pp. 258–269, 2013.

[58] A. Francillon, D. Perito, and C. Castelluccia, "Defending Embedded Systems Against Control Flow Attacks," *ACM Workshop on Secure Execution of Untrusted Code*, pp. 19–26, 2009.

[59] S. Das, W. Zhang, and Y. Liu, "A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 24, no. 11, pp. 3193–3207, 2016.

[60] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.

[61] Z. Guo, R. Bhakta, and I. G. Harris, "Control-flow Checking for Intrusion Detection via a Real-time Debug Interface," *International Conference on Smart Computing Workshops*, pp. 87–92, 2014.

[62] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace," *ACM Conference on Data and Application Security and Privacy*, pp. 173–184, 2017.

[63] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Efficiently Securing Systems from Code Reuse Attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, 2014.

[64] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted Flow Integrity Extension," *ACM/EDAC/IEEE Design Automation Conference*, pp. 74:1–74:6, 2015.

[65] R. de Clercq, J. Gtzfried, D. bler, P. Maene, and I. Verbauwhede, “SOFIA: Software and Control Flow Integrity Architecture,” *Computers & Security*, vol. 68, pp. 16–35, 2017.

[66] S. Mao and T. Wolf, “Hardware Support for Secure Processing in Embedded Systems,” *ACM/EDAC/IEEE Design Automation Conference*, pp. 483–488, 2007.

[67] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, “HCFI: Hardware-enforced Control-Flow Integrity,” *ACM Conference on Data and Application Security and Privacy*, pp. 38–49, 2016.

[68] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, “Integration of ROP/JOP Monitoring IPs in an ARM-based SoC,” *Conference on Design, Automation & Test in Europe*, pp. 331–336, 2016.

[69] Intel CET. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

[70] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, “Enforcing Kernel Security Invariants with Data Flow Integrity,” *Network and Distributed System Security Symposium*, 2016.

[71] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing Memory Error Exploits with WIT,” *IEEE Symposium on Security and Privacy*, pp. 263–277, 2008.

[72] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, “HDFI: Hardware-Assisted Data-Flow Isolation,” *IEEE Symposium on Security and Privacy*, pp. 1–17, 2016.

[73] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, “CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization,” *IEEE Symposium on Security and Privacy*, pp. 20–37, 2015.

[74] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks,” *IEEE Symposium on Security and Privacy*, pp. 969–986, 2016.

[75] J. R. Crandall and F. T. Chong, “Minos: Control Data Attack Prevention Orthogonal to Memory Model,” *IEEE/ACM International Symposium on Microarchitecture*, pp. 221–232, 2004.

[76] K. Biba, “Integrity Considerations for Secure Computer Systems,” *Defense Technical Information Center*, p. 68, 1977.

[77] IDA. `https://www.hex-rays.com/products/ida/index.shtml`.

[78] CoreSight Program Flow Trace. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf.

[79] Write XOR Execute. https://en.wikipedia.org/wiki/W%5EX.

[80] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[81] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): Chips that Remember and Compute," *IEEE International Solids-State Circuits Conference*, pp. 224–225, 1997.

[82] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[83] P. M. Nyasulu, *System Design for a Computational-Ram Logic-in-Memory Parallel-Processing Machine*. PhD thesis, 1999.

[84] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 192–201, 1999.

[85] A. K. Snip, D. G. Elliott, M. Margala, and N. G. Durdle, "Using Computational RAM for Volume Rendering," *IEEE International ASIC/SOC Conference*, pp. 253 – 257, 2000.

[86] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *International Symposium on Computer Architecture*, pp. 161–171, 2000.

[87] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, and et al., "The Architecture of the DIVA Processing-in-Memory Chip," *International Conference on Supercomputing*, pp. 14–25, 2002.

[88] J. Adibi, T. Barrett, S. Bhatt, H. Chalupsky, J. Chame, and M. Hall, "Processing-in-Memory Technology for Knowledge Discovery Algorithms," *International Workshop on Data Management on New Hardware*, pp. 2–es, 2006.

[89] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, "On the Implementation of Computation-in-Memory Parallel Adder," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 8, pp. 2206–2219, 2017.

[90] M. Scrbak, M. Islam, K. M. Kavi, M. Ignatowski, and N. Jayasena, "Exploring the Processing-in-Memory Design Space," *Journal of Systems Architecture*, vol. 75, no. C, pp. 59 – 67, 2017.

[91] H.-W. Tseng and D. M. Tullsen, "Data-triggered Multithreading for Near-Data Processing," *Workshop on Near-Data Processing*, 2013.

[92] M. L. Chu, N. Jayasena, D. P. Zhang, and M. Ignatowski, “High-level Programming Model Abstractions for Processing in Memory,” *Workshop on Near-Data Processing*, 2013.

[93] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads,” *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 190–200, 2014.

[94] X. Yang, Y. Hou, and H. He, “A Processing-in-Memory Architecture Programming Paradigm for Wireless Internet-of-Things Applications,” *Sensors*, vol. 19, no. 1, p. 140, 2019.

[95] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, “Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities,” *International Conference on Parallel Architecture and Compilation Techniques*, pp. 31–44, 2016.

[96] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing,” *International Symposium on Computer Architecture*, pp. 105–117, 2015.

[97] Radeon Rx 300 Series. https://en.wikipedia.org/wiki/Radeon_Rx_300_series.

[98] NVIDIA TESLA P100. https://en.wikipedia.org/wiki/Radeon_Rx_300_series.

[99] Hybrid Memory Cube - HMC Gen2. https://www.micron.com/-/media/client/global/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf.

[100] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, “Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube,” *International Conference on Architecture of Computing Systems*, pp. 19–31, 2016.

[101] LLVM. https://llvm.org/.

[102] SVF for Reaching Definition Analysis. https://github.tamu.edu/jyhuang/SVF.

[103] Bitonic Sorting Network. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm.

[104] L. O. Andersen, “Program Analysis and Specialization for the C Programming Language,” tech. rep., 1994.

[105] Intel Quartus Prime. https://fpgasoftware.intel.com/17.1/?edition=lite.

[106] SMCsim. https://iis-git.ee.ethz.ch/erfan.azarkhish/SMCSim.

[107] The gem5 Simulator. http://www.gem5.org/Main_Page.

[108] RIPE. https://github.com/johnwilander/RIPE.

[109] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," *Computer Security Applications Conference*, pp. 41–50, 2011.

[110] SPEC CPU 2006 Benchmark. https://www.spec.org/cpu2006/.

[111] S. Das, Y. Liu, W. Zhang, and C. Mahinthan, "Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

[112] The Source Code for Triggering Heartbleed Bug. https://github.com/mykter/afl-training/tree/master/challenges/heartbleed.

[113] ASAP 7nm Predictive PDK. http://asap.asu.edu/asap/.

[114] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," *Computing Research Repository*, 2017.

[115] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking Heterogeneous Cloud Functions," *Parallel Processing Workshops*, pp. 415–426, 2018.

[116] M. Malawski, "Towards Serverless Execution of Scientific Workflows - HyperFlow Case Study," *WORKS@SC*, November 2016.

[117] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," *Computing Research Repository*, 2017.

[118] TensorFlow Tutorials: Convolutional Neural Networks. https://www.tensorflow.org/tutorials/deep_cnn.

[119] Distributed TensorFlow. https://www.tensorflow.org/deploy/distributed.

[120] J. Yang, Y. Chen, S. Wang, L. Li, C. Meng, M. Qiu, and W. Chu, "Practical Lessons of Distributed Deep Learning," *International Conference on Machine Learning*, 2017.

[121] AWS, "AWS Step Functions." https://aws.amazon.com/step-functions/.

[122] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, pp. 400–407, 1951.

[123] Configuring Lambda Functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html.

[124] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[125] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *International Conference on Neural Information Processing Systems*, pp. 2951–2959, 2012.

[126] CIFAR-10 Dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[127] The MNIST Database. http://yann.lecun.com/exdb/mnist/.

[128] TensorFlow. https://www.tensorflow.org/.