Architectural Support for Software Security

A Thesis Presented

by

Juan Carlos Martínez Santos

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

June 2013
Abstract

Program execution can be tampered by attackers through exploitation of diverse sequential and concurrent errors. Many software and hardware approaches have been presented to validate data integrity and check program execution. Software approaches suffer large performance degradation that limits their application on run-time systems. Hardware approaches reduce the performance penalty, but current solutions incur high logic and storage overhead. To address these challenges, we propose to augment multiple system layers, i.e., hardware architecture, compiler, and operating system, in a coordinated manner to provide efficient monitoring with the same or better level of protection than current solutions. Our approaches consider security during the design process, along with other metrics such as cost, performance, and power so as to: 1) augment the attack detection rate in existing control-flow monitors, 2) reduce the overhead for doing dynamic information flow tracking in a single core architecture, 3) elevate the efficacy for doing dynamic information flow tracking in the multi-core scenario, 4) improve the efficacy and the efficiency of multi-thread data isolation.

In the first case, we found that the current control-flow monitor systems are still vulnerable to non-control data attacks. A malicious user can deceive the detection mechanism that the attack actions are part of the normal behavior. It is possible because of the inherent limitations in the information used to validate the execution profile at run-time, the execution signature. We proposed to leverage the existing branch prediction mechanism,
including the branch target buffer, to monitor the execution program. We incorporated information like the binary branch taking history and the expected (ahead) path into the program’s signature in order to make it stronger. We tested our approach under conditions that current approaches fail to detect non-control data attacks. Our solution can detect the traditional control-flow attacks as well as the non-control data attacks, reduce the implementation complexity, and get a low execution overhead.

Dynamic information flow tracking (DIFT) is a popular method to trace input propagation in a computer system so as to prevent memory corruptions. However, DIFT slows down the program execution and many implementations require large memory space for tag storage (also known as metadata). In order to alleviate these problems, we proposed compile-assisted aggregation of program variables according to their taints and memory page allocation. We also proposed modifications in the memory management unit and operating system for in-core run-time tracking. Our approach almost eliminates the storage overhead and significantly reduces the performance degradation caused by dynamic tracking and processing.

Furthermore, when adopting dynamic information flow tracking in multi-core architectures, we found that inconsistency issues between data and metadata processing may arise because data and metadata are processed in different places (cores) at different time. Current solutions enforce the same data order for metadata processing, and do not allow in-
dependent metadata to be accessed out-of-order. As a result, the execution overhead is large. We proposed a centralized architectural approach that uses the existing cache coherence unit to enforce the correct order for metadata when it is needed and meanwhile allow out-of-order metadata processing. Our results show a considerable overhead reduction compared to in-order processing, and significant improvement over the prior distributed approach in terms of implementation complexity and execution overhead.

Finally, ensuring data integrity and confidentiality in multi-thread applications is hard to accomplish because all the threads are running in the same application space. Software mechanisms for thread isolation can prevent an unauthorized thread from accessing other threads’ private data. However, the cost of in-lining checking slows down the application execution. We proposed to reduce the code complexity by moving the checking mechanism from the software domain to the hardware domain. We showed that with system library modification and operating system support, a hardware monitoring mechanism can provide higher security more efficiently.

In summary, our proposed enhancements to hardware architecture, compiler, and operating system contribute to the state of the art in the following ways:

- Elevating the security defense level of control-flow monitors by making the program signature stronger.
- Reducing the storage overhead and program execution slow-down of DIFT by ag-
gregating trusted and untrusted data in different pages.

- Allowing secure out-of-order metadata processing on multi-core DIFT and enforcing metadata access order only when data dependencies are present.

- Ensuring data integrity and confidentiality in multi-thread programs by moving the monitoring mechanism from the software domain to the hardware domain.
Acknowledgements

This dissertation was supported in part by National Science Foundation under a CAREER award - CNS 0845871.

This dissertation would not be possible without the generous help of many people during the course. I started my PhD studies five years ago thanks to a fellowship from the Government of Colombia, and I am very grateful to the institutions that have sponsored me during these years: Fulbright, Departamento de Planeación Nacional and COLCIENCIAS, from the Government of Colombia, Universidad Tecnológica de Bolívar, my home university, the Department of Electrical and Computer Engineering at Northeastern University, and LASPAU, the institution that administers the scholarship. Special thanks go to Dr. Patricia Martinez, Gilma Mestre, Mariana de Castro, William Cartagena, Marbel Marquez, and Jose Luis Villa from Universidad Tecnológica de Bolívar, and Ryan Keane and Lisa Tapiero from LASPAU.

I have run out of words to express my gratitude and appreciation to my research advisor, Dr. Yunsi Fei. I was fortunate to have the opportunity to work under her guidance. She is a very knowledgeable and caring professor, who has advised and supported me in every stage of my research. I have learned many important things from her, not only on technical sides, but also about many other aspects of life that would be of great value to my future endeavors.
I am also very grateful to the professors in my advisory committee. I want to thank Dr. Miriam Leeser and Dr. David Kaeli, from the Department of Electrical and Computer Engineering, and Dr. Erik-Oliver Blass, from College of Computer Science. I also want to thank my lab mates: Dr. Hai Lin and Tiansi Hu. They have been a great support since I started in the lab. I really appreciate their help and advice.

I could not have been successful in this dissertation without my family. Thanks to my dear wife, Sonia Helena, for her unconditional love and support, for encouraging me to reach my goals, for her valuable recommendations, for sharing the duties at home and helping me take care of our little daughters. Thanks to Anna Lucia and Maria Paula for making me happy every day. Thanks to my parents José and Elizabeth, to my brother Diego Fernando, and to my sister Elizabeth Juliana for their love and encouragement. Finally, I want to thank God for making everything possible.
Contents

1 Introduction .................................................. 21
  1.1 Contributions ............................................ 23
  1.2 Thesis Organization ...................................... 25

2 Background and Motivation .................................. 27
  2.1 Defeating Control and Non-Control Data Attacks ................. 28
  2.2 Dynamic Information Flow Tracking Technique ...................... 31
    2.2.1 Dynamic Information Flow Tracking on Multi-Threaded Programs
          with Shared Memory ..................................... 33
  2.3 Thread Isolation ........................................... 35

3 Leveraging Speculative Architectures ....................... 38
  3.1 Speculative Architecture for Control-Flow Transfer and Execution Path
       Validation ................................................. 39
    3.1.1 Training the Full Record Set (FRS) ...................... 42
    3.1.2 Size of Signature ..................................... 43
3.1.3 Extraction of Dynamic Program Execution Information for Validation ................................. 43
3.1.4 BTB Update and Administration ................................................................. 45
3.1.5 Architecture Support for Control-Flow Validation ........................................... 49

3.2 Security Analysis ......................................................................................... 52
3.2.1 Attack Detection Analysis ......................................................................... 53
3.2.2 Speed of Detection .................................................................................. 54
3.2.3 Security of the FRS ................................................................................ 55

3.3 Experimental Results .................................................................................. 55
3.3.1 Ambiguity Alleviation ............................................................................. 55
3.3.2 Storage Overhead .................................................................................... 57
3.3.3 Performance Impact of the Run-Time Validation Mechanism .................. 58

3.4 Related Work .............................................................................................. 62

4 Static Secure Page Allocation for DIFT .............................................................. 65
4.1 PIFT: Paged-dynamic Information Flow Tracking ...................................... 66
4.1.1 General Idea ......................................................................................... 67
4.1.2 Compilation Stage .................................................................................. 69

* Memory Map and Data Attribute ..................................................................... 70

* Code Duplication ............................................................................................ 73

4.2 Architectual Augmentation ........................................................................... 74
4.2.1 Wider Register File ................................................................................ 76
4.2.2 Memory Taint Bit Retrieval .............................. 77
4.3 Security Analysis ........................................... 78
  4.3.1 Protecting the Stack from Buffer Overflow .......... 79
  4.3.2 Avoiding Format String Exploitation ................. 79
  4.3.3 Protecting the Heap .................................... 80
  4.3.4 Defending Against other Attacks ....................... 82
  4.3.5 Testing Our Approach with a Real Attack ............ 83
4.4 Experimental Results ...................................... 84
  4.4.1 Software Implementation ............................... 84
  4.4.2 Static Memory Overhead ................................ 85
  4.4.3 Dynamic Memory Overhead ............................. 86
  4.4.4 Execution Overhead .................................... 87
4.5 Related Work .............................................. 88

5 Metadata Coherence Enforcement .......................... 92
  5.1 Our METACE Approach .................................... 93
    5.1.1 DIFT and Metadata Coherence Overview ............. 93
    5.1.2 Architecture Enhancement ............................. 94
    5.1.3 E-MOESI Protocol Implementation .................... 95
    5.1.4 Modeling Hardware Overhead of METACE ............ 97
    5.1.5 Overall Run-time Execution Overhead of METACE ... 101
  5.2 Metadata Hazard Analysis ................................ 102
6 Hardware Assisted Thread Isolation

6.1 HATI: Hardware Assisted Thread Isolation Mechanism for Multi-threaded C/C++ Programs

6.1.1 General Idea

6.1.2 Advantages of Our Hardware Assisted Approach

   * Performance
   * Dynamic Memory Usage
   * Protection on the Checking Process

6.2 Implementation Details

6.2.1 System Libraries

6.2.2 The Ownership Table: Software Handling

6.2.3 Architectural Enhancement

6.2.4 Operating System Support

6.3 Experimental Results
6.3.1 Benchmarks and Simulators ............................... 126
6.3.2 Dynamic Memory Overhead ............................ 128
6.3.3 Performance Degradation .............................. 129
   * Execution Overhead for Handling the Ownership Table ... 130
   * Run-Time Accessing Overhead .......................... 131
   * Context Switch Time Degradation ...................... 132
   * Overall Performance Overhead ......................... 132
6.3.4 Hardware Overhead .................................... 132
6.3.5 Security Evaluation .................................... 133
6.4 Related Work ............................................. 134

7 Conclusions ................................................. 137

  7.1 Main Conclusions ......................................... 137
  7.2 Future Work ............................................... 140
List of Figures

2.1 Complex program control-flow with ambiguity. .......................... 30

2.2 An example of consistent data access and metadata access. DIFT mecha-
nism ensures that metadata processing is later than data processing ($t_{31} >$
$t_{11}, t_{41} > t_{31}$). ......................................................... 34

2.3 Concurrent program examples with multiple subcomponents. ........... 36

3.1 Complex program control-flow with ambiguity resolution. ............... 40

3.2 Expected paths vector fetched at an indirect control instruction site .... 41

3.3 Ratio of the number of conditional branch executions over number of in-
direct instruction executions for SPECINT ............................ 44

3.4 Regular branch prediction flow enhanced with security features. Adapted
from [38] (p.124) ............................................................ 46

3.5 The architecture support in processor pipeline for control-flow validation . 50

3.6 Hardware support for expected path validation. Adapted from [105] (Fig.
5). ................................................................. 51

3.7 Extending the full-path-vector from profiling for the BTB entry ........... 52

3.8 Memory overhead for the Full Record Set (FRS) ........................ 58
3.9 Performance degradation of our validation mechanism for the BTB configuration of 512-1-1.................. 59

3.10 Normalized execution time with different BTB size to the case for a BTB of 512-1-1........................................... 60

3.11 Normalized execution time with different set associativity to the case for a BTB of 512-1-1........................................... 60

3.12 Normalized execution time with different history associativity to the case of a BTB of 512-1-1................................. 61

4.1 Virtual address space and page table........................................... 68

4.2 Code transformations at compile-time........................................... 70

4.3 Example of code duplication........................................... 74

4.4 Function argument at execution time........................................... 75

4.5 PIFT, Architecture design for paged dynamic information flow tracking ........................................... 75

4.6 Micro-benchmark for stack buffer overflow........................................... 80

4.7 Defending against stack buffer overflow........................................... 81

4.8 Micro-benchmark for format string vulnerability........................................... 81

4.9 Micro-benchmark for heap corruption........................................... 82

4.10 Detail of heap corruption attack........................................... 83

4.11 A real attack, taken from [21]........................................... 84

4.12 Global variable overhead........................................... 85

4.13 Code overhead........................................... 86
4.14 Stack overhead .............................................................. 87
4.15 Execution cycle overhead .............................................. 88

5.1 Our architecture enhancement for metadata coherence enforcement ........ 94
5.2 E-MOESI Cache Coherence Protocol ..................................... 96
5.3 Block diagram of METACE ............................................... 98
5.4 METACE actions when a data access entry from a main core is logged .......... 99
5.5 METACE actions when a coprocessor requests metadata access ................ 101
5.6 The state of METACE table after core 1 and core 2 access data in Figure 2.2, with a RAW between events $t_{12}$ and $t_{21}$. ......................... 103
5.7 An example with WAR and RAW for out-of-order metadata access .............. 104
5.8 The state of METACE table after main core data accesses in Figure 5.7 are executed ................................................................. 105
5.9 Increase of the shared bus utilization along with number of cores ............... 109
5.10 Execution time overhead of different DIFT implementations running 32 threads on 32 application cores, normalized to the execution time without DIFT monitoring ............................................................. 110
5.11 Execution time overhead with out-of-order access processing on 1, 2, 4, 8, 16, and 32 application cores and the same number of coprocessors, normalized to the DIFT execution time without metadata coherence enforcement .............. 111
5.12 Overhead composition in-order ........................................... 113
5.13 Overhead composition out-of-order ...................................... 113
6.1 The architecture enhancement. In-core and in-memory Ownership Tables
store virtual addresses. .................................................. 122

6.2 Dynamic memory overhead .................................................. 128

6.3 Number of entries in the Ownership Table ...................................... 129

6.4 Execution overhead for handling the ownership table ......................... 130

6.5 Execution time overhead due to run-time ownership checking and validation 131
List of Tables

3.1 Vulnerable Programs ............................................. 53
3.2 Architecture Parameters for Simulations ...................... 56
3.3 Number of Ambiguities Found ................................. 56
3.4 Details of a BTB Entry ........................................... 57
4.1 Parameters for Simulations ................................. 87
4.2 Comparison between Different DIFT Techniques ............ 91
5.1 Details of an entry for an 8-core 32-bit architecture ........ 108
5.2 Comparison of area for different table sizes versus an L1 cache (64KB, 64B line, 4-way associativity) and a processor .................. 108
6.1 Memory Management Routines .......................... 121
6.2 Parsec benchmarks. Key characteristics [15] .............. 126
6.3 CPU and Memory hierarchy configuration ..................... 128
6.4 Comparison between different software-based approaches and HATI with a 16-entry in-core OT .................................. 133
6.5 Details of an entry for a 32-bit architecture ................ 133
6.6 Comparison between a 32 entries size table and an L1 cache (64KB, 64B line, 4-way associativity).
Chapter 1

Introduction

As networking connections become pervasive for computer systems and embedded software contents increase dramatically, it becomes more convenient for hostile parties to utilize software vulnerabilities to attack embedded systems, such as personal digital assistants (PDAs), cell phones, networked sensors, and automotive electronics [60]. The vulnerability of embedded systems carrying sensitive information to security attacks, ranging from common cyber-crimes to terrorism, has become a very critical problem with far-reaching financial and social implications [25]. For example, security is still the largest concern preventing the adoption of mobile commerce and secure messaging [74].

However, security has been misinterpreted as the addition of cryptographic engines and security protocols to the system, which results in low efficiency and more attack surfaces. Security should be considered as another dimension throughout the design process, along with traditional design metrics like area, performance, and power [74].

Compared to the general purpose and commodity desktop system, an embedded system presents advantages in allowing deployment of meaningful countermeasures across
the system design. However, building a secure embedded system is a complex task that requires multidisciplinary research across different system layers and spanning various design stages, including circuits, processors, operating systems (OS), compilers, system platforms, etc. It is especially challenging to find efficient solutions granting system immunity to a broad range of evolving attacks, considering the stringent constraints of embedded systems on computing capability, memory, battery power, and the tamper-prone insecure environment. The challenges unique to embedded systems require new security approaches to cover all aspects of embedded system design from architecture to implementation.

Our research on security in embedded systems is to establish both effective and efficient architectural support in embedded processors for secure processing. Typical software security attacks include buffer overflows, fault injections, Trojan horses, and data and program integrity attacks [13, 77, 102]. These attacks exploit system vulnerabilities to allow malicious users to overwrite program code or data structures, leaking critical information, or launching malicious code. Hardware-based mechanisms are less susceptible to attacks since they can be made transparent to upper-level software attacks. They can also minimize error propagation and enable fast error isolation and robust recovery, as they can be carried out at a finer granularity without much execution time overhead.

Our investigations of security enhancement can be divided into two areas: single core architectures and multi-core architectures. In the area of single core architectures, we are focused on security mechanisms inside the processor. First, how to leverage existing
CHAPTER 1. INTRODUCTION

speculative architectures to protect non-control data in an efficient manner. Second, how to integrate hardware augmentation and software support for dynamic information flow tracking (DIFT) to prevent memory corruption. In the area of multi-core, we address specific issues of parallel processing. First, how to maintain data and metadata consistency for DIFT in multi-core systems with shared memory. Second, how to ensure thread isolation to avoid memory corruption and information disclosure in multi-threaded programs.

This dissertation explores the design and implementation of hardware and software co-design security schemes that can provide efficient and effective protection from a wide variety of low-level memory and high-level semantic attacks.

1.1 Contributions

This dissertation focuses on integration of enhanced in-core modules like the branch table buffer (BTB) and the memory management unit (MMU), or additional hardware modules, the compiler to perform security monitoring, and the operating system (OS).

The main contributions of this dissertation are as follows:

• Proposing a novel approach for protecting program execution at the micro-architectural level. Our approach enhances the on-chip branch target buffer (BTB) to monitor control-flow transfers and execution paths. It increases the security level compared with similar solutions by selecting a better indirect control signature (ICS), which alleviates the ambiguity between different control-flow paths with similar binary branch histories. Our security mechanism is able to detect more kinds of attacks and
has higher detection efficiency for the same attack than other solutions.

- Presenting PIFT (Page-level dynamic Information Flow Tracking), a hardware/software co-design solution to perform dynamic information flow tracking (DIFT) based on compile-time static taint analysis and secure page allocation with minimal hardware changes. PIFT reduces the memory overhead significantly by aggregating data according to their taints and using only one taint bit in the page table for each page. PIFT can reach the same security level as hardware-assisted approaches, i.e., addressing self-modifying code, just-in-time (JIT) compilation, and third-party libraries. Distinct from software approaches that annotate the code and add extra code to be executed, PIFT only involves system software augmentation, including compilation passes and OS support, without re-programming every individual application. Therefore, the performance degradation of application execution is very small.

- Introducing METACE (METAdata Coherence Enforcement), a light-weight centralized micro-architectural approach, to keep data and metadata consistency in multicore systems with shared memory and metadata processing decoupled on coprocessors. Our approach slightly modifies the existing cache coherence hardware to ensure that all data dependencies (read-after-write, write-after-read, and write-after-write) are preserved on the corresponding metadata (that we define as M-RAW, M-WAR, and M-WAW). METACE is unintrusive to cores or coprocessors with minor changes in the on-chip shared interconnections. It can work with both in-order and out-of-order program execution, and is compatible with various memory consistency
models (sequential, relaxed). Our evaluation of METACE shows that it incurs small execution overhead.

- Presenting HATI, an efficient hardware assisted thread isolation mechanism. Our approach bears similarity with software thread isolation approaches in terms of setting the data sharing patterns by modifying system libraries as in [56]. However, the runtime data access monitoring and ownership checking is assisted by extra hardware, rather than by the compiler or the operating system as in [10, 56, 40]. This greatly reduces the overall performance degradation and enables it to be used for run-time checking.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives the motivation of our control-flow transfer and execution path validation with shared data. It then provides an overview of dynamic information flow tracking (DIFT), and discusses the different proposed implementation of DIFT. It explains the problem of inconsistency between data and metadata under decoupled processing in multi-threaded programs. It also presents the security implications of running different threads in the same address space with private data.

Chapter 3 presents details of our control-flow checking mechanism and architecture for run-time validation based on the BTB architecture. It presents the security analysis of our approach. It also evaluates the performance impact of our solution and compares our work with related work.
Chapter 4 describes design aspects and run-time details of PIFT. It includes security analysis and experimental results. It also discusses relevant issues about PIFT and compares to the related work on DIFT for low-level memory-corruption security attacks and summarize our contributions.

Chapter 5 describes METACE in details. It explains how METACE handles critical data dependencies. It examines the impact of our design on performance, area, and security of the system.

Chapter 6 presents the design and implementation of HATI, a hardware/software thread isolation mechanism. It shows the performance and area overheads of the design. It also studies the security capabilities of the architecture and demonstrates its effectiveness at preventing security attacks.

Finally, Chapter 7 concludes the dissertation and proposes future directions for research.
Chapter 2

Background and Motivation

Program execution can be tampered by attackers through exploitation of diverse sequential and concurrent vulnerabilities. A vulnerability allows an attacker to disclose or corrupt information, to take control or gain privilege over the system, or to change the normal execution of the program.

Vulnerabilities are classified in different types [100]. The most exploited vulnerabilities are buffer overflow, integer overflow, and format string vulnerabilities [37, 48]. These vulnerabilities are present in programs written in C/C++.

Many software and hardware approaches have been presented to validate the integrity of data and check program execution. However, software approaches suffer large performance degradation that limits their application on run-time system. Hardware approaches reduce the performance penalty but current solutions imply high logic and storage overhead.

Our research on security in embedded systems is to establish both effective and efficient architectural support in embedded processors for secure processing. We propose to
augment different system layers, i.e., hardware architecture, compiler, and operating system (OS), to provide efficient monitoring with the same or better level of protection than current solutions.

In this chapter, we present the issues with the current solutions, their advantages and disadvantages, as well as the key points that motivate our work. The rest of this chapter is organized as follows. Section 2.1 describes the characteristics of control and non-control data attacks. Section 2.2 introduces the dynamic information flow tracking (DIFT) technique, and provides a thorough overview of the different methods of implementing information flow tracking. We also explain the problem of inconsistency between data and metadata when processing multi-threaded programs. Finally, Section 2.3 introduces the security implications of running different threads in the same address space.

2.1 Defeating Control and Non-Control Data Attacks

Reports from [1] show that control data attacks are the most dominant and critical security threats, which compromise control-flow transfers by altering the target address such as procedure call, return function, local jump/branch, and special non-local jump (setjmp/longjmp). When a control-flow attack is inflicted on a system, the program execution can be directed to either malicious code injected by the attacker or some existing code that would not otherwise execute at that moment. It is the execution of the compromised control instructions that deviate from its expected behavior.

Since control data attacks still follow the instructions’ semantic without explicitly vi-
olating any security policies, traditional measures such as code/data integrity checking or encryption/decryption alone cannot prevent it. Recently, there are some mechanisms presented to countermeasure control data attacks. Software-based approaches, such as StackGuard, StackGhost, and RAD [16, 22, 26, 35, 70], have shown effective against low-level corruption memory attacks. However, these approaches cause a significant performance penalty and normally require the use of standard dynamic libraries and the modification of the source code. On the other hand, hardware-based schemes are either based on observations of violation symptoms [52, 67, 90, 105] or tracking control source data [86, 27, 20, 92, 79, 28]. Their effectiveness, however, is often limited by inaccurate information being monitored or large memory/performance overhead.

Recently, some exploits have emerged that can change the direction of control instructions by overwriting local variables instead of changing the target address [32, 105, 82]. It is called decision data attack, where the attacker overwrites the data that determines whether to jump or fall through at a branch site. Although the control transfer (jump site and destination) is valid, the global control-flow is compromised. Thus, the attackers can extract important system information and take control of the application. This kind of attack is hard to detect by control-flow transfer validation only or tracking control source data, because a decision data attack does not violate “where to go” of control-flow (jumps from one site to another), but changes the “when” (schedule of control-flow transfers) by modifying the local variables which determine the jump decisions.

Control-flow validation approaches are techniques based on observation of violation
symptoms. There are several issues in program execution that complicate the validation process of control-flow transfers and execution paths, e.g., multiple-path situations, as shown in the example control-flow graph in Figure 2.1, where each node represents a basic block, and each edge is a control-flow transfer. Here, basic block \( P \) ends with an indirect instruction that may go to different target addresses for taken branches. There are multiple paths of basic blocks that lead to block \( P \) (see column 2), but not necessarily lead to block \( Z \). Without proper handling, they may result in ambiguities that will increase the false negative rate and degrade the effectiveness of the security countermeasures.

![Complex program control-flow with ambiguity](image.png)

**Figure 2.1:** Complex program control-flow with ambiguity.

For direct conditional branches, a program control-flow can be validated by using only binary decision information (where 1 represents a branch taken and 0 branch untaken - see column 3). However, when indirect branches are present \((P \rightarrow Y \text{ or } P \rightarrow Z)\), valida-
tion has to be performed against the address path (see column 2), which requires a lot of memory storage [105]. An alternative solution is to look backward at the history of branch decisions at indirect branches, and use a binary decision path (see column 3) [82]. Nevertheless, an attacker can take control of the application and change the normal control-flow by exploiting the ambiguity of decision history representation.

For example, in Figure 2.1, there are two paths that lead to basic block \( P \), which is the end of a super block and the beginning of the next one. Let us assume the path of \( A \rightarrow B \rightarrow C \rightarrow P \rightarrow Z \) is the correct, and the other path of \( J \rightarrow K \rightarrow L \rightarrow P \rightarrow Z \) is the infeasible. Their binary history paths are the same: 1-1-1. If an intrusion detection system (IDS) just uses the binary history path information, the incorrect path (\( J \rightarrow K \rightarrow L \rightarrow P \rightarrow Z \)) will escape the checking. Therefore, the decision history path is insufficient to validate execution. To prevent control-flow attacks, the IDS must validate both control-flow transfers and execution paths. In Chapter 3, we will show how to address this problem and other related issues.

### 2.2 Dynamic Information Flow Tracking Technique

Dynamic information flow tracking (DIFT) has been used to monitor the run-time behavior of applications, and has shown to defend against both high-level semantic attacks [28] and low-level memory corruption attacks [86, 58, 27, 65, 78, 20] effectively. The basic idea is to taint data inputs from untrusted sources with a tag (also called metadata), like network data and keyboard input, propagate the tag while data is being processed in the processor
core, and check the usage of tainted data in program’s important sites, like control-flow transfers and system calls. There has been many DIFT implementations in software (at both static and run time) [54, 99, 71, 50], and hardware [79, 20, 27, 92, 86].

In general, software-based approaches are more flexible with taint propagation and checking policies. However, static software approaches incur large code (memory) overhead and performance degradation and could not handle cases like self-modifying code, just-in-time (JIT) compilation, and multi-thread applications [93]. Dynamic software approaches support dynamic code generation, but the performance overhead due to the instrumentation limits their applicability.

On the other hand, hardware-based approaches address these drawbacks and reduce the performance degradation, but require a drastic redesign of the processor core for taint storage and processing (including propagation and checking). They are often limited by certain pre-set rules to avoid false alarms and cannot neutralize new emerging attacks. Most of the hardware-assisted DIFT schemes “couple” the taint storage tightly with the data. This requires drastic changes to both the processor logic and the register file, caches, memories, and busses for tag processing, storage, and propagating [86, 27, 28]. Such invasive implementations impose high design cost and incur hardware overhead. To reduce the implementation complexity, recent approaches have “decoupled” the metadata processing from data processing, leveraging the prevailing multi-core architecture or utilizing tightly “coupled” coprocessors [46, 29, 93, 94, 80, 106]. In this way, only minor changes are needed outside cores to synchronize data processing and metadata processing, conducted
at different places.

In both software and hardware approaches, the main sources of overhead are accessing and storing the metadata information. Chapter 4 shows how we address this problem and other related issues.

2.2.1 Dynamic Information Flow Tracking on Multi-Threaded Programs with Shared Memory

Metadata processing and its synchronization with data processing is an important factor that affects both the security effectiveness and the performance of DIFT implementations. When multi-threaded programs run on multi-core systems with shared memory, coherence issues may arise with decoupled metadata processing [93, 45]. Figure 2.2 shows an example of consistent data and metadata access [45]. Each application core (processing data) is associated with a coprocessor, which processes metadata. Memory-accessing events are shown on the individual time line of the main core or coprocessor. Data accessing order is defined by the programmer and ensured by the compiler. Decoupled metadata processing on a coprocessor is always later than the corresponding data processing on a main core, ensured either at static-time by the compiler or at run-time by the inter-core communications through first-in-first-out (FIFO) buffers or pipes. However, since the metadata processing on coprocessor 1 is independent of that on coprocessor 2, event $t_{41}$ may appear earlier than $t_{32}$. Therefore, coprocessor 2 might get a stale value of $\text{tag}(U)$. This incorrect metadata value would affect the effectiveness of DIFT significantly, incurring false alarms or false negatives.
A solution to avoid such inconsistency between data access and metadata access is to use transactional memory to institute atomicity between data and metadata [23, 62]. However, the transaction overhead limits its usability on modern architectures. Another solution is presented in [93] where the processor delays committing the data processing instruction until the extra hardware processes the metadata. This approach is applicable to “in-core” DIFT implementations, but would result in significant delay on “decoupled” implementations. Recently, one approach enforces access of metadata in exactly the same order that data is accessed [95]. However, this causes delay in execution of unrelated memory accesses. Another approach is to implement hardware locks for metadata access [45]. Each application core sets a lock on the corresponding metadata when it has updated the data, and the coprocessor releases the lock when the metadata update is done. Any other access to the locked metadata is stalled during the process. This approach is distributed and requires additional hardware on each coprocessor for the lock and holding memory.
accesses. In addition, the program execution would be slowed down significantly due to
extra traffic on the bus for enforcing metadata coherency. These solutions remove the co-
herence problem by enforcing metadata order processing. However, they are losing the
opportunity to process unrelated metadata access out-of-order. Chapter 5 shows how we
address these problems and provide an efficient solution.

2.3 Thread Isolation

The ubiquitous adoption of multi-core architectures in various computer systems and in-
frastructures has significantly accelerated the development of concurrent programs. How-
ever, concurrent programs imply a more complex programming model over the traditional
sequential model. For example, in the shared memory programming model, the execu-
tion of parallel threads may interfere each other’s data structures, either unintentionally by
buggy programs with thread synchronization errors or deliberately by malicious attackers
who utilize program vulnerabilities. This would negatively affect the privacy and security
of web servers, application servers, web browsers, and many other systems that require
the capacity to execute subcomponents in isolation from each other. The common reason
for such thread-interfering bugs and attacks is the shared address space for multi-threaded
programs. Adversaries can force a vulnerable thread to access or even modify the private
data of other threads, as there in no thread boundary checking. Buffer overflow, string
vulnerabilities, integer errors, and other vulnerabilities can all be exploited by a malicious
thread to gain control over other threads and inflict an attack.
CHAPTER 2. BACKGROUND AND MOTIVATION

One solution to prevent such cross-thread attacks is to spawn processes in concurrent programming rather than threads. Currently process isolation for program subcomponents has been broadly adopted in web browsers, like IE, Google Chrome, and Firefox, for tabbing. Explicit isolation between processes is supported by the operating system. Figure 2.3 (a) and (b) demonstrate the address space for multi-thread and multi-process, respectively. Having multiple address spaces would increase the memory footprint of the program, and incur expensive context switches. In addition, the inter-process communication (IPC) is complex and will slow down program execution. To alleviate the large resource overhead of multi-process programs, read-only pages can be shared among processes, as shown in Figure 2.3 (c). Only when a process needs to modify a data page, a new copy is allocated in its own private data segment, allowing other processes to use the unmodified shared version. When multiple processes access shared data, similar synchronization mechanisms, as shown in Figure 2.3 (b), must be used to avoid race conditions and inconsistency issues.

(a) Multi-thread without isolation, running in the same address space.
(b) Process isolation, each sub-component (thread) is encapsulated in a process and runs in a different address space.
(c) Process isolation with shared pages.

Figure 2.3: Concurrent program examples with multiple subcomponents.

For concurrent programs, Figure 2.3 (a) is both clear and potentially faster. It gives bet-
ter responsiveness and scalability. However, achieving proper and efficient thread isolation is very challenging in multi-threaded programs. The main issue is the lack of explicit program constructs to identify what data objects belong to a specific thread. Current thread isolation approaches only target *unintended sharing*, i.e., data being shared among threads in ways that the programmer does not expect [10, 56]. They focus on avoiding concurrent programming errors like data races, and the tools are mainly for program debugging. We aim to address the issue of *intentional cross-thread attacks*, which exploit vulnerabilities of multi-threaded programs and compromise integrity and confidentiality of other threads. Chapter 6 shows how we address this problem.
Chapter 3

Leveraging Speculative Architectures for Run-time Program Validation

Program execution can be tampered by malicious attackers through exploiting software vulnerabilities. Changing the program behavior by compromising control data and decision data has become the most serious threat in computer system security. Although several hardware approaches have been presented to validate program execution, they either incur great hardware overhead or introduce false alarms. We propose a new hardware-based approach by leveraging the existing speculative architectures for run-time program validation. The on-chip branch target buffer (BTB) is utilized as a cache of the legitimate control flow transfers stored in a secure memory region. In addition, the BTB is extended to store the correct program path information. At each indirect branch site, the BTB is used to validate the decision history of previous conditional branches and monitor the following execution path at run-time. Implementation of this approach is transparent to the upper operating system and programs. Thus, it is applicable to legacy code. Because of good code locality of the executable programs and effectiveness of branch prediction, the frequency
of control-flow validations against the secure off-chip memory is low. Our experimental results show a negligible performance penalty and small storage overhead.

The remainder of the chapter is organized as follows. Section 3.1 presents details of the proposed mechanism and architecture for run-time validation based on the branch target buffer (BTB) architecture. Section 3.2 presents the security analysis of our approach. Section 3.3 evaluates the performance impact of our approach. Section 3.4 compares our work with related work and summarizes our contributions.

3.1 Speculative Architecture for Control-Flow Transfer and Execution Path Validation

In Section 2.1, we show that a program control-flow can be validated by using only binary decision information (where 1 represents a branch taken and 0 branch untaken). However, when indirect branches are present, validation has to be performed against the address path, which requires a lot of memory storage [105]. An alternative solution is to look backward at the history of branch decisions at indirect branches, and use a binary decision path [82]. Nevertheless, an attacker can take control of the application and change the normal control-flow by exploiting the ambiguity of decision history representation.

We propose a hybrid solution for the above situation: each history path has to be associated with something that provides enough information about the history, and at the same time, allows an observation window as large as possible. We use the past branch site address (PBPC) in the history [57]. It allows us to differentiate the correct path from the incorrect ones. We also consider the number of basic blocks in the current super block.
CHAPTER 3. LEVERAGING SPECULATIVE ARCHITECTURES

With this size information, it is more difficult for the adversary to camouflage the attack. As shown in the right column of the table in Figure 3.1, the new path representation for basic block $P$ associated with target address (TPC) of $Z$ is $\text{size-C-1-1-1}$; i.e., the size of the super block, the last branch site address, and the direction history path.

![Diagram of program control-flow with ambiguity resolution](image)

<table>
<thead>
<tr>
<th>Address Path [32 x n bits]</th>
<th>Branch History [n bits]</th>
<th>Our Approach [(6+32+n) bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Path</td>
<td>A-B-C-P</td>
<td>1-1-1</td>
</tr>
<tr>
<td>Incorrect Path</td>
<td>J-K-L-P</td>
<td>1-1-1</td>
</tr>
</tbody>
</table>

Figure 3.1: Complex program control-flow with ambiguity resolution. Figure 2.1 has been modified to show the changes.

In addition to the binary history, information about future expected paths is used to check the decision of basic blocks in the next super block. We get an all-path-vector for the current target address (TPC) for a number of levels or until the program execution reaches the end of the super block. Figure 3.2 shows one example of an expected path vector for a depth of 2. Basic block $A$ is the end of last super block. One of the taken branches of $A$ goes to $B$, and $B$ will be the root of a binary tree. Because all the basic blocks between $B$ and next super block are direct conditional branches, their directions are...
sufficient for path validation. With a depth of 2, there are four paths in total: $B - C - E$ (with a decision path of 11), $B - C - F$ (10), $B - D - G$ (01), and $B - D - H$ (00). We use a 4-bit vector to represent the validity of each path. For example, if path $B - D - G$ (decision history path is 01) is invalid, and all the other three paths are valid, the vector will be 1101. For depth $n$, the vector size is $2^n$ bits, where each bit represents the validity of its corresponding path. Therefore, the vector size is much smaller than the maximum size of $2^n \times n$ bits for recording all the possible n-level paths.

![Figure 3.2: Expected paths vector fetched at an indirect control instruction site. The full-path-vector 1101 shows that path $B - D - G$ is incorrect.](image)

The expected path vector (EPV) is used during program execution to validate on the fly, making possible faster detection of attacks. Whenever a conditional branch is encountered and resolved, its direction decision (taken or no taken, i.e., 1 or 0) is used to cut the vector in half, with the higher half kept for decision 1 or the lower half for 0. If the vector reduces
to 0 (all elements are zero), an invalid path has been executed, and the anomaly is detected. For example, in Figure 3.2, the 2-level full-path-vector of node B is 1101. We assume the program is executing path B → D → G. When the control-flow transfer of B → D is resolved, the 0 decision keeps the lower half of the vector, 01. When the program proceeds to the next level, D → G transfer is taken (decision is 1), and the higher half of 01, 0, is kept. As 0 is detected, the execution of the invalid path B → D → G is caught by the monitoring mechanism.

In this Chapter, we have introduced the main aspect of our approach. In the rest of the Chapter, we show how to get the signature of the program, what is the size of the signature, how is done the process of reading and updating the signatures, and what are the hardware changes needed to support efficient and effective control-flow monitoring.

3.1.1 Training the Full Record Set (FRS)

The full record set (FRS) represents the normal behavior of a program, which theoretically contains all the legitimate indirect control transfers and execution paths. There are two ways to collect the records [81]. We can either extract the normal behavior through static analysis of the legacy code, or perform training, as many model-based approaches have done [105]. By running the applications in a certain secure environment with a lot of test data, the processor can regard all seen control-flow transfers and execution paths as normal ones. If the FRS does not completely cover the normal behavior, using it as a reference to validate program execution at run-time will incur false alarms (false positives). Previous work has shown that the total number of control-flow transfers actually converges quickly.
CHAPTER 3. LEVERAGING SPECULATIVE ARCHITECTURES

as the number of indirect control instruction increases [81].

3.1.2 Size of Signature

As our approach is sampling the program execution by checking only at indirect control instruction sites, it is important to make each checking cover a broad range of program execution and to improve the detection rate of invalid control-flow transfers and anomalous execution paths. For each super block, the execution path is first validated against the expected path vector along the execution. Then at the end of the super block, the dynamic branch decision history is validated against the history paths previously characterized. Both the expected path and history path should be long enough to cover most of the super block. Since the size of a super block varies greatly in programs, we examine the average size to determine the appropriate path length. Based on our profiling study, Figure 3.3 shows the ratio of the number of conditional branch executions over the number of indirect control instruction executions for several SPECINT benchmarks. The ratio varies greatly with applications, but for most benchmarks, there is one indirect instruction executed for every 5-6 conditional branches. Therefore, we have set the depth of expected path vector to be 6 and, the length of history paths to be 14 conditional branches, so that the total length is 20 levels.

3.1.3 Extraction of Dynamic Program Execution Information for Validation

For every indirect control instruction, the system has to validate its legitimacy. Since the full record set (FRS) is stored in a secure memory region, frequent off-chip full record
access for validation will degrade the program performance greatly. To reduce the performance penalty, we can either reduce the latency for accessing the full record or decrease the number of accesses. Previous work has employed a Bloom Filter scheme to reduce the off-chip hardware access latency to four cycles [81]. We focus on reducing the access frequency by leveraging the speculative architecture of branch target buffer (BTB).

When validating program execution, the system needs to collect dynamic information on the history of conditional branch directions (the last branch address and the size of the super block) to compare against the normal behavior obtained from static-time analysis or training. We dedicate three registers for dynamic information, which can only be accessed by the validation system: the branch history shift register (BHSR), the past branch program counter (PBPC), and the size of the current super block (SIZE). The \textit{branch history shift register} (BHSR) [82] is a shift register that stores the decisions of the last $n$ conditional instructions (assume the length of BHSR is $n$). It is updated after each direction is resolved.
Another register, the past branch program counter (PBPC), records the branch address of the last basic block. The last register, for the size of the current super block, counts the number of basic blocks belonging to the current super block.

At each indirect instruction site, the branch site (BPC) is used to look up the expected behavior either in the BTB or in the FRS. If neither a BTB entry nor an indirect control signature (ICS)\(^1\) in the FRS is found that matches the tuple of \{BPC, TPC, BHSR, PBPC, SIZE\}, the target address or history are invalid. An alarm is raised for the operating system to take control over the program. Otherwise, there is match, and the expected path vector is fetched to check the following basic blocks along the execution. Another alarm may be raised at any time within the next super block if the vector is reduced to 0.

### 3.1.4 BTB Update and Administration

Branch prediction mechanisms have been widely adopted for high-end embedded processors to alleviate pipeline stalls caused by conditional instructions. Predicting the branch direction while caching the branch target address has been the fundamental objective in many designs of efficient pipelined architectures. The branch target address is provided by the BTB, which has a cache structure consisting of a number of sets with the set associativity range from one to eight ways [68]. The upper part of the BPC is used as a tag to access the BTB.

\(^1\)Each indirect control signature (ICS) consists of one branch site (BPC) as a tag, one target site (TPC), multiples history paths (HPs) with the last branch address, the size (SIZE) of the super block, and the vector for the expected paths (EPV).
pipeline architecture. The solid objects and lines are for the regular branch prediction scheme. The flow is also extended with some enhancements for control-flow transfer and execution path validation, as shown in dashed lines and objects, which will replace the original operations.

![Flowchart](image)

**Figure 3.4:** Regular branch prediction flow enhanced with security features. Adapted from [38] (p.124)

When an instruction is fetched from the instruction memory (in *IF* stage), the same PC address is used to access the BTB. A hit at this point indicates that the instruction to be executed is a control instruction. The predicted target PC (TPC) in BTB is then used as the next PC (NPC) to fetch a new instruction in the next pipeline stage (*ID*), rather than
waiting until the later stage to use the NPC computed in \textit{EX} stage for fetching instruction, avoiding branch stalls. Meanwhile, according to the instruction type, a direction prediction or computation is performed. If the branch is computed to be taken (\textit{direction hit}), the TPC from the BTB will be compared with the computed NPC in the \textit{EX} stage. If these two values match (\textit{address hit}), both the branch direction and the target prediction are correct. However, one more validation has to be done for indirect control instructions in our approach.

In execution stage, the system has to compare the dynamic BHSR, PBPC, and SIZE against the ICS in the BTB entry. If the history matches as well, it is a \textit{history hit}, and the program execution will continue as normal. Otherwise, this is a \textit{history mis-prediction}: the history paths associated with the TPC in the BTB do not include the history just seen. The tuple of \{BPC, computed NPC, BHSR, PBPC, SIZE\} has to be sent to the external memory (FRS) for further validation. This is labeled 3 in Figure 3.4. Only when it misses again, a security alarm will be raised.

In traditional architecture, on an \textit{address mis-prediction} site, the BTB entry is just updated when the next PC is resolved from the instruction. However, since an indirect target address mis-prediction may also be caused by security attacks, in our enhanced architecture, the external memory has to be checked before the corresponding entry in the BTB is updated with the matched ICS. This site is labeled 2 in Figure 3.4. At a \textit{direction mis-prediction} site where there is a BTB entry for the instruction but the instruction is actually not taken, the entry is deleted and the fetched TPC is squashed. There is normally
a mis-prediction penalty for these remedy actions.

On the leftmost side of the flow diagram, if no entry is found in the BTB for the current PC, the instruction may be a regular data path instruction or a control instruction falling through. In the subsequent ID stage, if the instruction is found to be a taken control instruction, it is a direction mis-prediction, and the address is not cached in the BTB. Before a new entry is inserted to the BTB, the tuple of \{BPC, computed NPC, BHSR, PBPC, SIZE\} has to be validated against the record in memory. This site is labeled 1 in Figure 3.4. Since the BTB has limited capacity, a replacement policy, e.g., least recently used (LRU), may be employed to find a victim to evict for multi-associativity BTB.

A lot of studies have been presented on improving prediction accuracy to reduce the mis-prediction penalty and thus performance degradation [51, 89, 44]. We observe that for regular speculative architecture, the BTB has served as a target address cache for some of the taken branch instructions, and mis-prediction just affects performance. To turn the BTB into a cache for the full record set, we have to extend it first to include the path information as well. More importantly, we have to ensure its integrity, i.e., guarantee the BTB is free of corruption from the external memory. Thus, when we use the BTB as the reference to validate control-flow transfers and execution paths at run-time, on any mis-prediction site, including direction, target address, and history path, we have to look up the external full record for the current BPC before the BTB is updated.

Note that the BTB access time is still the same as in a standard architecture. As shown in Figure 3.4, the extra history path checking (labeled as 3 in Figure 3.4) only occurs
in the execution stage of pipeline after the TPC is compared with the computed NPC. Comparing the “history path” in the BTB entry with the run-time execution path is just a combinational operation. It can be done in parallel with the common direction prediction checking (labeled as 2 in Figure 3.4) without increasing the BTB access time or stalling the pipeline. Therefore, the execution overhead is only due to access the FRS each time a miss-prediction appears.

### 3.1.5 Architecture Support for Control-Flow Validation

Accessing the full record set should be reduced to the minimum to lower the performance degradation. Thus, a clear distinction has to be made between instructions that will point to a safe target address and the instructions that definitely require validation against the external FRS. Assuming that code integrity checking mechanisms, e.g., [31, 53], have been applied, and based on the analysis in Section 2.1, we can regard direct control instructions as always generating correct target addresses because the addresses are encoded in the instruction, and the instruction bits have been ensured correct. Hence, on BTB mis-predictions, these instructions can directly update the BTB with the computed next target address without validating against the full record set. Note that only the target address is needed for these BTB entries. In addition, the full record set in memory does not need to keep any information for direct control instructions. Only those indirect control instructions that are mis-predicted by the BTB will possibly incur full record lookups.

Figure 3.4 shows that the tuple of \{BPC, computed NPC, BHSR, PBPC, SIZE\} is sent to the full record set for validation at three sites (labeled 1, 2, 3), which represent:
direction mis-prediction, address mis-prediction, and history mis-prediction, respectively. Figure 3.5 illustrates the overview of our architecture support in a processor pipeline for control-flow validation. For every indirect branch instruction, the BPC is sent to the BTB during the fetch stage. After the execution stage, the dynamic information extracted, including next target PC (NTPC), BHSR, PBPC, and SIZE, is also sent to the BTB for validation. On any mis-prediction, the full record set (FRS) in the secure memory is accessed, and the corresponding indirect control signature (ICS) is brought into the processor. With the expected path vector fetched into the processor pipeline, the program execution is monitored at run-time on a basic block basis.

Figure 3.5: The architecture support in processor pipeline for control-flow validation

It is worth noting that by monitoring the expected path at run-time our approach can detect anomalies as soon it occurs. A hardware extension, similar to the one presented in [105], is added to monitor the execution. Figure 3.6 depicts the hardware. Each time the
branch direction is resolved, the multiplexer logic select the correct half, and a NOR gate will detect whether the selected half is zero or not. The total logic size will depend on how many levels we are monitoring.

Our BTB entry has a fixed size for the sake of simplicity. Consequently, the lengths of the history paths and expected path vector in the BTB entry are also pre-defined. However, due to the varying size of super blocks, the actual length that can be extracted dynamically is not fixed. We have a register to record the number of basic blocks in a super block, and the system uses it to mask the comparison between the dynamic BHSR and the static HPs in the BTB. The number of bits in each BTB entry for an expected path vector is $2^n$ if the preset number of levels is $n$. In the case the depth of a super block ($m$) is greater than the number of level ($m < n$), the signature is truncated. On the other hand, if the depth of a super block is $m$, and $m < n$, we only get an all-path-vector of $2^m$ bits. The vector is extended by duplicating each bit $2^{(n-m)}$ times, in order to be put in a BTB entry for run-time validation. Figure 3.7 shows an example where $n=4$ and $m=2$, thus, each bit

![Figure 3.6: Hardware support for expected path validation. Adapted from [105] (Fig. 5).](image-url)
in the vector obtained from training is duplicated $2^{(n-m)} = 2^2 = 4$ times to compose the expected path vector (EPV) in a BTB entry.

\[
\begin{align*}
\text{Profiling EP} &= 1010 \\
\text{BTB EPV} &= 1111000011110000
\end{align*}
\]

Figure 3.7: Extending the full-path-vector from profiling for the BTB entry

### 3.2 Security Analysis

There are two issues in evaluating the effectiveness of our approach. First, the approach should not produce false positives when no attack is introduced upon the system. Second, the approach should be able to detect as many attacks that attempt to exploit a software vulnerability as possible, i.e., the false negative rate should be low.

We assume that the profile for the FRS is complete, and hence the false positive rate is zero (or near zero). In order to evaluate the false positive rate of our approach, we run a set of applications from MiBench. We profile all possible options in each case. Then, we feed the application with a different inputs and check the results. The system does not raise an alert when there is no attack.

To evaluate the effectiveness of the attack detection, we test buffer overflow and format string attacks developed by John Wilander [97]. The buffer overflow testbench includes 20 different attacks that cover practically all possible buffer overflow attacks. Our approach detects all of them. The buffer overflow attacks are caught when the function \texttt{strlen} is
determining the length of the input. When the size of the input exceeds the size intended, our monitor generates a different dynamic signature (different PBPC and different history) that is not present in the FRS. Similarly, when a string format vulnerability is exploited, an unrecorded signature appears and the attack is caught.

In addition, we test real programs with well-known vulnerabilities: Polymorph (buffer overflow), Sendmail (integer errors), SSH server (integer errors), Traceroute (double free), and WU-FTPD (format string). Table 3.1 shows details of the attack capture sites. In all cases, the attacks are detected. The results indicate that our approach is effective in detecting different kinds of memory corruption attacks.

**Table 3.1: Vulnerable Programs**

<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability</th>
<th>Site Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polymorph</td>
<td>Buffer overflow</td>
<td>Different target address (TPC)</td>
</tr>
<tr>
<td>Sendmail</td>
<td>Integer errors</td>
<td>Unknown signature at prescan function</td>
</tr>
<tr>
<td>SSH server</td>
<td>Integer errors</td>
<td>Unknown signature at do_authentication function</td>
</tr>
<tr>
<td>Traceroute</td>
<td>Double free</td>
<td>Different past branch site (PBPC) at getenv function</td>
</tr>
<tr>
<td>WU-FTPD</td>
<td>Format string</td>
<td>Unknown signature at vfprintf function</td>
</tr>
</tbody>
</table>

### 3.2.1 Attack Detection Analysis

For control data attacks, our approach catches the attack when the tuple \{BPC, TPC\} is not found in the FRS as in Polymorph. This misbehavior will also be detected in a traditional control-flow monitor. However, when the attack does not overwrite a control data, but a decision data like in SSH, control-flow monitors as in [11, 55] cannot detect the anomaly by just validating the individual transitions without context (timing) information.
Control path monitors as [105, 81] can detect the aforementioned attacks. However, ambiguities in binary execution path representation may make different paths to be taken as one. As a result, attacks may escape detection like in Traceroute.

Our proposed system represents the program control-flow with many legitimate execution episodes, each of which includes an indirect jump transition and correct execution paths before and after the transition point. In this way, the context information in the control-flow is well maintained and enforced.

3.2.2 Speed of Detection

Another aspect is how fast the attack can be detected. The granularity of our control-flow validation is at the basic block level, i.e., on average 10-100 instructions. Then, our approach is faster than traditional control-flow monitors on detecting buffer overflow attacks like in Polymorph. In [11, 55], the attack will be detected when the program execution transfers a corrupted target address. However, since our monitor includes both future path prediction and history path for each indirect transition site, we may detect control-flow errors during execution earlier than the transition point.

For example, for a more sophisticated attack as shown in Sendmail, the attacker utilizes several bugs in the program to corrupt local variables without altering the control data. The detection time depends on how the modified data is used and how the attack changes the program’ signature. In this case, our approach detects the attack after a few instructions, when the execution path and the size of the super block do not match with any entry in the FRS.
3.2.3 Security of the FRS

There are several locations where the FRS can be altered: in the main storage (e.g., hard disk), during loading from the disk to memory, and in the main memory. We assume that the attacker cannot alter the FRS in the first two places. This is a reasonable assumption, and is commonly used by other approaches. To avoid overwriting the FRS in memory by the attacker (or the user), a dedicated memory region is allocated as read-only. Any attempts to overwrite the FRS will be denied, and an alarm will be activated.

3.3 Experimental Results

To assess the performance impact of our approach, we test a set of SPECINT applications. We modified the SimpleScalar Toolset [12] that models an in-order single-issue processor to profile a set of MiBench applications [36], and to determine how common the ambiguities for binary path representation are.

The architecture parameters for the simulator are listed in Table 3.2. We consider extra delays for accessing the full record set for validation when the branch prediction unit is enhanced with security features.

3.3.1 Ambiguity Alleviation

We define the path ambiguity as two or more different execution paths that have identical branch site (BPC), same target (TPC), and same binary history (BH), but different past branch sites (PBPC). Our profiling shows that several legal paths can share the same tuple of \{BPC, TPC, BH\}. The ambiguity results are shown in Table 3.3 for a set of applica-
Table 3.2: Architecture Parameters for Simulations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTB</td>
<td>512 sets, n-way set assoc.</td>
</tr>
<tr>
<td>Return address stack</td>
<td>8 entries</td>
</tr>
<tr>
<td>Branch miss penalty</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Bimod</td>
</tr>
<tr>
<td>Full record set acc. latency</td>
<td>50/100/150/200 cycles</td>
</tr>
<tr>
<td>Fetch/dispatch/issue width</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>5</td>
</tr>
<tr>
<td>Load/store queue</td>
<td>8 entries</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>18 cycles</td>
</tr>
</tbody>
</table>

From the MiBench suit. To handle the ambiguities and reduce the possibility for a successful attack, we include the past branch site (PBPC) and also the number of basic blocks in the super block (Size). With this information, we solve all the ambiguities for the tested applications. It is possible for a more complex application to have different paths that share the same signature \{BPC, TPC, BHSR, PBPC, SIZE\}. However, it will be much more difficult for an adversary to camouflage the attack.

Table 3.3: Number of Ambiguities Found

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>12bits of History</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitcount</td>
<td>4</td>
</tr>
<tr>
<td>dijkstra</td>
<td>8</td>
</tr>
<tr>
<td>patricia</td>
<td>4</td>
</tr>
<tr>
<td>qsort-large</td>
<td>15</td>
</tr>
<tr>
<td>qsort-short</td>
<td>22</td>
</tr>
<tr>
<td>susan-corner</td>
<td>16</td>
</tr>
<tr>
<td>susan-edge</td>
<td>5</td>
</tr>
<tr>
<td>susan-smoth</td>
<td>1</td>
</tr>
</tbody>
</table>
3.3.2 Storage Overhead

Compared to the conventional BTB, our enhanced BTB contains reference path information as well. For the length that we have chosen, i.e., 6 for the expected path vector and 14 for the history path, each BTB entry will be 182 bits long, 118 bits more than the traditional structure. Table 3.4 shows the details of a BTB entry for a 32-bit architecture, where the first two fields are the common ones in conventional BTB, and the rest ones are the extension for control-flow monitoring.

Table 3.4: Details of a BTB Entry.

| 32-bit BPC | 32-bit TPC | 8-bit superblock size | 32-bit PBPC | 14-bit BHSR | 64-bit EPV |

As an FRS is needed in a secure memory region, the memory footprint for the programs will be increased as well. Figure 3.8 shows the results for a set of SPECINT benchmarks, where for each application, the first bar represents the original program code size, and the second bar represents the memory overhead. We can see that the memory overhead for most of the benchmarks is quite small, a few kilo bytes, except for gcc, which requires 250KB. This is due to the large number of tuples of \{BPC, TPC, HPs\}. The average ratio of memory overhead over the program code size is 3.79%. The overhead is three times smaller than dynamic information flow tracking approaches like [86, 28] that use one bit tag per one byte memory (12.5%), and four times better than control-flow monitors like [55] that monitors hashed patterns.

Our approach does not change the size of the program, while in software approaches
like [11], the program is enhanced with hashes for intra- and inter-procedural control-flow transfers.

### 3.3.3 Performance Impact of the Run-Time Validation Mechanism

We examine the performance impact of our run-time validation mechanism. Our BTB configuration contains three parameters, i.e., the number of sets, the set associativity, and the number of history paths in each entry (the path associativity). Figure 3.9 demonstrates the resulting performance degradation for a set of SPECINT benchmarks with the BTB configuration of 512 sets, direct map, and one history path in each BTB entry. Looking up the full record set in memory to validate mis-predicted indirect instruction targets has resulted in negligible performance degradation. Even with the full record memory access time set at 200 cycles (more than 10 times the latency to access the main memory), for most of the benchmarks, the execution cycle overhead is 4.82% on average. For application gcc, the overhead is noticeable - up to 24.13%.
We examine the impact of each BTB parameter on the performance. We change the number of sets but keeping both the set and history path associativities as one. Figure 3.10 shows the simulation results for BTBs with the number of entries as 512, 1024, 2048, and 4096. The execution cycle overheads are normalized to the case for a BTB with 512 sets. For benchmarks gzip and mcf, the overhead cycles actually increase as the capacity of BTB increases. The result seems counter-intuitive, since the performance should improve as the size increases for a conventional BTB architecture [68]. However, since our enhanced BTB is used not only for control-flow transfer validation (TPC), but also for path validation, there are more memory accesses due to BTB cold misses as the BTB size increases. For other benchmarks, there is no significant difference for the performance as the BTB size increases. In these programs, the total number of indirect instructions is less than 512, so the performance overhead has already saturated. For mcf, the overhead starts saturating at 1024, which indicates that the number of indirect instructions for it is less than 1024.
CHAPTER 3. LEVERAGING SPECULATIVE ARCHITECTURES

Figure 3.10: Normalized execution time with different BTB size to the case for a BTB of 512-1-1

We change the BTB set associativity but keeping the other two parameters constant. Figure 3.11 demonstrates the resulting execution cycle overhead with different set associativities ($n = 2, 4, 8$) normalized to that of the direct map case. The result is comparable to the previous one. As the set associativity increases, the effective capacity of the BTB increases, and the overhead also increases or saturates due to similar reasons.

Figure 3.11: Normalized execution time with different set associativity to the case for a BTB of 512-1-1
We finally change the BTB history path associativity while keeping the other two parameters constant. Figure 3.12 demonstrates the resulting execution cycle overhead with different numbers of history paths in an entry ($n = 2, 4, 8$) normalized to the case with one history path per BTB entry. We see that as the history associativity increases, the execution overhead drops dramatically, especially for benchmark *bzip*. Note that since the number of BTB sets and associativity do not change, the BTB capacity is the same, and there is no increase in BTB cold misses. However, with more history paths associated with a selected TPC, the chance of finding a matched HP in the BTB is increased, and thus the external memory access time is reduced. When the overhead time becomes saturated at some point, for example, $n = 4$ for *gcc*, it means that most of the indirect branches have less than four history paths leading to it. For *mcf* and *bzip*, $n = 2$ is enough.

![Figure 3.12: Normalized execution time with different history associativity to the case of a BTB of 512-1-1](image-url)
3.4 Related Work

There has been a lot of research done on using specialized software [16, 22, 26, 70] and hardware [27, 35, 90, 86, 28] to detect specific attacks such as buffer overflows or format string vulnerabilities. These techniques focus on the attack itself and can prevent attackers from exploiting the particular vulnerability. They fall in the category of white-box approaches, where the attack mechanism has been analyzed in detail and specific precautions are taken. However, due to the many sources of vulnerabilities, it is desirable to employ a more general symptom-based mechanism to detect any abnormal operations. This is a kind of black-box approach, where the program execution is monitored at run-time and any deviation from its legitimate behavior is detected, no matter what is the cause.

Researchers have shown that the program behavior can be modeled at different granularity levels, including system call, function call, control and data flow [11, 55]. Simply checking control-flow information is not sufficient to detect decision data attacks and may cause high false negatives in some cases. In addition, storing control transfer data in a Finite State Automata (FSA) and checking against all normal traces is a tremendous design effort [11, 34, 59]. Some previous work has focused on increasing the reliability of the return address stack (RAS) to defeat the exploits at the function call level [101, 52, 67]. However, the control data for other non-RAS indirect branches can also possibly be tampered with by adversaries through contamination of memory.

Other finer granularity levels have been proposed to detect abnormal program behavior. There exist some research works that detect control-flow errors either by software or
CHAPTER 3. LEVERAGING SPECULATIVE ARCHITECTURES

hardware support. The software-based approach rewrites the binary code with a checking mechanism in every basic block [16]. Although this technique is flexible, it requires binary translation and increases both the program size and execution cycles dramatically. Hardware support for control-flow transfer validation is more efficient in performance, e.g., the hardware-assisted preemptive control-flow checking [72]. However, the hardware cost is large because it is necessary to duplicate the entire register file and the PC for validation. Illegal indirect branch transfers may slip the checking mechanism. Another hardware-based approach mainly focuses on direct jumps and uses a sophisticated co-processor for the complex control-flow modeling and checking [105]. The storage overhead for dynamic behavior reference is very large.

The most related previous work is the Bloom Filter-based run-time control-flow validation [81]. They focus on reducing the storage size for legitimate control-flow transfers and thus the hardware access latency. However, their Bloom Filter may introduce false negatives. In their follow-up work [82], validations are only performed at indirect branch sites with a binary history path (branch direction). This approach may result in a high false negative rate because it cannot solve path ambiguities due to the binary representation. Our approach reduces the ambiguity by associating the binary history path with the last branch address and super block size. In addition, our mechanism considers the branch correlation between adjacent super blocks by associating the history path with the future expected path. Thus, our approach has a faster detection time as well as a higher anomaly detection rate than the previous work in [82].
By utilizing the existing branch target address prediction mechanism, our proposed system achieves negligible execution cycle overhead even with a long latency for accessing the FRS.

In summary, our contributions include:

• We propose a hardware-based solution to perform control-flow transfer and execution path validation at run-time with minimal hardware changes.

• Different from other approaches, our approach reduces the ambiguity due the binary representation by associating the binary history path with the last branch site and the size of the super block.

• Our approach correlates consecutive super block by associating the binary history path with the future expected path. Therefore, camouflaging is harder for the attack and the detection time may be faster.

• We demonstrate that our approach is effective in detecting control data and decision data attacks with very small performance degradation by leveraging the speculative architecture of branch target buffer (BTB).
Chapter 4

Static Secure Page Allocation for Light-Weight Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT) is an effective security countermeasure for both low-level memory corruptions and high-level semantic attacks. However, many software approaches suffer large performance degradation, and hardware approaches have high logic and storage overhead. We propose a flexible and light-weight hardware/software co-design approach to perform DIFT based on secure page allocation. Instead of associating every data with a taint tag, we aggregate data according to their taints, i.e., putting data with different attributes in separate memory pages. Our approach is a compiler-aided process with architecture support. The implementation and analysis show that the memory overhead is little, and our approach can protect critical information, including return address, indirect jump address, and system call IDs, from being overwritten by malicious users.

In this chapter, we present PIFT, a novel hardware/software co-design approach for
efficient dynamic information flow tracking using secure page allocation. The rest of the chapter is structured as follows. Section 4.1 and 4.2 describe our approach at design time and run-time in details. Section 4.3 and 4.4 present the security analysis and the experimental results, and Section 4.5 reviews the related work on DIFT for low-level memory-corruption security attacks and summarizes our contributions.

4.1 PIFT: Paged-dynamic Information Flow Tracking

We propose a novel approach to taint the memory at the granularity of page, normally at the size of 4KB, and allocate data to memories according to their attributes, trusted (TR) or untrusted (UTR), at compile-time. Our approach differs from previous DIFT approaches in three ways. First, rather than tracking the information and tainting data at run-time, i.e., updating the taints of both registers and memory, our approach identifies taints of memory data statically. The compiler allocates trusted/untrusted information into different memory pages. In this case, the run-time taint propagation and updating is only for registers. Second, instead of associating each data value to a taint bit, we aggregate data according to their taints, i.e., putting trusted data in trusted memory pages, and untrusted data in untrusted memory pages. The whole page has only one taint bit stored in the page table that reduces the memory space overhead significantly. Finally, our approach requires less hardware augmentation to do DIFT taint processing compared to hardware approaches, demands less operating system (OS) support for taint handling, and more importantly, does not require double memory access (one for the data and the other one for the tag). As a
result, the performance overhead is lower.

4.1.1 General Idea

In PIFT, the attributes of data are obtained from a static taint analysis of the source code at compile-time first. The compiler then divides all the data into two categories: trusted (TR) and untrusted (UTR). At loading time, the loader (an OS service) stores the data variables in different types of pages and initializes the page tag. At execution time, when a data is accessed, the Memory Controller (another OS service) retrieves the taint from the data address. In addition, the Memory Controller also allocates dynamic pages according to the attributes. Figure 4.1 shows the mapping of the virtual address space to physical memory, which is managed by the page table where a tag is set for each page. By default, the text segment, containing instructions, is trusted (TR). The data segment, the stack, and the heap are divided into two types of regions, according to the data that they will keep. At run-time, trusted/untrusted information will be allocated into trusted/untrusted pages accordingly. As a result, the overhead for storing/accessing taints is basically zero in our approach.

To propagate taints between the off-chip memory and on-chip register file, we have to augment the processor architecture slightly. The register file is widened by one bit to hold the taint. We also include logic to calculate the propagation of the taints. Many propagation rules, e.g., those in [28], can be employed and selected through a configuration register. We adopt the OR-rule on all the involved operands. Note that different rules just require

---

1Details about architectural changes are given in Section 4.2.
Different from other DIFT approaches, our approach statically ensures that untrusted data will not be stored in a trusted page and vice versa. Thus, at run-time, our approach checks this condition along with program taint propagation. Violations of such a condition may indicate possible memory corruption attacks, and the processor will raise an exception which allows the OS or other service routines to examine the violation further. By allocating the critical information, like the return address and system call IDs, into trusted pages and enforcing the taint checking policy, our approach protects the memory systematically. Our solution eliminates the need to check these control sites at run-time, e.g., function return and system call. However, indirect jump sites need taint checking, where the target addresses in registers must be trusted.
4.1.2 Compilation Stage

At this stage, PIFT does static taint analysis and memory map generation, which includes data attribute assignment, instruction duplication, and data duplication.

Figure 4.2 shows the basic code transformations done by a modified compiler. Our approach does not change the source code. Instead, it changes the compiler front-end to add annotations to each variable of the program in the static single assignment (SSA) representation. These annotations contain the taint information for each variable and are used by newly added passes in the middle-end to do static taint analysis for all the variables.

We define five attributes: \textit{UNINITIALIZED}, \textit{UNKNOWN}, \textit{NOT\_TAINTED}, \textit{TAINTED}, and \textit{BOTH}. The static analysis checks the statements one by one. If a variable is only declared, as in \texttt{#define INDEX}, the initial attribute is \textit{UNKNOWN}. If the variable is declared and an immediate value is assigned, as in \texttt{#define MAX\_SIZE 128}, the initial attribute is \textit{NOT\_TAINTED}. Sometimes, the analysis finds variables that are used without a formal declaration, and it sets the attribute as \textit{UNINITIALIZED}. Later, when these variables are declared, the attribute will be updated. Function arguments, inside the function, are set by default as \textit{UNKNOWN}. This attribute is used during the intra-procedural analysis, and gets updated at the function call sites when the attribute of the passed in arguments is known.

The objective of the static analysis is to find the final attribute (\textit{TAINTED}, \textit{NOT\_TAINTED}, or \textit{BOTH}) for all variables. The taint information is held during all the compiler optimization passes, and it is passed on to the back-end to generate the reordered memory map.

When identifying the data sources and consequently, the initial taint values, our ap-
approach also does annotations for critical C standard library functions, which are found vulnerable to memory corruption attacks [97]. Although the vulnerable function is given as a black-box, each time it is invoked, the compiler checks how the taints of the arguments affect the taints of the return results (if available). For example, function `malloc(size)` returns a pointer to a newly allocated block, `size` bytes long, and we let the attribute of the returned pointer depend on the taint of variable `size`.

Memory Map and Data Attribute

The application memory space is divided into four segments: text, data, heap, and stack. We handle each segment to support dynamic information flow tracking specifically, adhering to certain overwriting policies that we set for memory integrity protection. We adopt a run-time policy that data is only allowed to be exchanged between pages with the same taint. Our static transformation ensures it at compiler time. We show later in Section 4.3 by enforcing this policy various memory corruption attacks are caught by our checking mechanism.

By default, the text or code segment is set up as trusted. Therefore, when it is loaded,
the memory page taints are trusted. In general, this segment is read-only for program instruction fetching. However, in the case of applications with dynamically generated code (i.e., self-modifying code), the text segment could be overwritten. In these scenarios, the new code must be from trusted sources (i.e., trusted Java’s bytecode). If a piece of code is invoked from another section (i.e., the stack segment), a similar policy is applied. The code in all cases must be trusted.

The data segment normally contains initialized data used by the program (constants and global variables) and the BBS segment \(^2\). The constants are known at compile time and will keep their values throughout the execution. Hence, they are tagged as trusted and allocated on trusted memory pages. In contrast, although global variables may be initialized to some values, the values may change, and their attributes could be either trusted or untrusted at different execution times. If the different data stored in a global variable have the same attribute throughout program execution, the compiler fixes the tag value and allocates the global variable to a proper page. If the static taint analysis shows that a global variable can have both attributes, PIFT duplicates the global variable, one copy for trusted usages and the other for untrusted usages.

PIFT implements intra- and inter-procedural data flow analysis at the compiler *middle-end* to get a list of variables and their attributes, and then identify which global variables need to be declared twice. The duplication results in data overhead and slight code size increase. However, it helps us to protect critical information, like data pointers in the data

\(^2\)The BSS segment, also called block-started-by-symbol, includes all uninitialized variables declared at the file level as well as uninitialized local variables declared with the static keyword.
segment, from being overwritten by malicious untrusted information.

Heap segment is a memory section dynamically allocated to store temporary information. If the information comes from a trusted source, a trusted page (or pages) will be allocated. Otherwise, untrusted pages will be allocated. The taints are associated with the pages until the pages are released. At compile-time, the linker (the last step in the compiler back-end) assigns the right attribute to each heap chunk, and therefore the operating system (OS) can allocate the heaps on different pages.

In addition to dynamic data, the heap also holds critical metadata used by the memory management unit (MMU) for memory chunks. Attackers can exploit programming errors to overwrite the metadata, like forward and backward pointers that link to available memory chunks, to change the execution flow. If the MMU stores these critical pointers in a trusted page, our approach can avoid the possibility that they are corrupted by a heap overflow or double free. In [103], it shows that by modifying the memory controller (an OS service) and using a lookup table, the heap’s metadata can be separated and located in a different memory region. We propose to hold the metadata in trusted pages.

The last segment, stack, requires special considerations. At the beginning of the back-end code generation, some variables are assigned to the stack, and some are mapped onto registers. The stack variables could be function arguments, local variables, and special registers. Hence, the stack segment can hold both trusted and untrusted information. For example, frame pointer and return address are trusted. However, the attributes of function arguments and local variables depend on the static analysis, and they may need to
be allocated on different pages. In order to separate trusted data from untrusted data, we modify the way how the compiler allocates each variable on the stack. By default, each variable on the stack is indirectly addressed through the stack pointer and a relative offset (calculated at compile-time based on the position where the variables will be held). In our modified compiler, trusted variables and untrusted variables are allocated with a different offset. The offset is large enough to avoid stack overlapping, and is page-aligned as well. This modification helps to protect critical data, including return address, system call IDs, and function call pointers. The idea of multiple stacks has been presented in [104], where data is placed on different stacks according to their types, like array, integer, etc. We use attributes to differentiate them.

**Code Duplication**

As mentioned above, our compiler specifies the attribute of each variable and places it in an appropriate memory page. In the cases where a variable must be duplicated, the compiler has to generate a new statement to use the duplicated variable. Figure 4.3 shows an example using the GIMPLE\(^3\) representation. In the example, the system identifies that the attribute of variable \(c_3\) can be trusted and untrusted, so instead of merging \(c_1\) and \(c_2\) in \(c_3\) (by the \(\phi\) operator), it keeps both copies. Subsequently, the system duplicates an keeps both \(d_1\) and \(d_2\), one trusted (dependent on \(c_1\)) and the other untrusted (dependent on \(c_2\)). Due to statement duplication, the size of the code can increase. In Section 4.4, we present the experimental results.

\(^3\)GIMPLE representation is a tree based intermediate language used during *middle-end* compiler optimization.
There is an issue with function calls because the arguments’ attributes may be trusted or untrusted at different sites. Figure 4.4 shows an example. When an argument has different attributes at different call sites, as shown in Figure 4.4 for \textit{var1} and \textit{var3}, the function needs to be duplicated, and the function call sites need to be modified. During the static analysis, the system finds that the attribute for the first argument is BOTH. Thus, the system duplicates the function and propagate both attributes \textit{NOT_TAINTED} and \textit{TAINTED}. Although the statement duplication suggests an exponential increase of the code, our static taint analysis shows that function duplication is a rare condition. In most of the cases, the BOTH attribute is masked by the OR rule propagation, when this variable is operated with a TAINTED variable.

## 4.2 Architectural Augmentation

In the above section, we explain how the compiler sets the variable attributes and how the variables are aggregated to different kind of pages. In this section, we show how the
OS, especially the memory controller, should handle each segment of the memory map at run-time. We also describe what changes are needed inside the processor to support taint propagation and checking in order to detect security attacks on the application.

Figure 4.5 shows the architecture design. PIFT allows the compiler to aggregate data based on their taints, i.e., trusted data and untrusted data will be put in different pages. At run-time, when the OS allocates memory for the current process, it initializes the page taint accordingly in the Page Table (in memory). Each time the processor fetches an instruction
from the memory, the Memory Management Unit (managed by the OS) retrieves the Page Taint from the Page Table. The taint for the instruction has to be trusted in order to prevent malicious code injection and execution. This is ensured by a Taint Checking module at the instruction fetch (IF) stage.

During execution (EX) stage, the tag is propagated inside the processor by the Taint Propagation module. There are two locations for such modules. One is at the instruction decode (ID) stage where the taints of the source operand registers are known, and the current instruction is decoded. The other is at the memory (MEM) stage where the taint of memory data is retrieved from the page table, e.g., for LOAD instructions.

For taint checking, the Memory Management Unit module ensures that when data is being written to the memory (in the MEM stage), the combination of taints for the source and destination is allowed by the overwriting policy. In addition, the Taint Checking module checks that the jump target address is always trusted (in the MEM stage). Otherwise, an exception is raised.

Details about architectural changes are given below.

4.2.1 Wider Register File

Each register is widened by one bit to hold the taint. For taint propagation between registers or between memory and registers, we include glue logic to read and write these taints. “OR” operations are performed on the source taints except several special cases. For example, when the instruction has only one source operand, the taint is propagated directly without any operation. Another case is when a destination register is cleared using
the XOR instruction \((xor \ r1, r1, r1)\) or the AND instruction \((and \ r1, r1, 0)\), the register’s taint is trusted independent of the source operands taints. In addition, when an instruction loads an immediate value \((movl \$10,(%esp))\), the destination register or variable is always trusted. These special cases are all considered in the \textit{Taint Propagation} module, as shown in Figure 4.5.

### 4.2.2 Memory Taint Bit Retrieval

During taint propagation between registers and memory, the taint associated with a variable (in fact, with the memory page that holds the variable) should be retrieved on any LOAD or STORE instruction. At sites of LOAD instructions, when the memory address is used to look up the data memory/cache for data, it is used to look up page table for taint as well. At sites of STORE instructions, the taint for the memory location is retrieved in a similar manner, and the overwriting policy of no trusted/untrusted data moved to untrusted/trusted page is enforced with security alarms raised on violations.

In systems that support virtual memory, the page table has to be looked up for any memory-access instruction for data retrieval anyway. Therefore, there is no extra overhead for retrieving the taint bit stored in the page table.

At compile time, the static analysis assigns the tag for each memory segment. At runtime, when the OS loads the program, all the page attributes are set, and the tag information is kept in the page table. If the page table is full and a new page request arises with a page fault, a victim page has to be chosen and swapped out from the memory to disk. Correspondingly, its entry in the page table also has to be deleted. However, its taint has to
be saved in memory (some special region) for later reuse. When data on this victim page is needed again, the page will be swapped of the memory, and its taint will be recovered from the special memory region into the page table. Note that this page taint memory is managed by the OS, and the page fault handling process is extended slightly to both store the page taint on swapping the page out and recovering it on swapping the page in.

4.3 Security Analysis

There are two considerations in evaluating the effectiveness of PIFT. First, the approach should not produce false alarms when no attack is launched upon the system. Second, the solution should be able to detect attacks that attempt to overwrite critical information.

To evaluate the false alarms, we run several SPEC CINT2000 benchmarks [6]. For each application, the source code is compiled twice, one with a regular compiler and the other with our modified compiler. The obtained functionality is the same, and the applications run without false alarms. Details about how our compiler affects the execution are given in Section 4.4.

To evaluate the security effectiveness of PIFT, we use three micro-benchmarks to test if our system can detect overwrites of critical memory positions by untrusted information. These benchmarks were used in [20] to show the effectiveness of DIFT. To simulate dynamic information flow tracking at run-time, we used a modified version of tracegring, an instrumentation framework to track the flow of tainted data, which is a plug-in for Valgrind [64] developed by Avalanche project [43].
4.3.1 Protecting the Stack from Buffer Overflow

The first micro benchmark intends to experiment with stack buffer overflow attacks. As shown in Figure 4.6, it reads and prints an array of 10 elements (by definition). Because the function \textit{fscanf} does not check the input size, the program can read more elements and overwrite the stack of function \textit{main}. As a result, the stack can be smashed, and the return address can be overwritten.

Using PIFT, the overwriting is avoided because the return address and other trusted variables are put on a trusted page while \textit{array} is put on an untrusted page. The buffer overflow cannot affect the return address. Details of the attack are shown in Figure 4.7. The memory management unit (MMU) allows \textit{array} to grow, but only within untrusted pages. When it continues to overwrite an adjacent trusted page, the MMU will halt the execution, and the OS will take control of the application. In an X86 architecture, the attack is detected at this point because the taint attributes of the source index register (ESI) and the destination index register (EDI) differ, which are used by the system call, \textit{fscanf}, to perform copying a chunk of data in a memory region.

4.3.2 Avoiding Format String Exploitation

The second micro benchmark, shown in Figure 4.8, demonstrates how to exploit the format string vulnerability when the function \textit{printf} is invoked incorrectly. This vulnerability allows an attacker to send a format string like “1111 %p” (in \textit{array}), to reveal information on the process’ memory. With other format strings like “1111 %n” in \textit{array}, the attacker
can even overwrite the process’ memory.

PIFT catches the attack when the system tries to use the format directive \%p. In normal conditions, the format directive \%p is stored in a register that is used by the function \printf to handle the formatted printing. With PIFT, the attribute of this register is untrusted because the content comes from the user input. The taint of the format directive register gets to propagate to a register used by a conditional jump in the \printf function, and our checking policy does not allow untrusted target address.

4.3.3 Protecting the Heap

The third micro benchmark, shown in Figure 4.9, shows a heap corruption attack. Two arrays of eight elements, array and p, are allocated. One of them, array, is used to hold an untrusted string. Because the function scanf does not check the size of input, an attacker can introduce more elements and overwrite critical information (forward and
CHAPTER 4. STATIC SECURE PAGE ALLOCATION FOR DIFT

Figure 4.7: Defending against stack buffer overflow

backward link pointers) held in the memory next to array.

At static time, the compiler cannot determine where \( p \) and \( array \) will be allocated. The exact position is only determined at run-time. However, the static analysis can determine the attribute during compilation. The attribute of each chunk of memory is passed to the MMU when the program is loaded. When a new chunk of memory is needed, the MMU

```c
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100

int main (int argc, char **argv) {
    char array[MAX_SIZE];
    snprintf(array, MAX_SIZE, argv[1]);
    printf(array);
    return 0; }
```

Figure 4.8: Micro-benchmark for format string vulnerability
will allocate it in the right page. In this example, \texttt{array} is used to hold untrusted data from \texttt{scanf} function. Details are shown in Figure 4.10.

Our approach prevents the attack because \texttt{array} is now put in an untrusted page without the possibility of overwriting the critical meta-data or any trusted chunk of memory.

```c
#include <stdio.h>
#include <string.h>
#define SIZE 8

int main () {
    char * array;
    char * p;
    array = malloc(SIZE);
    p = malloc(SIZE);
    scanf("%s", array);
    free(array);
    return 0; }
```

Figure 4.9: Micro-benchmark for heap corruption

4.3.4 Defending Against other Attacks

For system calls and \texttt{return-to-libc} attacks, our approach is also effective because the system ensures that the IDs for system calls are always stored in a trusted page. PIFT prevents the attacker from invoking an arbitrary system call, but it does not prevent the attacker from requesting a system call with untrusted corrupted arguments.

For high-level semantic software attacks, e.g. cross-site scripting (XSS) attacks, the security weakness is the failure to validate properly inputs from the user. PIFT will be useful as well because the system ensures that only code from a trustworthy source can be executed. In addition, PIFT will detect any attempt to corrupt trusted information by an untrusted source.
Overall, PIFT can detect the attacks that intend to take control of the application by overwriting critical information like return address, system call IDs, and function pointers or by intending to execute untrusted code. Although the attacker can still overwrite some memory positions, no crucial information will be changed due to different page allocation and security policies. Our approach can be complemented with other techniques that watch the run-time execution of a program for abnormality [57].

### 4.3.5 Testing Our Approach with a Real Attack

Figure 4.11, taken from [21], shows the code of the function `do_authentication()` used to authenticate users on the SSH server. The `do_authentication()` function uses the local variable `auth` to check if authentication process has been done. The variable `auth` is set
to start at zero (0). Then, the internal function \texttt{packet\_read()} is used to read the password. Because the function \texttt{packet\_read()} is vulnerable \cite{33}, the attacker can overwrite the \texttt{auth} variable to one (1). As a result, the attacker can gain the access the system.

Our approach addresses this problem by allocating the \texttt{auth} variable on the trusted stack page, so the attacker cannot overwrite it.

```c
void do_authentication(char *user, ...) {
  int auth = 0;
  ...
  while (!auth)
  {
    /* Get a packet from the client */
    type = packet_read();
    switch (type)
    {
      ...
      case SSH_CMSG_AUTH_PASSWORD:
        if (auth_password(user, password))
          auth = 1;
        case ...
      }
  
  if (auth) break;
  }
  /* Perform session preparation. */
  do_authenticated(...); }
```

Figure 4.11: A real attack, taken from \cite{21}

### 4.4 Experimental Results

#### 4.4.1 Software Implementation

At compile-time, our implementation includes two parts: static taint analysis (that includes variable, code, and function duplication) and memory map generation.

Our compiler is built on top of Vulncheck \cite{84}, an extension of GCC 4.2.1 for detecting
code vulnerabilities. Instead of detecting vulnerabilities, our approach does taint analysis to initialize and propagate variable taints.

4.4.2 Static Memory Overhead

There are two sources of memory overhead: global variable and statement duplication. Figure 4.12 shows that the overhead for global variable duplication is 6% on average. Small variable duplication results in small statement duplication, as shown in Figure 4.13. The average code overhead is less than 1%.

![Figure 4.12: Global variable overhead](image)

In general, the total amount of memory used for global variables is less than a hundred bytes. Thus, getting 6% in global variable duplication in a program with several hundreds of kilobytes is a very small overhead. Nevertheless, how the variable duplication affects the code size is more important. The results show that 6% of the global variables (less than ten bytes) leads to a code increase of 1%, around 10 kilobytes for a program of 1 megabyte.
4.4.3 Dynamic Memory Overhead

The static analysis reveals that no program requires heap duplication, just separation. However, that is not the case for stack. Using drd, a Valgrind tool for monitoring the process behavior [64], we measured how much each stack grows during execution. Figure 4.14 shows the result. For some applications, the stack is just split into two orthogonal stacks (untrusted and trusted). The overall effective size is the same, so that the overhead is zero, like 176.gcc and 197.parser. For some applications, the majority of the stack needs to be duplicated (e.g., local variables that are trusted and untrusted at different usages), and the overhead is near 100%, e.g., 164.gzip, 181.mcf, 255.vortex, and 300.twolf. For other applications, it is a mix of separation and duplication, and the overhead depends on the composites of stacks, e.g., 256.bzip2.

The stack size for the applications that need stack duplication is on average 8 kilobytes. Thus, 8 kilobytes over a few megabytes (for code size) is a small overhead.

Figure 4.13: Code overhead
4.4.4 Execution Overhead

To measure the performance impact of our approach, we simulate a set of SPEC CINT2000 applications on cache-grind, a Valgrind tool for cache and branch-prediction profiling. The architecture parameters for the simulator are listed in Table 4.1.

Table 4.1: Parameters for Simulations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>Pentium 4 D</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.80GHz</td>
</tr>
<tr>
<td>I1</td>
<td>16KB, 8-way, 32 byte line size</td>
</tr>
<tr>
<td>D1</td>
<td>16KB, 8-way, 64 byte line size</td>
</tr>
<tr>
<td>L2</td>
<td>2MB, 8-way, 64 byte line size</td>
</tr>
<tr>
<td>Conditional Branch Predictor</td>
<td>16,384 2-bit saturating counters</td>
</tr>
<tr>
<td>Branch Target Address</td>
<td>512 entries</td>
</tr>
<tr>
<td>L1 access latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>L2 access latency</td>
<td>200 cycles</td>
</tr>
<tr>
<td>Mispredicted Branch latency</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Simulator</td>
<td>Cache-grind</td>
</tr>
</tbody>
</table>

In order to evaluate performance, we run two sets of program simulations and compare their number of cycles. One set is compiled and executed without any modifications, and
the other set is compiled with our enhanced compiler. Figure 4.15 shows the effect on performance degradation, less than 2% on average, which is negligible. The execution overhead is caused by an increase in the cache misses due to the memory duplication. When data with different attributes are allocated into different pages, the original spatial locality is reduced, and hence there will be more capacity-limit cache misses. At the same time, code size may get larger and average memory access time to code pages may increase as well.

![Figure 4.15: Execution cycle overhead](image)

4.5 Related Work

A lot of research work has been done for enforcing secure program execution through DIFT on computer systems. In most cases, four aspects of the system may get involved or require modifications to different extents: application, operating system, compiler, and hardware.

Software approaches allow much flexibility in policies for taint propagation and checking, and cover most of the known vulnerabilities [54, 99, 71, 17, 50, 69, 83]. However, they
suffer a significant performance slowdown, e.g., as shown in Securifly [54] with 37% execution overhead, 76% in Taint-enhanced [99], 363% in LIFT [71], and 35% in GIFT [50]. In addition, they cannot track information flow inside binary libraries or system call functions [99, 69], and do not support multi-threaded code [54, 71]. To reduce the amount of information that must be tracked and also the size of extra code, [17] uses static analysis and a declarative annotation language to associate symbolic tags with data at run-time. In [69], a value-range propagation algorithm is combined with taint analysis to help programmers apply an effective boundary-aware programming style. In [83], static taint analysis and program transformations are performed at compile-time to guarantee that program execution is free of unauthorized flow.

Hardware-assisted approaches address the problems of software approaches by introducing changes inside the processor core for taint storage and processing. The changes include widening the memory, register file, and buses, and introducing logic to initialize, propagate, and check the taints [86, 28, 19, 47, 39]. The taint initialization could be done by the operating system [86, 47], or by new instructions [28]. The policies for propagation and checking could be set up at the beginning of the execution [86], or reconfigured at run-time [28]. Some approaches use a special memory region to store the taints instead of widening the memory and buses [93, 19]. They may introduce more latencies for cache misses when the processor needs to retrieve the taint for data from the memory.

There also exists some hardware/software co-design work that utilizes the inherent processor architectural features without changes to the processor or memory system. An
approach presented in [18] leverages the speculative execution hardware support in modern commodity processors, like the Itanium processor, for DIFT. They utilize the deferred exception mechanism, using the exception token bit extended on each general-purpose register for the taint bit. However, they have to set up a bitmap for data memory for their taints. They use software-assigned policies to specify security violations, and software techniques for taint propagation.

To reduce the performance degradation of DIFT process, one recent tendency is to run separately the normal application and the DIFT process on different cores utilizing multi-core architecture [66, 19, 76, 41]. In [66], the application runs speculatively on one core without any security check. On another core, the application runs with DIFT. The system checks if the results of the two cores are the same. If they differ from each other, the system will roll the application back to the previous checkpoint. In [19], one core is used as a centralized lifeguard of the whole system, responsible for propagating the taints and checking the usage of them while the other cores are executing the applications. In [76], a log-based architecture (LBA) system provides hardware support for logging a program trace and delivering it to the monitoring processors. In [41], a memory-layout diversification technique and an efficient delta execution are used to reduce the overhead associated with executing several replicas.

The closest techniques to our page-level tainting are presented in [104, 103, 58]. The idea of splitting data onto multiple stacks and heaps based on their types was first proposed in [104, 103], where their data types are an attribute easy to retrieve from the source
code. Work in [58] presents the basic concept of page-level tainting by manually assigning variables onto pages based on profiling. Our approach is much more sophisticated, and implements the automatic taint analysis and page allocation process in a compiler with moderate hardware support. Our hardware/software co-design approach works efficiently and effectively for dynamic information flow tracking. Table 4.2 summarizes the comparison between our approach and other DIFT implementations.

Table 4.2: Comparison between Different DIFT Techniques.

<table>
<thead>
<tr>
<th>DIFT Steps</th>
<th>Software</th>
<th>Hardware-assisted</th>
<th>PIFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>application extension/compiler</td>
<td>OS support</td>
<td>compiler</td>
</tr>
<tr>
<td>Propagation</td>
<td>application extension</td>
<td>extra HW logic</td>
<td>extra HW logic</td>
</tr>
<tr>
<td>Storage</td>
<td>dedicated memory region</td>
<td>widened memory or dedicated region</td>
<td>page table</td>
</tr>
<tr>
<td>Checking</td>
<td>application extension</td>
<td>extra HW logic</td>
<td>extra HW logic</td>
</tr>
</tbody>
</table>

In summary, we propose a hardware/software co-design solution to perform dynamic information flow tracking based on compile-time static taint analysis and secure page allocation with minimal hardware changes. PIFT reduces the memory overhead significantly by aggregating data according to their taints and using only one taint bit in the page table for each page. We demonstrate that our approach can reach the same security level as hardware-assisted approaches, i.e., addressing self-modifying code, just-in-time compilation, third-party libraries, as long as the attributes (trusted or untrusted) of the sources of these codes are known a prior. Different from software approaches that annotate the code and add extra code to be executed, PIFT only involves system software, including compilation passes and operating system support, without reprogramming the applications. Therefore, the performance degradation of application execution is very small.
Chapter 5

Micro-Architectural Support for Metadata Coherence in Multi-core Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT) has shown to be an effective security measure for detecting both memory corruption attacks and semantic attacks at run-time on a wide range of systems from embedded systems and mobile devices to cloud computing. When applying DIFT to multi-thread applications running on multi-core architectures, the data processing and metadata processing are normally decoupled, i.e., being performed in different places at different times. Therefore, if the metadata access is not in the same order as data access, inconsistency issues may arise, which would reduce the security effectiveness of DIFT. Avoiding such inconsistency between data access and metadata access, i.e., maintaining metadata coherence, has become a challenging issue. In this paper, we propose METACE (METAdata Coherence Enforcement). METACE includes architectural enhancement in the memory management unit and leverages the existing cache coherence hardware and protocol to enforce metadata coherence. It introduces minimum changes to
cores, coprocessors, and the memory hierarchy. It covers the complete set of data dependencies without deadlocks and is compatible with different memory consistency models. Our approach does not require modification of the source code. METACE supports out-of-order metadata access resulting in less performance degradation than previous approaches.

The rest of the Chapter is structured as follows. Section 5.1 describes our approach in details. Section 5.2 explains how METACE handles different critical data dependencies. Section 5.3 presents the experimental results. Section 5.4 reviews the related work on metadata processing for DIFT and summarizes our contributions.

### 5.1 Our METACE Approach

#### 5.1.1 DIFT and Metadata Coherence Overview

Our approach assumes off-core metadata processing similar to [46]. The application is being executed on main cores, and each main core is associated with a coprocessor for metadata processing. The main core notifies its coprocessor with important information about the instructions in execution through an FIFO buffer, as shows Figure 5.1. The coprocessor propagates the metadata and checks the usage of metadata in critical sites.

For different instructions running on the core, there are different ways of synchronization between the core and the coprocessor. For data processing like ALU execution instructions or memory access instructions, the metadata is processed by the coprocessor only after the instruction has been committed by the application core. For control instructions like system calls and indirect jumps, the core suspends after issuing the instruction
5.1.2 Architecture Enhancement

METACE involves both architectural augmentation and cache coherence protocol extension. The current elaborate full cache coherence protocol is MOESI [61]. We modify the on-chip Coherence Unit \(^1\) by adding a table to track all data access events on the shared bus from applications cores. Each memory event adds an entry to the table, with the ID for the core requesting data access, the data address, and the action executed (read or write).

The coherence unit (CU) already snoops on the shared bus and controls the cache line

---

\(^1\) Coherence Unit (CU) is an on-chip module connected to the shared bus and the memory controller. It monitors all cache tags and controls cache coherence by transferring data between caches or between cache and the external memory [88]. CU is currently present in modern processors like SPARC T3 [85].
status according to the MOESI protocol. METACE leverages the CU to fill up the new structure without much overhead. Metadata coherence is then enforced according to the table.

Figure 5.1 shows the architectural enhancement of METACE. The table resides in the Coherence Unit and is associated with the shared bus in a multi-core architecture. For a multi-processor architecture, there are multiple CUs and therefore multiple tables, which are connected to other processors (CUs) via Coherence Links.

### 5.1.3 E-MOESI Protocol Implementation

We examine the original cache coherence protocol, MOESI, and find that it does not expose all the data access requests from main cores to the shared bus. We slightly augment the cache coherence protocol to capture the required information. Figure 5.2 shows the finite state machine for the extended MOESI cache coherence protocol (E-MOESI). Our modifications, shown in bold font, allow all the requests to be seen by the Coherence Unit and the METACE table without changing the cache coherence functionality.

Each cache line has five possible states: M (modified), O (owned), E (exclusive), S (shared), and I (invalid). The state transitions are responding to both local signals from the cores and global signals from the shared bus (appearing before the “/” on the edge label). Some state transitions generate output signals to the bus (appearing after the “/” on the edge label). For example, Read Hit and Write Hit mean the cache line receives a data read and write request from the core, respectively, and both are satisfied by the cache (a local cache hit). When Bus Read appears as the input on a state transition, it means
Figure 5.2: E-MOESI Cache Coherence Protocol. The text in bold indicates our modifications. *Bus Read* signal exposes read hit actions and *Bus Write* signal exposes write hit actions.

another core is requesting to read the data but does not find it in its private cache (a remote *cache miss*).

The MOESI cache controller only broadcasts the *Bus Read* signal on a *Read Miss* event. In a *Read Miss* event, the cache line status must be invalid (I). The requesting core is stalled for data retrieving, and a *Bus Read* signal is broadcasted with the requested address. Other caches respond to *Bus Read* signal by looking up their cache lines. If a copy is found in another cache (*snooping hit*), the data is copied to the requesting core’s cache from the snooped cache, and the requesting core’s cache line status changes from invalid (I) to shared (S). If no copy is found in any other cache, the data is copied from memory to the requesting core’s cache, and its status changes from invalid (I) to exclusive...
With the original MOESI, data access requests that result in a cache hit are not seen on the bus. We change the cache coherence protocol to broadcast a new signal, Bus Read', on cache hits. In the case of Read Hit, the requesting core’s cache line stays in its current status (M, O, E, or S), and the Bus Read' signal is broadcasted into the bus. Similarly, on a Write Hit of modified (M) or exclusive (E) data, Bus Write signal is not generated under the original MOESI. Therefore, METACE table cannot see such requests. In our approach, the cache broadcasts a new signal, Bus Write', in these cases. When the Bus Write' and Bus Read' signals are observed, no extra action is activated in other caches. This avoids additional bank/port contention on the other caches.

With these protocol changes, the cache coherence functionality is not affected at all, and all the data access requests are now seen by the Coherence Unit and the METACE table. The extra signals on the shared bus would consume some bandwidth and affect the execution time. We have considered this overhead in our experiments.

### 5.1.4 Modeling Hardware Overhead of METACE

A naive in-order implementation of METACE works like an FIFO structure with a head pointer and a tail pointer, where an entry is appended to the FIFO’s end each time a data access instruction is committed. Only the head entry can be evicted when a coprocessor’s metadata access request matches it. The sequential access order of the FIFO structure determines that the first entry is blocking all the metadata accesses of other entries, similar to [95]. However, accessing metadata in exactly the same order as data access is not neces-
sary and can stall non-critical requests, which introduces execution overhead on metadata processing. For example, for consecutive data readings by different cores, the order may not need to be preserved for metadata reading.

To support out-of-order metadata access and meanwhile maintain metadata coherence, a **fully-associative cache structure** is used for the table. Each entry is extended with two fields, *Key ID* and *Lock ID*, which associate a set of entries with data dependencies among them. Within a set, there is only one entry holding the key, but there may be multiple entries with the same lock. Figure 5.3 shows the block diagram of the METACE table. The METACE logic is the controller that takes bus requests, manages the METACE table, and sends signals to the requesting main core or coprocessor.

![Figure 5.3: Block diagram of METACE.](image)

Figure 5.4 depicts the actions executed inside the METACE modules when an entry is added. Each time a main core gains access to a memory data, a new entry should be added to the table. The METACE logic looks up the table for an available slot, and checks data dependency between it and the existing valid entries. If an RAW, WAR, or WAW dependency is discovered, a lock is set in the new entry and a key is assigned to the previous entry that must be attended first. For RAW, there may be multiple reads associated
with one data write. Thus, the lock will be duplicated for the multiple read entries.

Figure 5.4: METACE actions when a data access entry from a main core is logged.

The *logging process* is activated by data access requests from main cores, and should be completed before the corresponding coprocessor requests metadata access. It is passively logging, and it is not on the execution path of neither main cores nor coprocessors. Even in basic 5-stage pipelines, the time window for the logging process (5 cycles) is long enough to accommodate the actions shown in Figure 5.4, which include table lookup, checking
CHAPTER 5. METADATA COHERENCE ENFORCEMENT

entry dependencies, and setting keys and locks. Therefore, memory event logging does not affect the program execution at all. Only when the table is full, a signal is generated to stall the execution on all main cores. When some table entries are evicted by metadata processing on coprocessors, the execution will resume.

Figure 5.5 shows the actions executed by the METACE modules when a coprocessor requests metadata access. The METACE table is looked up to find an entry with the corresponding data access that matches the request. If an entry is found and it is not locked, i.e., it does not have any data dependency to preserve, an acknowledgment (ACK) signal is returned to the requester and the coprocessor gains access to the metadata instantly. If the matched entry holds a key for a lock, the corresponding lock is removed from other dependent entries. Then, the matched entry is deleted. The unlocked entries have no dependency any more and will be deleted when their corresponding metadata accesses are executed. If the matched entry is locked, the METACE logic rejects the metadata request by issuing an no-acknowledgment (NACK) signal to the request coprocessor. The coprocessor waits for other metadata access requests to unlock the entry. With such structure, any entry in the table can be evicted as long as certain conditions are met allowing out-of-order metadata processing.

The METACE logic includes a fair bus arbitration mechanism that handles the shared communication bus. When a coprocessor is NACKed, the mechanism will schedule the coprocessor only when the dependent entry is resolved and removed from the table. As the METACE table is located in the on-chip coherence unit (CU), the access delay for looking
Figure 5.5: METACE actions when a coprocessor requests metadata access.

up the table is equivalent to the access time to an on-chip translation looksaside buffer (TLB).

5.1.5 Overall Run-time Execution Overhead of METACE

Overall, there are some common execution overheads due to off-core DIFT techniques (shown in Section 5.1.1). Extra metadata accesses cause more memory accesses. The main
application core may be stalled if the FIFO buffer between the core and the coprocessor is full, or the current instruction (critical control site) has to wait until the coprocessor checks if the instruction is allowed according to the DIFT security policy.

There are also METACE-specific factors that affect the performance, including increase of the traffic on the shared bus due to the enhanced cache coherence protocol (shown in Section 5.1.3), limited METACE table capacity which stall data processing when it is full and extra time to check the METACE table for access order (shown in Section 5.1.4).

In our experiments, we have considered all the aforementioned performance-affecting factors and analyzed their effects.

### 5.2 Metadata Hazard Analysis

#### 5.2.1 Dealing with M-RAW, M-WAR, and M-WAW Hazards

To show how the METACE preserves RAW dependency on the corresponding metadata, we revisit the scenario presented in Figure 2.2, where two cores and two coprocessors execute the application and process the metadata in the correct sequence. Figure 5.6 shows the state of the table after memory access events are executed. Note that a lock is set on entry 3, and the key is assigned to entry 2. At the time coprocessor 1 requests to read tag(T) (event $t_{31}$ shown in Figure 2.2), the table is looked up for the request. Because the request (including the core ID, the address, and the action) matches the head entry and there is any lock, METACE allows coprocessor 1 to read the tag and the first entry is deleted from the list. Then, when coprocessor 1 requests to write tag(U), the system
allows the action, and the second entry is deleted from the list. Meanwhile, the lock #1 is removed from the dependent entry 3. When the coprocessor 2 requests read tag(U), the METACE also allows the action.

In the case of data-metadata inconsistency, i.e., coprocessor 2 requests to read tag(U) (event $t_{41}$) earlier than coprocessor 1 writes tag(U) (event $t_{32}$), the table is looked up. There is an entry matching the request, entry 3 in Figure 5.6. However, the entry is locked, and therefore METACE logic sends a NACK signal to coprocessor 2, which is taken as a cache miss signal. The metadata reading will be allowed only after coprocessor 1 writes tag(U) and unlocks entry 3.

To avoid M-RAW hazards, we set locks on read events by write events, but the system is still susceptible to M-WAR hazards. Similarly, read actions should also be logged and
lock the dependent write if a WAR data dependency is found.

An M-WAW hazard will appear when two cores write to the same location without reading operations in between, and the compiler does not remove such dynamically dead code. This hazard will be resolved in the similar way as we handle M-RAW and M-WAR hazards.

5.2.2 Out-of-order Metadata Access

Figure 5.7 shows an example where two set of instructions have both WAR and RAW dependencies on two data (B and A), respectively. After the application cores execute the memory accesses, the table looks like what is shown in Figure 5.8. There are two data dependencies between coprocessor 1 and coprocessor 2, RAW on A (lock #1) and WAR on B (lock #2). Because there is no lock for entry 1 and 2, the coprocessor 1 will not be stalled if it issues a request for reading tag(A) (event t32) earlier than reading tag(B) (event t31), i.e., out-of-order metadata access. Similarly, coprocessor 2 can read tag(C) (event t42) any time because there is no lock (data dependency) on it.

![Figure 5.7: An example with WAR and RAW for out-of-order metadata access](image-url)
### Figure 5.8: The state of METACE table after main core data accesses in Figure 5.7 are executed

Overall, METACE keeps a log of all the data dependencies and enforces them on metadata access. It allows out-of-order metadata processing when no dependency is found.

## 5.3 Evaluation

### 5.3.1 Experimental Setup

We use Multi2Sim [91], a cycle accurate multiprocessor simulator, to model and simulate different multi-core configurations and measure the performance impact of our approach. We extend the default cache module, *NMOESI*, to make all the data accesses visible to the centralized table.

Our baseline system has a two-level cache hierarchy similar to SPARC T3 [85]. The cache parameters has been set at the same values as in [45] to facilitate fair comparisons. The L1 instruction and data cache are 4-way 64KB each, the unified L2 cache is 4-way 32MB, and the tag cache is 4-way 8KB. All cache line sizes in the hierarchy are 64B. The LRU replacement policy is used for all caches. The access latencies for L1 instruction cache, data cache, and tag cache are all 3 cycles, and that for L2 cache is 6 cycles. The L2 cache miss penalty is 160 cycle. The goal of using a large L2 cache is to reduce the

<table>
<thead>
<tr>
<th>Entry</th>
<th>Valid</th>
<th>Core ID</th>
<th>Address</th>
<th>Action</th>
<th>Key ID</th>
<th>Lock ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B</td>
<td>Read</td>
<td>#2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>C</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>A</td>
<td>Write</td>
<td>#1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>Read</td>
<td>#1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>C</td>
<td>Read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>B</td>
<td>Write</td>
<td>#2</td>
<td></td>
</tr>
</tbody>
</table>
number of accesses to the main memory so that the effect of METACE on performance can be clearly exposed.

We model decoupled off-core DIFT monitors on coprocessors with a 16-entry FIFO interface, similar to [29, 46]. The amount of tags is proportional to the size of the program. We use: (i) 1-bit tag per byte, (ii) OR rule for tag propagation, and (iii) system calls and indirect jumps as the checking sites. The metadata is stored in the application’s memory space. A translation table filled in at the program loading time is used to map the location for the metadata from the address of the corresponding data, as in [62].

The tag cache is similar to a regular data cache, except for manipulation at the bit-level granularity. A 32-bit enable mask is used for either reading a one-bit tag from the 32-bit retrieved word or updating one bit. As we set the same line size for both application core caches and coprocessor tag caches, a single metadata cache line can pack tags for eight data cache lines ideally, providing good spatial locality and therefore alleviating the performance slowdown. If one coprocessor attempts to access a metadata that is not altered, and it locates in a cache line with another dirty metadata, the cache coherence protocol requires to reload the whole line. This false sharing may increase the metadata cache miss rate. Details about how the architectural changes affect the execution performance are given in Section 5.3.3.

We choose a diverse set of CPU-intensive parallel benchmarks from the PARSEC Suite [15] to test the impact of our metadata coherence mechanism. We use the simsmall input set. To identify performance bottlenecks, the number of threads is equal to the num-
ber of cores running the application. The benchmarks are compiled for x86 architecture using *pthread* synchronization primitives.

For each architecture configuration, we run the benchmarks on the multi-core system for three scenarios: 1) with coprocessors performing taint processing but without metadata coherence enforcement; 2) enforcing in-order metadata access with the additional hardware in the form of FIFO; 3) out-of-order metadata access with a fully-associative cache-like table.

### 5.3.2 Hardware Overhead for a Fully Associative METACE Table

Our experiments show that the size of the METACE table depends on the number of the thread running and not the size of the input set. With more threads, the METACE table is filled up more often, thereby either having a more severe performance impact due to stalling waiting on a METACE table entry to be freed up, or the implementation cost will be greater due to needing a large METACE table. By scaling the number of entries of the table to the number of cores, our simulations show that the selected sizes (16, 32, 64, 128 and 256 entries for 2, 4, 8, 16 and 32 cores, respectively) are sufficient to allow the application run without stalling the main application due to a full table. These values take in account the worst scenario found during the simulation stage, where each core was able to gain access to eight memory address before the corresponding coprocessors.

Table 5.1 shows the details of a table entry for a 32-core 32-bit architecture. Each entry is 64-bit. The first bit indicates the valid status of the entry. When deleting an entry, the valid bit is set to zero. The next three fields hold the data access information:
the requesting core ID, data address, and access action. The key and lock identification numbers are each 8-bit, to support up to 256 dependencies. The last field is not used in the current implementation. It can be used in multi-process scenarios to label the entry ownership by different processes, so on context switch only a portion of the table is flushed rather than the entire table.

Table 5.1: Details of an entry for an 8-core 32-bit architecture.

<table>
<thead>
<tr>
<th>Valid</th>
<th>Core ID</th>
<th>Address</th>
<th>Action</th>
<th>Key ID</th>
<th>Lock ID</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>5 bits</td>
<td>32 bits</td>
<td>1 bit</td>
<td>8 bits</td>
<td>8 bits</td>
<td>9 bits</td>
</tr>
</tbody>
</table>

We use CACTI 5.3 [49] to estimate the on-chip area overhead of our METACE table, and compare it with the on-chip L1 cache under 65nm process technology. Table 5.2 shows the simulation results. In the worst scenario, the area overhead for a 256-entry table is around 0.12% of the die size of the SPARC T3 Processor.

Table 5.2: Comparison of area for different table sizes versus an L1 cache (64KB, 64B line, 4-way associativity) and a processor.

<table>
<thead>
<tr>
<th>Area</th>
<th>32 entries</th>
<th>64 entries</th>
<th>128 entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mm^2$</td>
<td>0.044</td>
<td>0.070</td>
<td>0.152</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area</th>
<th>256 entries</th>
<th>L1 cache</th>
<th>SPARC T3 [85]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mm^2$</td>
<td>0.463</td>
<td>2.335</td>
<td>371</td>
</tr>
</tbody>
</table>

We count the total number of $Bus \ Read'$ and $Bus \ Write'$ signal broadcast in each configuration to have an idea about the power/energy impact of our approach. Figure 5.9 shows the additional activity in the shared bus and the tendency when the number of threads
increases. In the worst case, the overhead is around 3.5%, which also show that the contention in the shared bus is low. We can see that the increase remains constant up to 8 cores. However, the tendency for more cores depends on the application. For blackscholes and canneal the tendency remains constant. For fluidanimate and swaptions the extra activity tends to decrease. For streamcluster the abrupt change is due to the point dimensions (32) is the same number of cores, which stress the shared channel more than other configurations.

![Figure 5.9: Increase of the shared bus utilization along with number of cores.](image)

### 5.3.3 Performance Evaluations

We evaluate the impact of our approach on the execution time. Figure 5.10 shows the execution time overhead for the three cases, the application running with: (i) off-core DIFT monitoring only, denoted as DIFT, (ii) off-core DIFT monitoring and enforcing in-order metadata access, denoted as in-order, and (iii) off-core DIFT monitoring and allowing out-of-order metadata access, denoted as out-of-order, all normalized to the execution time for the application running without any DIFT. The system has 32 cores and 32 coprocessors, supporting 32-thread per application.
We can see that our version of DIFT with out-of-order metadata coherence enforcement has a low performance degradation compared to application execution without DIFT, 9.7% on average. For the in-order processing, the execution time overhead is slightly higher, 12% on average. For applications with low and medium data sharing like *blackscholes* and *fluidanimate*, the execution time overhead is lower because there are few data dependencies existing. For the other three applications, *canel* and *streamcluster*, with high or medium data exchange, and *swaptions*, with coarse data granularity, the execution time overhead is larger.

Adding metadata coherency enforcement to DIFT mechanism introduces very low overhead on top of off-core DIFT, less than 5.6% on average for our out-of-order metadata access. These performance results are similar to the 7% execution overhead over DIFT for a distributed approach [45]. The advantages of our approach are less invasive (without change the distributed cache controllers), easier and cheaper to scale with more cores, and can be easily extended for multi-processor architecture. In these cases, the execution
overhead would increase linearly instead of exponentially as happens in [45].

Figure 5.11 shows the performance degradation of our out-of-order metadata processing implementation when the number of cores changes, normalized to DIFT without metadata coherence enforcement. The source of overhead is the extra traffic added to the shared bus to monitor data accesses and enforce metadata coherence. Overall, the execution time overhead is small, around 4% for 8 applications cores, and less than 6% for 16 and 32 cores application cores. With more cores requesting access to the bus, there are more bus contentions for cache coherency and metadata coherency.

For single-thread applications, there is no coherency issue. Therefore, the approach proposed in this paper should not be activated at all. If single-thread application is running, there will be power/energy wasted on extra traffic on the bus, but the program performance will not be affected much. Figure 5.11 shows that the performance overhead is 4%.

It is interesting to note that as the number of cores increases, the overhead is not always increasing. The more cores, the more contentions on the shared bus for metadata coherency enforcement, and therefore the absolute execution overhead is increasing. However, the
baseline performance (DIFT without metadata coherence enforcement) is not always improving along the number of cores, because the cache coherency mechanism may even increase DIFT’s execution time when the number of cores increase. Therefore, the normalized overhead due to metadata coherence enforcement does not always deteriorate.

To investigate more details of the performance slowdown of METACE, we break down the overhead of enforcing metadata into three parts: DIFT overhead, MOESI overhead, and enforcement overhead. Figure 5.12 and Figure 5.13 show how it is composed for in-order and out-of-order implementation respectively. DIFT overhead represents the cost of decoupled metadata processing (initialization, propagation, and checking) on the off-core coprocessors. MOESI overhead is the extra overhead added to the system to make all the data accesses visible to the table (shared bus traffic). The last segment is the portion of the overhead for enforcing metadata coherence by looking up the table and checking the conditions. In Figure 5.12, in-order overhead is the extra overhead for enforce strict order metadata processing. In Figure 5.13, out-of-order overhead is the potion for allowing out-of-order metadata processing. Each application is running with 32 threads on 32 cores. As we can see, the out-of-order overhead is small, confirming our analysis in Section 5.1.5. The overhead for metadata processing (DIFT) and especially the coherence protocol enhancement (E-MOESI) are the biggest components of the overall overhead, indicating potentials for future performance optimizations.
CHAPTER 5. METADATA COHERENCE ENFORCEMENT

5.4 Related Work

DIFT tagging techniques can be classified into two categories [46]. The first type is “in-core” metadata processing, where the tag processing is integrated in the core pipeline. This requires drastic changes to both the processor logic and the register file, caches, memories, and busses for tag processing, storage, and propagating [86, 27, 28]. Such invasive implementations impose high design cost and incur hardware overhead. The second type is
“decoupled” processing, where the metadata processing is decoupled from data processing to reduce the implementation complexity.

There are two types of “decoupled” metadata processing approaches, offloading DIFT on multi-cores [95, 63, 80, 66, 76, 87, 75], and off-core DIFT on coprocessors [46, 29, 45]. In offloading approaches, the original application extended with communication hooks runs on one core, and a replica with DIFT monitoring runs on another core [80, 66, 87]. In [76, 75], the extra core monitors the program execution without running the application. These techniques use shared memory to synchronize the data core and the monitor core. They are not intended to handle multi-thread applications with shared data except for [95], which uses hardware support to make the order of metadata updates sequentially consistent with data access adding a significant slowdown (40%) due to synchronization.

Off-core approaches attach a coprocessor to the main core and let the coprocessor monitor the application at run-time. A hardware interface (e.g., FIFO) handles the communication and synchronization between the main core and the coprocessor. Off-core approaches are applied to both single-thread applications [46, 29] as well as multi-thread [45]. In [45], the main core captures inter-thread data dependencies by monitoring the cache coherence traffic and enforces them on metadata processing on coprocessors. The added complexity to the coherence mechanism difficulties its scalability.

To guarantee consistency between data and metadata, many works enforce atomicity by encapsulating both data and metadata accesses [62, 63]. Chung et al use transactional memory for metadata and data atomicity [23]. However, wrapping data and metadata ac-
cess in a transaction requires costly hardware support for transactional memory access. Software approaches, like LIFT [71], instruments the application by inserting metadata operations after the corresponding data accesses. The execution overhead caused by atomicity and the instrumentation reduce its usability.

In summary, METACE targets multi-thread applications running on multi-core architectures. We use off-core DIFT architecture, while the approach also applies to off-loading approaches with different performance impact. The main idea of METACE is to log memory accesses, identify data dependencies, and preserve them on corresponding metadata. METACE reduces the stall time of metadata processing by allowing out-of-order metadata processing on independent requests, rather than enforcing strict sequential data access order on all the metadata, as in [95]. METACE utilizes the current cache coherence protocol to monitor data memory accesses as in [45]. The distinct difference between both approaches is that our monitor hardware is centralized while the one in [45] is distributed. METACE enhances the Coherence Unit instead of modifying each cache controller distributively. Our approach has a lower traffic in the shared bus. Our centralized structure shows a linear scalability instead of the exponential scalability shown in [45]. Overall, METACE is more resource-efficient and incurs less execution overhead.
Chapter 6

Hardware Assited Thread Isolation

On a resource-sharing platform, running software subcomponents in isolation is critical to protect user’s privacy and data security. In client-server applications, thread isolation is required to prevent private data that only belongs to certain threads from being read or modified by other unauthorized threads running in the same address space. However, the current programming languages (C/C++) and compilers do not provide such support for multi-threaded programs. In this Chapter, we present HATI, a hardware assisted thread isolation approach. Different from software approaches, where both data access right setting and run-time monitoring of data objects access are embedded in applications and therefore result in significant dynamic memory usage and performance degradation, HATI leverages on-chip hardware modules to reduce the run-time validation time. It introduces much smaller memory overhead and very low performance degradation.

The rest of the chapter is structured as follows. Section 6.1 describes our approach in details. Section 6.2 presents the design and implementation of our thread isolation mechanism. Section 6.3 shows the experimental results. Section 6.4 reviews the related
work on thread isolation and summarizes our contributions.

6.1 HATI: Hardware Assisted Thread Isolation Mechanism for Multi-threaded C/C++ Programs

6.1.1 General Idea

HATI is a hardware assisted thread isolation mechanism. It integrates changes to system libraries that handle the program’s dynamic memory management, the operating system (OS), and the architecture, to enforce at run-time thread isolation in an effective and efficient manner.

System libraries are modified to intercept dynamic memory events and thread spawning. Memory management routines like `malloc()`, `new()`, `free()`, `delete()`, etc., extract important information on data objects and log it into an Ownership Table \(^1\) at run-time. Each time a thread is spawned, a new table is created and initialized based on a heritage policy. Details about the creation and update processes are given in the next section.

Figure 6.1 shows the architectural design of our approach. The Ownership Table becomes part of the process context, and resides in the system space for accessing protection. At run-time, data accesses by threads are monitored, and the intended sharing policies for the heap data objects are enforced \(^2\). To keep low overhead for run-time access, the Ownership Table, a structure of lists, is sorted and indexed by threads ID, so memory segments allocated by the same thread will be keep in one list. To keep the Ownership Table access

---

\(^1\) The Ownership Table is a data structure that keeps the ownership state of memory objects [56].

\(^2\) Text segment is by default only-read and shared, stack segment is private to each thread and protected by the OS, and data segment is shared by default.
time low, our approach proposes an extra on-chip hardware module, a TLB (Translation Lookaside Buffer)-like structure, to cache the active part of the Ownership Table.

The on-chip hardware helps to monitor each memory access and checks if it complies with the ownership policy in an efficient manner. To keep confidentiality and integrity of the collected information, the OS is slightly modified to allow the extra hardware to access the Ownership Table in main memory at run-time. The table can only be modified by the system libraries, and it cannot be accessed by the current thread or any other thread running in the user space.

6.1.2 Advantages of Our Hardware Assisted Approach

Our hardware assisted thread isolation approach has a number of advantages compared to the previous software approaches.

Performance

There are a number of execution overheads incurred by our thread isolation approach.

1. Creating and initialization of the Ownership Table during the spawning of a new thread.

2. Updating the Ownership table during a memory allocation or deallocation event.

3. Run-time accessing to the Ownership table for access right checking.

4. Thread context switching.
In HATI, overhead 1 and 2 are the same as those of software approaches like in [10, 56, 40]. Overhead 3, thanks to the hardware support, is significantly reduced and enables our approach to be used for run-time validation. Overhead 4 appears only in our approach, but it slightly affect the overall performance. Details about how the performance degradation is estimated and a comparison with software approaches are given in Section 6.3.

**Dynamic Memory Usage**

In software approaches, the compiler adds a few checking instructions per memory access into the program, which implies recompilation and highly increases the size of the executable file as well as the dynamic memory footprint of the running application. In our approach, the modifications are just in the system libraries. The application source code does not need recompilation, and the linker will statically link the application object files and the extended library object files, which slightly affects the size of the program executable and keeps the dynamic memory requirement unchanged.

**Protection on the Checking Process**

Instead of embedding the checking mechanism in code and locating the ownership table in the user space, our solution isolate both from being accessed by any user. The run-time ownership checking process involves accessing the Ownership Table in main memory (only modifiable through the system libraries), the on-chip table (a TLB-like structure that caches part of the ownership table), the enhanced hardware interface between the core and main memory, and the checking logic. Communications between the Ownership Table and
the on-chip table are controlled by the operating system, i.e., by the memory controller, the scheduler, and the context switch mechanism. With these features, no user can read nor change the information stored in the Ownership Table or the on-chip table. This is the main advantage of our solution over the software approaches.

6.2 Implementation Details

6.2.1 System Libraries

Our approach intercepts the `pthread_create()` function. During the creation, the child thread heritages the parent’s Ownership Table. This gives it the capability to access all the memory segments previously allocated by the parent thread with the same rights. Custom sharing models can be supported. For example, set a segment that only can be accessed only by selected threads with custom privileges. This would require source code annotations and compiler support as shown in [10, 56], actions that are out-of-scope in this work.

In addition, our approach intercepts memory management routines instead of modifying the source code, as shows in [56]. Table 6.1 shows a list of functions intercepted. First column includes the group of functions in the standard C library, and the second column explains the specific function. Third and fourth columns are for function routines in C++. When a new memory variable is allocated/deallocated, e.g., at sites of `malloc()` or `free()`, information about the owner thread, the base address, the size, and the access rights is collected and put in the ownership table. If the segment belong to a shared structure, the corresponding capability lists that share the structure are also updated. To avoid walk-
ing all the tables and reduce the searching time, auxiliary tables are used to bookkeep the shared segments. Our approach supports standard C/C++ memory management routines. Adding support for third party libraries, as in [2], can be done in a similar way.

<table>
<thead>
<tr>
<th>C routine</th>
<th>Function</th>
<th>C++ routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>allocate memory block</td>
<td>new</td>
<td>allocate storage space</td>
</tr>
<tr>
<td>calloc</td>
<td>allocate space for array in memory</td>
<td>new[ ]</td>
<td>allocate storage space for array</td>
</tr>
<tr>
<td>realloc</td>
<td>reallocate memory block</td>
<td>delete</td>
<td>deallocate storage space</td>
</tr>
<tr>
<td>free</td>
<td>deallocate space in memory</td>
<td>delete [ ]</td>
<td>deallocate storage space of array</td>
</tr>
</tbody>
</table>

### 6.2.2 The Ownership Table: Software Handling

The Ownership Table can be indexed by objects (access list) or by domains (capability list). In either case, the total area or size of the table is the same. However, the way how the structure is organized makes the run-time access time different. For implementation, we care for access time. Then, capability list gives the best way to index the ownership table and therefore is adopted. As the capability list is not necessary to be accessed by all the threads, an individual structure per thread can be held locally. This feature allow us to cache efficiently part of the capability list on an in-core TLB-like structure, as showing in Figure 6.1. As the thread only reads its own capability list, the validation is faster than the access the global table. Deleting an entry, however, must look up all the capability lists for shared data to remove the corresponding entry in all the possible lists. A table is used to rapidly find data objects that are shared between multiple capability lists and therefore reduce the overhead for looking up all capability lists.
6.2.3 Architectural Enhancement

The ownership is shown as a set of small capability lists indexed by thread identification number (TID). The Ownership Table is accessed through the memory bus during the allocation and free events and also whenever the on-chip TLB-like ownership table has a read miss (see Figure 6.1).

As our approach does not need extra instructions for accessing the ownership table, we embed extra logic to perform run-time references to the ownership table. The Ownership Table is hold in the program address space protected for user access. HATI caches part of the capability list of the current thread in the in-core OT. The partial information stored on the in-core OT is saved and restored each time a context switch occurs. This process increases the thread context switch time. Details about these effects are given later.

Initial analysis has shown that the whole Ownership Table in the main memory is small, only a few entries per thread. If the on-chip table is large enough, the entire Ownership
Table (capability list) can get uploaded during the context switch. It will reduce accesses to the main memory at the cost of context switch time increase. Otherwise, there will be performance degradation due to on-chip ownership table misses. To handle such misses, the system requires support from the memory management controller to retrieve the missing entry from the main memory. These actions may be on the system execution critical path, because the system has to validate the data memory access actions before performing them.

To reduce the performance overhead for multiple access to the main memory, HATI does not only bring the entry requested when a missing ownership table is retrieved, but also information about other segments is cached. The on-chip OT is modeled as a fully associative cache with an LRU replace algorithm. A local (cached) capability list is kept coherent in the same way as a shared cache. When a segment is freed, the corresponding entry is invalidated in the local on-chip OT, and an invalidation signal is broadcasted to other shared lists. These features have been modeled during our experiments.

As data object-thread access right checking is on every data memory access, there may be two memory accesses in the worst case: one for the ownership information retrieval and the other for data objects, with misses on both the on-chip ownership table and data cache lines.

When there are multiple active threads, there are multiple thread contexts. Each thread context holds the state of the configuration registers, the register file, the TLBs, and in our approach, the on-chip ownership table. Saving and restoring the thread context (including
the new on-chip table) is controlled by the scheduler and the OS context switch mechanism.

The Ownership Table holds virtual address. This allows the check occurs before the address translation, then HATI can perform the check in parallel with the translation. At the same time, using virtual address avoids costly table synchronization when the OS changes the address space mapping or a TLB shoot-down occurs.

### 6.2.4 Operating System Support

Our approach requires little OS support at different points. At the loading time, the OS is in charge of allocating an empty Ownership Table in memory and setting the access rights. This process is done before the program starts. Therefore, it does not introduce any performance overhead for program execution. Similar support is needed when a new thread is spawned. In this case, the OS allows the clonation of the parent’s Ownership Table.

HATI is only protecting dynamic data, as static data is by default shared rather than private. The memory controller detects dynamic memory access and enforces ownership protection. Any memory access to other data segment (for example, BBS, global variables, or constants) does not require checking, and the access is granted by default. Text segment is only-read and shared, and stack segment is private to each thread and protected by the OS.

At run-time, during a thread context switch, our mechanism has to flush the on-chip ownership table and bring in the entries for the scheduled new threads. Thread context switches also happen when the OS moves a thread from one processor to another. In this
case, the on-chip ownership table information is moved along the thread context. Context switch time can become an issue if the scheduler continuously changes contexts. All these conditions have been modelled in our design and implementation.

During execution, HATI requires OS support whenever a miss happens to the on-chip Ownership Table. The memory controller has to access the capability list of the current thread and bring the miss entry. In the case an entry is not found in the Ownership Table, an alarm is raised. The OS kills the application and generates a log report. This information can be used later for more analysis. More sophisticated treatments are possible, for example, kill the thread that tries to access other’s private data and keep the rest of the application running. However, these actions are out-of-scope in this paper.

6.3 Experimental Results

The main advantages of doing hardware thread isolation over software approaches are performance and protection on the checking process. There is less performance degradation than the software approaches, which embed the checking mechanism in the code. Therefore, HATI incurs less dynamic memory usage and does not demand complex compiler support. Meanwhile, the checking process is well protected from the user. Programs in user space cannot access the extra on-chip hardware, the ownership table, or the function modules to handle the Ownership Table.
6.3.1 Benchmarks and Simulators

We test our approach with ten multi-threaded programs divided into two groups: network bounded and memory-intensive multi-thread applications. The first five (*aget*, *pbzip2*, *pfscan*, *stunnel* (using openSSL library), *NullHTTPd*) are the same as those used in SharC [10] and Dynamic Ownership [56] so as to simplify comparison with our approach. These programs use different threads synchronization methods, e.g., event, mutex, and semaphore. We use NullHTTPd to evaluate our approach’s effectiveness in avoiding cross-thread attacks [3]. We also choose application from the Parsec Suit v2.0 [15] (*blackscholes*, *fluidanimate*, *swaptions*, *canneal*, and *streamcluster*) that use different synchronization methods. Table 6.2 shows the benchmarks used to evaluate our approach, along with their key characteristics of granularity and data usage. These benchmarks demonstrate the generality of our approach, show the worst case scenario, and show clearly all the overheads introduced.

<table>
<thead>
<tr>
<th>Program</th>
<th>Granularity</th>
<th>Working Set</th>
<th>Sharing</th>
<th>Exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>coarse</td>
<td>small</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>canneal</td>
<td>fine</td>
<td>unbounded</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>fine</td>
<td>large</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>streamclusters</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>swaptions</td>
<td>coarse</td>
<td>medium</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>

Application *aget* is set to download the compressed linux-2.6.0 (41MB) from the Linux Kernel Archives web site [7]. Application *pbzip2* compresses a 15MB file. In *pfscan*, the application searches for the string “hello” in the design folder of OpenSPARC T1 suit
Benchmark *stunnel* encrypts three connections to a bare echo server and measures the time to send and receive 10,000 messages. We measured the throughput of *NullHTTPd* with the maximum number of simultaneous connections set at 50 and, we used the Apache Benchmark application [8] to test it. We stressed the application by increasing the number of simultaneous clients until the throughput stops increasing. The results reported were obtained with 30 simultaneous clients fetching a small web page 10,000 times. For the Parsec programs, we spawn four threads and used the large input set to get the performance [15].

We first ran a program without any modification and take the execution time as the reference. We then execute the benchmarks with our approach activated. We wrote our own memory profiler using PIN [5] to calculate the memory overhead for the Ownership Table. We use Multi2Sim [91], a cycle-accurate multiprocessor simulator, to estimate the performance overhead. The CPU and memory configuration are given in Table 6.3. The memory hierarchy consists of a private L1 cache (unified for instructions and data) for each CPU core, which is also shared by the multiple threads scheduled onto the same core. An on-chip interconnect based on a single switch connects all L1 caches with a common L2 cache, and a bus connects the L2 cache with the main memory module.

We profile application execution to obtain dynamic data objects allocation/deallocation information, including `malloc()`, `free()`, etc., for C/C++ programs. We then perform memory access checking on the execution trace. The execution time overhead, the main memory overhead, and the access time of the extra hardware for checking are all collected.
Table 6.3: CPU and Memory hierarchy configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>x86</td>
</tr>
<tr>
<td>Number of cores</td>
<td>4</td>
</tr>
<tr>
<td>Number of hardware threads</td>
<td>1</td>
</tr>
<tr>
<td>L1</td>
<td>128 sets, 2-way, 256 block size</td>
</tr>
<tr>
<td>L2</td>
<td>512 sets, 4-way, 256 block size</td>
</tr>
<tr>
<td>L1 latency</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 latency</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>

6.3.2 Dynamic Memory Overhead

Dynamic memory overhead refers to the peak memory usage at run-time of the application, which includes the extra Ownership Table in addition to the original memory usage. Figure 6.2 shows that the maximum memory usage is very small, less than 1.4%, in the worst case (stunnel). The reason for these values is the small size of the Ownership Table versus the program’s dynamic memory usage.

![Figure 6.2: Dynamic memory overhead](image)

The memory overhead for the Ownership Table is proportional to the number of new
memory chunks allocated by the current thread (and the parent thread) into the heap. For each allocation, an entry is added to the table. Eight bytes per entry is sufficient for a 32-bit architecture to include information including \{base address, size, and rights\}. Figure 6.3 shows the maximum number of entries at run-time. We can see that stunnel has a high number of entries. This is because each connection (50 in total) starts a thread (50 threads simultaneously) that at the same time allocates multiple variables. However, the ratio between the overhead and the original memory footprint is low, less than 1.4%.

![Figure 6.3: Number of entries in the Ownership Table](image)

### 6.3.3 Performance Degradation

Performance degradation is the increase of the execution time with our data access checking mechanism. We run simulations for each benchmark without checking and with HATI enable. We take into account the execution overhead due to the creation and initialization of the Ownership Table during the spawning of a new threads as well as updating the table in each dynamic memory management event. We also consider the overhead for run-time
CHAPTER 6. HARDWARE ASSISTED THREAD ISOLATION

checking and validation. Finally, we estimate the effects over the thread context switching time.

**Execution Overhead for Handling the Ownership Table**

Figure 6.4 shows that the performance overhead for handling the Ownership Table is 3% on average. From Figure 6.3, we expect that programs with bigger ownership tables as *pf scan* and *stunnel* would have bigger performance degradation. However, *streamcluster* shows a different pattern. Even though the size of the Ownership Table is a few entries, the execution overhead is bigger than similar benchmarks. The reason of this behavior is the particular combination of medium granularity of the shared data and medium exchange features between threads that forces more frequent accesses to the in-memory Ownership Table and update more frequently the in-core OT.

![Figure 6.4: Execution overhead for handling the ownership table](image)
Run-Time Accessing Overhead

We estimate the execution overhead due to run-time validation. The on-chip TLB-like ownership table that works as a cache reduces the overhead for accessing the Ownership Table (in main memory) for each memory access. As a cache, the size of this table is important to keep the number of accesses to the Ownership Table low.

We estimate the overhead for different on-chip table sizes: 1, 2, 4, 8, 16, and 32 entries. The results given in Figure 6.5 show that the overhead can be up to 38% (for canneal), 12% on average when the system uses an on-chip ownership table with one entry. Figure 6.5 also shows that the overhead can be reduced around 2% when the on-chip ownership table has 32 entries. The optimal size for the on-chip Ownership Table, based on the results, is 16 entries. With this size, the worst overhead is around 3% with moderate hardware overhead. Details about the hardware overhead are given in the next section.

Figure 6.5: Execution time overhead due to run-time ownership checking and validation

*streamcluster* shows a different pattern without much improvement on the overhead when the number of entries increases. The reason of this behavior is the particular combi-
nation of medium granularity of the shared data and medium exchange features between threads forces more accesses to both in-memory and in-core OT.

**Context Switch Time Degradation**

Adding an on-chip Ownership Table to the thread’s context will increase the context switch time, which will affect the overall performance. We simulate the effects of doing it by increasing the penalty for context switch between a few cycles (low data transfer, up to 4 entries) to several hundreds of cycles (high data transfer, up to 32 entries). After several trials, we don’t see significant differences between the execution time running the application with different context switch penalties. It indicates that adding the on-chip Ownership Table information to the thread’s context does not affect the performance.

**Overall Performance Overhead**

Overall, the performance overhead is the combined effect of software handling (fixed), the checking and validation mechanism (depend on the size of the in-core OT), and the context switch time. Table 6.4 shows a comparison between HATI with a 16-entry in-core OT and different software approaches. The shown results are from the data reported by the authors. For fair comparison, we use the same programs running in a similar architecture with similar configurations (number of threads, size of input set, etc.).

**6.3.4 Hardware Overhead**

Table 6.5 shows the details of a table entry for a 32-bit architecture. Each entry is 64-bit. The first field indicates the base address. We use CACTI 5.3 [49] to estimate the
Table 6.4: Comparison between different software-based approaches and HATI with a 16-entry in-core OT.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Memory Overhead</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SharC [10]</td>
<td>26.1%</td>
<td>9.2%</td>
</tr>
<tr>
<td>Ownership [56]</td>
<td>10.4%</td>
<td>25.8%</td>
</tr>
<tr>
<td>HATI</td>
<td>0.4%</td>
<td>5.4%</td>
</tr>
</tbody>
</table>

on-chip area overhead of our in-core Ownership Table, and compare it with the on-chip L1 cache under 65nm process technology. Results are shown in Table 6.6. The area overhead for a 32-entry table is around 0.094% of the die size of the Xeon Processor E5 family (81 mm² [42]).

Table 6.5: Details of an entry for a 32-bit architecture.

<table>
<thead>
<tr>
<th>Base address</th>
<th>Size</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>24 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

Table 6.6: Comparison between a 32 entries size table and an L1 cache (64KB, 64B line, 4-way associativity).

<table>
<thead>
<tr>
<th></th>
<th>32 entries</th>
<th>L1 cache</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area [mm²]</td>
<td>0.076</td>
<td>2.335</td>
<td>3.25%</td>
</tr>
<tr>
<td>Energy [nJ]</td>
<td>0.062</td>
<td>0.363</td>
<td>17.07%</td>
</tr>
<tr>
<td>Power [W]</td>
<td>0.089</td>
<td>0.438</td>
<td>20.31%</td>
</tr>
<tr>
<td>Access Time [ns]</td>
<td>1.035</td>
<td>1.371</td>
<td>Faster</td>
</tr>
</tbody>
</table>

6.3.5 Security Evaluation

HATI protects against cross-thread heap overflow vulnerabilities. A real-world example of the attack is the vulnerability found in NullHTTPd [3]. With HATI enable, this attack is
detected when the attacker try to overwrite arbitrary words in a memory segment through the `free()` function, which does not belong to the malicious thread.

### 6.4 Related Work

There are some software techniques for fault isolation and process isolation, including code re-writing [96], in-line monitoring [30], process memory protection [9, 14, 98], and traditional capability-based access control mechanisms in custom operating systems [24, 73]. These techniques have added fine-grained protection and translation on data objects. However, they do not deal with multiple threads running in the same process context specifically. Their applicability to thread isolation is either limited or may only work for a specific architecture [96]. The performance slowdown is large, from 2x to 8x [30], and the separate subcomponent executions introduce complex inter-process communication [9, 14].

The most relevant thread isolation solutions are SharC [10], Dynamic Ownership Checking [56], and Ribbons [40]. These are all software-based solutions. In SharC [10], annotations are added to the source code to label data objects sharing patterns. These annotations are interpreted by the compiler. Static analysis helps to detect violations of defined rules. Dynamic checking is used to eliminate false positives and false negatives. The approach requires bookkeeping all the memory allocations, as well as checking every memory access. It increases the program code size and also the execution time significantly.

In Dynamic Ownership Checking approach [56], the programmer has to write ownership assertions that describe the sharing policies. The compiler does intra-procedural
analysis and inserts checkpoints before and after each memory access. Memory allocation and deallocation functions in C programs, including malloc(), free(), and memcpy(), return the necessary information about the size of the buffer and the memory map for data object sharing. The compiler adds a guard band between memory objects, and an exception will be raised if a thread attempts to overwrite the guard band. An ownership table is used to check each memory access. Attempts to access uninitialized memory or use dangling pointers can be detected. The main drawback of the approach is the size of the instrumented code. In addition, each memory access requires two accesses to the ownership table, one for address validation (not overwriting the guard band), and the other for data right checking and management.

In Ribbons [40], the main idea is to have an intermediate sharing pattern between thread (full sharing) and process (full isolation), i.e., ribbon. The source code (Java) is annotated, and therefore the compiler adds extra code to each memory declaration for sharing policy. Data sharing and access monitoring are also performed in the JVM (Java Virtual Machine), which interacts with the host machine OS to enforce memory access rights. The main advantage of this technique is a small increase in the size of the bytecode. However, the main cost is the code instrumentation. In addition, the reference table used to keep the critical information is kept in the user-space. Hence, it is open to corruptions.

In summary, the focus of this paper is to design a hardware-software framework for run-time thread isolation. HATI aims to protect both the confidentiality and integrity of a thread’s private data via explicit thread isolation that can be used in a running application.
CHAPTER 6. HARDWARE ASSISTED THREAD ISOLATION

The isolation mechanism will be responsible for setting the thread access rights of data objects (private, shared read-only, shared read/write, etc.) in multi-threaded programs. It then monitors at run-time whether the running application follows the established rules or not. Different from software approaches, where both access right setting and sharing pattern checking are embedded in the application, which increases the program size and the execution time significantly, our approach results in small increases on the application footprint and very low performance degradation with adequate hardware support.
Chapter 7

Conclusions

The research on security is far to be done. With the adoption of new technologies and the adaptation of the current ones, the challenges are always in continue change. In the this section, we summarize conclusions of our work as well as propose future work, which we expect to further advance this fascinating area.

7.1 Main Conclusions

This dissertation explores the potential of hardware/software co-design techniques to provide comprehensive protection from a wide variety of attacks on real-world applications. It addresses the critical issues of ensuring software protection on multi-thread applications running in multi-core architectures. This work focuses on integration of enhanced in-core modules like the branch predictor and branch table buffer (BTB), the memory management unit (MMU), the coherence unit (CU), the cache coherence protocol, the compiler, and the operating system (OS).
The conclusions of this dissertation are the following:

- In this dissertation, we have proposed a practical micro-architecture-based approach for run-time control-flow transfer and execution path validation. With the aid of speculative architecture of branch target buffer (BTB), we narrow down the insecure instructions to indirect control instructions, and sample the validations only at indirect control sites. The anomaly detection rate of our approach is higher than previous related work because our approach not only validates the history path, but also monitors the next branch decisions at run-time. Our approach results in very little performance degradation with minor storage overhead.

- Using a taint bit for each page helps us check data structure boundaries indirectly. We enforce that trusted memory cannot be overwritten by untrusted data and vice versa. In this way, data read from local disk like program instructions, initialized variables, and constant tables are trusted, and the attacker cannot overwrite it with data obtained from untrusted sources, like input/output (I/O) ports. In this dissertation, we propose a flexible, efficient, and light-weight hardware/software co-design solution for Dynamic Information Flow Tracking with compile-time page allocation. It achieves the same level of security as hardware-assisted DIFT, with less hardware augmentation. The overhead for taint storage is reduced almost to zero. The little memory and hardware overhead for implementing the proposed page-level taint processing yield a negligible performance overhead for the system.
• Decoupled metadata processing may result in data-metadata inconsistency issue, affecting both the security effectiveness and performance of DIFT implementations. This dissertation presents METACE, a centralized architectural enhancement in the coherence unit that enforces metadata coherence for dynamic information flow tracking in multi-thread applications running on multi-cores with shared memory. By monitoring the cache events, our approach identifies data dependencies and avoids data-metadata inconsistency. The extra hardware is unintrusive since it only introduces an extra module on the shared bus, rather than in the cores or coprocessors. Our simulation results show that the overhead for doing DIFT is dependent on the application’s features of data sharing and data exchanging. The results have demonstrated that our centralized solution only slightly affects the overall execution time, and can be implemented with much lower complexity and higher resource efficiency than the distributed approach.

• Running subcomponents in isolation is critical to user’s program security and privacy. Achieving proper and efficient isolation from each other, however, is very challenging in a multi-threaded program. This dissertation presents HATI, a hardware assisted mechanism for thread isolation. Compared with previous protection approaches, our implementation effectively protects a thread’s private data from being disclosed, and the integrity of a thread’s data from being corrupted by an unauthorized thread, in an efficient manner with the support of hardware that enables it to be used for run-time validation. Our approach splits the tasks of handling and
accessing the Ownership Table into the software and hardware domain, respectively. Our feasibility study indicates that the memory footprint and run-time overhead of providing this fine-grained protection is small. In addition, our approach protects the Ownership Table in a way that no user can read or modify its content. This yields a great advantage over previous approaches that handle the checking and monitoring in the same user space.

7.2 Future Work

While there has been significant interest in Dynamic Information Flow Tracking with compile-time page allocation in industry and academia, there remain several challenges to the widespread adoption of PIFT in the real world. More study is required to find methods for dealing with complex data structures (with both tainted and untainted fields) or shared data memory programs. Either case, the solutions would require efforts at static time and at dynamic run-time. The compiler would need to recognize and split shared structures when they are dual tagged. The coherence unit (hardware and protocol) would require adjustments.

In Chapter 5, we present our centralized architectural enhancement in the Coherence Unit that enforces metadata coherence for dynamic information flow tracking in multi-thread applications running on multi-cores with shared memory. This only includes a
specific architecture. As future work, we propose to test our approach in different architectures like Nehalem, and evaluate the effect of our approach on the performance with different architecture features, including L3 cache, distributed Coherence Units, and multiple processors.

In Chapter 6, we restrict our approach to a simple sharing model that helps us to protect the integrity and confidentiality of thread’s private data without annotating the source code. As future work, we propose to support more complex sharing models. This would require source code modifications and compiler support to work in conjunction with the modified system libraries and the additional hardware.
Bibliography


[85] SPARC T3-1, SPARC T3-2, SPARC T3-4 and SPARC T3-1B Server Architecture. Sun Oracle, February 2011.


