EFFICIENT AND VERIFIABLE TIMING CHANNEL PROTECTION FOR MULTI-CORE PROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yao Wang

January 2017

EFFICIENT AND VERIFIABLE TIMING CHANNEL PROTECTION FOR MULTI-CORE PROCESSORS

Yao Wang, Ph.D.

Cornell University 2017

Modern computing systems are becoming increasingly vulnerable to timing channel attacks that leak confidential information through the timing of microarchitectural events. Many timing channel attacks are caused by the interference between different programs in the shared resources of a multi-core processor. For example, an attacker program's cache lines can be evicted by a victim program, which allows the attacker to infer secret information about the victim. Timing channel attacks pose serious threats to contemporary computing systems because they can bypass traditional defense mechanisms such as access control. Previous studies have even demonstrated a practical timing channel attack to recover the keystrokes of a user in the commercial Amazon EC2 cloud.

In this thesis, we explored new timing channel attacks and developed timing channel protection schemes for some of the hardware resources in a multi-core processor. Specifically, we discovered new timing channel attacks in the shared on-chip networks and memory controllers. We proposed multiple protection mechanisms for on-chip networks, caches and memory controllers. Our protection schemes cover three high-level approaches: bi-directional protections, unidirectional protections and protections that trade off security for performance. We evaluate our protection schemes and show that the proposed schemes are effective against timing channel attacks while achieving performance improvements over previous protection schemes. Finally, we implemented some of the

protection mechanisms in RTL and used SecVerilog to verify the information flow control in hardware. The results show that the protection mechanisms indeed remove timing channels at the gate level.

BIOGRAPHICAL SKETCH

Yao Wang received his Bachelor of Science (with Honors) degree in Electronic Engineering from Tsinghua University in 2010. After that, he began to pursue his doctoral degree at Cornell University in the department of Electrical and Computer Engineering, with a focus on timing channel protection and hardware security.

This document is dedicated to my parents and my beloved wife, Xi Yan.

ACKNOWLEDGEMENTS

Six years ago, I arrived at the US for the first time as a student to pursue a PhD degree in computer engineering. I can still remember that I could barely speak English at that time, and was a bit frightened by the unfamiliar environment. Six years later, I have published seven research papers and managed to finish this 200-page long thesis. To be honest, I feel quite proud of myself. But I am also clearly aware that these achievements would not have been possible without the help of many people, whom I feel fortunate to meet in my life.

I would like to express my deepest thanks and appreciation to my PhD advising committee members for the help and support they generously provided. First and foremost, I want to thank my advisor Professor Edward Suh. Ed showed me that a good researcher is rigorous, passionate, and hardworking. I do not dare to say I am a good researcher, but I was certainly getting closer and closer to the standard under Ed's guidance in the past six years. One thing I did not enjoy in graduate school was the very night before a paper deadline, but Ed was always by my side, working with me together, which certainly made some pleasant memories about the not-so-pleasant paper deadlines. I also would like to thank Ed for being so kind and tolerant with me. I made plenty of mistakes during my PhD studies, but Ed was always very gentle and patient when advising me to avoid those mistakes. The next person on my thank list is Professor Christopher Batten. In my first year, I was very fortunate to take Chris's class of Computer Architecture, which I consider as one of the best classes out there. Due to my language issue, I was falling behind and almost lost faith in myself. Chris may have noticed that and gave me some encouragements that I desperately needed at that moment. I really appreciate the help. I also want to thank Chris for the many things he did for the CSL (Computer Systems Lab) com-

munity, which was really not an easy effort. Last but not least, I would like to thank Professor Andrew Myers. It was an honor to work and interact with Andrew in the last couple of years in my PhD. I enjoyed every discussion we had on our research collaborations. I also want to thank Andrew for being very understanding and encouraging, providing help and support when I needed.

I would like to thank all CSL professors. Special thanks to Professor Dave Albonesi. His class of memory systems was a lot of fun and gave me confidence in myself. Another special thanks to Professor Jose Martinez for teaching me the interesting and painful cache coherence protocols. I also want to thank Professor Zhiru Zhang for his help and guidance. Outside of CSL, I was fortunate to interact with some of the best professors in Cornell University and I thank all of them. I specially want to thank Professor Kevin Tang for his guidance in my first couple of years in graduate school.

I felt quite honored to be part of the CSL community. During my six years of PhD studies, I witnessed senior students graduating and new faces showing up, many of whom have helped me enormously in this long journey. I want to thank everyone in Suh Research Group for contributing to my research ideas. It was a great pleasure collaborating with these guys. I want to thank Ruirui Huang and KK Yu for being such good friends and sharing their PhD experiences as senior students. I also would like to thank Daniel Lo for generously helping me with technical questions and providing insightful suggestions throughout my PhD studies. Special thanks to Andrew Ferraiuolo and Benjamin Wu, whom I have the privilege to work with. Thanks also goes to Tao, of course, for sharing ideas on new technologies and providing great suggestions all the time. I also thank Rui Xu for his pioneer work on using SecVerilog. Outside of my research group, I was grateful to meet some of the best people in my life. Saugata Ghose was

always ready to provide help whenever I needed. He was such a reliable friend. The same goes to Shreesha Srinath. I enjoyed every lunch conversation we had and of course, thank you so much for letting me stay in your house before my thesis defense. I also want to thank Rob Karmarzin for being such a trustworthy officemate and introducing my cat, BaBao, to me and my wife. Special thanks to Maia Kelner, who very patiently taught me how to drive and familiarized me with American culture. There were so many people who have helped me along the way that it will be overly long if I enumerate all of them—Jon Tse, Ben Hill, Xiaodong Wang, Ji Kim, Berkin Ilbeyi, just to name a few. I wish all the students in CSL can succeed in their future endeavors.

I also had the opportunities to work with some brilliant students outside of CSL. Danfeng Zhang, who is now a professor in Pennsylvania State University, had collaboration with me on several research projects. I enjoyed our collaboration a lot and also learned about language-based security from him. I would like to also thank Nithin Micheal, whom I was grateful to work with in my early research projects.

Finally, I wish to express my deepest thanks to my family. My parents, Sixue Wang and Mingyao Qian, taught me the importance of self-discipline and hard-working, which got me this far in my academic journey. Thanks so much for raising me up and supporting me in all these years. My wife, Xi Yan, has always been by my side and backing me up. She is the one who gives me the strength and courage to overcome any difficulties I may face. I love her with all my heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Multi-core processors are widely used in modern computing systems such as cloud servers and mobile devices. In the architecture of multi-core processors, some hardware resources are often shared by multiple cores to improve hardware efficiency. Each core can run a different application and contends for the shared resources such as caches, on-chip networks and memory. When competing for the shared hardware resources, applications can cause interference to each other, hence affecting individual program's execution time. This timing interference, unfortunately, can sometimes lead to timing channels, which are a type of an information channel that leaks confidential information through timing. Attacker applications can exploit timing channels to launch attacks to steal confidential information from victim applications. For example, previous work [RTSS09] has demonstrated a timing channel attack that reveals the keystrokes typed by a user through shared data caches on commercial Amazon EC2 servers.

Timing channel attacks introduce serious threats to modern computing systems. In cloud computing, a user's applications are scheduled to run concurrently with other applications, which could be malicious and try to infer confidential information that belongs to the user through timing channel attacks. A similar problem exists in mobile devices today. If a user downloads an application that contains malware, this malicious application might run concurrently with the user's banking application, stealing confidential information such as passwords. Timing channel attacks are hard to defend against because they do not rely on physical access to the system and can bypass to-

day's security defense mechanisms such as access control. Even worse, a recent study [HKR+15] shows that timing channel attacks can also achieve a high leakage rate (500Kbps).

There exists extensive work on various hardware timing channel attacks and their countermeasures. However, most of the previous work focuses on cache timing channels. In this thesis, we explore possible timing channel attacks and their countermeasures in shared hardware resources including not only caches, but also on-chip networks and memory controllers. We find all these shared resources are vulnerable to timing channel attacks, and we develop efficient protection schemes that defeat these timing channel attacks. We further improve the performance of our protection schemes by introducing uni-directional protection when programs have asymmetric security requirements, or by trading off security for performance. Finally, we develop two processors in RTL and formally verify that both designs are free of timing channels using a tool called SecVerilog [ZWSM15].

1.1 A Case Study on Realistic Timing Channel Attacks

Previous work have proposed numerous timing channel attacks that exploit software or hardware vulnerabilities. Although many of these attacks were demonstrated in a confined environment, quite a few studies conducted their attacks in realistic environment settings including mobile devices and commercial VMs in cloud computing. These realistic timing channel attacks provide strong evidence that timing channel attacks are not only theoretically possible, but also practically viable as a way to leak confidential information. One notable work is a successful cache timing channel attack demonstrated on Amazon's

EC2 cloud computing platform by Ristenpart et al. [RTSS09]. This section reviews Ristenpart's attack in more detail and also discusses other realistic timing channel attacks.

1.1.1 Ristenpart's Attack on Amazon's EC2

As the cloud computing infrastructure continues to quickly evolve, more and more entities begin to export their computations and data into the cloud. While cloud computing brings several appealing benefits including economies of scale and dynamic provisioning, it also introduces new security threats as the computation and data of different entities (possibly rivals) could share the same physical machine. Traditional virtualization techniques (virtual machines) provides some degree of isolation between different entities, but does not offer complete non-interference because attacks (e.g., side-channel attacks) can bypass the boundaries between VMs. By far the best known example of a third-party compute cloud is Amazon's EC2, which is the target of the timing channel attack demonstrated by Ristenpart et al. [RTSS09]

The proposed attack requires three steps to succeed. First, the attacker finds out the location of the victim VM in the cloud infrastructure to narrow down the search space. Second, the attacker tries to gain co-residence with the victim VM by spawning a set of VMs and checking for co-residence. Co-residence means the attacker VM and the victim VM reside on the same physical machine, which is the basis for many timing channel attacks. Lastly, the attacker VM extracts confidential information about the victim VM via a cross-VM attack, taking the

form of timing channel attacks in this case. The three steps are described in more detail below.

Determining the Location of the Victim VM

When a user launches a VM, the VM is assigned to a single physical machine within the EC2 network for its lifetime. At the time of this attack being proposed, Amazon's machines were located in two regions, one in the United States and the other in Europe. Each region contains three availability zones. A user can specify a region and an availability zone to launch his or her VMs. Meanwhile, a user can also specify an "instance type" of the VM. Different instance types provide different computational power, memory and persistent storage. There were five Linux instance types, referred to as 'm1.small', 'c1.medium', 'm1.large', 'm1.xlarge', and 'c1.xlarge'.

In this step of the proposed timing channel attack, the attacker tries to determine the location (region and availability zone) of the victim VM in order to narrow down the search space of gaining co-residence. To achieve this goal, Ristenpart et al. did a survey study on instance mapping in Amazon EC2. They iteratively launched 20 VMs for each of the 15 availability zone/instance pairs and studied the relationship between the internal IP address of a VM and its zone or instance type.

Figure 1.1 shows the internal IP address mapping regarding different availability zones. The results suggest that the Amazon EC2 internal IP address is cleanly partitioned between availability zones. For example, samples from Zone 3 were assigned addresses within the range between 10.252.0.0/16 and

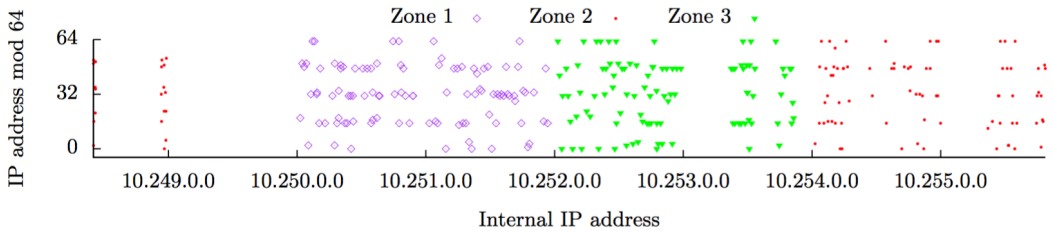


Figure 1.1: Internal IP address mapping regarding availability zones [RTSS09].

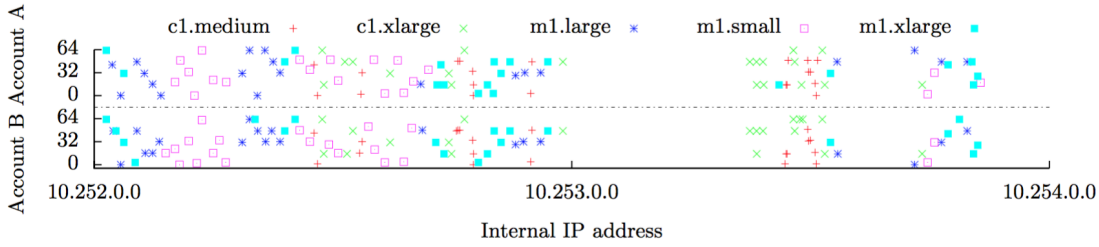


Figure 1.2: Internal IP address mapping regarding instance types [RTSS09].

10.253.0.0/16. While it is not surprising that availability zones show disjoint IP assignment, the experiments also shows a VM's instance type affects its IP address with considerable regularity. Figure 1.2 shows the internal IP address mapping regarding different instance types. This mapping study suggests that all /24 addresses between two consecutive Dom0 /24 addresses inherit the former's associated instance type. Here, Dom0 is a privileged virtual machine that is used to manage guest images, their physical resource provisioning, and any access control rights in Xen hypervisor [BDF+03]. For example, if one Dom0 VM is associated with m1.small with an IP address of 10.254.8.0/24 and the next Dom0 VM is associated with c1.medium with an IP address of 10.254.11.0/24, then any VMs that have IPs in prefixes 10.254.9.0/24 and 10.254.10.0/24 are associated with m1.small instance type.

The derived IP address mapping rules can dramatically reduce the number of instances needed before a co-resident placement is achieved. Basically, an attacker can enumerate public EC2-based web servers using external probes and translate responsive public IPs to internal IPs via DNS queries within EC2. With the internal IP address of a victim VM and the derived mapping rules described above, the attacker can figure out the availability zone and instance type of the victim VM. The attacker can then launch its VMs of the same instance type in the same availability zone to increase the chance of gaining co-residence.

Gaining Co-Residence with Victim VMs

Ristenpart et al. proposed several easy-to-implement network-based co-residence checks. Namely, instances are likely co-resident if they pass one of the checks listed below.

- Matching Dom0 IP address
- Small packet round-trip times
- Numerically close internal IP addresses (e.g., within 7)

As a matter of fact, an instance's network packet's first hop is always the Dom0 privileged VM. For the first check, an attacker can determine its own Dom0 IP from the first hop of any packet, and determine the Dom0 IP of another instance by performing a TCP SYN traceroute to the instance and inspecting the last hop. The second check is based on the fact that the packet round-trip times are usually smaller when two VMs are co-resident. The basis of the third check is that the same Dom0 IP will be shared by instances with a continuous sequence

of internal IP addresses. These checks were verified by comparing to the results of a hard-disk-based covert channel between EC2 instances and showed almost 100% accuracy.

Two strategies were proposed to gain co-residence with victim VMs. The brute-force strategy has an attacker launch many instances over time and targets on a set of victim VMs. In this strategy, the attacker repeatedly runs probe instances of the target type in the target zone, using the information gathered from the first step. Each probe instance performs the co-residence checks to determine if it is co-resident with any of the victim VMs. 1785 probe instances were launched in their experiments, which results in co-residency with 141 victim VMs out of 1686 target victims. Thus this strategy achieved 8.4% coverage of the target victim set. The second strategy exploits *placement locality* in Amazon EC2. Basically, if two instances run sequentially (the first terminated before launching the second) or run at around the same time, they are likely to be assigned to the same machine. In this attack strategy, the attacker detects when a victim VM is launched, and then engages in *instance flooding*: running as many instances in parallel as possible in the same availability zone and of the same instance type. Because of the placement locality exhibited by the EC2 placement algorithm, the attacker is able to achieve co-residence with a target victim VM 40% of the time (launching just 20 probe instances)—a very high successful rate. The assumption that the attacker can detect when a victim VM is launched is based on that an attacker can monitor a server's state by constant network probing. More interestingly, an attacker might be able to trigger new victim instances due to the use of auto scaling systems. The success of gaining co-residence with victim VMs was even demonstrated on commercial instances from companies including RightScale and rPath.

Carrying Out Timing Channel Attacks

Once an attacker places its VM on the same physical machine as a victim VM, he or she can extract confidential information (e.g., cryptographic keys) from the victim VM through various timing channel attacks. Ristenpart et al. demonstrated a cache-based timing channel attack that allows an attacker to measure the cache usage of a victim VM that shares the same cache with the attacker VM. The attack can also be used to estimate the traffic rates of a co-resident web server, which may correlate with confidential information. Both attacks were demonstrated in the Amazon EC2 environment, which represents the realistic environment settings in cloud computing. Although the demonstrated attacks were relatively simple, the main contribution of this work is that it shows a practical attack step by step in the real world. Since attackers can easily gain co-residence with victim VMs, various timing channel attacks that are based on hardware resource sharing (including the ones that will be described in this thesis) become viable to the attackers.

1.1.2 Other Realistic Timing Channel Attacks

As an extension to Ristenpart's attacks [RTSS09], Zhang et al. [ZJRR12] proposed a more fine-grained cache-based timing channel attack between VMs that share the same physical machine. In this attack, the attacker is able to probe the cache very frequently by abusing interprocess interrupts (IPIs), and filter out the noise due to hardware and software features. The fine-grained cache probing allows the attacker to extract a private ElGamal decryption key [EG85] from a co-resident victim VM. The attack was demonstrated on an Intel Core 2 Q9650

processor. It shows that probing over the course of just a few hours can provide the attacker enough information to reconstruct the victim's 457-bit private key with high accuracy—fewer than 10,000 possible keys need to be searched to find the right one!

Another line of work on timing channel attacks exploits timing difference in write accesses on deduplicated memory pages. Memory deduplication is a technique to reduce the memory footprint of a system by combining identical pages. However, write access to these deduplicated pages leads to a page fault, which takes much longer to process than write access to normal pages. This timing difference serves as the basis for several timing channel attacks. Suzaki et al. [SIYA11] proposed an attack that can determine whether specific applications are running in a co-located virtual machine in the cloud. Owen et al. [OW11] demonstrated that it is possible to efficiently fingerprint operating systems of co-resident virtual machines using memory deduplication attacks. Attackers can use OS fingerprinting to learn the type and version of an operating system so that they can launch attacks that are specific to the target OS. Gruss et al. [GBM15] presented a page deduplication attack in sandboxed JavaScript, which allows a remote attacker to collect private information such as whether a program or website is currently opened by a user. The proposed countermeasure was to disable page deduplication completely.

Many of the timing channel attacks take the form of passive attacks, which means the attacker can gain information about the victim system without noticeably affecting the behavior of the target system [ZF05]. Passive attacks are quite difficult to detect, and they can result in hazards to security and privacy. Because of these potential attacks, many companies refuse to outsource their com-

putational tasks to public computing infrastructures. According to a recent IDCI survey, 74% of IT executives and CIOs cited security as the top challenge preventing their adoption of cloud services [SK11b]. It is thus important to build a secure computing environment that can defeat various kinds of attacks, including timing channel attacks. However, current countermeasures against timing channel attacks suffer from either inflexibility or high performance overhead, which make them hard to deploy in practice. As a result, timing channel attacks remain to be a continuing threat in computing systems in the near future. This thesis explores some flexible and efficient timing channel protection schemes as a first step towards building systems that are invulnerable to timing channel attacks.

1.2 Threat Model

To defend against timing channel attacks, we first describe our threat model, which defines security objectives and identifies vulnerabilities and threats. On a high level, our threat model assumes a multi-core processor as the target platform and a hierarchical security policy.

1.2.1 Baseline Architecture

The timing channel attacks and protection schemes in this thesis target a multi-core processor, as shown in Figure 1.3. The processor has multiple in-order or out-of-order cores. All cores share a large last-level cache (LLC), which is connected to the private caches through on-chip networks. The networks can

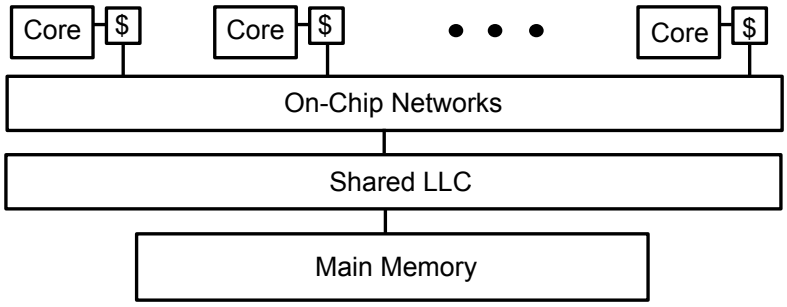


Figure 1.3: Baseline architecture.

be any topology that is applicable, such as bus, ring or mesh networks. The LLC is connected to a memory controller, which handles memory requests and responses.

In this multi-core architecture, three major hardware components (on-chip networks, LLC, memory controller) are shared among all the cores. Since each core can run a different application, a malicious application can steal information from another application through timing channel in these shared components. This thesis focuses on discovering possible timing channel attacks and developing their countermeasures in these shared components.

1.2.2 Hierarchical Security Policy

Our threat model considers timing channel protection among security domains. A security domain is defined as a set of software modules (VMs, processes, threads) that do not need timing isolation among them. As an example, consider the cloud computing platform in Figure 1.4. A security domain may contain multiple VMs if these VMs belong to the same user. The hypervisor/OS is

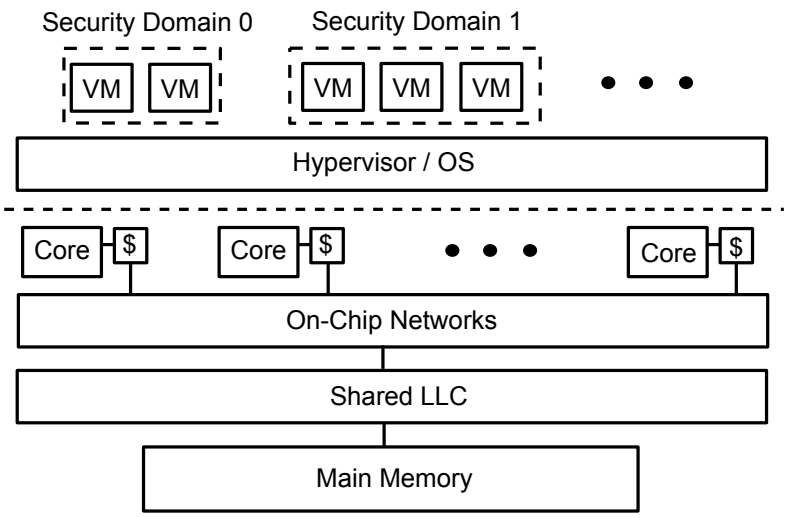


Figure 1.4: Security domains in cloud computing.

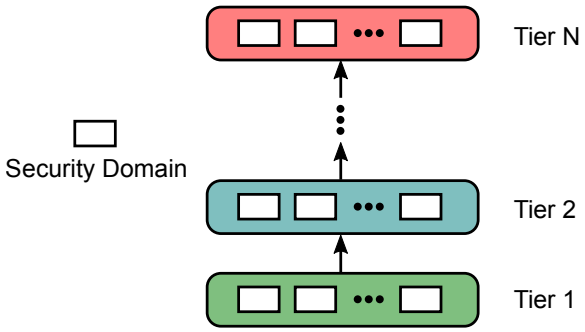


Figure 1.5: Hierarchical security policy.

responsible for scheduling these VMs to the underlying processor and tagging each VM with the correct security domain ID.

Different security domains may pose different security requirements. This thesis assumes a hierarchical security policy that defines the relationship between security domains, as shown in Figure 1.5. There are N security tiers, and each tier contains an arbitrary number of security domains. The security policy follows the rule below:

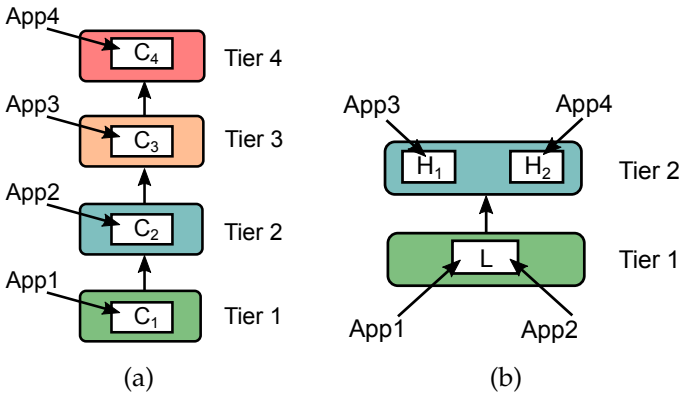


Figure 1.6: Common security policies.

- Information is allowed to flow from security domain A to security domain B if and only if $A \in$ Tier i, $B \in$ Tier j and $i < j$.

The security policy has two implications. First, information may not flow between two security domains in the same security tier. This is useful for the use case where multiple mutually distrusting high security applications (e.g., banking, medical) are running concurrently. Second, information **is** allowed to flow from a lower security tier to a higher security tier. This represents the use case where low security applications (e.g., simple games) are running together with high security applications. Information is allowed to flow from low security applications to high security applications, but not the other way around.

The security policy is general enough to support common use cases. For example, a government or military database system usually employs the multi-level security (MLS) policy, which consists of four hierarchical levels. This MLS policy can be represented in our security policy by four security tiers with each tier containing one security domain, as shown in Figure 1.6(a). The security policy can also represent the mobile scenario with two security tiers, as shown in Figure 1.6(b). Tier 1 only has one security domain L, which contains pub-

lic applications. Tier 2 has two security domains, with each security domain containing a different high security application.

1.2.3 Attack Assumptions

Timing channel attacks can take many different forms. In general, timing channel attacks can be divided into software-based attacks and hardware-based attacks. Software-based attacks exploit the vulnerabilities in software, while hardware-based attacks rely on the microarchitectural features of a processor. This thesis focuses on defending against hardware-based timing channel attacks. More specifically, we consider attacks that exploit interference in shared hardware resources between **concurrently running** programs. Hardware timing channels within a program or caused by context switches are out of the scope of this thesis. As an exception, a couple of secure cache designs are proposed to defeat cache timing channels within a program in Section 3.6.

Both unintentional and intentional information leaks are considered in this threat model. Thus, I aim to prevent both side-channel and covert-channel attacks. The goal of a side-channel attack is to gain access to the secret information possessed by the victim, which does not intend to leak the secret. The attacker can intentionally create interference in the shared hardware resources and make timing measurements on its own operations. The attacker then tries to correlate the timing information with the secret. In a covert-channel attack, the attacker already possesses a secret, but is limited in how it can share this secret. For example, a malicious 3rd party web application may try to leak a user's data when the cloud infrastructure restricts its network connections so that it cannot

directly communicate with malicious servers. The attacker can try to bypass such restrictions using a timing channel to communicate with a co-resident VM whose network connection is not restricted. The attacker can communicate the secret by deliberately modifying its workload to cause timing variations to the colluding attack program's execution.

The threat model assumes a strong attacker. The attacker can generate any pattern of cache accesses as well as memory requests, which gives the attacker the capability to create interference that favors timing channel attacks in the shared resources. Another assumption is that the attacker is able to measure the exact timing of its own memory accesses so that it can analyze the timing information with high accuracy. On the other hand, the hypervisor/OS is assumed to be trusted and not compromised by the attacker. The hardware counters are controlled by the hypervisor/OS and are not accessible by the attackers. For most of this thesis except for Section 3.6, we assume different security domains do not share data that can be both read and written. However, different security domains are allowed to share read-only data (e.g., program code, shared library.).

1.3 Limitations of Software-Only Solutions

One approach to protect against timing channel attacks is to write or rewrite software in a way that defeats known timing channel vulnerabilities. However, this approach has several limitations. First, software solutions are often designed to deal with only specific timing channel attacks. The software written with a timing channel protection method is still vulnerable to new timing

channels that have not yet been revealed. For example, the Montgomery multiplication [Mon85] and Chinese Remainder Theorem implementation of Diffie-Hellman [DH06] or RSA [RSA78] were believed to defeat the timing attack designed by Kocher [Koc96a]. However, this implementation is shown to be prone to a later attack proposed by Brumley et al. [BB03]. The second limitation of software solutions is the lack of precise control on execution time. To illustrate the idea, consider a simple code snippet below.

```
1 Victim:
2       int secret, h1, h2, l1;
3       init(secret, h1, h2, l1);
4
5       if (secret)
6           h1 = l1
7
8       h2 = h1
```

If the value of *secret* is 1, line 6 is executed. Otherwise, line 6 is skipped. The timing difference reveals the value of *secret*—a timing channel. One naive solution to eliminate the timing channel is adding dummy operations to balance the control flow, as shown below.

```
1  Victim:
2      int secret, h1, h2, l1, l2;
3      init(secret, h1, h2, l1, l2);
4
5      if (secret)
6          h1 = l1
7      else
8          l2 = l1
9
10     h2 = h1
```

At first glance, this software fix seems to work because both branches of the *if* statement have the same number of instructions. However, it ignores the underlying hardware features such as caches. If the value of *secret* is 1, $h1$ is brought into a cache, hence the execution time of line 10 is short because of a cache hit. If the value of *secret* is 0, the access to $h1$ in line 10 becomes a cache miss. The timing difference between a cache hit and a cache miss represents a timing channel that can reveal the value of *secret*.

The timing channel shown in the example above is caused by the cache interference between different instructions within the same program. Timing channels may also be introduced by interferences between different processes in shared hardware resources. For example, a victim program's cache accesses may evict cache lines of an attacker in a shared LLC, which allows the attacker to infer confidential information about the victim. It is also possible that a victim's memory requests delay the memory requests of an attacker when competing for the same memory channel, resulting in illegal information leakage. Many soft-

ware solutions are prone to similar timing channels because software cannot strictly control the execution time of each instruction.

Software-only solutions are insufficient to defend against these micro-architectural timing channel attacks due to the limitations discussed above. Protection mechanisms must also be implemented in hardware to completely eliminate the sources of timing channels. To this purpose, my thesis will focus on designing efficient timing channel protection mechanisms in hardware.

1.4 Thesis Contribution and Organization

This thesis proposes timing channel protection mechanisms for shared hardware resources in a multi-core processor. The goal of these protections is to defend against timing channel attacks between concurrently running programs while minimizing the performance overhead introduced by the protection. Depending on the specific security policy, timing channel protections in this thesis are divided into three approaches.

If two security domains are mutually distrusting (i.e., they are in the same security tier), the protection scheme needs to ensure that information does not flow in either direction. This kind of protection is called **bi-directional protection**. Bi-directional protection requires complete noninterference between security domains, which poses restrictions on how the shared resources can be used by different security domains and can significantly degrade performance.

On the other hand, if two security domains are not in the same security tier (e.g., L and H_1 in Figure 1.6(b)), the protection scheme only needs to guarantee

the information does not flow from the higher security tier to the lower one. This kind of protection is called **uni-directional protection**. Uni-directional protection allows the runtime information from the security domain in a lower security tier to be used to decide how the hardware resources are allocated to higher tiers. With the help of this runtime information, uni-directional protection can generally provide more efficient resource allocations or scheduling decisions compared to bi-directional protection, as will be shown in later chapters.

Both bi-directional and uni-directional protections prevent any illegal information flows according to the security policy, i.e., they enforce zero information leakage. However, users sometimes are willing to sacrifice security for better performance. A third approach to defend against timing channel attacks allows controllable amount of information leakage between security domains while trying to maximizing the system performance. This approach enables a trade-off between security and performance, which allows the users to tune the performance of their systems based on the amount of information leakage they can tolerate.

In the rest of this thesis, all three design approaches are used to develop efficient timing channel protection schemes for the shared hardware resources in a multi-core processor. The primary contributions of this thesis are summarized below:

- We construct a timing channel attack against on-chip networks. We propose a bi-directional protection scheme called Temporal Network Partitioning (TNP), as well as a uni-directional protection scheme called Reverse Priority with Static Limit (RPSL). Both schemes eliminate the timing channels in on-chip networks.

- We identify timing channel attacks against caches and propose several protection schemes. We found that timing channel attacks are possible through cache coherence protocols and defeat the attacks with simple time-division multiplexing mechanisms. We develop a secure dynamic cache partitioning scheme (SecDCP) that defeats cache timing channel attacks with uni-directional protection. SecDCP improves performance by up to 43% and by an average of 12.5% over static cache partitioning. Furthermore, we propose a secure cache design based on a high-associativity cache (ZCache [SK10]) to support more security domains. We also find a possible attack caused by the relocation process of ZCache. To defend against timing channels within a program, we propose a couple of secure cache designs (DirtyCache, RelCache) that supports software security labels.
- We construct timing channel attacks against shared memory controllers. We propose two completely secure memory controller designs (TP and SecMC-NI). SecMC-NI improves the performance of the best known scheme by 45%. To further improve the performance, we also propose a memory controller that enables a tradeoff between performance and security (SecMC-Bound).
- We implement some of the protection mechanisms in RTL. Two secure processors are developed and verified using a tool called SecVerilog [ZWSM15]. The results show that the protection mechanisms indeed remove timing channels at the gate level.

Table 1.1 summarizes the contributions of this thesis. The rest of the thesis is organized as follows. Chapter 2 describes our findings on new timing

Hardware	Bi-directional Protection	Uni-directional Protection	Performance and Security Tradeoff
On-Chip Networks	TNP	RPSL	
Caches	ZCache-Based Protection	SecDCP DirtyCache RelCache	
Memory Controllers	TP SecMC-NI		SecMC-Bound

Table 1.1: Primary contributions of this thesis.

channel attacks against on-chip networks and our protection schemes using bi-directional and uni-directional protection. Chapter 3 discusses previous timing channel attacks on caches and introduces our secure cache designs. Chapter 4 presents our studies on new memory controller timing channel attacks and protection mechanisms. Chapter 5 discusses our experience of designing two secure processors in RTL and verifying their timing channel security using SecVerilog. Finally, Chapter 6 summarizes related work and Chapter 7 concludes the thesis and discusses future directions.

1.5 Collaboration, Previous Publications, and Funding

This work would not have been possible without the effort of all my collaborators. My advisor, Professor Edward Suh, was the key collaborator in brainstorming all the timing channel protection ideas. Andrew Ferraiuolo helped develop the simulation infrastructure needed to evaluate the memory controller protection schemes. His expertise in timing channel protection also helped me quickly identify security vulnerabilities in the early design of protection schemes for caches and memory controllers. The hierarchical security policy

was also inspired by Andrew's work on Lattice Priority Scheduling [FWZ+16]. Danfeng Zhang is the creator of SecVerilog, which is the tool that allows me to verify the security of a hardware design down to the gate level. His work on language-based protection framework inspires my work on the secure cache design that supports software security labels. Danfeng also helped evolving the secure cache design as well as the secure dynamic cache partitioning scheme, SecDCP. Professor Andrew Myers was a great resource for insights into developing efficient timing channel protections and applying SecVerilog in hardware designs. Andrew also offers a lot of educational suggestions in terms of paper writing. I would also like to acknowledge Rui Xu for his work in developing secure processors using SecVerilog. His experience in SecVerilog helped me quickly solve difficulties I encountered when verifying secure processors with SecVerilog. I also want to thank Professor Christopher Batten and his group (Ji Kim, Shreesha Srinath, Chris Torng, and Berkin Ilbeky) for providing me with the source code of the PARC-tiny processor for RTL verification.

This thesis includes my work of several previous publications, including timing channel attacks and protections for on-chip networks [WS12], caches [WFZ+16] and memory controllers [WFS14, WWS17]. I helped Danfeng Zhang to test and validate the design of SecVerilog [ZWSM15] by applying SecVerilog to the design of a secure MIPS processor. The results of the MIPS processor design are included in this thesis. I also helped Andrew Ferraiuolo to develop a uni-directional protection scheme for memory controllers called Lattice Priority Scheduling [FWZ+16], which is not included in this thesis.

In terms of funding, the work in this thesis was supported in part by NSF CNS-0746913, NSF CNS-0708788, NSF CNS-0905208, NSF CNS-0746913,

NSF CCF-0905208, NSF CNS-1513797, AF FA8750-11-2-0025, ONR N00014-11-1-0110, ARO W911NF-11-1-0082, and donations from Intel Corporation.

CHAPTER 2

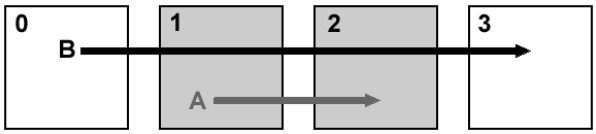
ON-CHIP NETWORKS

This chapter discusses timing channel attacks and protection schemes for shared on-chip networks. Section 2.1 describes timing channel attacks we found in shared on-chip networks. Section 2.2 describes a bi-directional protection scheme based on Temporal Network Partitioning (TNP). We then propose a more efficient uni-directional protection scheme called Reverse Priority with Static Limits (RPSL) in Section 2.3.

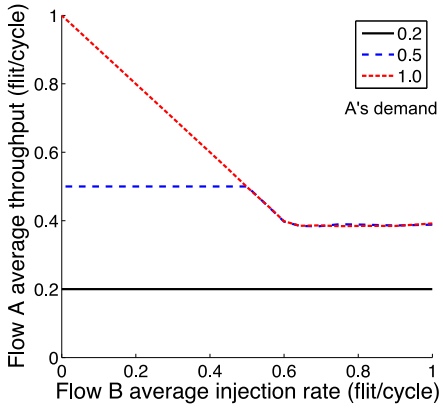
2.1 Attacks

2.1.1 On-chip Network Interference

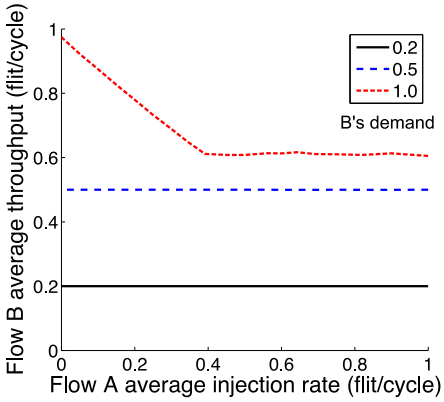
On-chip network interference happens when network flows from multiple applications contend for shared resources such as links and buffers. As an example, Figure 2.1(a) illustrates a simple scenario where two flows, Flow *A* and Flow *B*, share a common link between Node 1 and 2 on a typical 1-D mesh with 4 virtual channels. Because only one flow can use the link in each cycle, the network performance of one flow, such as throughput and latency, can be affected by the other flow's demand. Figure 2.1(b) and Figure 2.1(c) show the throughput of Flow *A* and *B* respectively as a function of the other flow's injection rate, using a cycle-level network simulator called Darsim [LSC+10]. In the experiments, we fixed one flow's injection rate (0.2, 0.5, or 1 flit/cycle) and measured its delivered throughput while varying the other flow's injection rate from 0 to 1 flit/cycle. As shown in the figure, when a flow has a low injection rate (0.2 flit/-



(a) Network and flow setup.



(b) A's throughput, varying B's demand.

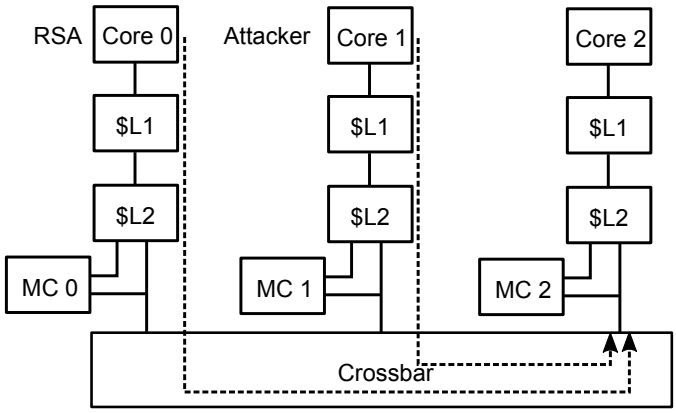


(c) B's throughput, varying A's demand.

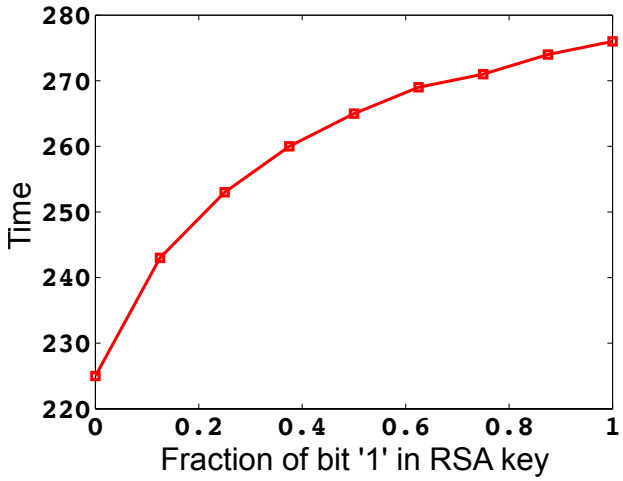
Figure 2.1: A simple network interference example.

cycle), its throughput is not significantly affected by the other flow's injection rate because a round-robin arbitration allocates roughly half of the link capacity to each flow, which is enough to satisfy the demand. However, when the injection rate is high (1 flit/cycle), a flow's throughput is highly dependent on the other flow's injection rate.

This network interference can be exploited as a vector to launch timing channel attacks. Essentially, an attacker can measure its run-time network throughput or latency to infer what the victim is doing (e.g., the victim's run-time network demand) and further correlate this information with the victim's secret.



(a) A platform setup.



(b) Attack program's execution time.

Figure 2.2: A side-channel attack on RSA.

2.1.2 An RSA Attack

As a more concret example, let us consider a timing channel attack on the RSA [RSA78] algorithm, as shown in Figure 2.2. RSA is a public key cryptographic algorithm that is widely used to secure electronic communications through encryption or digital signatures. In one of the RSA implementations, the core algorithm performs a modulo multiplication of two large numbers (often 1024 or 2048 bits) depending on each bit in a secret key, and is shown to be prone to tim-

ing attacks [Koc96b]. Essentially, the algorithm examines each bit in the key and only performs a multiplication if the bit is 1. In this example, the multiplication operation in RSA causes additional network traffic to the memory controller 2 (MC2) due to cache conflicts. Meanwhile, the attack program runs a loop that generates cache misses in every iteration, which also sends a lot of memory requests to MC2. Packets from the RSA program and the attack program share the output port of the crossbar and experience network interference. The experiments were performed using McSim [ALOJ13], which provides timing models for a multi-core platform based on Intel's Pin [LCM+05] tool. Figure 2.2(b) shows the attack program's execution time as the number of 1s in the RSA key varies. As shown in the figure, the execution time has a direct correlation with the fraction of 1s in the key. This result suggests that the attacker can roughly estimate the number of 1s in the key based on its own execution time, which can greatly reduce the search space to find the correct key.

2.2 Temporal Network Partitioning (TNP)

We propose to use temporal network partitioning to defend against timing channel attacks for on-chip networks. In this protection scheme, we statically allocate virtual channels (VCs) in each input port of a router to security domains. We then use oblivious arbitration [DT04] to have security domains take pre-determined turns to use the crossbars and links on a per-cycle basis. In the case of two security domains, each security domain can be allocated with a half of input virtual channels for exclusive use, and be allowed in the switch allocation and link traversal in every other cycle. Effectively, this scheme statically allocates a half of network resources to each domain. Since the allocation is

completely static, the attacker is unable to infer any dynamic demand about the victim using on-chip network interference.

TNP removes a timing channel in both directions between two security domains. However, it can incur significant performance overheads because resource allocation cannot be dynamically adjusted to match the actual network demands from each security domain. For example, if one security domain has a high bandwidth demand while the other domain has a very low demand, TNP only allows the domain with the high demand to use at most half of the network bandwidth. To solve this problem, TNP can be configured to statically allocate more bandwidth to the security domain with higher demand. However, since the bandwidth allocation is always static, TNP cannot efficiently deal with bursts or changes in different phases of an application.

Another bi-directional protection scheme called "SurfNoC" [WGO+13] has been proposed recently to improve the performance of TNP. SurfNoC also builds on top of time division multiplexing the network resources among different security domains, but came up with a better scheduling of security domains in each router, which allows a packet to be forwarded to the next router immediately after it arrives. This smart scheduling avoids the unnecessary delay for a packet to stay in the input buffers and wait for its turn.

2.3 Reverse Priority with Static Limit (RPSL)

TNP completely eliminates timing channels in on-chip networks, but can be quite inefficient because all network resources are statically allocated. In this

section, we propose a more efficient protection scheme that allows dynamic scheduling of network traffic through uni-directional protection.

2.3.1 Priority-Based Arbitration

In the multi-level security model, one goal is to prevent information flows from the high security domain to the low security domain. To achieve this goal without statically allocating resources, we propose a priority-based arbitration for resources such as the router crossbar. The basic idea is to assign a high priority to low security traffic so that its behavior is not affected by high security traffic. In other words, when flows from two security domains compete for the switch traversal, the low security domain always wins, in both input port and output port arbitrations in a separable allocator. To remove interference in buffers, virtual channels are statically allocated to each security domain. This static allocation removes head-of-line blocking between packets from different security domains. In this way, the low security flows are not affected by the dynamic demands of the high security flows. At the same time, this approach allows a security domain to dynamically use more network resources when the other domains have low demands.

2.3.2 Denial-of-Service Protection

While assigning a higher priority to a lower security domain can prevent illegal information leakage without statically restricting network resources, the approach introduces an obvious concern for fairness. In fact, the strictly higher

priority allows a malicious program in the low security domain to easily perform a Denial-of-Service (DoS) attack on high security programs by sending packets at a high injection rate and occupying all network resources.

To prevent the DoS attack, we add an additional mechanism that monitors and limits the traffic amount of the low security domain in a way that is independent from the demand of the high security domain. This mechanism sets a static limit per port on the number of flits that can be sent by the low security domain over a certain time interval. Once the limit is reached, the port does not send the low security flits until the next interval, allowing the high security flits to go through. For example, if we set the limit to be 80 and time interval to be 100, it means the low security flows can at most send 80 flits every 100 cycles. The remaining cycles can only be used by the high security flows. Note that the static limit does not create a timing channel because it does not depend on the high security traffic. Also, the limit can change over time for better performance as long as it does not reflect sensitive information from the high security domain.

2.3.3 Multiple Security Domains

It is relatively straightforward to extend the protection scheme to support more than two security domains. Suppose we have N security domains, and we rank them based on their security tiers. The flows from the lowest security domain get the highest priority to use the router crossbar and vice versa. To extend to N security domains, RPSL needs N-1 static limits to avoid DoS attacks, one for each domain except for the highest security domain. The limits may specify the

maximum bandwidth usage over a certain period for corresponding security domains or the aggregate maximum usage for each security domain and below.

2.3.4 Evaluation

Experimental Setup

We evaluate the proposed protection scheme, named RPSL, compared to the baseline without protection and TNP. The simulations are performed by modifying Darsim [LSC+10]. The network is configured with four virtual channels per port, and uses iSLIP [DT04] as the baseline allocator. We assume a low security domain and a high security domain. For TNP, we allocate half of the virtual channels and switch time slots to each security domain. For RPSL, we allocate half of the virtual channels to each domain while prioritizing the low security flows in arbitration. The static limit in RPSL is set to be 80 flits per 100 cycles. Each packet consists of eight data flits and one head flit. All the experiments are done with a warm-up period of 20,000 cycles, followed by simulation of 100,000 cycles.

Security Evaluation

We study the security of each protection scheme using the simple example shown in Figure 2.1. In the experiments, we assign Flow A to be the low security flow and Flow B to be the high security one. In practice, an attack program will measure its throughput over time while keeping its injection rate high. The variations in the observed throughput is used to obtain information about the

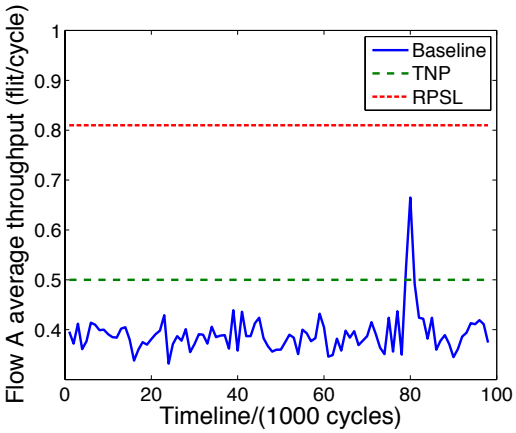


Figure 2.3: Flow A's throughput over time with varying demands from Flow B. Flow A: low security, Flow B: high security.

victim. To mimic such an attack process, we keep Flow A's injection rate at 1 flit/cycle while varying the injection rate of Flow B over time. More specifically, we randomly change the injection rate of Flow B among 1/3, 1/2, and 1 flit/cycle every 1000 cycles.

Figure 2.3 shows the dynamic throughput of Flow A. In the baseline scheme, the throughput of Flow A varies significantly over time, which reflects the dynamic demands of Flow B. This shows the baseline scheme is vulnerable to timing channel attacks. On the other hand, the throughput of Flow A stays constant under TNP and RPSL, regardless of the changing injection rate of Flow B, which shows the information leakage from the high security domain to the low security domain is prevented.

Performance Evaluation

We evaluate the performance of the protection schemes on a 6-by-6 mesh network with two security domains. As shown in Figure 2.4, domain A, which is

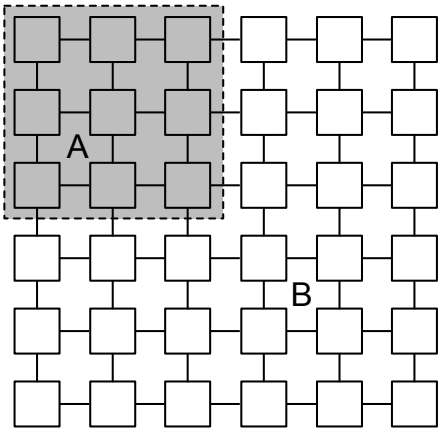
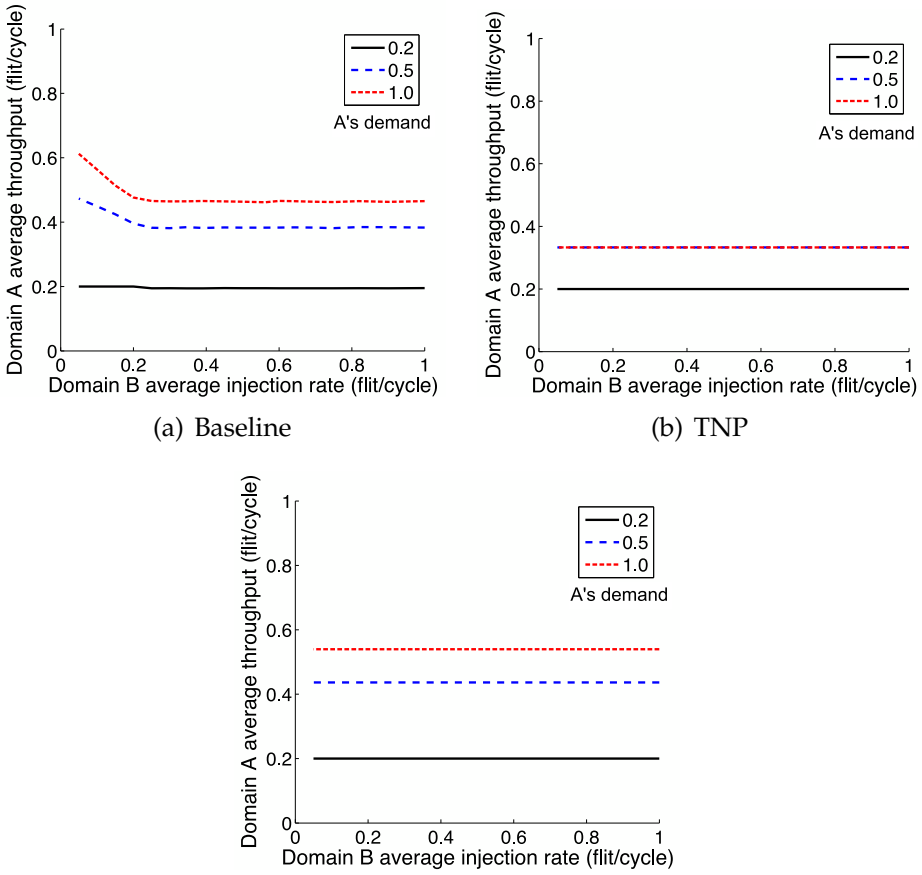


Figure 2.4: Experimental setup in a 6-by-6 mesh network.

the low security domain, was given one-fourth of the cores. We place the cores in contiguous locations because this minimizes the communication cost within a security domain. We simulated a transpose traffic pattern in which a node communicates with the node that is symmetric with respect to the diagonal. We use Y-X routing in this experiment. As a result, flows from different domains share some links in the network, potentially causing interference. We tested three injection rates (0.2, 0.5, 1.0 flit/cycle) for the low security domain. We plot the average throughput of the low security domain while varying the average injection rate of the high security domain, which shows if there exists a timing channel from the high security domain to the low security domain.

Figure 2.5 shows the performance results. As expected, in the baseline scheme, the throughput of domain *A* (low security) changes with the injection rate of domain *B* (high security). In contrast, domain A's throughput stays constant under both TNP and RPSL. Again, this shows both schemes provides an effective protection against timing channels. On the performance side, a distinct difference between the results of TNP and RPSL is that domain *A* achieves higher average throughput in RPSL than in TNP when domain *A*'s injection rate



(a) Baseline

(b) TNP

(c) RPSL (limit = 0.8 flit/cycle)

Figure 2.5: Low security (domain A) throughput as a function of the high security (domain B) demand for transpose traffic pattern.

is high (0.5 or 1 flit/cycle), which shows the performance advantage of RPSL over TNP scheme by allowing more efficient sharing of network resources. We also tested hotspot traffic pattern and the results show similar trends.

The experiments above keep the injection rate constant in each simulation run. To mimic the real application behavior, we simulated the transpose traffic pattern again but randomly vary the injection rate of each flow at run time. We measure the throughput of each domain as well as the aggregate throughput of the system, as shown in Figure 2.6. We normalize the actual throughput over throughput demand, which equals to the number of flits received divided by

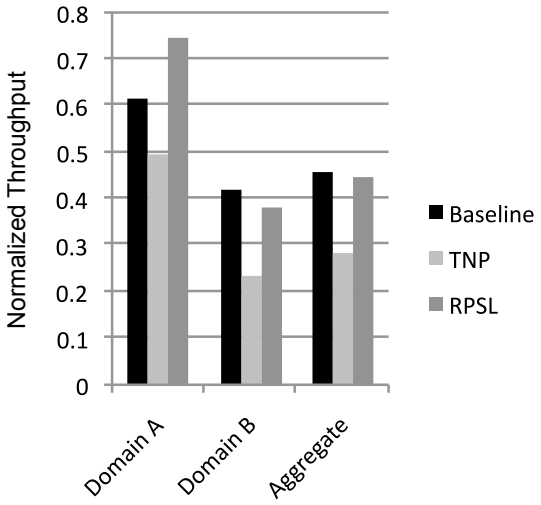


Figure 2.6: Impacts of timing channel protection on the network throughput.

the number of flits injected. For domain *A*, RPSL scheme achieves higher performance than that of the baseline scheme. This is because RPSL gives domain *A* a higher priority to use the router crossbar. For domain *B*, the performance of RPSL is slightly lower than that of the baseline. However, the aggregated performance of RPSL is comparable to the baseline. By contrast, TNP incurs a significant performance overhead due to the static allocation of network bandwidth. RPSL adapts to the dynamic network demands much better than TNP, hence achieving much higher performance.

CHAPTER 3

CACHES

Cache timing channels have been studied extensively in previous work. In this chapter, we will discuss cache timing channel attacks and their countermeasures in detail. Section 3.1 summaries previous cache timing channel attacks. Section 3.2 describes existing work on cache timing channel protections. In Section 3.3, we propose a new cache timing channel attack based on cache coherence protocol and a protection scheme against it. Section 3.4 discusses our uni-directional protection scheme called SecDCP (Secure Dynamic Cache Partitioning). Section 3.5 explores possible timing channel attacks and protections in a high-associativity cache (ZCache) to support more security domains. Section 3.6 describes a couple of cache designs that support software security labels to protect against timing channels within a program.

3.1 Attacks

3.1.1 Cache Interference

Caches are vulnerable to timing channel attacks because cache misses take much longer to finish than cache hits. When multiple programs are running concurrently and sharing the last-level cache, a program's cache lines can be evicted by other programs, causing this program to incur more cache misses and hence experience longer execution time. This cache interference among different programs can be exploited to infer confidential information about a program. Even if a program is running alone, it can still be prone to timing channel attacks

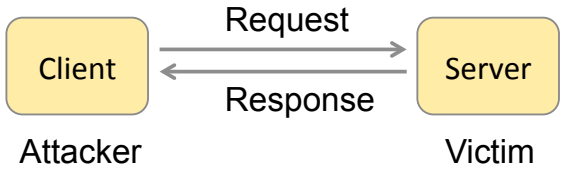


Figure 3.1: Attack model for internal interference.

since its cache lines can evict each other. Given different input sets, the program can show different execution times, which could be used to correlate with the program's secret.

The on-chip network interference is caused by contention in network resources on a per-cycle basis. The scheduling decision in one cycle does not have to affect the decision in the next cycle. This is quite different from the cache interference, which is based on the cache state. After a cache line is fetched into the cache, it will keep affecting the timing of following cache accesses until it is evicted. As a result, the protection schemes for caches work quite differently from the ones for on-chip networks. Before introducing the protection schemes, we first describe existing cache timing channel attacks in more detail. Based on the source of cache interference, we can generally categorize cache timing channel attacks into the ones based on internal interference and the ones based on external interference.

3.1.2 Internal Interference

Attacks based on internal interference exploit the cache interference within a victim program. Figure 3.1 illustrates the attack scenario. In this type of attacks, the attacker sends requests to the victim and measures the timing of responses.

The attacker then performs a timing analysis to recover a secret of the victim. Previous work [Ber05] has successfully demonstrated such an attack on AES encryption algorithm. The pseudocode for the victim program is shown below:

```
1  Victim:
2       k_i = segment of the key;
3       x_i = segment of input message;
4       ...
5       (Table[j] is evicted by the
6            victim code itself)
7       r = Table[k_i ⊕ x_i];
8       (if k_i ⊕ x_i = j, the encryption
9            takes longer than otherwise)
```

In this AES algorithm, AES tables are used to speed up computation. The AES key is used to index into the AES table. However, due to internal interference, some of the table entries are evicted by the victim code itself, hence cache accesses to these table entries become cache misses, which delays the encryption responses to the attacker. Using a known encryption key k_i', the attacker tries different values of input messages and find the one (x_i') that gives longest encryption time. The attacker is able to derive j, which equals to $k_i' \oplus x_i'$. The same process is repeated for an unknown key k_i, and the corresponding x_i can be found out. Using the following equation, the attacker can successfully recover the unknown key k_i.

$$k_i' \oplus x_i' = k_i \oplus x_i \tag{3.1}$$

In general, attacks based on internal interference rely on the fact that the timing of the victim's responses is dependent on the victim's secret. The attacker

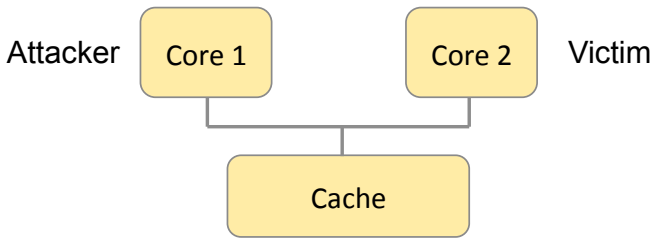


Figure 3.2: Attack model for external interference.

does not have to be on the same physical machine as the victim. Example attacks include Bernstein's attack [Ber05] and cache collision attacks [BM06].

3.1.3 External Interference

Attacks based on external interference exploit the cache interference between different programs. The interference can be caused either by a context switch between two programs on the same core, or by resource contention between programs that are running concurrently in a multi-core processor. We focus on the latter case in this study. Figure 3.2 shows the attack scenario. The cache is shared by the attacker and victim programs, hence one program's cache lines can be evicted by the other program. The attacker is able to detect which cache lines have been evicted by the victim, given that cache misses take much longer time to finish than cache hits. Hence, the attacker can derive the index of cache lines that have been accessed by the victim. Previous work [Per05] has shown that this information is sufficient to recover a part of the AES key of a victim program.

In the attack described above, the information is leaked through the addresses of cache accesses. Other than addresses, we found that the number of

cache accesses can also leak information, which often takes the form of a covert channel attack. In this attack, two attackers collude to communicate a secret through cache interference. Assume two attackers, *A* and *B*, share the cache. Attacker *B* first primes the cache with its own cache lines. After that, attacker *A* either issues many cache accesses to evict attacker *B*'s cache lines to indicate a bit '1', or issues no cache accesses to indicate a bit '0'. By probing the cache again, attacker *B* can learn whether a bit '1' or '0' is sent by attacker *A*.

In general, attacks based on external interference rely on the fact that the victim evicts the attacker's cache lines or accesses shared cache lines based on secret information. The attacker can detect the evictions of its own cache lines or the accesses of shared cache lines through timing and then infer the secret information. The attacker must reside on the same physical machine as the victim to make this attack succeed. Example attacks include prime-probe attacks [Per05, OST06], evict-time attacks [OST06] and flush-reload attacks [GBK11].

3.2 Previous Protection Schemes

Previous work proposes many protection schemes to defend against cache timing channel attacks. Software solutions [OST06, BGNS06] rely on rewriting the software to remove known timing channels. They are ad hoc and new attacks may be possible after the software is rewritten. For instance, RSA implementation using Montgomery multiplication and Chinese Remainder Theorem was believed to be secure against timing channel attacks, but was proven wrong by later work [BB03]. Furthermore, software solutions usually incur large perfor-

mance overhead (2x - 4x slowdown) [BGNS06]. Hardware solutions provide more general and efficient protection by modifying the cache architecture.

Various secure cache designs [Pag05, WL07, WL08, LL14] haven been proposed recently. The approaches taken by these designs fall into two categories: randomization and partitioning. In the randomization approach [WL07, WL08, LL14], the memory-to-cache mapping is randomized to obfuscate the attacker's measurement. Randomization approach essentially adds noise to an attackers observation. A general problem with this approach is its vulnerability to covert channel attacks, which can collect a large number of data samples to remove the noise and recover confidential information. Moreover, randomization only hides which cache line has been accessed by the victim, but does not hide the number of cache accesses. In covert channel attacks, an attacker can intentionally manipulate the number of cache accesses to send confidential information. In the partitioning approach [Pag05], the cache is statically divided into multiple partitions. Each partition can only be used by one process, thereby eliminating the cache interference between a victim and an attacker. However, static cache partitioning incurs high performance overhead because the partition size cannot adapt to the runtime cache demand of each process.

3.3 Cache Coherence

Multi-core processors use cache coherence protocols to allow data to reside in multiple private caches. Unfortunately, we found that coherence operations can lead to a new timing channel, which was not discussed previously. Coherence operations can lead to timing interference through coherence bus contention or

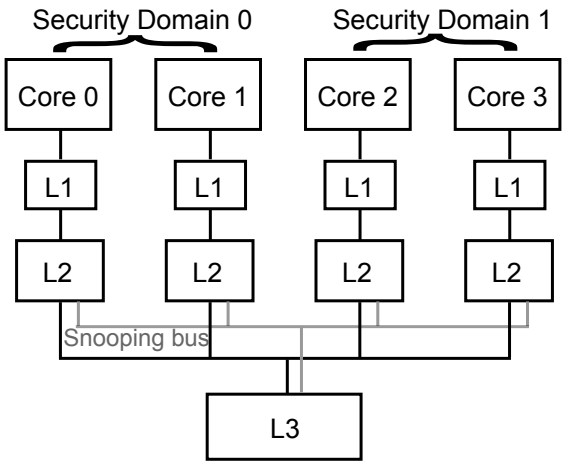


Figure 3.3: System architecture for a cache coherence attack.

contention for cache ports. Even when there is no shared data between security domains, interference on the shared coherence networks can lead to a timing channel.

3.3.1 Cache Coherence Attack

We demonstrate a timing covert channel attack through cache coherence using a simulated 4-core system. The architecture of the system is shown in Figure 3.3. Each core has private L1 and L2 caches, and the four cores share an L3 cache. The four L2 caches are connected with a snooping coherence bus, which uses a MOESI protocol. Note that a similar attack is viable for directory coherence protocol.

We assume two mutually distrusting security domains. Security domain 0 (SD0) runs on core 0 and core 1 while security domain 1 (SD1) runs on core 2 and core 3. We assume different security domains can share read-only data (e.g., program code). SD0 has two threads, each running on a different core.

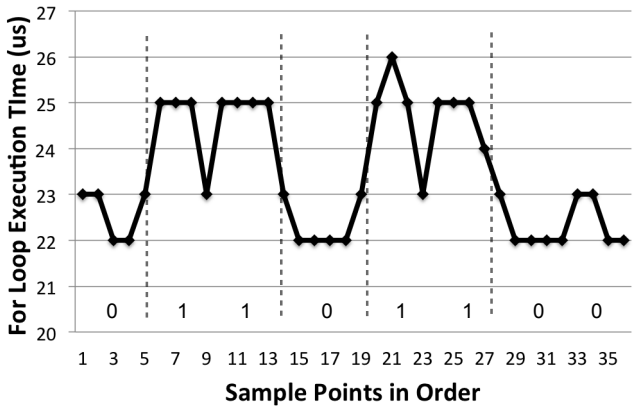


Figure 3.4: SD0's timing observation.

Both threads run a *for* loop, and write to shared data during each iteration. Before each write is performed, one of the L2 caches has to forward the data to the other through the snooping bus and invalidate its own copy. SD0 repeats this process and records the time for each *for* loop execution.

To communicate a secret, SD1 sends a '0' bit by doing nothing and sends a '1' bit by spawning multiple threads that write to shared data. When SD1 does nothing, SD0's cache coherence traffic is not interfered by SD1. When SD1 writes to shared data using multiple threads, SD1 also generates coherence traffic on the bus, which delays the coherence traffic from SD0. As a result, SD0 observes a longer execution time of the *for* loop. Figure 3.4 shows the execution time of the *for* loop that SD0 observes. The execution time has a clear correlation with the secret '01101100' sent by SD1.

3.3.2 Cache Coherence Protection

Cache coherence protocols have two sources of timing interference, namely bus contention and port contention. Similar to on-chip networks, we can use temporal network partitioning to remove interference between security domains. However, protection for cache coherence is more complicated in two aspects.

First, a coherence request from security domain *A* can be sent to a cache that belongs to another security domain *B*. When the receiving cache sends a response (e.g., ack), the response must use *A*'s time slots in the on-chip network. Otherwise, the response can interfere with *B*'s own network traffic, creating an interference from security domain *A* to security domain *B*. Second, in the MOESI protocol, a private cache that has a "Owned" state of a data must forward the data to another cache that requests the data, even if the other cache belongs to a different security domain. This operation requires using the cache pipeline and can delay the requests from the core, which may result in timing interference between security domains. To remove this contention, we modify the coherence protocol so that the data is served from the shared cache instead of the private cache whenever the data is owned by a different security domain. Because different security domains only share read-only data, the shared cache should always has an up-to-date copy.

3.4 Secure Dynamic Cache Partitioning (SecDCP)

3.4.1 Background

Previous work [SDR02, QP06, XL09, SK11a] has studied dynamic cache partitioning techniques to improve cache performance by utilizing the runtime cache demand information of each application to adjust the partition sizes. However, previous dynamic partitioning schemes are prone to timing channel attacks due to the following reasons.

First, the partition size depends on the cache demand of confidential applications. Previous dynamic cache partitioning schemes consider the cache demands of all processes to decide the partition size. This means the partition sizes can reveal the cache demand of each process. By observing the change in its own partition size, an attacker application is able to infer the runtime demand of a confidential application, and further correlate this demand information with a secret.

Second, the change in runtime partition sizes is not strictly enforced. When changing the cache partition size, previous dynamic cache partitioning methods [QP06, XL09, SK11a] enforce the new partition size at replacement time. If the size of a cache partition decreases, its cache lines are not immediately evicted. Instead, they remain in the cache until some cache lines from another process replace them. This means the eviction of one process's cache lines depends on other processes' cache accesses—a vulnerability for timing channel attacks.

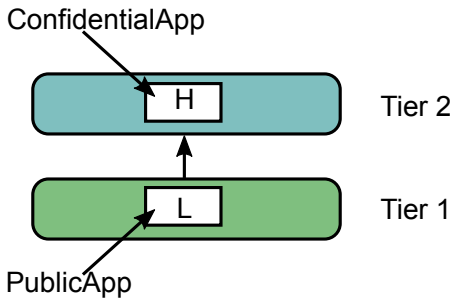


Figure 3.5: Two security domains.

Since dynamic cache partitioning is vulnerable to timing channel attacks, prior secure partitioned cache designs have used static cache partitioning, incurring significant performance overhead. In this section, we show that secure dynamic cache partitioning is feasible under hierarchical security policy (see Figure 1.5). We observe that the hierarchical security policy is inherently asymmetric. We only need to prevent information flow from confidential applications to public applications. Taking advantage of this asymmetry, we propose Secure Dynamic Cache Partitioning (SecDCP), which significantly improves the performance of static cache partitioning while meeting security requirements.

To illustrate the ideas of SecDCP, we first describe SecDCP design for a simple hierarchy with two security domains and then extend it to the general hierarchical security policy.

3.4.2 SecDCP for Two Security Domains

Consider the security policy in Figure 3.5. We assign security domain H to the confidential application and security domain L to the public application. The goal is to prevent information flow from H to L through cache timing channels.

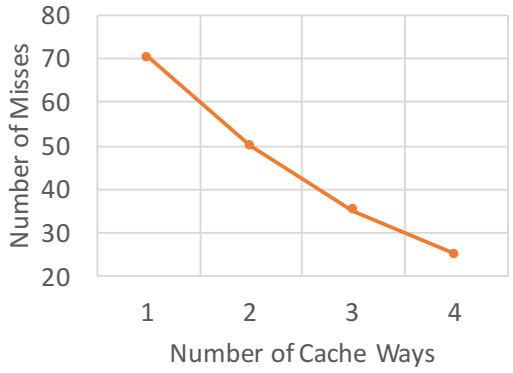


Figure 3.6: Generated miss curve by UMON.

To achieve this goal, SecDCP follows a couple of design rules; 1) partition size is independent of confidential applications, and 2) operations for changing partition size leak no information about confidential applications.

We divide SecDCP into two parts: *Partition Allocation Algorithm (PAA)* and *Partition Enforcement Mechanism (PEM)*. *PAA* is responsible for allocating the size of each partition at run time. *PEM* is responsible for enforcing the new partition size after *PAA* picks a new cache allocation.

Partition Allocation Algorithm (PAA)

Unlike previous dynamic cache partitioning techniques in which the cache allocation is dependent on the demands of both L and H, *PAA* only uses the demand from L to allocate the partition size. We divide the time into epochs of length T. At the end of each epoch, *PAA* allocates a new partition size based on the demand of L in current epoch. Inside each epoch, we use utility monitors (UMON) [QP06] to collect the utility information of L. UMON generates miss curves shown in Figure 3.6. The miss curve is a function that maps the number

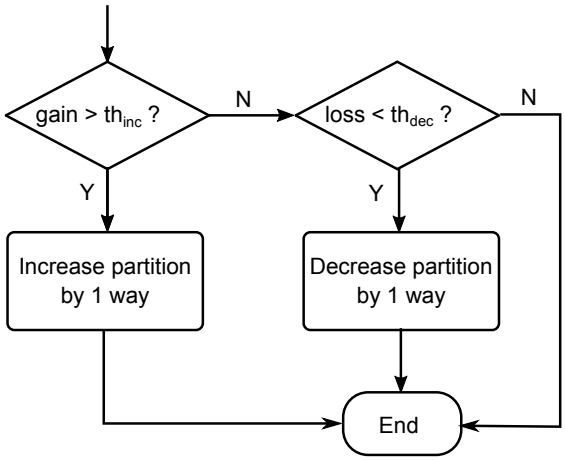


Figure 3.7: Partition Allocation Algorithm.

of cache ways to the number of cache misses. Using the miss curve, we can calculate the gain for increasing the cache size as well as the loss for decreasing the cache size. Assume the current partition size for L is X ways, we define the gain and loss as follows:

$$gain = [MISS(X) - MISS(X+1)]/MISS(X) \tag{3.2}$$

$$loss = [MISS(X-1) - MISS(X)]/MISS(X) \tag{3.3}$$

MISS(X) means the number of cache misses with X cache ways. Gain indicates the percentage of cache misses for L that can be reduced when we increase L's partition size by one cache way. Loss indicates the percentage of additional cache misses for L that will be introduced when we decrease L's partition size by one cache way.

The gain and loss parameters are fed into *PAA*, as shown in Figure 3.7. *PAA* defines two threshold values, th_{inc} and th_{dec}, to determine the new partition size. If the gain is larger than th_{inc}, L's partition size increases by one way. Otherwise, if the loss is smaller than th_{dec}, L's partition size decreases by one way.

Note that *PAA* only considers the demand of L when allocating the partition size. This design choice is made to prevent information leakage from H to L. When L increases its partition size, H's partition size decreases accordingly. Although H may suffer from the reduced partition size, overall system performance can still improve since L will get significant improvement (higher than th_{inc}). To prevent unfairness and starvation, our default design always reserves one cache way for H. On the other hand, when L decreases its partition size, H's partition size increases accordingly. This creates an opportunity for H to improve its performance with more cache ways, while the performance drop of the L partition is bounded (less than th_{dec}).

Partition Enforcement Mechanism (PEM)

After *PAA* allocates a new partition size, PEM enforces the new allocation in a way that leaks no information from H to L. A naive approach is to flush the entire cache way whenever a way is reallocated to another security domain. But flushing can introduce significant performance overhead. PEM improves the performance by treating increases and decreases of partition sizes differently.

We associate each cache line with a 1-bit identifier (*ID*) to indicate which security domain fetched the cache line. When L's partition size decreases, PEM checks the *ID*s and flushes all the cache lines that belong to L in the adjusted cache way before reallocating it to H. This flush is independent of accesses from H, so no information is leaked from H to L. On the other hand, when L's partition size increases, PEM does not flush the cache way that gets reallocated from H to L. As a result, H's cache lines in this cache way can be evicted by L's cache lines, and H can learn about the cache accesses of L. However, this

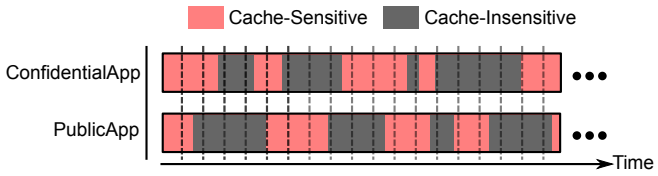


Figure 3.8: Application Phases.

information flow is benign according to our security policy. The optimization improves the performance of H by allowing H to get cache hits in its cache lines in L's partition until L evicts them.

Efficiency Discussion

SecDCP dynamically partitions the cache at run time by utilizing cache demand information from public applications, while previous dynamic cache partitioning schemes such as utility-based partitioning [QP06] use information from all applications to decide partition sizes. How does SecDCP make good partitioning decisions when it only sees information from one side?

It is known that applications often show phase behavior. To simplify analysis, we categorize application phases into cache-sensitive and cache-insensitive phases. Assume a confidential application is running concurrently with a public application, as shown in Figure 3.8. The state of concurrent application phases, (PublicApp, ConfidentialApp), can be divided into four cases: (I, I), (I, S), (S, I), (S, S). Here, I represents a cache-insensitive phase and S a cache-sensitive phase. We analyze performance case-by-case:

- (I, I): The cache partition size does not affect performance much, so SecDCP and utility-based partitioning achieve similar performance.

- (I, S): Both schemes allocate most of the cache to the confidential application because the public application does not need much cache.
- (S, I): Utility-based partitioning will allocate most of the cache to the public application. SecDCP's allocation will achieve the same outcome if the threshold is properly chosen.
- (S, S): Utility-based partitioning will find an allocation that benefits both applications the most, whereas SecDCP's allocation is dependent on the threshold value.

In a summary, SecDCP achieves almost the same performance as utility-based partitioning when the public application is cache-insensitive. For the other two cases, the performance of SecDCP relies on the accuracy of a threshold value. In our experiments, SecDCP uses the same threshold value (20%) across different benchmarks and achieves similar performance as utility-based partitioning. Our analysis and experimental results indicate that using information from one side is enough for dynamic partitioning in many cases.

3.4.3 SecDCP for General Case

We now describe SecDCP for a more general security policy as shown in Figure 1.5, where there are multiple security tiers. It may seem straightforward to extend SecDCP for a general security policy, but this extension turns out to be non-trivial.

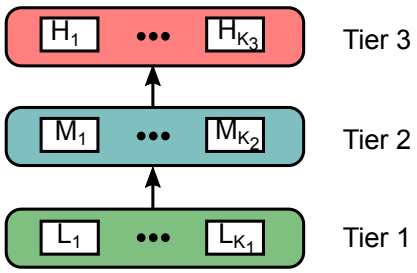


Figure 3.9: Security policy example.

Partition Allocation Algorithm (PAA)

For a general security policy, *PAA* uses UMON to track the cache demand of each security domain separately. Using the allocation algorithm, *PAA* is able to figure out the change in partition size for each security domain. However, a new problem arises when *PAA* tries to reallocate the partition size. In a general security policy, if one security domain needs to increase its partition size, it can take a cache way from any security domain that is in a higher tier. We found that security vulnerabilities exist if *PAA* is not designed carefully. Without loss of generality, we discuss the insecure designs in the case of 3 security tiers, as shown in Figure 3.9. There are K_i security domains in tier i ($1 \leq i \leq 3$).

Case 1: Suppose L_1 wants to increase its partition size. Meanwhile, M_1 wants to decrease its partition size and M_2 wants to keep its partition size the same. A performance oriented algorithm tends to pick a cache way from M_1 to reallocate to L_1. If M_2 knows L_1 is cache-sensitive, but observes that it does not lose any cache way, it can infer M_1 may be decreasing its partition size. Hence, M_2 can learn the runtime cache demand of M_1.

Case 2: Assume M_1 wants to increase its partition size. M_1 gradually takes cache ways from H_j where ($1 \leq j \leq K_3$) until there is no any available cache way

in security tier 3. Now security domain M_2 starts to request for larger partition size. However, M_2 cannot get more cache ways because tier 3 has no extra cache ways. If M_2 knows that applications in tier 1 are cache-insensitive, then M_2 can infer that a security domain in tier 2 has high cache demand.

The pitfalls can be summarized as follows: 1) Dynamic arbitration between incomparable security domains based on cache demand. 2) First-Come, First-Serve (FCFS) allocation.

Our secure allocation design avoids the aforementioned pitfalls. We assume a general security policy that consists of N security tiers with each tier containing K_i incomparable security domains where $1 \leq i \leq N$. For the convenience of description, we use the notation $C_{i,j}$ to denote the jth security domain in security tier i.

If a security domain $C_{i,j}$ wants to increase its partition size, *PAA* will first check security tier $(i + 1)$. There are K_{i+1} security domaines in tier $(i + 1)$. We assume each security domain has P cache ways that can be reallocated to other security domaines initially. There are $K_{i+1} * P$ available cache ways for allocation in tier $(i + 1)$. To avoid the FCFS allocation pitfall, *PAA* reserves certain number of cache ways for each incomparable security domain. Since there are K_i incomparable security domaines in tier i, each security domain in tier i can get at most $\lfloor K_{i+1} * P/K_i \rfloor$ cache ways from tier $(i+1)$. Hence, *PAA* checks the reserved cache ways for security domain $C_{i,j}$ in tier $(i + 1)$. If there exists one cache way that is currently occupied by a security domain $C_{i',j'}$ and $i' > i$, then this cache way can be reallocated to $C_{i,j}$. If multiple cache ways satisfy the condition, *PAA* picks one cache way using a strict round-robin policy, thereby avoiding the pitfall of dynamic arbitration based on cache demand. If no cache way satisfies the

condition, *PAA* will then move to tier $(i + 2)$ and repeats the same procedure. The algorithm traverses through the security tiers from tier $(i + 1)$ to tier N, and terminates when a target cache way is found or the search reaches tier N.

If security domain $C_{i,j}$ wants to decrease its partition size, *PAA* will simply reallocates the extra cache way to one security domain in tier $(i + 1)$. If there are multiple security domaines in tier $(i + 1)$, *PAA* picks one security domain using strict round-robin policy.

Partition Enforcement Mechanism (PEM)

PEM does not flush the cache way when the partition size increases, since we allow cache lines belonging to a lower security domain to evict cache lines belonging to a higher security domain. However, when a partition size decreases, *PEM* needs to carefully flush specific cache lines to avoid timing channel attacks. Assuming a security domain $C_{i,j}$ gives up one cache way to be reallocated to another security domain $C_{i+1,j'}$, *PEM* will check each cache line in the reallocated cache way. If the security domain of a cache line is incomparable with or lower than $C_{i+1,j'}$, this cache line needs to be flushed.

3.4.4 Evaluation

Experimental Setup

We use an architecture simulator, gem5 [BBB+11], to evaluate the performance of SecDCP. We model a multi-core system that is configured with private L1 caches and a unified shared L2 cache, as shown in Table 3.1. We ran multipro-

Core count	2 / 4
Core model	2GHz Out-of-Order, ARM ISA
L1 caches	Private, 32kB, 2-way set associative, split D/I
L2 caches	Shared, 1MB, 8-way set associative / Shared, 2MB, 16-way set associative
Memory	200-cycle latency, 8GB/s peak memory BW

Table 3.1: System configuration.

cache-insensitive	astar, libquantum, gobmk, h264ref, hmmer, sjeng
cache-sensitive	bzip2, mcf, soplex, xalan

Table 3.2: Program categorization.

gram workloads that consist of SPEC CPU2006 programs. We fast-forward the simulation until every program in the workload has reached 1 billion instructions, after which the simulator starts detailed timing simulation. The simulation terminates when every program has run at least 250 million instructions. We calculate the IPC (Instruction Per Cycle) for each program using only the first 250 million instructions simulated.

We use 10 SPEC CPU2006 programs to form our multiprogram workloads. We use profiling to categorize these programs [QP06], as shown in Table 3.2. Cache-insensitive programs do not benefit significantly from cache size increase because they have few cache accesses or because their working set fits in a small cache size. In contrast, cache-sensitive programs can continuously benefit from increasing cache size up to the entire cache. For convenience, we mark cache-sensitive programs with the letter *S* and mark cache-insensitive programs with the letter *I*. We use weighted speedup as the metric to evaluate the performance. For a system with programs running concurrently, weighted speedup is defined as the sum of each program's IPC normalized to the IPC when the program is

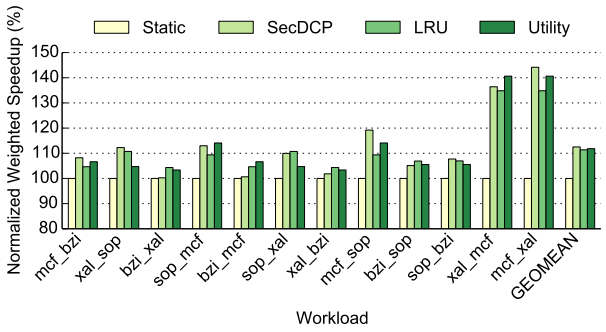


Figure 3.10: Performance for SS workloads.

running by itself:

$$Weighted\ Speedup = \Sigma(IPC_i/SingleIPC_i) \tag{3.4}$$

Performance for Two Security Domains

We first study the performance of SecDCP when there are two security domains, as shown in Figure 3.5. We use the naming convention program1_program2 for a workload, meaning that program1 belongs to L and program2 belongs to H. We compare SecDCP with three other schemes. The first scheme is static partitioning, which statically divides the cache into two equally-sized partitions. The second scheme has no partitioning and the entire cache is managed using LRU replacement policy. The last scheme is utility-based partitioning [QP06] that dynamically partitions the cache using runtime cache demands of both programs. For SecDCP scheme, we set the threshold value, th_{inc} and th_{dec}, to be both 20% in these experiments. Out of the four schemes, static partitioning and SecDCP are secure while LRU and utility-based partitioning are insecure against cache timing channel attacks.

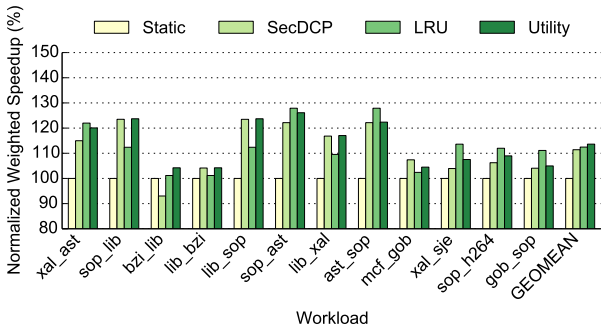


Figure 3.11: Performance for SI workloads.

Figure 3.10 shows the results for SS workloads, which are mixes of two cache-sensitive programs. We normalize the weighted speedup of each scheme to that of static partitioning. For these workloads, SecDCP achieves 12.5% improvement over static partitioning on average. In some cases, the improvement can reach as much as 40%. This is because SS workloads are cache-sensitive, requiring efficient utilization of the limited cache space. Static partitioning incurs high performance overhead since it cannot adapt the partition sizes to meet the runtime cache demand of each program. SecDCP increases the partition size of L when the L program is able to improve its performance with larger partition size. On the other hand, SecDCP decreases the partition size of L when it will not hurt the performance of the L program by much, thus giving more cache space to the H program, which can opportunistically improve its performance. On average, SecDCP achieves similar performance to no partitioning and utility-based partitioning.

Figure 3.11 shows the performance for SI workloads, which are randomly selected mixes of a cache-sensitive program and a cache-insensitive program. On average, SecDCP achieves an 11.4% improvement over static partitioning. For SI workloads, SecDCP will gradually reduce the partition size of the cache-

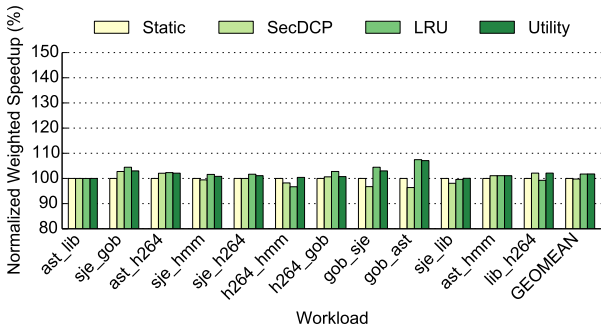


Figure 3.12: Performance for *II* workloads.

insensitive program, greatly improving the performance of the cache-sensitive program. Utility-based partitioning can also achieve the same goal, which is why we see similar speedups between SecDCP and utility-based partitioning for each workload. We also see in some cases (e.g., bzi_lib) that SecDCP performs worse than static partitioning. This is because the threshold value we choose (i.e., 20%) does not match the cache demand of *bzip*2. In this workload, *bzip*2 keeps decreasing its partition size because the loss (defined in equation 3.3) is less than 20%. However, *libquantum* is a streaming program that cannot utilize the increasing cache size. System performance decreases as a result. If we swap the programs (i.e., lib_bzi), *bzip*2 gets most of the cache because *libquantum* gives up most of its cache ways. As the figure shows, SecDCP outperforms static partitioning by 5% for lib_bzi.

Figure 3.12 shows the performance for *II* workloads, in which both programs are cache-insensitive. Since the performance of the program does not depend much on cache size, all schemes perform similarly.

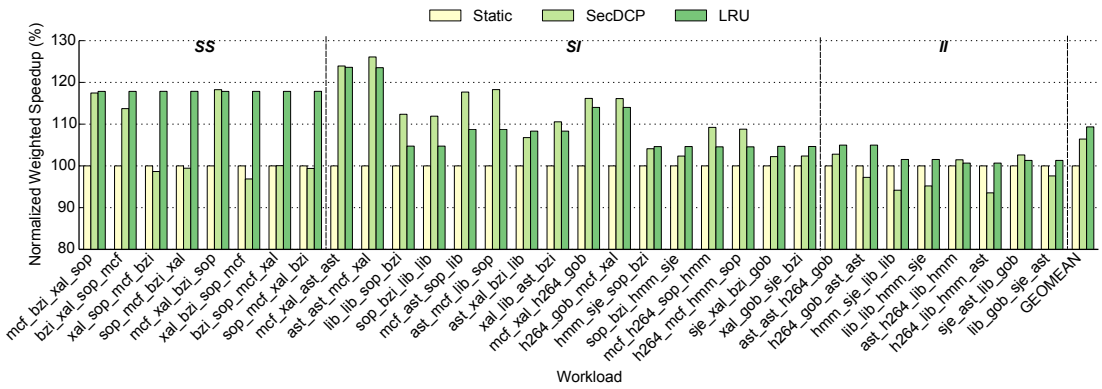


Figure 3.13: Performance for a linear security policy.

Scalability

We then study the performance of SecDCP when the system scales to four cores. We use a 2MB, 16-way set-associative L2 cache, which is shared by four programs. We consider two different security policies. The first policy is a linear security policy, as shown in Figure 1.6(a). Since we did not observe a big difference between no partitioning and utility-based partitioning in 2 core experiments, we do not include utility-based partitioning in 4 core experiments for simplicity. The static partitioning scheme divides the cache into four partitions, each being a four-way cache. We picked 32 four-program workloads, which consists of SS, SI and II types (two programs from each category). The results are shown in Figure 3.13. SecDCP achieves 6.4% improvement over static partitioning on average. For quite a few benchmarks, the improvement can reach 20%. However, we do see some workloads for which SecDCP performs worse than static partitioning. This is due to the imprecision of the threshold value we choose (20%).

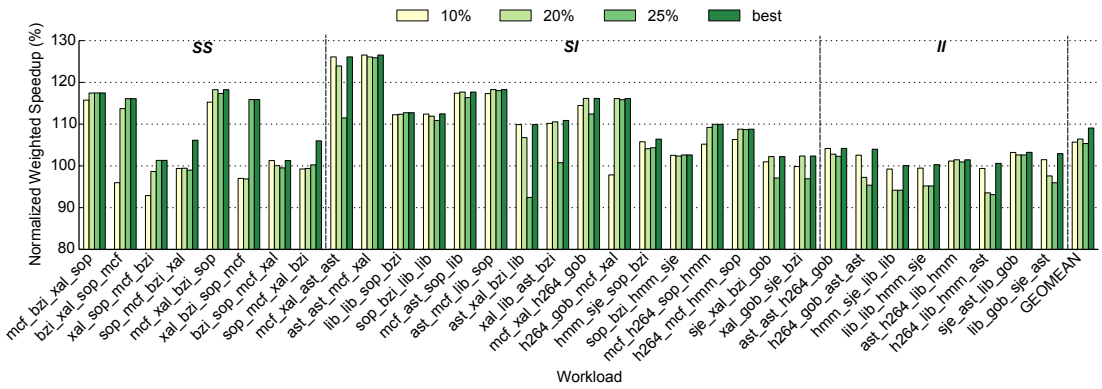


Figure 3.14: Performance with different thresholds.

To validate our judgement, we tried different threshold values from the set {2%, 5%, 10%, 15%, 20%, 25%}. We show the results for three threshold values in Figure 3.14. We also plot a bar (labeled as "best") that represents the best performing threshold for each workload. As can be seen, the threshold value can affect the performance significantly for most workloads, hence picking a good threshold is important. By comparing against the profiling results for individual program, we found that the optimal threshold value depends on the steepness of a program's miss curve. A steeper miss curve usually indicates a larger threshold value. For most workloads, using 20% as the threshold value performs reasonably well, only within 2.5% worse than the best performing threshold on average.

The second policy is shown in Figure 1.6(b). In this security policy, the two public applications belong to the same security domain. The static cache partitioning scheme divides the cache into 3 static partitions. The partition size for L is 8 cache ways because two programs share the partition, while the partition sizes for H_1 and H_2 are both 4 cache ways. In the previous threshold study, we found 20% is a good threshold value for SecDCP. However, this is not the case

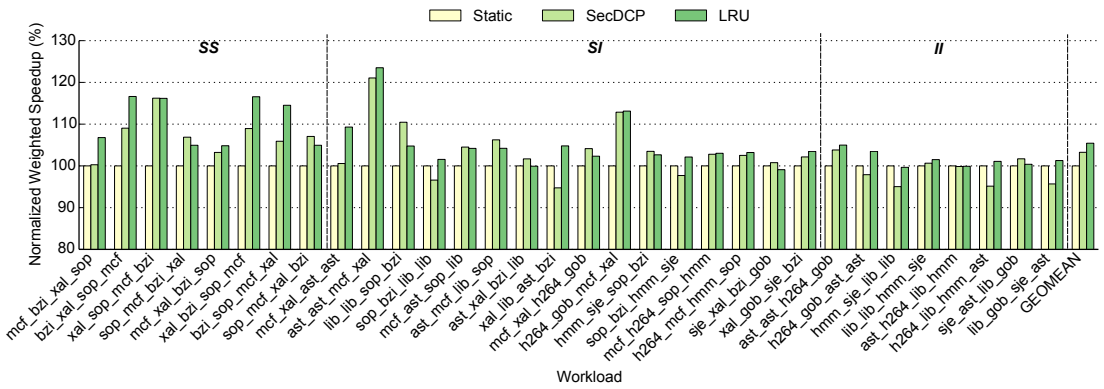


Figure 3.15: Performance for mobile security policy.

when a security domain contains two programs. Consider the case that two programs have the same amount of cache misses, but one program can achieve 20% less cache misses with one extra cache way while the other program does not benefit from more cache ways. In this case, we still want to give one extra cache way to these two programs although the overall reduced cache misses is only 10%. Hence, we need to set the threshold value lower, roughly equals to 20%/(# of programs in a security domain). We performed the same threshold study for this security policy, which shows that SecDCP with a 10% threshold performs much better than a 20% threshold. Figure 3.15 shows the performance of SecDCP with a 10% threshold. SecDCP achieves up to 21% and on average 3.3% improvement over static partitioning. The speedup is lower than that of the linear security policy, because the performance of the baseline static cache partitioning improves compared to the linear security policy by allowing the two public applications to share a cache partition.

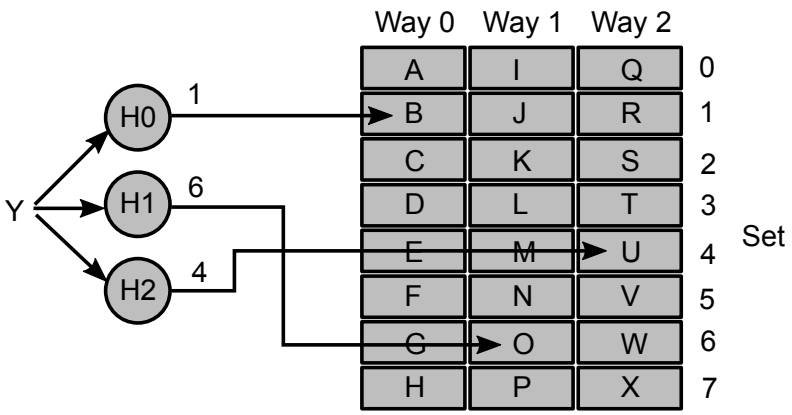


Figure 3.16: ZCache architecture.

3.5 Protection Based on a High-Associativity Cache (ZCache)

Static partitioning and SecDCP both partition the shared cache at the granularity of cache ways. Although they are secure against timing channel attacks, the number of security domains that can be supported simultaneously is constrained by the cache associativity. To overcome this limitation, we explore timing channel protection schemes based on ZCache [SK10], a high-associativity cache that can potentially support large number of security domains.

3.5.1 Background

ZCache is a novel high-associativity cache design. In a normal set-associative cache, the number of replacement candidates equals to the associativity. In ZCache, the number of replacement candidates can be much larger than the physical associativity. As a result, a better replacement candidate can be chosen for eviction to improve cache performance. Essentially, ZCache increases

Addr	Y	B	O	U	...
H0	1	1	7	5	...
H1	6	3	6	0	...
H2	4	5	2	4	...

Table 3.3: Hash values for each address.

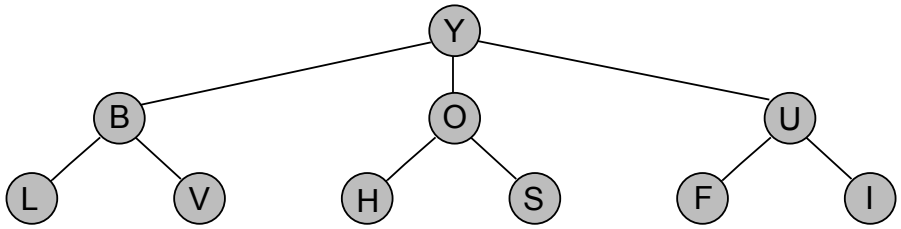


Figure 3.17: Tree of replacement candidates.

the effective associativity of a cache. This is achieved by using different hash functions to decide the cache set number for each cache way.

To illustrate how ZCache works, let us consider a 3-way cache shown in Figure 3.16. A new cache block Y is fetched into the cache. Each cache way has a different hash function ($H0$, $H1$, $H2$) to decide the set number. As a result, Y is mapped to set 1 in way 0, set 6 in way 1 and set 4 in way 2. Now Y has 3 replacement candidates, $\{B, O, U\}$, to choose from. However, since a cache block can reside in different sets in different cache ways, ZCache can generate more replacement candidates by considering the possible locations of $\{B, O, U\}$ in other cache ways. Given the hash values in Table 3.3, we can derive more replacement candidates in a tree structure, as shown in Figure 3.17. In this example, we only expand the tree to two levels, but more levels can be generated if needed. Suppose the replacement policy choose cache block V for eviction. A series of relocation operations are required to maintain the cache organization. The relocation process is shown in Figure 3.18(a). Y replaces B, B replaces V, and V is evicted. Figure 3.18(b) shows the final cache state.

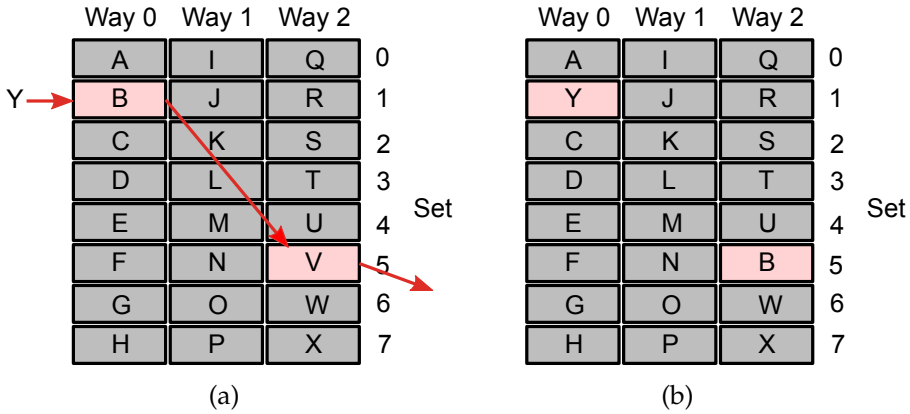


Figure 3.18: Relocation to accommodate incoming cache block.

The advantage of ZCache is the large amount of replacement candidates during an eviction, even with limited physical cache associativity. With more replacement candidates, the replacement policy can pick a better cache line for eviction to improve the performance. As the ZCache paper points out, the effective cache associativity depends on how good the replacement decision can be compared to a fully-associative cache, not the physical cache associativity. However, ZCache does not defend against cache timing channel attacks, because it still allows a victim's cache lines to evict an attacker's cache lines, and the eviction is not randomized. In the following discussion, we explore how we may extend ZCache to support timing channel protection.

3.5.2 Same-Domain Replacement

To avoid interference between different security domains, we first explore a scheme that enforces same-domain replacement. In this scheme, a new cache block can only evict a cache block that belongs to the same security domain. This is illustrated in Figure 3.19. We use uppercase letters to denote the cache lines from one security domain, and lowercase letters to denote the cache lines

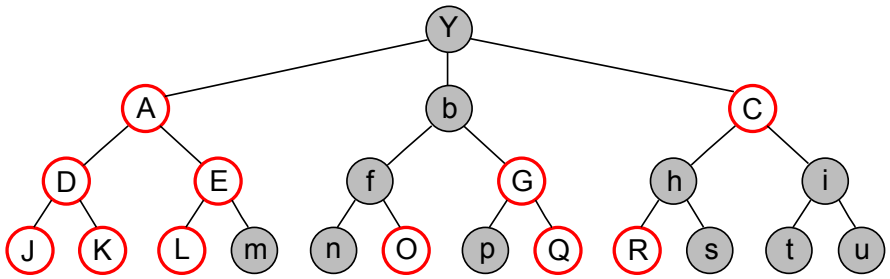
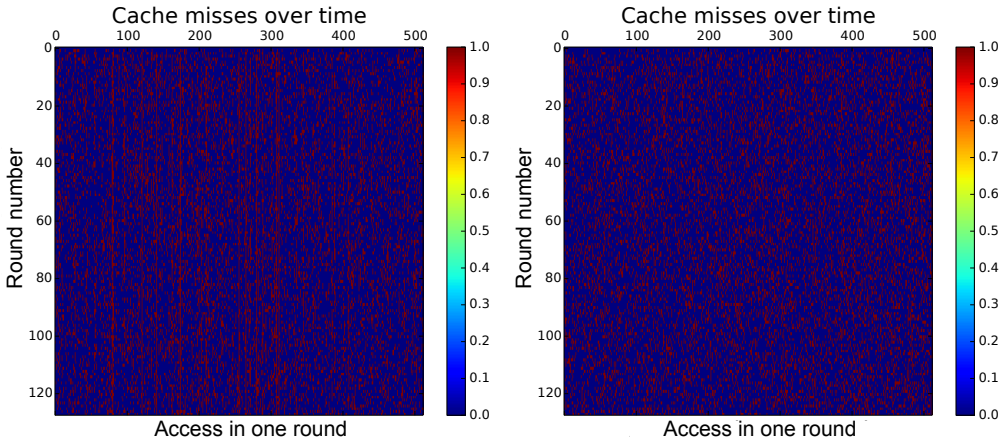


Figure 3.19: Same-domain replacement.

from another security domain. All eligible replacement candidates are marked white. If the tree does not contain any cache lines from the same security domain as the new cache line, the new cache line is directly sent to the CPU and not cached. Under this replacement policy, a victim program can never evict an attacker's cache lines. However, a victim program can change the locations of an attacker's cache lines through relocation process. For example, if the cache line being evicted is O, both cache lins b and f will be relocated to another cache way.

At first glance, allowing relocations between different security domains may seem benign. However, we discover a practical covert channel attack by exploiting this relocation process. We assume two attackers share a ZCache while one attacker (sender) is trying to send confidential information to the other attacker (receiver). Each attacker belongs to a different security domain. The cache has 4 physical cache ways and 128 cache sets. Each attacker occupies 256 cache lines—half of the cache capacity. The receiver randomly generates 512 addresses. In each round, the receiver accesses these 512 addresses one by one, and records the hit or miss information for each address through timing. After that, the receiver waits for a time interval and starts the next round, repeating the same operations. The receiver performs 128 rounds, so it can gather 512*128 data



(a) Sender does nothing. (b) Sender issues random accesses.

Figure 3.20: A relocation-based attack.

points. On the other hand, the sender performs different operations to encode one bit. To send a bit '0', the sender does not issue any cache access. To send a bit '1', the sender randomly issues 2,560 cache accesses in each round, which can relocate the receiver's cache lines. If the receiver is able to distinguish these two cases, then one bit is communicated successfully.

Figure 3.20(a) and Figure 3.20(b) shows the hits and misses that the receiver observes in each case. The *X* axis is the data point in one round (512 data points) and the *Y* axis denotes different rounds. A red data point indicates a cache hit while a blue data point indicates a cache miss. Although both graphs appear to be rather random, there is a subtle difference between the two graphs. In the graph that assumes no accesses from the sender (Figure 3.20(a)), there exist some red lines that cross all the way from top to bottom. These red lines are addresses that always get cache hits, which means they never get evicted from the cache. In contrast, in the graph that assumes random accesses from the sender (Figure 3.20(b)), no such red lines exist. This observable difference easily distinguishes the two cases and can be exploited for timing channel attacks.

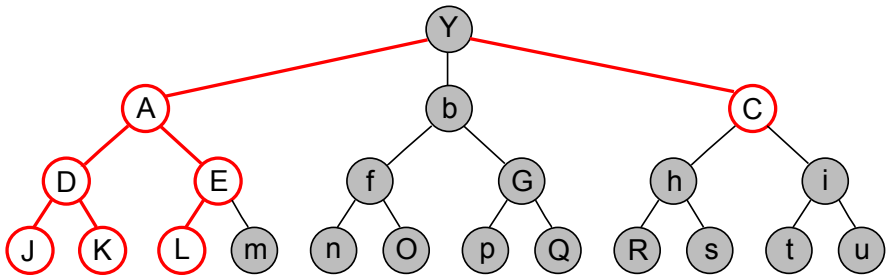


Figure 3.21: Strict same-domain replacement.

In the first case, some addresses are never evicted from the cache because they do not conflict with any other addresses in the 512 addresses in their **current** cache way. As a result, they stay in a fixed cache location and keep providing cache hits. However, in the second case, these addresses are very likely to get relocated by the cache accesses from the sender under the same-domain replacement policy. After these addresses are relocated to another cache way, they may conflict with the other addresses given that a different hash function is used. As a result, they may get evicted because of cache conflicts. This attack shows that even allowing relocation between different security domains can introduce timing channels if the attacker is able to collect a large number of data points. Same-domain replacement may be sufficient to defend against side channel attacks, but is still prone to covert channel attacks, which can send many data points using this relocation process.

3.5.3 Strict Same-Domain Replacement

To eliminate the timing channel caused by relocation between different security domains, we can add further restrictions on the eligible replacement candidates. Specifically, we restrict the replacement candidates to be only the cache lines

whose path to the root node all consists of cache lines from the same security domain. This replacement policy is illustrated in Figure 3.21. The eligible candidates are marked white. If no eligible replacement candidates exist in the tree, the data is supplied to the CPU directly without caching.

In this strict same-domain replacement scheme, a security domain can neither evict nor relocate cache lines that belong to another security domains. This design provides complete non-interference, hence is secure against timing channel attacks. However, the number of eligible replacement candidates can drop significantly compared to the "relaxed" same-domain replacement scheme, especially if we scale up the number of security domains. To alleviate this scaling issue, we propose to apply set partitioning on top of strict same-domain replacement. Since hash functions are already part of the ZCache implementation, set partitioning can be implemented efficiently by slightly modifying the hash function, hence introducing negligible hardware overhead. For example, if a cache that has 4 cache ways and 2048 cache sets needs to support 4 security domains, we can map the first two security domains to the first 1024 sets and the last two security domains to the last 1024 sets by modifying the hash functions as follows:

$$Hash(address)\%1024 + (security\ domain\ ID/2) * 1024 \tag{3.5}$$

3.6 Defending Against Timing Channels within a Program

3.6.1 Reuse-Based Attacks

So far, we've only discussed cache timing channel protection schemes to defend against attacks based on external interference, in which different programs that share a cache belong to different security domains. However, these protection schemes are ineffective against attacks based on internal interference. In this type of attacks, the attacker can reside on a remote machine and communicates with the victim. The execution time of the victim code (e.g., performing an encryption) may correlate with the victim's secret, hence the attacker is able to infer the secret through timing channels. As an example, consider the following victim code:

```
1  if (secret)
2        h = l1
3  else
4        h = l2
5  l1 = 0
```

If the secret bit is '1', line 2 is executed and $l1$ is fetched into the cache. When line 5 gets executed, the memory access results in a cache hit. However, if the secret bit is '0', line 4 is executed instead. When line 5 gets executed, the memory access leads to a cache miss. The attacker is able to tell the value of the secret simply by observing the victim's execution time. This attack is a type of reuse-based attacks [BM06], which exploits the fundamental feature of a cache—demand fetch. Essentially, when a memory block is accessed, it will be

fetched into the cache, so the following accesses to this same memory block will become cache hits and take less time.

Random fill cache [LL14] defends against reuse-based attacks by fetching a random memory block within some range of the target block into the cache instead of the target block itself. In the attack example above, random fill cache does not fetch $l1$ when line 2 gets executed. It will randomly fetch some memory block that is near $l1$ into the cache, thus line 5 may or may not be a cache hit. However, random fill cache requires the security-critical data to be in continuous memory regions to guarantee security, which is restrictive and sometimes can be impractical. Besides, random fill cache may incur significant performance overhead because many unuseful cache lines may be fetched into the cache.

3.6.2 Language-Based Protection

Instead of completely abandoning demand fetch, which is the workhorse of a cache, we propose to disable demand fetch only when demand fetch will leak secret information. This can be accomplished with the help of a language-based protection framework proposed by Zhang et al. [ZAM12]. In this framework, software is labeled to indicate which cache accesses need to be protected to prevent timing channel leakage. Each instruction in the program has a read label and a write label. The read label defines timing of an instruction, setting an upper bound on the label of hardware that can affect the execution time of this instruction. The write label defines hardware states that an instruction can influence, setting a lower bound on the label of hardware that the instruction can

modify. We defines two labels, L (low) and H (high), which satisfies $L \subseteq H$. Applying the labels to the previous victim code, we have the following labeled code, in which the first label in brackets is the read label, and the second one is the write label.

```
1  if (secret)   [H, H]
2      h = l1    [H, H]
3  else
4      h = l2    [H, H]
5  l1 = 0        [L, L]
```

Since line 2 is labeled with [H, H], $l1$ is stored in the cache and labeled as H. When line 5 gets executed, it cannot access $l1$ through the cache because the read label of this instruction is L, which means the execution time of this instruction can only be affected by cache lines that are labeled as L. Hence, the memory access results in a cache miss and the data is fetched from the memory as if the data is not in the cache. With this design, the value of the secret will not be revealed through the total execution time.

However, this solution will possibly introduce two copies of the same data in the cache. A correct cache design needs to ensure these copies are consistent with respect to the memory. Indeed, the language-based protection [ZAM12] only specifies the timing behavior of the underlying hardware and omits the actual hardware implementation. It is therefore the hardware designer's responsibility to implement a hardware design that is functionally correct and conforms to the requirements of read and write labels. We target to implement such a cache design in this section.

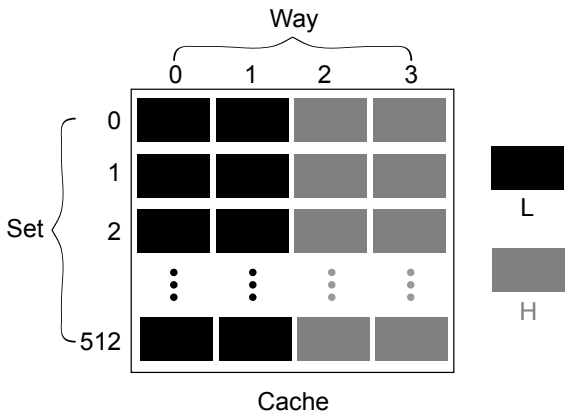


Figure 3.22: A partitioned cache.

3.6.3 Baseline Partitioned Cache

We assume the baseline cache is divided into two partitions, *H* and *L*. The partition is done at the granularity of cache ways. Figure 3.22 shows an example partitioned cache with four cache ways. The cache lines in *H* partition all share the same security label *H*, while the cache lines in *L* partition have security label *L*. When an instruction fetches a cache block from memory, the write label of the instruction decides which partition the cache block will be stored.

3.6.4 Problem: Dirty Bit Leakage

During our design of a secure cache, we found that the dirty bit of a cache line can also lead to information leakage. To demonstrate the problem, consider the following victim code.

```
1  if (secret)   [H, H]
2      h1 = l1   [H, H]
3  else
4      h2 = l1   [H, H]
5  h3 = h1       [L, L]
6  l2 = l3       [L, L]
```

Assume the secret bit is '1', both $h1$ and $l1$ are fetched into the cache and stored in the H partition. When line 5 gets executed, however, access to $h1$ should be a cache miss since the read label is L. Hence, the cache fetches $h1$ from memory. When the memory response arrives at the cache, we discard the data from memory and move the existing copy of $h1$ from H partition to L partition. This is because the copy in the cache is the most updated version of $h1$. However, this design brings up a security issue.

Assume $l2$ and $l3$ in line 6 are fetched into the cache and evict $h1$ as a result. Since $h1$ is dirty, the cache needs to write it back to memory. Without a writeback queue, the eviction of a dirty block may stall the cache from processing other requests. Even if a writeback queue exists, the queue can get full and stall the CPU from accessing the cache. On the other hand, if the secret bit is '0', then $h1$ stays clean after line 5. In this case, when $h1$ gets replaced by $l2$ or $l3$, the cache can simply invalidate $h1$ without writing it back to memory. Comparing the two cases, we find that the dirty bit of a cache line may reveal confidential information through timing channels. A secure cache design must handles the leakage caused by the dirty bit.

3.6.5 Solution 1: Mark All Cache Lines Dirty (DirtyCache)

We propose a simple cache design that removes the timing channel leakage through dirty bit. The basic idea is to make dirty bit contain no secret information. To achieve this, our design marks every cache line being fetched into the cache dirty, hence the name "DirtyCache". Since all cache lines are dirty, dirty bit contains no useful information for the attacker. Beside a dirty bit, each cache line also has a one-bit security label T to indicate its security level. We divide the cache into two partitions, L partition and H partition. Cache lines in L partition has a security label L while cache lines in H partition has a security label H. Next, we detail the DirtyCache design by describing its operations under different security labels of an instruction. As before, the first label in the bracket is the read label, and the second label is the write label.

Instruction Label = (L, L)

- Cache hit: the cache controller checks the security label T of the cache line being accessed.
  - If $T = L$, the cache controller treats the hit as a normal hit in a normal cache.
  - If $T = H$, the cache controller treats the access as a cache miss because the instruction should not see this cache line given its read label is L. A cache line in L partition is chosen for eviction and a memory request is issued for the access. When the memory response returns, the cache controller moves the cache line from H partition to L partition, changing its label from H to L.

- Cache miss: a cache line in L partition is chosen for eviction and a memory request is issued for the access.

Instruction Label = (H, H)

- Cache hit: this is treated as a normal cache hit as in a normal cache because the instruction is able to see all the cache lines in the cache.
- Cache miss: A cache line in H partition is chosen for eviction and a memory request is issued for the access.

Instruction Label = (L, H)

- Cache hit: the cache controller checks the security label T of the cache line being accessed.
  - If $T = L$, the cache controller treats the hit as a normal hit in a normal cache.
  - If $T = H$, the cache controller treats the access as a cache miss because the instruction should not see this cache line given its read label is L. A cache line in H partition is chosen for eviction and a memory request is issued for the access. When the memory response returns, the cache controller discards the memory response and uses the existing copy in the cache.
- Cache miss: a cache line in H partition is chosen for eviction and a memory request is issued for the access.

Instruction Label = (H, L)

- Cache hit: this is treated as a normal cache hit as in a normal cache because the instruction is able to see all the cache lines in the cache.
- Cache miss: A cache line in L partition is chosen for eviction and a memory request is issued for the access.

The complication of the DirtyCache design is mostly in handling an access that has a read label of L while the data exists in the H partition of the cache—the timing channel in our code example. DirtyCache eliminates the timing channel with a simple design. However, marking all cache lines dirty can introduce significant performance overhead, because any eviction of a cache line requires writing the data back to memory, regardless of whether the cache line has been written or not. To minimize this performance overhead, we propose a more efficient cache design called "RelCache" that gets rid of the DirtyCache restrictions on dirty bits.

3.6.6 Solution 2: Use Relative Dirty Bits (RelCache)

In this cache design, we allow one memory block to have two copies with different security labels in the cache. For example, the cache can contain both memory block M with label L and memory block M with label H. To maintain the coherence of different copies, we make the dirty bit of the L copy to be **relative** to the H copy. In other words, the L copy is dirty only if the L copy is different from the H copy. The dirty bit of the H copy is defined the same as a normal cache (relative to the memory). As we will see below, this design choice is necessary

to efficiently remove the timing channel caused by the dirty bit. We describe the design of RelCache by showing how the cache controller reacts to cache hits and misses under different security labels. For cache hits, we discuss read and write accesses separately because a read access does not affect the dirty bit while a write access can change the dirty bit of a cache line.

Instruction Label = (L, L)

- Read access & cache hit: the cache controller checks the security label T of the cache line getting hit (C_1).
  - If $T = L$, the cache controller treats the hit as a normal hit in a normal cache.
  - If $T = H$, the cache controller treats the access as a cache miss. A cache line C_2 in L partition is chosen for eviction. If C_2 has another copy C_2' with label H in the H partition, the cache controller checks the dirty bit of C_2. If C_2 is dirty, the cache controller write it back to memory and invalidate C_2', since C_2' is a stale copy. If C_2 is clean, C_2 is simply invalidated. If C_2 is the only copy in the cache, the cache controller evicts C_2 normally. After the eviction, the cache controller sends a memory request to fetch the data for the read access. When the memory response arrives at the cache, the cache discards the response and copy the data from cache line C_1 to the L partition. The new cache line is marked as clean (relative to the H copy) with a security label L. In this design, the dirty bit of the new cache line does not contain any secret information from H since it is always clean.

  - If there are two copies in the cache, the cache responds with the L copy, which is the most updated version of the data.

- Write access & cache hit: the operations in this case are identical to the previous case, except that the new cache line is marked as dirty because the access is a write.

- Cache miss: A cache line C_2 in L partition is chosen for eviction. If C_2 has another copy C_2' with label H in the H partition, the cache controller checks the dirty bit of C_2. If C_2 is dirty, the cache controller write it back to memory and invalidate C_2', since C_2' is a stale copy. If C_2 is clean, C_2 is simply invalidated. If C_2 is the only copy in the cache, the cache controller evicts C_2 normally. After the eviction, the cache controller send a memory request to fetch the data. When the memory response arrives at the cache, mark the new cache line as clean with label L.

Instruction Label = (H, H)

- Read access & cache hit: If there are two copies in the cache, the cache responds with the L copy. Otherwise, the access is treated as a normal cache hit.

- Write access & cache hit: the cache controller checks the security label T of the cache line getting hit (C_1).

  - If $T = L$, the controller checks the dirty bit of C_1. If C_1 is clean, the access writes to the cache and also write back to memory. This is because the write label of this instruction is H, which disallows changing the dirty bit of L partition. By writing back to memory, C_1 remains

to be clean in the cache. On the other hand, if C_1 is dirty, the access only needs to write to the cache.

  - If $T = H$, the access writes to the cache.
  - If there are two copies in the cache, the controller writes to the H copy and invalidates the L copy.

- Cache miss: A cache line C_2 in H partition is chosen for eviction. After the eviction, the cache controller send a memory request to fetch the data. When the memory response arrives at the cache, mark the new cache line as clean with label H.

Instruction Label = (L, H)

- Read access & cache hit: the cache controller checks the security label T of the cache line getting hit (C_1).
  - If $T = L$, the cache controller treats the hit as a normal hit in a normal cache.
  - If $T = H$, the cache controller treats the access as a cache miss. A cache line C_2 in L partition is chosen for eviction. If C_2 has another copy C_2' with label H in the H partition, the cache controller checks the dirty bit of C_2. If C_2 is dirty, the cache controller write it back to memory and invalidate C_2', since C_2' is a stale copy. If C_2 is clean, C_2 is simply invalidated. If C_2 is the only copy in the cache, the cache controller evicts C_2 normally. After the eviction, the cache controller sends a memory request to fetch the data for the read access. When the memory response arrives at the cache, the cache discards the response and uses the existing copy in the cache.

  - If there are two copies in the cache, the cache responds with the L copy, which is the most updated version of the data.

- Write access & cache hit: the operations in this case are identical to the previous case.
- Cache miss: A cache line C_2 in L partition is chosen for eviction. If C_2 has another copy C_2' with label H in the H partition, the cache controller checks the dirty bit of C_2. If C_2 is dirty, the cache controller write it back to memory and invalidate C_2', since C_2' is a stale copy. If C_2 is clean, C_2 is simply invalidated. If C_2 is the only copy in the cache, the cache controller evicts C_2 normally. After the eviction, the cache controller send a memory request to fetch the data. When the memory response arrives at the cache, mark the new cache line as clean with label H.

Instruction Label = (H, L)

- Read access & cache hit: If there are two copies in the cache, the cache responds with the L copy. Otherwise, the access is treated as a normal cache hit.
- Write access & cache hit: If there are two copies in the cache, the controller writes to the L copy. Otherwise, the access is treated as a normal cache hit.
- Cache miss: A cache line C_2 in L partition is chosen for eviction. If C_2 has another copy C_2' with label H in the H partition, the cache controller checks the dirty bit of C_2. If C_2 is dirty, the cache controller write it back to memory and invalidate C_2', since C_2' is a stale copy. If C_2 is clean, C_2 is simply invalidated. If C_2 is the only copy in the cache, the cache controller evicts C_2 normally. After the eviction, the cache controller send a memory

request to fetch the data. When the memory response arrives at the cache, mark the new cache line as clean with label L.

RelCache removes the dirty bit leakage by maintaining a relative dirty bit when there are two copies of a data in the cache. This design avoids the restriction of dirty cache, which marks every cache line dirty, reducing the performance overhead of timing channel protection. Although RelCache can introduce two copies of a data, which can waste some cache space, the amount of shared data between H and L instructions is usually small in common programs. With the language-based protection framework, RelCache presents a secure and efficient solution to defend against timing channel attacks within a program.

CHAPTER 4

MEMORY CONTROLLERS

While timing channel attacks and their countermeasures have been studied in the context of shared caches, to the best of our knowledge, timing channels through a shared memory channel have not yet been studied at the hardware architecture level. In this chapter, we discuss our findings on memory controller timing channels and propose several protection schemes. Section 4.1 describes timing channel attacks we found in a shared memory controller. Section 4.2 proposes a protection scheme based on Temporal Partitioning (TP). Section 4.3 introduces a more efficient protection scheme called SecMC-NI by interleaving requests that access different banks and ranks of DRAM. To further improve the performance of protection schemes, we propose SecMC-Bound, which enables a tradeoff space between performance and security in Section 4.4.

4.1 Attacks

4.1.1 Memory Controller Interference

Memory requests from different security domains contend for the shared DRAM memory and can affect the latency of each other, which opens a timing channel. This timing channel is contention-based, much like the on-chip network timing channel. However, the memory timing channel is much more complicated because of the way memory works. In DRAM memory, a memory transaction lasts for tens of cycles, during which it can affect the scheduling of following memory transactions. Multiple memory transactions can be in flight

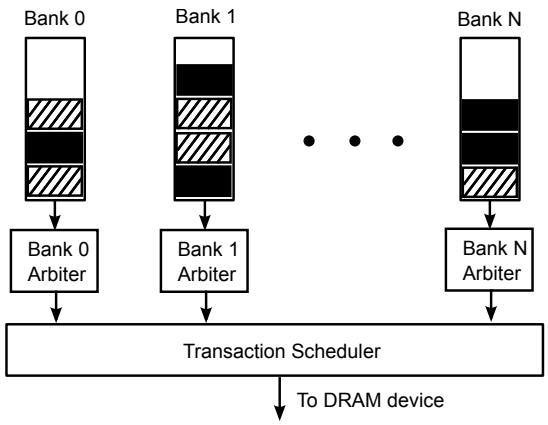


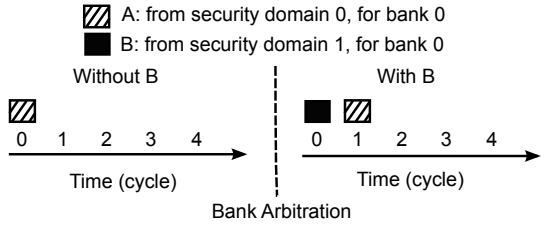
Figure 4.1: A conventional memory controller.

simultaneously in a pipelined fashion, given that they satisfies the timing constraints of a specific DRAM. For example, two memory read requests to different rows of the same bank must be separated by at least $t_{RAS} + t_{RP}$ cycles. More details about the timing constraints in DRAM memory can be found in [JNW07].

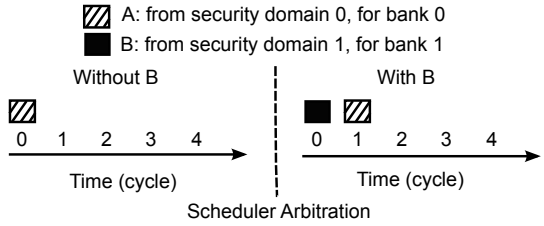
To provide timing channel protection for memory controllers, let us first understand the sources of interference in conventional memory controller architecture, which is shown in Figure 4.1. A successful memory access takes the following steps:

1. It is enqueued into one of the request queues based on the address.
2. It wins bank arbitration in the bank arbiter.
3. It wins transaction scheduler arbitration.
4. It gets sent to the DRAM devices.

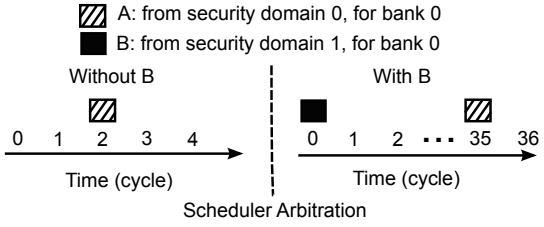
The First-Ready First-Come First-Served (FR-FCFS) [RDK+00] scheduling algorithm is used for the baseline memory controller. As shown below, there are three sources of interference in the baseline memory controller.



(a) Bank Arbitration Interference



(b) Scheduler Arbitration Interference



(c) DRAM Device Interference

Figure 4.2: Interference in memory controllers.

Queueing Structure Interference.

The baseline memory controller has a separate queue for each combination of the ranks and banks (e.g., if there are 3 ranks and 4 banks, a typical memory controller will have 12 queues). This ensures that requests for each bank are put into a separate queue. Although this queueing structure is beneficial for exploiting bank-level parallelism in DRAM accesses, it introduces interference among memory accesses from different security domains. In this queueing structure, a queue can mix memory requests from different security domains, which are denoted by different patterns in Figure 4.1. As shown in Figure 4.2(a), Request *A*

from security domain 0 can be delayed in the queue by Request *B* from another security domain 1 if the bank arbiter schedules Request *B* prior to Request *A*.

Interference in the queueing structure also occurs whenever the memory controller stalls the requests to a particular bank when that bank's request queue is full. If security domain 0 fills the request queue of bank 0 and stalls the memory requests from security domain 1, security domain 1 can learn that security domain 0 is sending many memory requests to bank 0.

Scheduler Arbitration Interference.

The transaction scheduler also causes interference. As can be seen in Figure 4.2(b), suppose Request *A* and Request *B* are accessing different banks and they both win bank arbitration in cycle 0. Without Request *B*, Request *A* wins the scheduler arbitration and is sent to the DRAM at cycle 0. However, if Request *B* exists and arrives in the queue earlier than Request *A*, the FR-FCFS scheduler will favor Request *B* in arbitration, thus delaying Request *A* to the next cycle. This interference changes the timing of Request *A*.

DRAM Device Interference.

Resource contention in DRAM device components such as the command bus, the data bus, banks, and ranks can also create timing channels. For example, assume Request *A* and *B* are from different security domains and intend to access the same bank of a rank. Request *A* arrives at the queue at cycle 2 and Request *B* arrives at cycle 0. Without Request *B*, Request *A* wins bank arbitration and scheduler arbitration at cycle 2. However, if Request *B* exists and is scheduled

at cycle 0, Request A cannot win scheduler arbitration at cycle 2 even if it wins bank arbitration, because the DRAM device cannot serve two memory requests to the same bank concurrently. In an open page policy, if the second request is a row hit, it needs to wait until the first request finishes I/O gating. In a close-page policy, the second request needs to wait even longer, because it cannot be scheduled until the bitline is precharged. As shown in Figure 4.2(c), Request *A* is not scheduled until cycle 35 because the bank has been busy serving Request *B*.

The interference problem is not limited to FR-FCFS scheduling algorithm, but exists for most of current memory scheduling algorithms (i) because queueing structures mix requests from different security domains, (ii) because the arbitration of the transaction scheduler depends on the dynamic demands of different security domains, (iii) and because of the properties of DRAM devices. All these sources of interference can be used as timing channels to derive the memory usage characteristics of security domains, which can leak secret information. Next, we show two timing channel attacks that exploit the memory interference.

4.1.2 A Side-Channel Attack on RSA

This side channel attack shows how a private key of an RSA decryption program can be compromised by exploiting the interference in memory accesses. The system setup is shown in Figure 4.3. The system has two cores, each with a private direct-mapped L1 cache. The RSA decryption algorithm runs on Core 0 while an attack program is running simultaneously on Core 1.

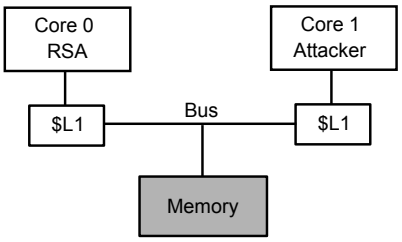


Figure 4.3: System setup for an RSA attack.

```
x = C
for j = 1 to n
    x = mod(x^2, N)
    if d_j == 1 then
        x = mod(xC, N)
    end if
next j
return x
```

Figure 4.4: Square and multiply algorithm for RSA: C is the encrypted message, x is the decrypted message, N is the product of two large prime numbers, d is the RSA private key, and n is the number of bits in the key.

The RSA decryption algorithm uses a private key to decrypt an encrypted message. It is often implemented with the square and multiply algorithm to perform fast exponentiation, as shown in Figure 4.4. In this implementation, the bits in the private key are checked one by one, and a modulo operation is performed only when the bit is '1'. In this attack example, the memory addresses are configured so that when this modulo operation is performed, a cache miss occurs. In other words, the number of memory requests for the RSA algorithm is directly dependent on the number of '1' bits, or the Hamming weight, of the private key. The attacker issues memory requests to the DRAM continuously and measures the time to finish those requests.

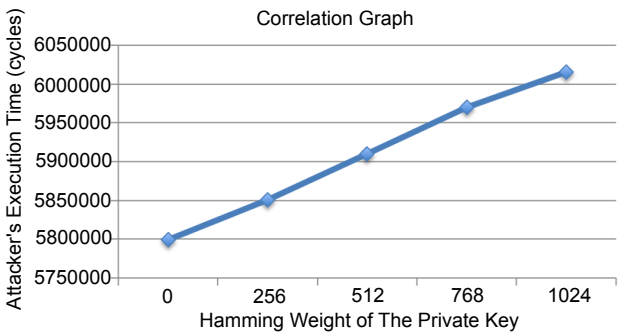


Figure 4.5: RSA side-channel attack example.

Figure 4.5 shows the execution time of the attack program as a function of the Hamming weight of the private key. As can be seen, the attacker's execution time has a direct correlation with the Hamming weight, meaning that the attacker can estimate the number of 1s in the private key by simply measuring its own execution time. The attack can succeed because the memory controller is shared and the interference between the memory requests from different programs leaks information.

4.1.3 A Covert-Channel Attack

In this shared memory covert-channel example, one adversary tries to send information to another adversary when direct communication between them is disallowed. The system setup is similar to Figure 4.3 except that now Adversary 0 runs on Core 0 and Adversary 1 runs on Core 1. The goal of Adversary 0 is to send the sequence '10010110' to Adversary 1. Adversary 0 achieves this goal by dynamically changing its memory demand, which affects the latency of memory requests from Adversary 1. To send a '0', Adversary 0 does not issue any memory request for a period of time. To send a '1', Adversary 0 sends many

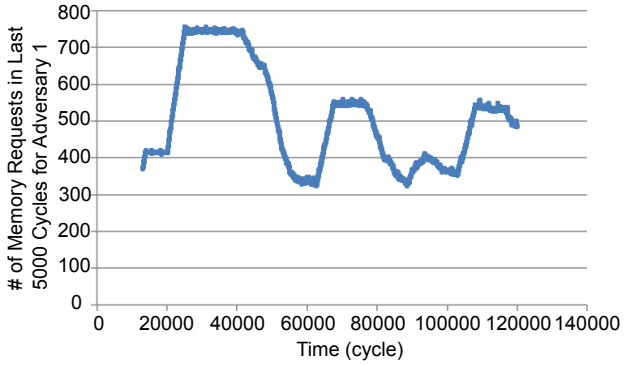


Figure 4.6: A covert-channel attack example.

memory requests. Meanwhile, Adversary 1 keeps sending memory requests and tracks the dynamic throughput it can achieve using a software counter.

Figure 4.6 shows the memory throughput observed by Adversary 1 over the last 5,000 cycles. As can be seen, the throughput shows a pattern that corresponds to the bit stream that Adversary 0 intends to send. When the throughput is low, Adversary 1 can infer that Adversary 0 is sending a lot of memory requests, and interprets the bit being sent as a '1'. Otherwise, the bit being sent is a '0'. Using the interference in the memory, Adversary 1 can fully recover the information that Adversary 0 sends, proving the feasibility of this covert channel attack.

4.2 Temporal Partitioning (TP)

This section introduces a timing channel protection scheme called Temporal Partitioning that eliminates aforementioned sources of memory interference. Tem-

poral partitioning enforces complete noninterference between different security domains, hence providing bi-directional timing channel protection.

4.2.1 Protection Mechanisms

Queueing Structure Protection

To eliminate interference among memory requests from different security domains in the same request queue, the queueing structure proposed in this design includes queues for each combination of ranks and security domains instead of each combination of ranks and banks. Figure 4.7 shows the new queueing structure. With per security domain queueing structure, memory requests from different security domains are separated and stored in different queues, and therefore, bank arbitration cannot cause interference among them. Interference can still exist between requests in the same queue, however, they belong to the same security domain, so the interference is benign. In order to exploit bank parallelism, this scheme also requires scheduling logic that scans the queue for requests to an idle bank. Similar logic is also used in a conventional open page memory controller to find requests to open rows.

Scheduling Protection

Concurrent memory accesses from multiple security domains cause both arbitration interference and DRAM device interference. These two types of interference can be eliminated if only one security domain uses memory resources at a time. Thus, we propose Temporal Partitioning (TP), which divides the time into

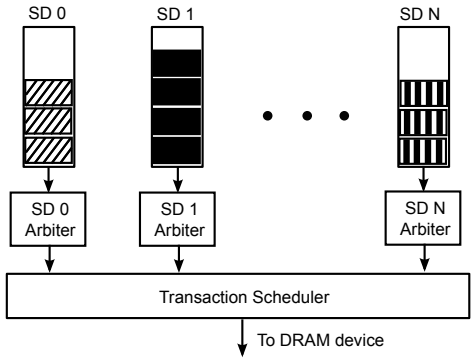


Figure 4.7: Queueing structure per security domain.

Figure 4.8: Static time-slot allocation in temporal partitioning.

fixed-length turns. During each turn, only requests from a particular security domain, which we say is active, can issue.

Figure 4.8 illustrates the high-level idea of TP. The length of a turn is defined as T_{turn}. During each turn, requests of the active security domain are scheduled normally, but requests from other domains are not allowed. While requests within each domain can cause interference with each other, such intra-domain interference is benign as they cannot leak information to another domain. TP allows requests of the active security domain to be dynamically scheduled, which is good for performance. At the end of each turn, the next security domain is selected and activated using a fixed, static schedule. The implementation discussed in this work uses a strict round-robin static schedule, however, any other static schedule will suffice.

Row Buffer Policy

In a DRAM cell, requests for data already in the row buffer (sense amplifier) are much faster than others. In an open page row-buffer management policy, the most recently activated row is left in the row buffer of the bank until another row in that bank must be accessed. This policy is beneficial for workloads that have a lot of row locality (and are therefore likely to reuse data already in the row buffer). In contrast, a close-page policy immediately precharges the bank in anticipation of an access to a different row. The close-page policy has better memory access times for consecutive accesses to different rows, although it can no longer exploit row locality. Therefore, close-page policies are preferable for workloads with little row locality.

Since during a turn the active security domain is allowed to cause interference among its own memory requests, it seems reasonable, at first inspection, to allow any of these policies or a hybrid scheme to best suit the particular workload. However, the scheme as described thus far does nothing to handle the data available in the row buffer before switching to the next turn in an open-page policy. As a result, an adversary can learn about the data access pattern of another security domain through the timing difference between row buffer hits and row buffer misses.

This channel can be eliminated by issuing a precharge command to every bank at the end of a turn. Unfortunately, contemporary DRAM chips cannot meet the power criteria necessary to issue a precharge to every bank in a sufficiently small time interval. Further, precharging only the banks that were actually accessed does not work as this implies a variable number of precharges at

Read Transaction	$t_{RAS}+t_{RP}$
Write Transaction	$t_{CWD}+t_{BURST}+t_{WR}+t_{RP}+t_{RCD}$

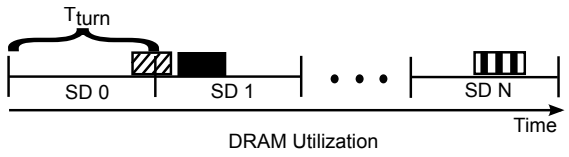
Table 4.1: Close-page DRAM timing analysis.

the end of the turn and causes yet another timing channel. The TP protection scheme thus requires a close-page policy for security and better performance.

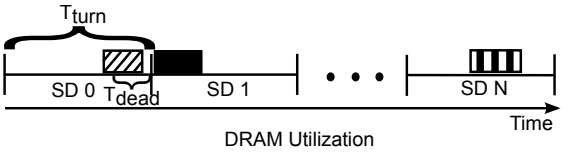
Dead Time

With only the aforementioned restrictions, a memory transaction could be issued before the turn changes, but remain in flight at the beginning of the next turn, possibly causing interference to memory requests from the next security domain. This interference is illustrated in Figure 4.9(a). Therefore, an interval of time, called the **dead time**, is required at the end of each turn to prevent new transactions from being issued. The dead time must be long enough to complete any in-flight transactions before the turn transitions, as shown in Figure 4.9(b). In other words, the dead time must be no less than the worst case time T_w to drain either a read or write transaction. The times required to drain either of these transactions (and precharge the bitline after the access) are shown in Table 4.1 using DRAM timing parameters, which can be found in commercial DRAM datasheets. Based on our study of several commercial DRAM datasheets, the time to drain write transactions is usually longer than the time to drain read transactions. Therefore,

$$T_{dead} = T_w = t_{CWD} + t_{BURST} + t_{WR} + t_{RP} + t_{RCD}. \tag{4.1}$$



(a) Interference caused by in-flight transactions.



(b) Dead time.

Figure 4.9: Dead time to remove interference from in-flight transactions.

Refresh Timing Channel.

In a conventional memory controller, no transactions can be issued when a bank is being refreshed. However, when a bank that needs to be refreshed is already being accessed, the refresh is stalled until the in-flight commands to that bank are completed. This means the actual time the refresh takes place depends on the memory transactions and therefore memory access patterns and data of the active security domain. Figure 4.10 shows the interference caused by a stalled refresh. Without Request A, the refresh can finish before the end of SD 0's turn, and Request B can be issued as normal. However, if Request A exists and it delays the refresh, then it is possible that the refresh cannot finish until the next turn, because the time to finish a refresh, t_{RFC} is larger than T_{dead}. This indirectly delays the schedule of Request B.

This type of interference is caused by a refresh crossing the border between two consecutive turns. To eliminate this interference, the dead time can be increased to at least as long as the time to complete a refresh, t_{RFC}, plus the time required to drain any in-flight transaction, T_w. However, this is overly conser-

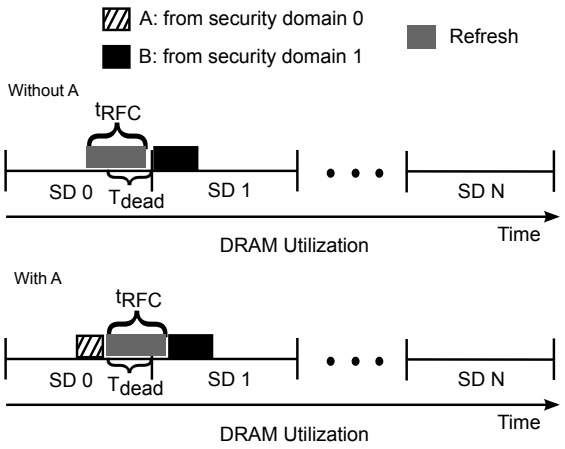


Figure 4.10: Interference from a stalled refresh.

vative if done unilaterally for each turn. Instead, since the originally scheduled time for each refresh is public information, only turns during which refresh is scheduled will have a dead time that is increased by t_{RFC}. This eliminates the refresh timing channel for any chosen turn length value since, if the turn length is greater than $t_{RFC} + T_w$, any refresh issued during a particular turn will always finish before the end of that turn. If the turn length is less than $t_{RFC} + T_w$, the dead time is larger than the turn length and the active security domain is blocked for its entire turn. Because there is no access from the active security domain, there is no interference between a memory access and a refresh.

Turn Length Tradeoff

The length of a turn affects the performance of temporal partitioning. There is no upper limit on the turn length, but the turn length should at least be greater than T_w to avoid starvation. When T_{turn} is equal to T_w, at most one request can be scheduled in one turn and it can only be scheduled at the first cycle of the turn because of the dead time. The optimal turn length depends on the work-

load and cache configuration among other system parameters. The tradeoffs involved are best explored by characterizing the sources of overhead in temporal partitioning.

The first source of overhead is the dead time, which wastes memory bandwidth for a fixed interval at the end of each turn. The dead time comes at the end of every turn, therefore the overhead depends on the number of turns. As the turn length increases, the number of turns will reduce. As a result, the dead time overhead is less with a longer turn length. On the other hand, as the turn length increases, the maximum time a request can spend blocked in the transaction queue while its security domain is inactive also increases. Therefore, a longer turn length will be desirable when the throughput is the main concern and a shorter turn length will be desirable when the latency is important.

4.2.2 Performance Optimizations

Bank Partitioning (BP).

In TP, the memory bandwidth loss due to the dead time represents one of the largest sources of overhead. The dead time ensures that memory requests from two consecutive turns cannot interfere with each other by draining all in-flight transactions at the end of a turn before issuing any memory request from a new turn. Unfortunately, the dead time needs to be quite conservative in order to avoid interference even in the worst case where requests from two turns access the same bank of a rank.

If it can be guaranteed that requests from two consecutive turns cannot access the same bank, the dead time can be significantly reduced because in-flight transactions do not need to be drained before issuing requests from a new domain. TP can use bank partitioning among security domains or turns to guarantee this property. For example, different security domains can be mapped to different banks in the main memory. Alternatively, TP can restrict which memory banks can be used at the beginning and the end of each turn to ensure that there cannot be bank conflicts between two consecutive turns. With this optimization, the dead time can be reduced to the worst case time interval between two consecutive memory accesses to different banks. Considering the power constraint and different combinations of consecutive memory accesses to different banks, the dead time can be determined by the following equation:

$$T_{dead} = max(t_{FAW} - 3 * t_{RRD}, t_{CWD} + t_{BURST} + t_{WTR}, \\ t_{CAS} + t_{BURST} + t_{RTRS} - t_{CWD}). \quad (4.2)$$

For the DRAM module we used in the experiments, this new dead time is only 18 cycles compared to 43 cycles without bank partitioning.

Application-Aware Turn Length.

In the baseline design, temporal partitioning divides the memory bandwidth evenly among security domains using the static round-robin scheduling with the same turn length for all security domains. In order to distribute the memory bandwidth more effectively for a given workload mix, TP can be optimized to use a different turn length for each security domain and also schedule turns in an order that matches the workload characteristics. As long as the turn lengths

and schedule are not affected by the dynamic memory demand of each security domain, temporal partitioning still ensures that there is no timing channel between security domains.

4.2.3 Evaluation

Experimental Setup

We evaluate the security and performance of temporal partitioning through simulation studies. DRAMSim2 [RCBJ11] is used to model the memory controller as well as ranks, banks, and channels of a DRAM. To study the performance impact on realistic benchmarks, DRAMSim2 is integrated into an architecture simulator, ZSim [BBB+11]. We model a multi-core processor, and each core has a private L1 cache. The L2 cache is shared, but partitioned to remove interference among cores. This configuration was chosen to evaluate the impact of memory timing channel protection schemes without the impact of cache interference. We model a single memory channel with 8 ranks and 8 banks per rank. The detailed configuration parameters are shown in Table 4.2.

We use multi-program workloads constructed from SPEC CPU2006 benchmark suites to evaluate the performance. To evaluate the performance under different memory intensities, we run eight copies of the same program on the 8-core processor. Each program fast-forwards for 1 billion instructions and then enters detailed simulation mode, in which the memory requests are simulated in cycle-accurate manner. The simulation terminates when all programs have executed at least 100 million instructions. We only take the first 100 million in-

Processor	
ISA	x86
Core count and frequency	2/4/8 cores, 2.0GHz
ROB size	128
Issue width	4
Cache	
L1 I-cache	32KB/4-way
L1 D-cache	32KB/4-way
L2 Cache	2/4/8 MB 8 ways per core
DRAM	
DRAM bus frequency	667MHz
DRAM configuration	1 channel, 8 ranks, 8 banks/rank
Total capacity	16GB
DRAM Timing Parameters (DRAM cycles) $t_{RC} = 34, t_{RCD} = 10, t_{RAS} = 24, t_{FAW} = 20$ $t_{WR} = 10, t_{RP} = 10, t_{RTRS} = 1, t_{CAS} = 10$ $t_{RTP} = 5, t_{BURST} = 4, t_{CCD} = 4, t_{WTR} = 5$ $t_{RRD} = 4, t_{REFI} = 7.8us, t_{RFC} = 107$	

Table 4.2: Configuration parameters for ZSim and DRAMSim2 simulators.

structions of each program to calculate the IPC. Figure 4.11 shows the L2 misses per kilo instructions (MPKI) for different SPEC benchmarks.

We use weighted speedup as the performance metric. For a system with concurrently running programs, weighted speedup is defined as the sum of each program's IPC normalized to the IPC when the program is running by itself:

$$Weighted\ Speedup = \Sigma(IPC_i/SingleIPC_i) \tag{4.3}$$

The weighted speedups are normalized to the insecure baseline, which is FR-FCFS scheduling [ZR97] in our experiments. We used 24 SPEC benchmarks and show 12 representative benchmarks in graphs. The benchmarks are ordered based on the memory intensity (MPKI) with more memory intensive bench-

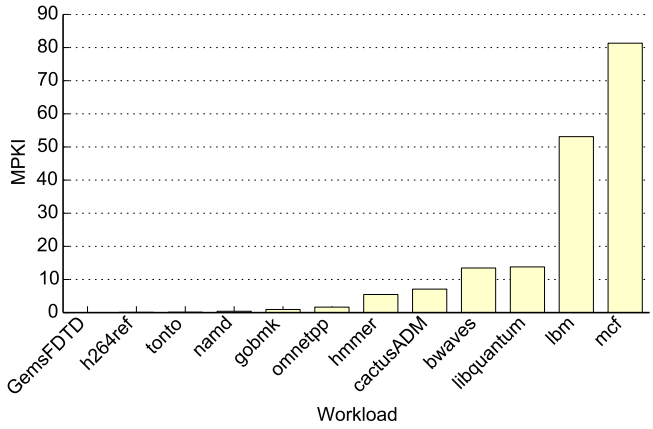


Figure 4.11: Memory intensity study of SPEC2006 benchmarks.

marks on the right. Note that the geometric mean of each figure is calculated using all 24 benchmarks.

Security Evaluation

Temporal partitioning eliminates the memory interference by modifying the queueing structure and the scheduling algorithm of a conventional memory controller. To test if memory interference has indeed been eliminated, multi-program workloads comprised of SPEC2006 benchmarks are run to record the timing of memory requests. We use gem5 [BBB+11] to collect memory request traces in 10 million instructions for each benchmark, then these traces are used in pairs of two ($T0$, $T1$) to study the security of a two-core system, in which each core runs in a different security domain. The traces are fed into DRAMSim2 to simulate the cycle-level behavior in the memory controller and DRAM device.

To verify that temporal partitioning can protect against timing channel attacks, a benchmark, $T0$, from one security domain is fixed and run with a dif-

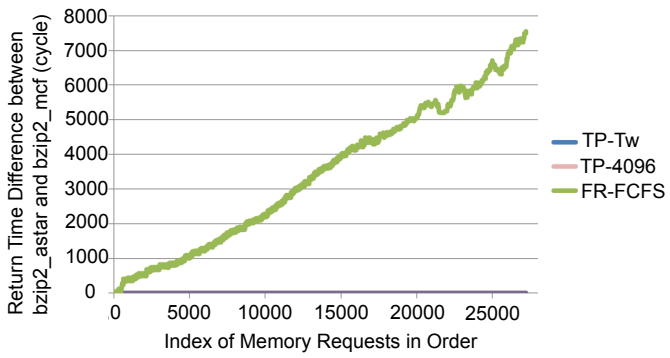


Figure 4.12: Memory return time difference of $T0$ running with difference $T1$s.

ferent benchmark, $T1$, from another security domain. If the memory controller completely eliminates interference, the return time of each memory request in $T0$ should always be the same regardless of what benchmark $T1$ is. The results for a fixed $T0$ with different $T1$s are compared. Figure 4.12 shows one example of the comparison. The Y axis is the return time difference for each memory request in $T0$ when *bzip* is used for $T0$ and $T1$ is changed from *astar* to mcf. Two different turn lengths are used for TP, namely T_w and 4096 memory cycles. As can be seen, both $TP - T_w$ and $TP - 4096$ show a flat line that equals 0, meaning the timing of $T0$'s memory requests are not affected by which benchmark $T1$ is. In contrast, The result for FR-FCFS shows a huge difference after $T1$ changes from one benchmark to another, which indicates the existence of memory interference and a timing channel. Every possible combination of benchmark pairs was compared in this way, and the results show that with temporal partitioning protection, the return time of every memory request from $T0$ stays the same regardless of what benchmark $T1$ runs.

The timing channel protection is still effective when there are more than two security domains. To test the security of increasing the number of security do-

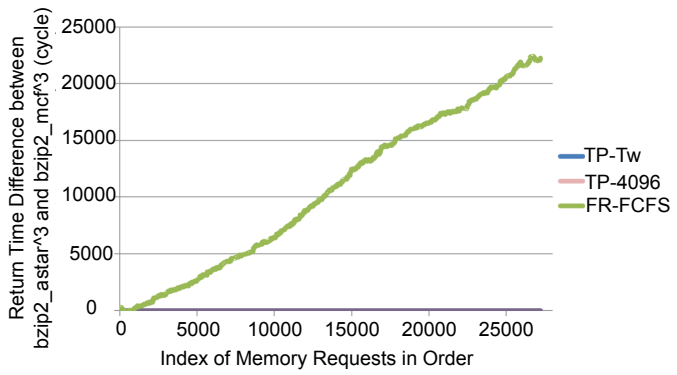


Figure 4.13: Memory return time difference with 4 security domains.

mains, experiments are run with four traces. $T0$ is kept constant and ($T1$, $T2$, $T3$) are changed from ($astar$, $astar$, $astar$) to (mcf, mcf, mcf). These two combinations were intentionally chosen because $astar$ is not memory-intensive while mcf is very memory-intensive. The results for the case where $T0$ is $bzip2$ are shown in Figure 4.13. Similar to the results for two security domains, the comparison passes for all combinations, which shows that TP eliminates the memory interference for multiple security domains. This security evaluation is run for all benchmarks in SPEC2006.

Performance Evaluation

Performance Overhead. The static scheduling and the dead time in temporal partitioning introduce performance overhead compared to an insecure baseline. To quantify this performance overhead, the SPEC2006 benchmarks are run with the baseline memory controller and with TP. In these experiments, the turn length of TP is set to be 128 cycles.

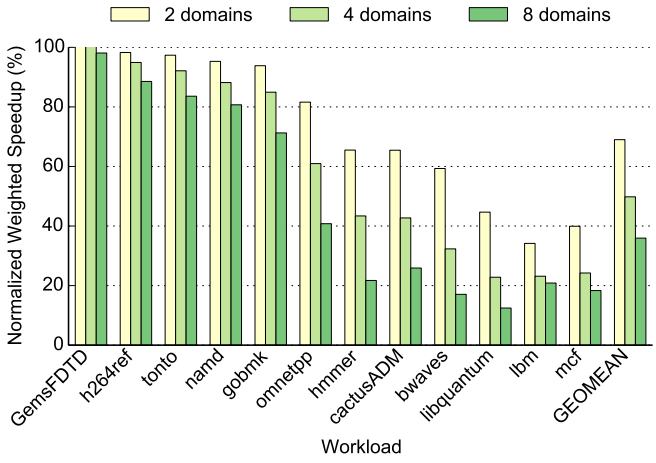
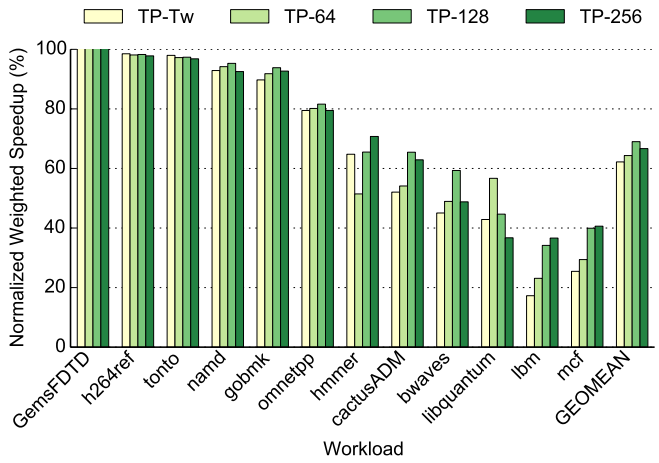


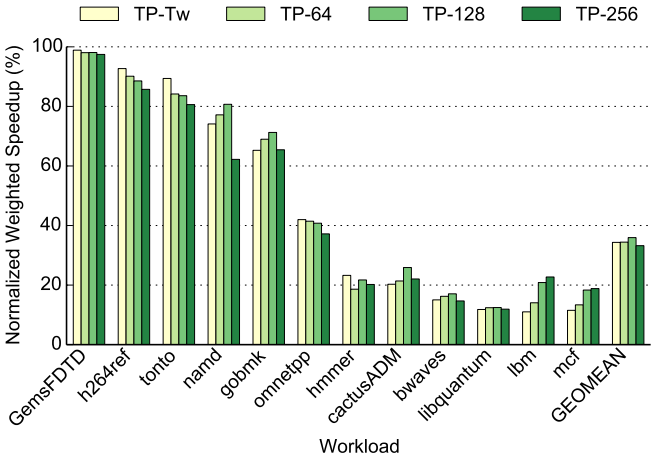
Figure 4.14: Performance overhead of TP.

Figure 4.14 shows the performance comparison between the baseline and TP. The weighted speedup of TP is normalized to that of the insecure baseline. Figure 4.14 also shows the performance of different number of security domains to study the scalability effect.

For 2 security domains, TP achieves 70% performance of the baseline. For non-memory intensive benchmarks, TP incurs negligible performance overhead compared to the baseline. However, as the memory intensity increases, the performance gap between TP and the baseline grows significantly. The overhead exceeds 60% for benchmarks such as *lbm* and *mcf*. The performance overhead mainly comes from long queueing delay of the static turn scheduling and wasted bandwidth during the dead time. This performance overhead is also reflected as the number of security domains scales up. For 8 security domains, TP incurs 63% performance overhead on average compared to the baseline. This is mainly because a security domain needs to wait for more turns before issuing its requests as the number of security domains increases.



(a) 2 security domains



(b) 8 security domains

Figure 4.15: Effect of turn length on performance overhead.

Effect of Turn Length. As we discussed earlier, the turn length of TP scheme can affect the performance of security domains. Longer turn length allows more requests to be issued in each turn, but also leads to longer queueing delay for some requests. Figure 4.15 shows the performance of TP with different turn lengths. Again, all the performance results are normalized to that of insecure baseline. Results for both 2 and 8 security domains are presented.

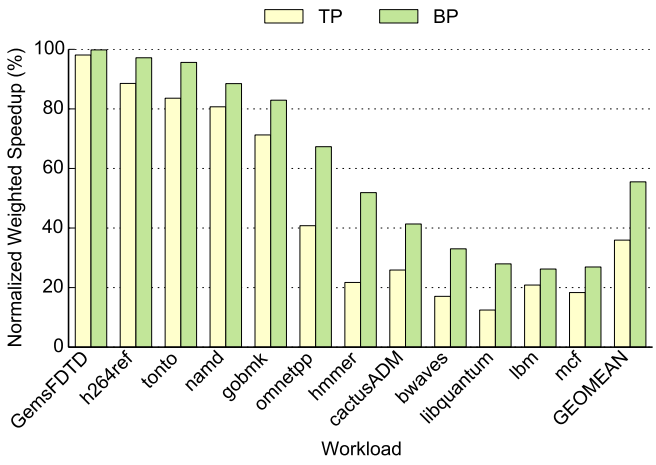


Figure 4.16: Performance improvement with bank partitioning.

For most benchmarks, the performance of TP increases initially as the turn length increases. However, once the turn length reaches some certain value (mostly 128 in these experiments), the performance of TP starts to decrease. These results illustrates the trade-off between delays and bandwidth associated with a turn length. Benchmarks that are delay-sensitive (e.g., libquantum) tend to favor shorter turn length, while benchmarks that are bandwidth-sensitive (e.g., mcf) tend to favor longer turn length.

Bank Partitioning. We studied the effectiveness of bank partitioning as a performance optimization technique. As discussed earlier, bank partitioning reduces the dead time from 43 cycles to 18 cycles, hence significantly reducing wasted bandwidth. Figure 4.16 shows the performance comparison between TP and BP for 8 security domains. Thanks to the shorter dead time, BP outperforms TP by 50% on average, achieving 56% performance of the insecure baseline.

4.3 SecMC-NI

Temporal Partitioning defeats memory timing channel attacks, but comes with high performance overhead due to the wasted bandwidth in dead time. In this section, we propose SecMC-NI (NI stands for noninterference), a secure memory controller that provides efficient memory scheduling by tightly interleaving requests from different ranks and banks.

4.3.1 Existing Protection Schemes

After we propose Temporal Partitioning (TP) as a timing channel protection mechanism, a few new techniques [FWZ+16, SGS+15] have been developed to improve the performance of TP while maintaining the security guarantee against memory timing channels. Researchers have proposed to use rank partitioning (RP) with temporal partitioning. In rank partitioning, the physical memory is partitioned among different security domains. Rank partitioning [SGS+15] restricts security domains to place data in different ranks. This restriction ensures that memory requests from consecutive turns always access different ranks. The dead time between requests thus can be much shorter (6 cycles using our DRAM model). However, rank partitioning seriously constrains data placement, making it difficult to deploy in practice. If the number of security domains is large, rank partitioning may be simply infeasible due to the limited number of ranks. For example, typical systems have no more than 8 ranks per channel. In cloud computing, rank partitioning may imply that a system cannot keep more than 8 virtual machines in memory. Rank partitioning also leads to high memory fragmentation. VMs with a small memory footprint

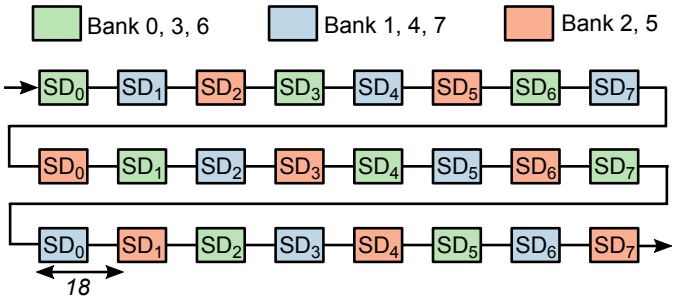


Figure 4.17: Bank triple alternation schedule example.

waste allocated memory while VMs with a large memory footprint suffer from insufficient memory.

To avoid the disadvantages of rank partitioning, Shafiee et al. [SGS+15] proposed Bank Triple Alternation (BTA), which divides the memory banks into three bank groups. Consecutive memory requests are restricted to always access different bank groups, which ensures that there cannot be any bank conflict between turns. This enables using short (18 cycle) turns. As a drawback, only a subset of banks can be accessed in each turn.

Figure 4.17 shows an example BTA schedule for 8 security domains and 8 banks. The 8 banks are divided into 3 bank groups, and each security domain (SD_i) can only access one of the bank groups in each turn. The schedule ensures that consecutive memory requests always access different banks, hence the time interval between them can be as short as 18 cycles using our DRAM timing parameters.

However, the fixed scheduling of BTA leads to several inefficiencies. First, if two requests from the same security domain access different banks in the same bank group, the second request needs to wait for 24 (3*8) turns, or 432 cycles, even though they only need to be separated by 18 cycles under timing constraints. Second, requests arriving at a "bad" time can be delayed significantly.

For example, if a request for bank 0 from SD_0 arrives at cycle 6, it must wait for another 24 turns. Finally, BTA does not consider ranks when determining the schedule. Requests to different ranks are still separated by 18 cycles even though they only need to be separated by 6 cycles. Next, we show our new memory controller design, SecMC-NI, that removes the inefficiencies in BTA.

4.3.2 SecMC-NI Algorithm

The inefficiencies in BTA come from the static nature of the scheduling algorithm—only a fixed set of banks can be accessed in each turn. The design principle of SecMC-NI is to allow a security domain to access any bank or rank in its turn while improving the peak bandwidth through interleaving requests to different banks and ranks. The scheme is inherently dynamic as memory requests are scheduled based on each domain's own access pattern rather than a fixed schedule.

As in other temporal partitioning schemes, SecMC-NI divides time into turns and use strict round-robin scheduling to schedule security domains. Only one security domain can issue requests in each turn. For the convenience of description, we define some parameters as follows:

T_{turn}: Turn length in clock cycles

S: Total number of security domains

T_{bank}: Minimum number of cycles between requests that access different banks

T_{rank}: Minimum number of cycles between requests that access different ranks

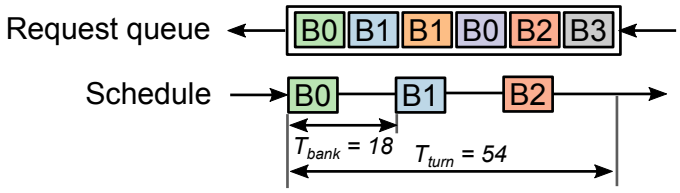


Figure 4.18: SecMC-NI scheduling example.

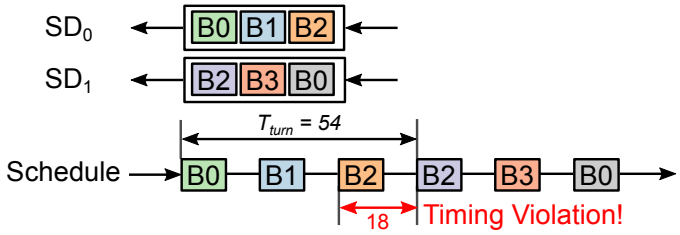


Figure 4.19: Bank conflict in SecMC-NI scheduling.

Request Selection Algorithm

In each turn, request selection algorithm picks requests from a security domain to be issued. The requests are chosen based on the following rules. First, requests must access different banks. This rule ensures that we can schedule the chosen requests at a rate of one request every T_{bank} cycles. Second, the algorithm enforces the maximum number of requests to be scheduled is $\lfloor T_{turn}/T_{bank} \rfloor$. This ensures that the picked requests can fit into one turn. Finally, the selected requests are scheduled at cycle 0, T_{bank}, $2 \cdot T_{bank}$, ..., $(\lfloor T_{turn}/T_{bank} \rfloor - 1) \cdot T_{bank}$ within the turn.

As a concrete example, consider the schedule in Figure 4.18. The notation *Bi* indicates a memory request for bank *i*. The requests are enqueued from the right side of the queue and dequeued from the left side, as indicated by the arrows. Let's assume $T_{turn} = 54$ and $T_{bank} = 18$. Although there are two requests for bank 0 and bank 1, only one of the requests gets issued. Because at most 3 requests can be issued in this turn, the request to bank 3 remains in the queue.

Compare to BTA, the scheduling algorithm of SecMC-NI is more flexible. As long as the requests are accessing different banks, they are allowed to be issued together in a turn. However, this additional flexibility introduces new complexities. Consider the example in Figure 4.19. Two different security domains are ready to schedule their requests. If the requests are scheduled in their arrival order, the resulting schedule will violate DRAM timing constraints due to a bank conflict. To avoid such timing violations, SecMC-NI uses a reordering algorithm to adjust the order of memory requests.

Reordering Algorithm

For reordering, SecMC-NI uses a small buffer to keep track of the schedule in the previous turn. After the requests are selected for the current turn, SecMC-NI checks each of these requests against requests in the previous turn. If Req_0 in the current turn accesses the same bank as Req_1 in the previous turn, Req_0 will be placed at the same relative position as Req_1. As a result, these two requests are separated by T_{turn} cycles. To allow accesses to the same bank in two consecutive turns, T_{turn} must be long enough (43 cycles) to satisfy DRAM timing for accesses to the same bank. After the reordering, the history buffer is updated with the schedule of the current turn. Using this reordering algorithm, the schedule of SD_1 in Figure 4.19 becomes $B0, B3, B2$, which satisfies DRAM timing constraints.

While reordering removes timing violations, it introduces a security concern because the scheduling order of a security domain may be affected by the previous security domain. To avoid the information leak through reordering and enforce strict non-interference, SecMC-NI delays sending memory responses back

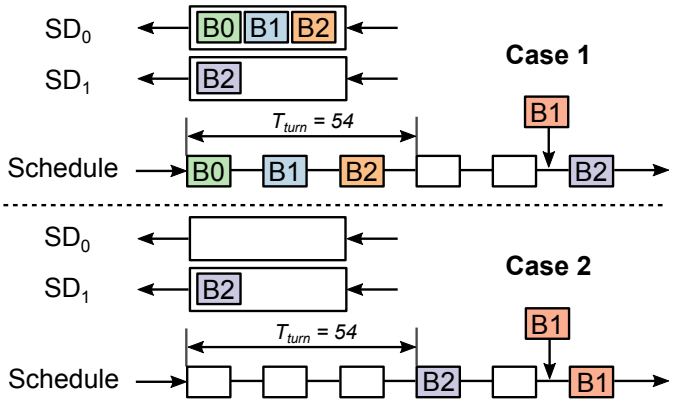


Figure 4.20: Insecure scheduling example.

to CPU until all memory requests in a turn finish memory accesses. SecMC-NI then sends the memory responses in the arrival order of memory requests.

Another subtle security issue is that the schedule in a turn must be determined at the beginning of a turn. If requests arrive in the middle of a security domain's turn, they cannot be scheduled in the current turn even if there are available slots. To understand the problem, consider the example in Figure 4.20. In the first case, SD_0 has three requests scheduled in its turn. SD_1 only has one request to bank 2, which is scheduled at the last slot. During SD_1's turn, a new request to bank 1 arrives for SD_1. However, because the last slot is already taken, $B1$ cannot be scheduled in this turn. In the second case, SD_0 has no requests. As a result, $B2$ from SD_1 is scheduled in the first slot. When $B1$ arrives in the middle of the turn, it can be scheduled in the last slot. This example shows that SD_0 can affect the timing of SD_1's requests, which is insecure. SecMC-NI avoids this insecure scheduling by determining the schedule at the beginning of a turn. $B1$ will not be scheduled in both cases.

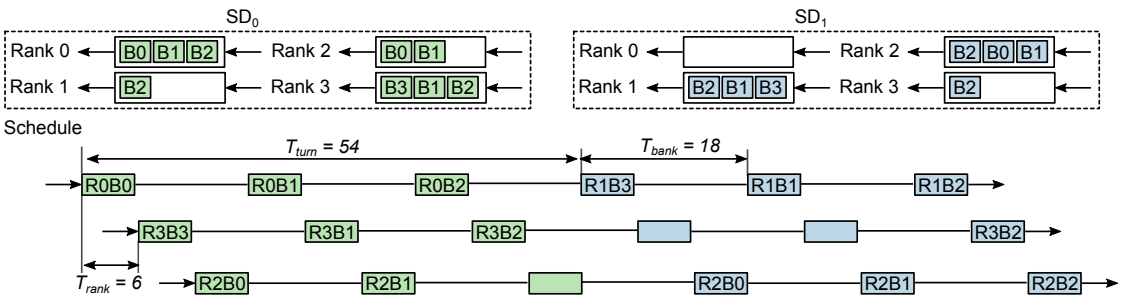


Figure 4.21: SecMC-NI scheduling with rank interleaving.

Interleaving Requests From Different Ranks

So far, we only considered which banks that memory requests access. However, we can construct a far more efficient scheduling if we consider both the rank and bank of a memory request. The basic idea is to construct a separate schedule for each rank, and interleave these schedules to form the final schedule. The request selection algorithm first picks the $\lfloor T_{bank}/T_{rank} \rfloor$ ranks with the most pending requests. For each chosen rank, the algorithm then picks the requests to different banks as described earlier. Once all requests are selected, SecMC-NI interleaves requests from different ranks to construct the final schedule. The ranks needs to be reordered based on the previous turn's schedule to avoid timing violations, similar to the bank reordering described above.

As a concrete example, consider the example in Figure 4.21 with the following timing parameters: $T_{turn} = 54$, $T_{bank} = 18$, $T_{rank} = 6$. The requests are grouped into separate queues based on which rank they access. SecMC-NI first picks the ranks with the most pending requests. With these timing parameters, at most three ranks can be interleaved. For domain 0, rank $\{0, 3, 2\}$ are selected to be issued in this turn. For each rank, requests to different banks are selected as described earlier. Each rank constructs its scheduling pipeline separately,

and these scheduling pipelines are shifted and combined to construct the final schedule as shown in Figure 4.21. The notation $RiBj$ indicates a request that accesses bank j in rank i. We separate each scheduling pipeline by T_{rank} cycles to avoid timing violations between requests to different ranks. For domain 1, rank {2, 1, 3} are selected to be issued. To avoid timing violations between security domains, SecMC-NI reorders the scheduling pipelines of a security domain so that each rank's scheduling pipeline is in line with previous security domain. For example, SD_1's ranks are scheduled in the order of {1, 3, 2} given that SD_0's rank schedule is {0, 3, 2}. This ensures that the requests that access the same bank and rank are separated by at least T_{turn} cycles. In the best case, a security domain can issue 9 requests in a single turn using SecMC-NI, which significantly improves the peak throughput.

Address Randomization

Although SecMC-NI has the same peak bandwidth as rank partitioning, it can only do so if requests are evenly distributed across different ranks and banks. If all requests access the same rank and bank, only one access can be issued per turn. To improve the scheduling efficiency, address randomization is implemented in SecMC-NI. The randomization maps a physical address to a randomized DRAM address by XORing a random bit vector so that requests are more evenly distributed across ranks and banks.

Security

SecMC-NI completely removes timing channels among security domains because the memory latency of each security domain is independent of accesses from other domains. The scheduling algorithm ensures that the same set of accesses are scheduled for each turn no matter which addresses are accessed by other domains. The ordering within a turn is hidden by delaying responses until all requests in a turn finish.

4.3.3 Peak Bandwidth Comparison

We compare the peak bandwidth utilization of SecMC-NI with that of existing schemes under our DRAM timing parameters (T_{bank} = 18 and T_{rank} = 6). The worst case time between two requests that have a bank conflict is 43 cycles. Hence, TP can issue one request every 43 cycles if the minimum turn length is used, resulting in only 9% utilization (assuming a burst length of 4). For bank partitioning (BP) and BTA, at most one request can be issued every 18 cycles, resulting in a bandwidth utilization of 22%. Rank partitioning (RP) can issue one request every 6 cycles, leading to 67% utilization. SecMC-NI's bandwidth utilization depends on the turn length. For T_{turn} = 43, at most 6 requests can be issued in a turn, resulting in a utilization of 56%. For T_{turn} = 54, SecMC-NI can issue 9 requests in a single turn in the ideal case, achieving the same bandwidth utilization (67%) as rank partitioning. Tabel 4.3 summarizes the bandwidth utilization of different schemes.

Scheme	TP	BP	BTA	RP	SecMC-NI
Bandwidth Utilization	9%	22%	22%	67%	67%

Table 4.3: Bandwidth utilization comparison between different schemes.

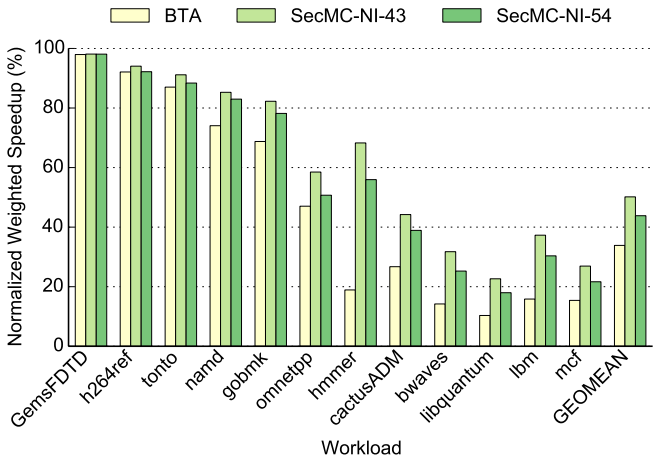


Figure 4.22: Performance comparison between SecMC and BTA.

4.3.4 Evaluation

We integrated DRAMSim2 into ZSim to evaluate the performance of SecMC-NI. We modeled a multi-core processor with 8 cores. Each core runs a benchmark from a different security domain. The experimental setup is similar to that described in Section 4.2.3.

Effect of Turn Length

We configure SecMC with two different turn lengths: 43 cycles and 54 cycles. The 43-cycle turn allows up to 6 requests per turn (3 ranks, 2 banks per rank). The 54-cycle turn allows up to 9 requests per turn (3 ranks, 3 banks per rank). Figure 4.22 shows the performance comparison between the two turn lengths. For programs with low memory intensity, BTA and SecMC perform almost equally well, reaching 80% of the baseline's performance. However,

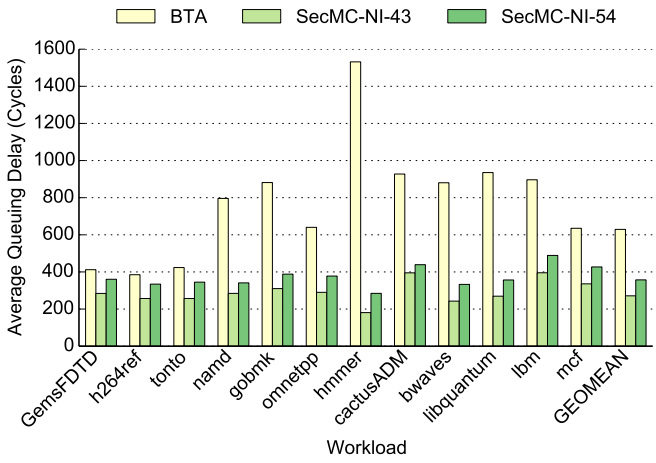


Figure 4.23: Queuing delay comparison between SecMC and BTA.

for memory-intensive programs, SecMC significantly outperforms BTA, almost doubling the performance of BTA in many cases. This shows the advantages of SecMC as it allows more flexible scheduling with a higher peak throughput by interleaving requests to different ranks and banks.

To better understand the results, Figure 4.23 shows the average queueing delay of memory requests under each scheme. The queueing delay is calculated as the difference between the time that a memory request arrives at a request queue and the time that the memory request gets issued to DRAM. Due to the inflexible schedule, BTA incurs high queueing delays (~600 cycles). SecMC-NI-43 cuts this queueing delay by half on average.

Comparison between SecMC-NI-43 and SecMC-NI-54 leads to another interesting observation. Although SecMC-NI-54 has a higher theoretical bandwidth utilization than SecMC-NI-43, the longer turn length of SecMC-NI-54 increases the average queueing delay of memory requests. Hence, we see SecMC-NI-43 actually outperforms SecMC-NI-54. For the rest of the evaluation section, we use 43 cycles as the turn length for SecMC-NI.

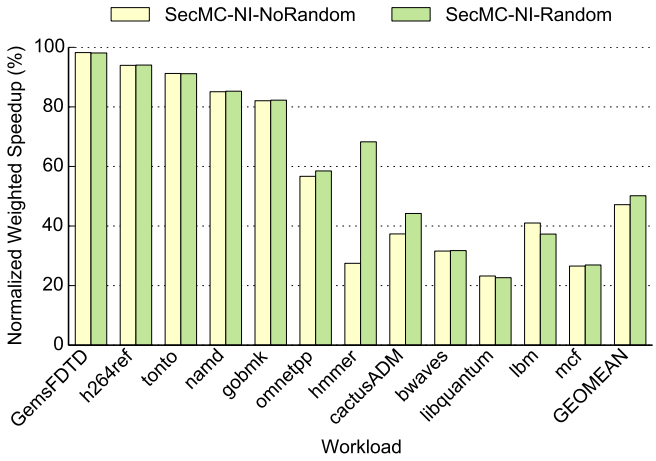


Figure 4.24: SecMC with and without address randomization.

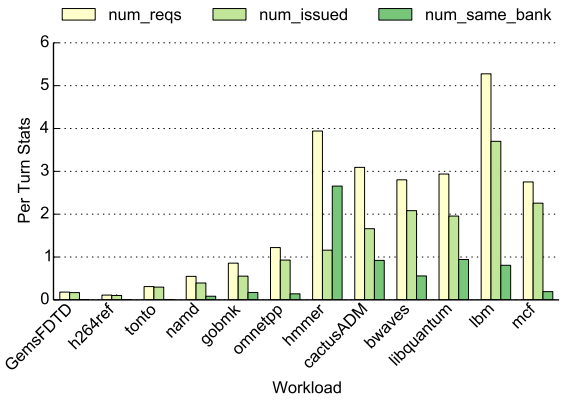


Figure 4.25: SecMem scheduling statistics.

Effect of Address Randomization

We then study the impact of address randomization on the performance of SecMC-NI. The purpose of address randomization is to distribute memory requests more evenly across different ranks and banks, thus helping SecMC-NI to issue more requests in a turn. We compare the performance of SecMC-NI with and without address randomization, as shown in Figure 4.24. On average, adding address randomization only improves the performance of SecMC by 2%. However, for some benchmarks such as *hmmer*, the improvement is significant.

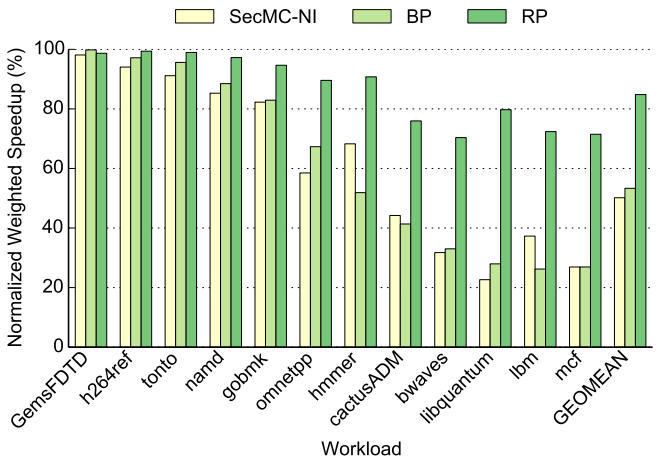


Figure 4.26: Performance comparison between SecMC and spatial partitioning.

To understand why *hmmer* benefits significantly from address randomization, we profiled the simulation without address randomization to record scheduling statistics in each turn. Figure 4.25 shows three scheduling statistics. *num_reqs* represents the average number of requests in the queue for the active security domain in each turn. *num_issued* represents the average number of requests that are issued in each turn. *num_same_bank* represents the average number of requests that try to access the same bank of a rank. The value of *num_same_bank* indicates the number of requests that cannot be scheduled together in a turn. As the figure shows, the value of *num_same_bank* is large for *hmmer*, meaning that a lot of the requests in *hmmer* incur a bank conflict. With the help of address randomization, *hmmer*'s requests are distributed more evenly across different banks, which explains the huge performance gain for *hmmer* when address randomization is enabled. For programs with many bank conflicts, address randomization is an effective way to improve the performance.

Comparison with Spatial Partitioning

SecMC-NI outperforms the current state-of-the-art (BTA) that places no restriction on memory allocation by 45% on average. Here, we compare SecMC-NI with spatial partitioning schemes. Figure 4.26 shows the performance comparison between SecMC-NI, bank partitioning (BP), and rank partitioning (RP). On average, SecMC-NI achieves similar performance as bank partitioning. For some benchmarks (e.g. *hmmer*, *lbm*), SecMC-NI even outperforms bank partitioning. This is mainly because SecMC-NI provides higher peak bandwidth; the consecutive requests only need to be separated by 6 cycles in SecMC-NI, while the minimum time interval between two requests in bank partitioning is 18 cycles. However, the results also show a significant performance gap between SecMC-NI and rank partitioning, even though SecMC-NI has the same theoretical bandwidth utilization as rank partitioning. This performance gap results from two factors. First, the delay for a request to wait for its turn in SecMC-NI is much longer than that of rank partitioning, as the minimum turn length is 43 cycles. Second, SecMC-NI can reach the peak bandwidth only when there are enough requests in a security domain that access different banks and ranks to utilize all the scheduling slots. This condition is usually not met for most benchmarks. Next, we propose a more efficient protection scheme called "SecMC-Bound" as an attempt to close this performance gap.

4.4 SecMC-Bound

4.4.1 Intuition and Overview

Schemes that completely remove timing interference such as TP and SecMC-NI are inherently inefficient because they need to guarantee noninterference even for the worst-case traffic. In particular, accesses from different secure domains must be far enough apart (43 cycles) in case they access the same bank of a rank, which requires a long turn length. Long turns lead to a significant queueing delay. In practice, however, normal traffic patterns should only experience a small number of bank conflicts.

In this section, we introduce a new scheme, named SecMC-Bound, which enables a trade-off between security and performance with an theoretic bound on information leakage. Allowing some timing interference enables the scheme to use dynamic scheduling optimized for common-case behaviors. SecMC-Bound hides common timing variations by delaying each response to a pre-determined expected response time, which is independent of accesses from other security domains. An access with significant interference may violate the expected response time. The information leak through such timing variations is controlled by delaying responses to the worst-case time of a completely secure scheme (such as TP or SecMC-NI) after a limited number of violations over each period.

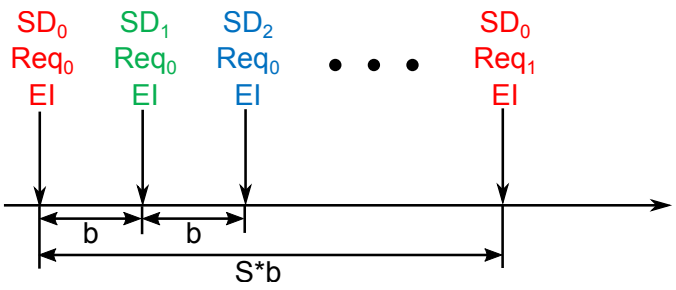


Figure 4.27: Expected issue times for memory requests.

4.4.2 SecMC-Bound Algorithm

As in other secure memory controller designs, SecMC-Bound assumes that there are per-domain input and output queues. Memory requests are stored in the input queue of the corresponding security domain, and responses from DRAM are stored in the output queue before being returned to the last-level cache (LLC).

Expected Times

Instead of relying on turn-based scheduling to remove timing interference between different security domains, SecMC-Bound assigns expected issue time to each memory request when the request is enqueued at the memory controller.

The expected issue time (EI) for a request defines the clock cycle when this request is expected to be issued to DRAM. Here, we introduce a parameter b, which represents the time interval between EIs of two consecutive requests from different security domains. Figure 4.27 shows an example of expected issue times. The EI for the i^{th} memory request from security domain s can be calculated by Equation 4.4.

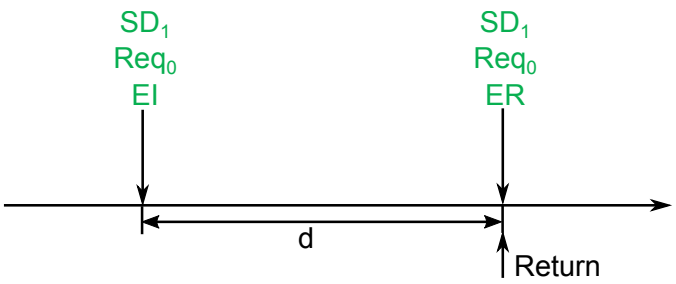


Figure 4.28: Expected response times for memory requests.

$$t_{EI}(s,i) = S \cdot b \cdot j + s \cdot b \tag{4.4}$$

where j is the minimum integer such that $t_{EI}(s,i) > t_{EI}(s,i-1)$ and $t_{EI}(s,i) >$ *enqueue time*. S represents the total number of security domains as we defined previously. Note that EI is only an estimate that is used for arbitration. In each clock cycle, the DRAM scheduler selects the request 1) with the lowest EI 2) among ones that can be issued (satisfying all DRAM timing requirements). The actual issue time of a request does not need to match its EI. For example, if there exist lots of bank conflicts, requests are likely to be issued later than their EIs because of high memory interference.

To hide the memory interference from potential attackers, we delay each memory response to return at certain clock cycles. The return time for a memory request is called expected response time (ER), which defines the clock cycle when this request will be returned to the CPU. We introduce another timing parameter d, which is defined as the difference between a request's EI and its ER. Figure 4.28 shows an example of expected response time for a request from SD_1. The function of d is to hide the interference between different security domains. Although requests from different security domains can delay each other in memory scheduling, this interference cannot be observed by an attacker as

long as every request is returned exactly at its ER, since the ER of a request from one security domain is independent of other security domains. Given the parameter d, ER can be trivially calculated with Equation 4.5.

$$t_{ER}(s,i) = t_{EI}(s,i) + d \tag{4.5}$$

ER Violation

When a request is completed, the corresponding response is put into a per-domain output queue. To hide timing variations in the DRAM scheduling, SecMC-Bound delays each response until its ER. The built-in delay (d) in ER hides small timing interference.

However, there is no guarantee that every request can finish before its ER and be returned at exactly ER. If the interference between memory requests is very intense, or d is set too small, it is likely that some requests cannot finish before their ERs. If a request is returned after its ER, we count it as an ER violation. ER violations represent information leakage as they allow an attacker to observe timing variations caused by memory interference. If a request is allowed to be returned at any clock cycle after its ER, an attacker is able to observe the exact delay value, hence extracting significant amount of information from one ER violation. To limit the amount of information that one ER violation leaks, we restrict a request to return only at predetermined delay values, i.e., d_i ($1 \leq i \leq W-1$). We also specify a worst-case delay value, d_W, in a way that a request is guaranteed to finish before $ER+d_W$, or worst-case response time

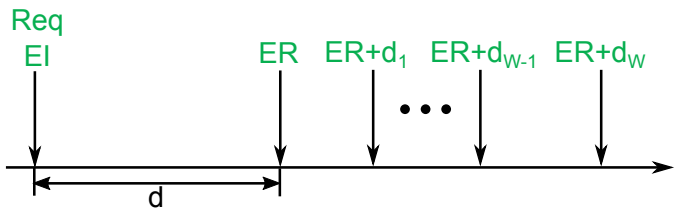


Figure 4.29: Predetermined delay values.

(WR). We will describe how to calculate the worst-case response time in the next section.

Figure 4.29 shows the predetermined delay values for a request. If a request finishes at cycle t_{finish}, it will be returned at cycle $ER+d_j$, where j is the minimum integer such that $ER + d_j > t_{finish}$. Under this scheme, a request that violates ER can only have W possible delays, which limits the amount of information that an attacker can extract from one ER violation.

SecMC-Bound increments a counter (m) to record the number of ER violations. Both EI and ER of the request are also incremented by d_k to account for the visible delay. The following memory request uses this updated EI as the previous request's EI in the constraint ($t_{EI}(s,i) > t_{EI}(s,i-1)$) when calculating its EI.

Worst-Case Times

This section describes how to calculate the worst-case times. The worst-case time represents the timing that the controller can guarantee a request to finish for any traffic pattern. The worst-case time is determined using a secure scheduling algorithm such as TP or SecMC-NI.

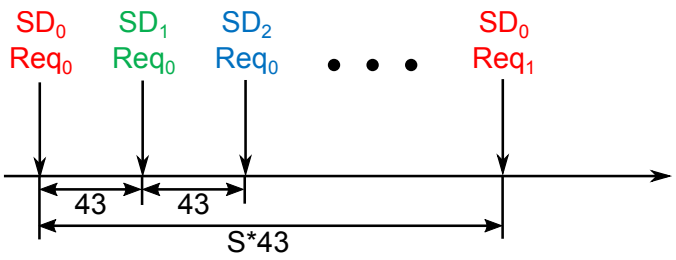


Figure 4.30: TP used as the worst-case scheduling algorithm.

As an example, consider using TP with the minimum turn length (T_{turn} = 43) as the worst-case scheduling algorithm. Under this TP scheme, one request can be issued every 43 cycles, and security domains take turns to issue their requests. Figure 4.30 shows an example TP schedule. With TP being used as the worst-case scheduling, the worst-case issue time (WI) for the i^{th} request from security domain s can be calculated by Equation 4.6.

$$t_{WI}(s,i) = S \cdot T_{turn} \cdot j + s \cdot T_{turn} + offset_{refresh} \tag{4.6}$$

where j is the minimum integer such that $t_{WI}(s,i)$ is greater than $t_{enq}(s,i)+T_{turn}$ and $t_{EI}(s,i-1)+S \cdot T_{turn}$. Here, $t_{enq}(s,i)$ is the enqueue time, T_{turn} is the minimum turn length (43) and $offset_{refresh}$ represents the delay due to DRAM refresh cycles. Note that the EI of the previous request ($t_{EI}(s,i-1)$) in the constraint incorporates delays due to ER violations and represents time after the actual issue-time. Therefore, the WI of a request is defined to be at least $S \cdot T_{turn}$ cycles away from the previous request's actual issue-time and at least T_{turn} cycles away from the time that the request is enqueued.

The above construction provides an enough margin for a memory controller to enforce the worst-case time for any traffic patterns. If there is a request whose WI is less than T_{turn} cycles away, our DRAM scheduler enforces a dead time so

that no new requests can be issued and all in-flight transactions will be drained. This guarantees that the request can be issued by its WI. Effectively, the scheduling follows TP for one turn. This scheduler design ensures that every request is issued before its WI. In turn, the worst-case response-time (WR) can be computed using a DRAM access latency:

$$t_{WR}(s,i) = t_{WI}(s,i) + t_{RCD} + t_{CAS} + t_{BURST} \tag{4.7}$$

Limiting the Number of *ER* Violations

SecMC-Bound enables a trade-off between performance and the number of *ER* violations through the parameters b, d, and W. Larger b or d will reduce the number of *ER* violations, but at the cost of increased memory latencies.

The number of *ER* violations directly correlates with the amount of leaked information. In order to bound the information leakage, SecMC-Bound limits the number of *ER* violations that can happen in a certain time interval. To enforce the limit, SecMC-Bound maintains counters for the number of *ER* violations (m) and the number of read requests (n) for each security domain. SecMC-Bound can be configured to only allow up to M *ER* violations over a period (N read requests or C cycles).

If m reaches the limit M for one security domain, SecMC-Bound switches to the conservative worst-case mode for that security domain and delays every response until its *WR*. As a result, there cannot be any more *ER* violations for that security domain. Note that this restriction does not change the DRAM schedul-

ing or accesses from other security domains. Once a period is over, the counters are reset and the output queue again uses *ER*s to delay responses.

Handling Conflicts of Response Time

If no *ER* violation happens, the *ER*s of any two requests are always different, i.e., at most one response needs to be returned to the CPU at any cycle. This property is guaranteed by equation 4.5. For two requests in the same security domain, the request that arrives later always has a higher *ER*. For two requests from distinct security domains, the different security domain IDs guarantee that their *ER*s always differ by a multiple of b cycles.

However, if *ER* violations do happen, it is possible that two requests from different security domains are assigned the same *ER* because they use different delay values (d_i where $1 \leq i \leq W$). Since a memory controller can only return one memory response to the CPU in each clock cycle, one of these responses have to be delayed—a possible timing channel. To avoid the conflicts of response time, we apply time multiplexing to the memory controller's response queue. Assume a request from security domain s finishes and the corresponding response is stored in the response queue. Instead of returning this response at t_{ER}, we further delay this response to cycle t'_{ER}, which satisfies:

$$0 \leq t'_{ER} - t_{ER} < S \,\&\, t'_{ER} \; mod \; S == s \tag{4.8}$$

t'_{ER} calculates the next available time slot for security domain s to return its response given a strict round-robin schedule. The extra delay does not leak information because it comes from a static schedule. Although this so-

lution means some responses will be delayed, the number of delayed cycles is bounded by the number of security domains (S). The performance overhead of this tiny delay is almost negligible.

4.4.3 Performance Optimizations

Avoiding Worst-Case Times

When the number of ER violations m reaches the limit M, SecMC-Bound switches to the worst-case mode, which incurs significant performance overhead to that security domain. To reduce the chance of entering the worst-case mode, we can gradually increase the value of d as the number of violations increases. As an example, assume that the limit (M) is 3 violations over 1 million requests. We set the initial value of d to be d_{init} and adjust the value of d based on the number of ER violations (m) for a security domain.

- If $m = 0$, $d = d_{init}$.
- If $m = 1$, $d = d_{init} + delay_1$.
- If $m = 2$, $d = d_{init} + delay_2$.
- If $m = 3$, $d = d_{init} + d_W$.

Because d increases with m, an ER violation is less likely to happen when m is large. As a result, SecMC-Bound is unlikely to enter the worst-case mode. After a period, the counter m is reset to 0 and d is reset to d_{init}. The optimization can be applied to any value of M by defining d for each possible value of m. Note that

a similar optimization can be applied to b to also reduce the chance of entering the worst-case mode.

Dynamic Tuning of d_{init} and b_{init} Value

SecMC-Bound uses two design parameters d and b to determine expected response times. Intuitively, these two parameters represent different points in the security-performance trade-off space, and we may be able to improve performance with minimal impact on security if we can properly choose d and b depending on application characteristics. For example, applications with infrequent memory accesses may not experience any ER violation even with a small d, which results in a shorter memory latency.

The optimization in Section 4.4.3 adjusts the value of d within each period, but the initial value of d (d_{init}) is still fixed. Here, we discuss how the value of d_{init} can be dynamically adjusted.

In this scheme, we adjust d_{init} at the end of each period based on the number of ER violations (m) observed in that period. The updated d_{init} is used in the following period.

- If $m = 0$, $d_{init} = d_{init}$ - 10.
- If $m \geq M - 1$, $d_{init} = d_{init} + 10$.
- Otherwise, $d_{init} = d_{init}$.

The above algorithm decreases d_{init} if there was no ER violation in the previous period ($m = 0$), and increases d_{init} if the number of ER violations almost

reached the limit ($m \geq M - 1$). The algorithm can be adjusted with different thresholds to change d_{init} more aggressively. For example, a more aggressive scheme may decrease d_{init} when $m < M - 1$. This design can decrease d_{init} to a lower value, but is also likely to result in more ER violations. In our experiments, we used the shown algorithm, which more conservatively change d_{init}.

The same approach can be applied to dynamically adjust the initial value of b (b_{init}). Dynamic tuning of d_{init} and b_{init} has two benefits. First, the dynamic tuning allows the scheme to adapt to different phases of an application. Memory-intensive phases may require large d_{init} and b_{init} to reduce the number of ER violations, while less memory-intensive phases can use smaller d_{init} and b_{init} to improve the performance. Second, the dynamic tuning allows adapting to different workloads without manual designer efforts. The scheme will automatically adjust itself to meet the specified limit on the ER violations for a given workload.

Combining with Spatial Partitioning

We observe that SecMC-Bound is orthogonal to spatial partitioning techniques such as rank/bank partitioning. In fact, combining SecMC-Bound with spatial partitioning can yield higher performance than simply using spatial partitioning. In rank/bank partitioning, the requests are scheduled in a static round-robin fashion. Even if a security domain is not memory-intensive and generates few memory requests, its scheduling time slots are reserved and cannot be utilized by other memory-intensive security domains. In contrast, SecMC-Bound allows dynamic scheduling among security domains and hides the timing variations by delaying memory responses. Hence, memory-intensive programs

could get higher bandwidth when other programs are not using the memory. In a mixed scheme, the memory controller can start with SecMC-Bound. When a security domain reaches the limit on number of violations, the mixed scheme can switch to the spatial partitioning schemes for complete security. As long as SecMC-Bound provides better performance than spatial partitioning before the switch happens, the overall system performance still beats that of applying spatial partitioning alone.

4.4.4 Information Theoretic Bound

SecMC-Bound is designed so that the response time of each memory request only depends on requests within its security domain, except for ER violations. The added delay on an ER violation is the only property that depends, partially, on memory requests from other security domains, leading to potential timing channels. Here, we present an information theoretic analysis that enables us to conservatively compute a quantitative upper bound on the rate of information leakage based on the number of ER violations.

We start by making the following definitions. Let $\mathbf{x}$ be the history of all memory requests, from all security domains, from time $-\infty$ to the present. Let $\mathbf{Y}$ be the vector of delays $(y_1, y_2, ...y_n)$ seen by a receiver program that places n read requests such that $y_i \in \{0, d_1, ...d_{W-1}, d_W\}$ $\forall i$ s.t. $0 < d_1 < ...d_W$. Note that any $y_i > 0$ is considered an ER violation.

In the context of a covert channel between malicious programs, channel capacity is the most natural information theoretic metric. It can be computed as: $C = \max_x I(\mathbf{X}; \mathbf{Y}) = \max_x \{H(\mathbf{Y}) - H(\mathbf{Y}|\mathbf{X})\}$, using the usual information theoretic

definition of entropy. Because $\mathbf{Y}$ is deterministic given $\mathbf{X}$ for a memory controller, $I(\mathbf{X};\mathbf{Y}) = H(\mathbf{Y})$. In general, mathematically analyzing the precise distribution of $\mathbf{Y}$ is non-trivial. To simplify the analysis, we conservatively assume that all probability distributions over the possible values of $\mathbf{Y}$ are attainable. In this case, the entropy of $\mathbf{Y}$ is maximized when the distribution is uniform over all $\mathbf{y} \in \mathbf{Y}$ and the channel capacity can be simply computed with the number of possible $\mathbf{y} \in \mathbf{Y}$.

Given that SecMC-Bound limits the ER violations to be at most M out of N requests, the number of possible values of $\mathbf{Y}$ can be computed using a basic combinational analysis. The leakage is simply the logarithm of this value:

$$\max_{\mathbf{X}} I(\mathbf{X};\mathbf{Y}) = \log_2\left(\sum_{m=0}^{M} W^m \binom{N}{m}\right). \tag{4.9}$$

In the context of an unintentional side channel where a malicious listener snoops on the activity of other programs, maximum leakage, which can be computed as $\mathcal{L}(\mathbf{X} \to \mathbf{Y}) \equiv \log_2(\sum_y \max_{p(x)>0} p(y|x))$, is a popular metric for the rate of information leakage. In our case where $\mathbf{Y}$ is deterministic in $\mathbf{X}$, maximum leakage also reduces to the logarithm of the number of possible values of $\mathbf{Y}$. Hence, the bound for maximum leakage is the same as the bound for covert channel capacity.

Practical Channel Capacity. The above bound is calculated under conservative assumptions, which are not true in practical systems. The bound assumes that all conceivable $\mathbf{Y}$ distributions are possible and that a malicious program can choose any of these distributions at will. However, in practice, memory accesses are much more likely to result in no ER violation or a low delay even on an ER violation. The bound also captures the information leak from *all* programs to

a receiver even though attackers in practice only control a subset of programs and other accesses add uncontrolled noise. The bound also assumes that an attacker can control and measure all memory accesses at a cycle granularity. This is unlikely in practice. For example, caches affect memory requests and it is difficult to maintain perfect synchronization between concurrent programs.

In that sense, we provide a conservative bound for channel capacity. The practically achievable channel capacity is likely be at least two to three orders of magnitude lower than our information theoretic bound under ideal assumptions. For example, a previous study [HKR+15] showed the covert-channel capacity around 500 Kbps based on memory contention even when every memory access could be used to leak information in theory. Fortunately, our experimental results show that even our conservative bound can be used to provide good performance with a guarantee on low information leakage. We leave the further refining the bound considering practical limitations as future work.

4.4.5 Evaluation

We study the performance of SecMC-Bound under different security guarantees. The experimental setup is the same as SecMC-NI. In these experiments, we also used a more diverse workloads that consist of different SPEC benchmarks. We randomly constructed 20 such workloads. We only show the results for representative 10 workloads in figures, but found that all 20 workloads show similar trend. The 10 workloads we used are shown in Table 4.4.

mix-1	ton les gam gro gam h264 mcf bwa
mix-2	gcc cac h264 zeu gam cac gam ton
mix-3	sje per zeu gam les les gcc gcc
mix-4	xal gro gcc bzi dea gob ton hmm
mix-5	nam zeu gam per omn gcc ton zeu
mix-6	per xal cac dea gob les cac gob
mix-7	per gcc cac mcf zeu per omn lib
mix-8	sop gam mil gro hmm sje les lib
mix-9	ton xal omn gcc ton h264 h264 hmm
mix-10	zeu hmm lbm les mcf mcf mcf bwa

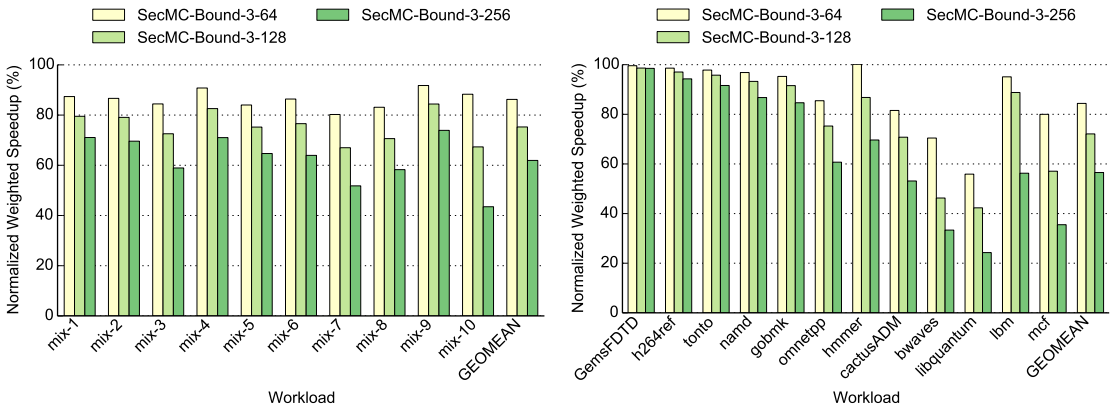
Table 4.4: Mixed workloads.

Parameter Sweep for b and d

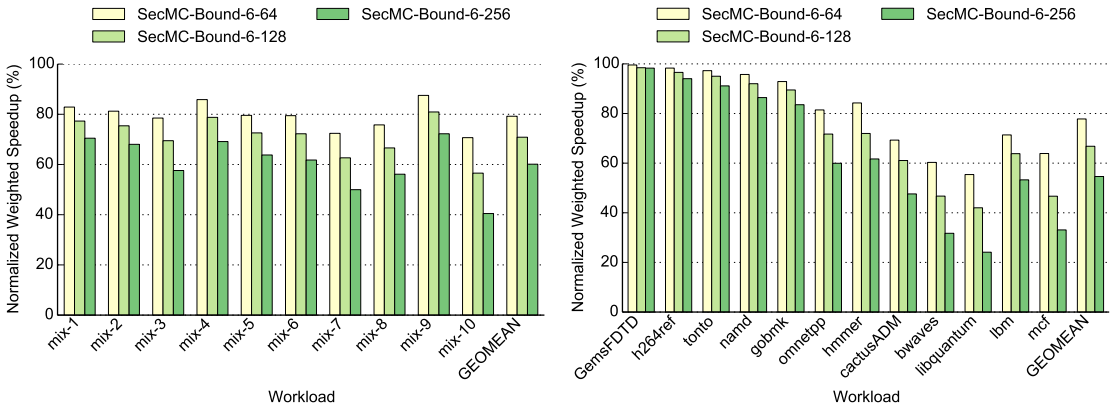
To explore the design space, we tried different values for the design parameters, b and d. We used $\{3, 6, 18\}$ for b and $\{64, 128, 256\}$ for d. Note that in these experiments, we do not restrict the number of violations over a period. For these experiments, we used $W = 3$ with $d_1 = 30$, $d_2 = 160$ and d_3 to be the worst-case delay. The performance was normalized to the insecure baseline.

Figure 4.31 shows the performance of SecMC-Bound across different parameter values. We use the notation SecMC-Bound-b-d to represent the scheme with different b and d. For a fixed b, the performance of SecMC-Bound decreases as d increases because a smaller d leads to an earlier expected response-time (ER). A large d increases the memory latency. For a fixed d, the performance decreases as b increases. This is because b affects the expected issue-time (EI), hence also indirectly affecting the expected response-time (ER). A larger b leads to a later expected response-time.

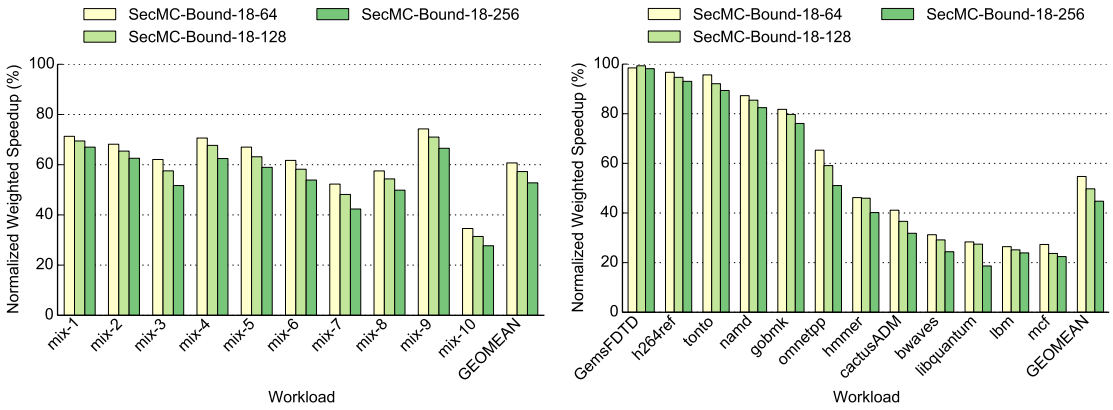
In summary, small b and d improve the performance of SecMC-Bound. For example, SecMC-Bound-3-64 achieves nearly 90% of the baseline's performance, which even beats rank partitioning. However, the high performance of



(a) b=3.

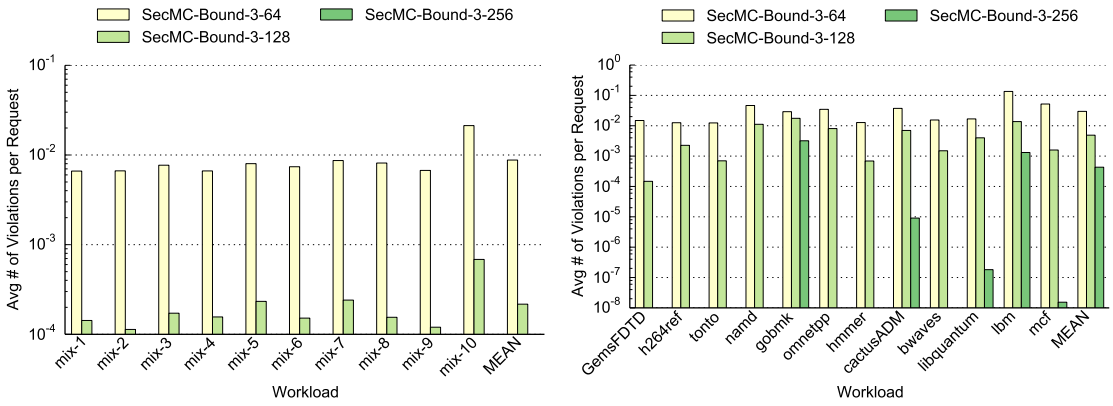


(b) b=6.

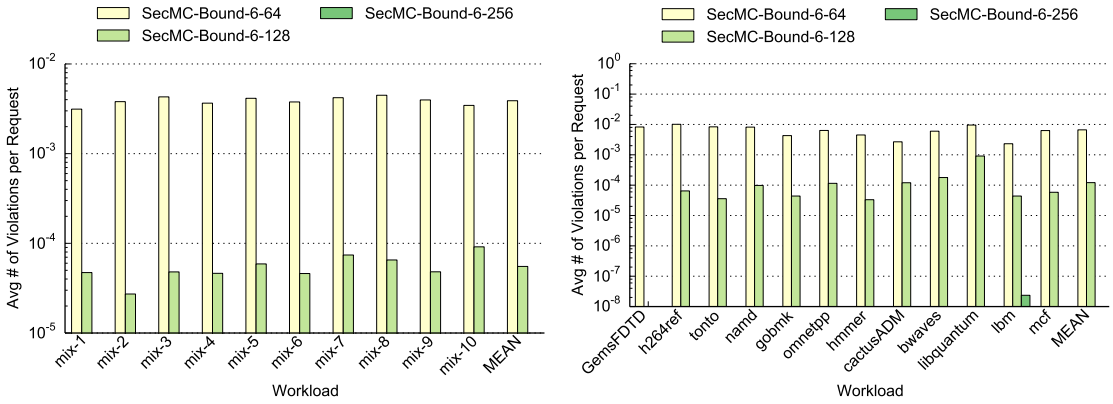


(c) b=18.

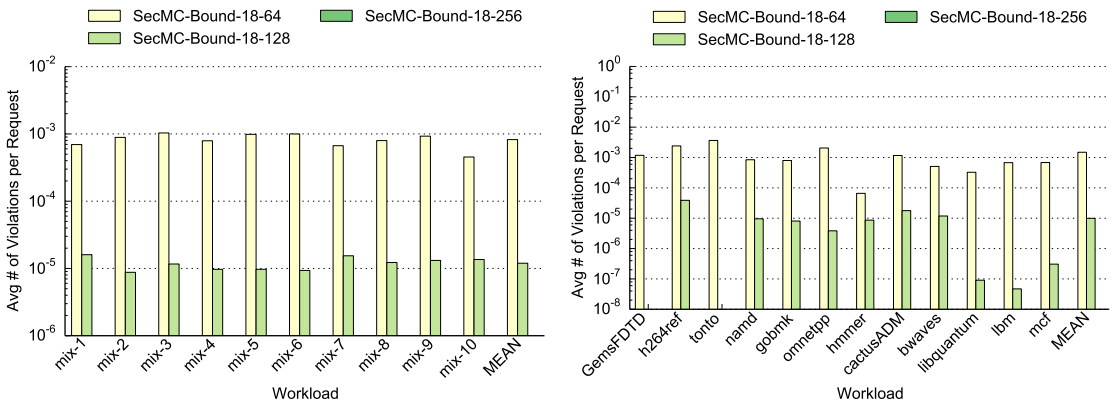
Figure 4.31: Performance with different value of b and d.



(a) b=3.



(b) b=6.



(c) b=18.

Figure 4.32: Number of violations with different value of b and d.

SecMC-Bound-3-64 comes with a lower security level. To see how the security is affected, we plot the average response time violations per memory request in Figure 4.32. Note that the y-axis uses a log scale. As the figure shows, small b and d leads to more frequent violations, which indicates more potential information leak. Fortunately, the number of response time violations decreases exponentially as we increase d. Even though a large d leads to lower performance, SecMC-Bound-6-256 still achieves 60% of the baseline's performance, which is better than SecMC-NI and BTA.

Figure 4.31 and 4.32 show that SecMC-Bound provides a large trade-off space between performance and security. Users can tune the performance of their memory controller based on how much security they are willing to sacrifice.

Limiting the Response Time Violations

We study the performance of SecMC-Bound when we apply the mechanism to enforce the number of violations in a period. Figure 4.33 shows the performance results for two sets of workloads with b = 6 and d = 160. We use four different limit values. "4 in 1,000" means we allow 4 violations in every 1,000 requests. Once the number of violations for a security domain reaches the limit in a period, this security domain enters the worst-case mode, in which it always uses the worst-case response time derived from TP. As can be seen from the results, enforcing a limit on the number of violations introduces some performance overhead to SecMC-Bound. As the limit gets lower, a security domain is more likely to enter worst-case mode, which lowers its performance.

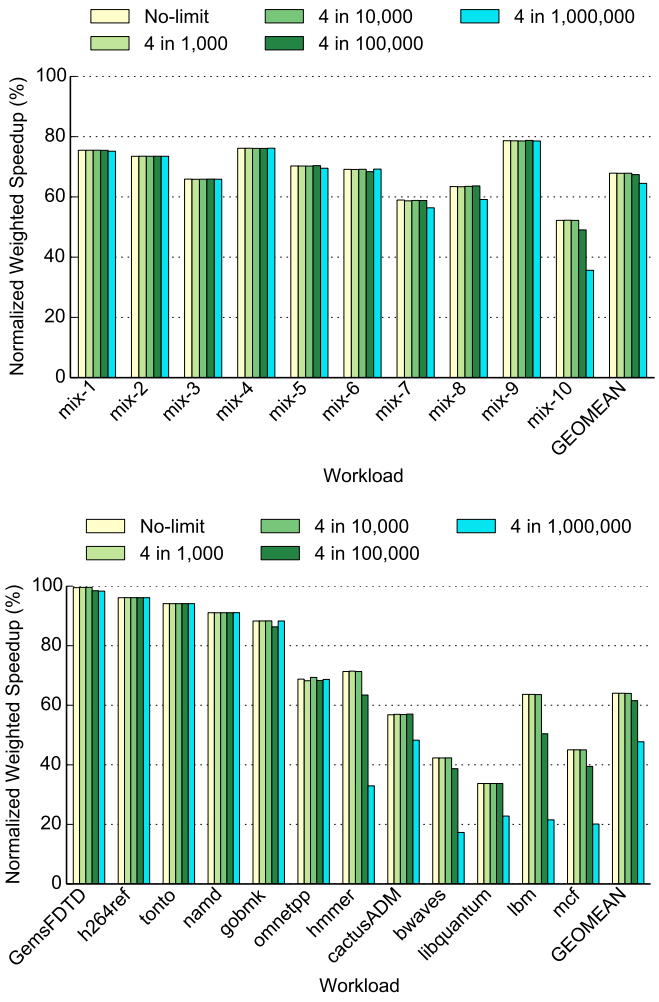


Figure 4.33: Performance with limit on violations.

Yet, enforcing a limit on the number of violations is necessary to provide a bound on the information leak. Figure 4.34 shows the leakage rate for each of the limit. As we lower the limit, the leakage bound drops accordingly. A designer may choose the limit based on the assets he or she wants to protect. For example, if the asset is a large file such as an HD movie, even leaking hundreds of bits per second may be acceptable. On the other hand, if the scheme needs to protect a small secret such as a cryptographic key, the limit will need to be much lower.

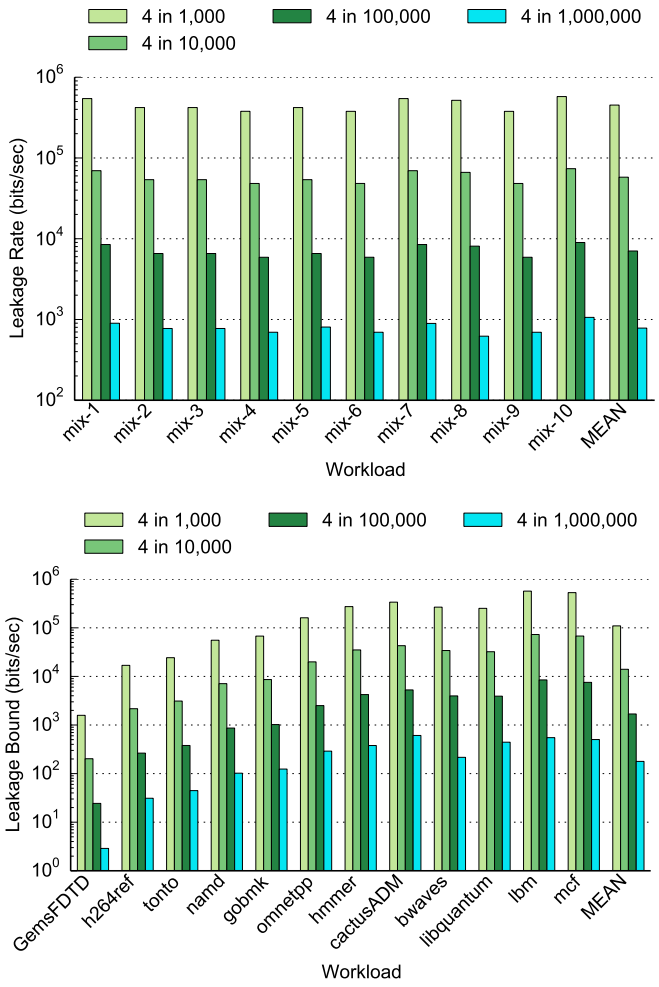


Figure 4.34: Leakage rate with limit on violations.

Optimization 1: Avoiding Worst-Case Times

The high performance overhead for the low limit in Figure 4.33 is mainly caused by the worst-case mode where all responses are delayed to their WR. One of the proposed optimizations reduces the chance of reaching the ER violation limit and entering the worst-case mode by gradually increasing the value of d. To see the effectiveness of this optimization, we ran experiments with this optimization implemented—the value of d increases by 10 whenever an ER violation happens within each period. Figure 4.35 shows the performance results for the

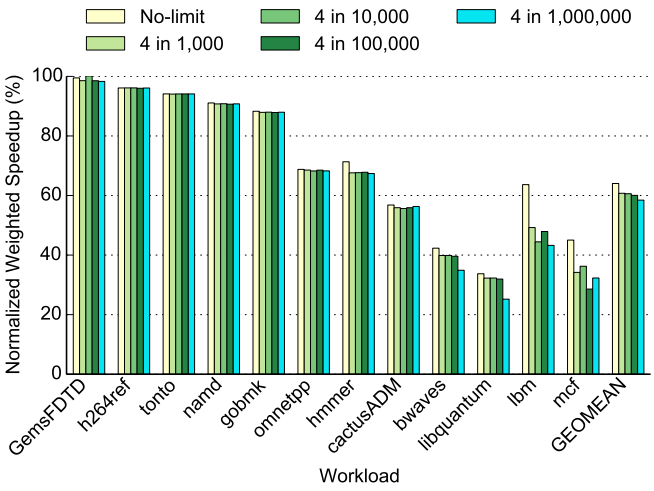


Figure 4.35: Performance after optimization.

single-benchmark workload with this optimization. Compared to the results without the optimization shown in Figure 4.33, the performance drastically increases for the low ER violation limit (4 in 1,000,000). The result suggests that this optimization is effective in preventing a security domain from frequently entering the worst-case mode when the limit is low. However, the performance is slightly degraded for cases with a high limit. Security domains do not often reach the ER violation limit in these cases, hence increasing the value of d actually increases the average memory latency. The results suggest that the optimization should be used when a security domain is likely to reach the limit on ER violations in a period.

Optimization 2: Dynamic Tuning of d_{init} Value

We study the performance impact of dynamically tuning the d_{init} value based on the number of ER violations in the previous period. In this experiment, the initial value of d (d_{init}) is set to 160 and b is set to 6. Figure 4.36 shows the performance with this optimization under different ER violation limits. When the

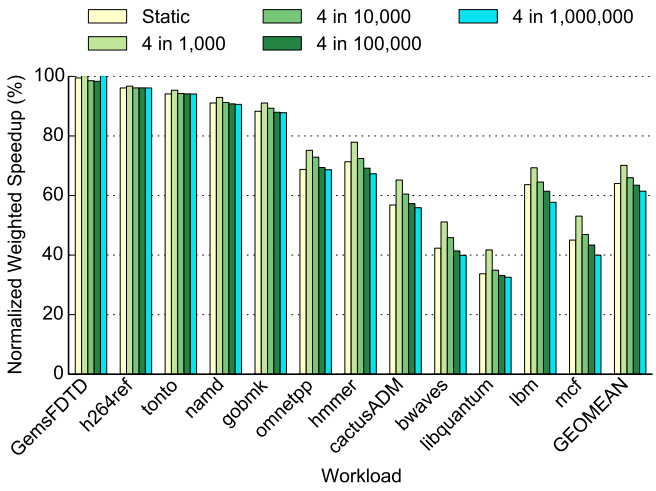


Figure 4.36: Performance by tuning $d0$ value.

limit is high (4 in 1,000), we see a noticeable performance improvement over the static case with a fixed d_{init}. This is because the optimization drastically reduces the value of d_{init} (e.g., from 160 to 72) while the static case uses a fixed d_{init} value (160). However, when the limit becomes lower, the performance improvement gradually drops down to zero. For a lower limit with a longer period, m over a period is more likely to be non-zero and d_{init} often cannot be reduced.

Optimization 3: Combining with Spatial Partitioning

We study the performance of SecMC-Bound when combined with spatial partitioning techniques. In this experiment, we use $b = 3$ and $d = 24$ ($t_{RCD} + t_{CAS} + t_{BURST}$). We choose small b and d values because spatial partitioning should alleviate the interference between security domains, thus requests can be issued and returned faster. We do not enforce a limit on the number of violations so that we can see the best possible performance of the mixed scheme. The performance results for the mixed workloads are shown in Figure 4.37. Combining SecMC-Bound with spatial partitioning significantly outperforms using spatial

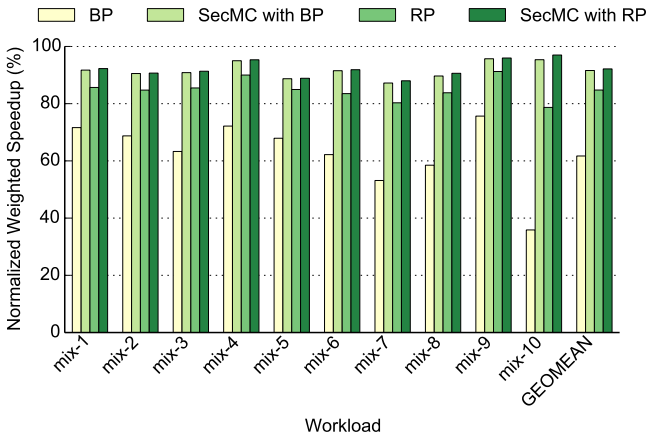


Figure 4.37: Combining SecMC-Bound with spatial partitioning.

partitioning alone. However, the combined schemes do incur timing violations on some requests. If we apply the enforcement mechanism to limit the number of violations in each period, it is likely that we will see some performance loss in the mixed scheme. Nevertheless, since the mixed scheme switches to RP/BP after the violation limit is reached, the overall performance of the mixed scheme will still beat that of RP/BP.

Comparison with Previous Schemes

Figure 4.38 shows the performance comparison between SecMC-Bound and other secure memory controller schemes. In this experiment, we use $b = 6$ and $d = 160$ to represent the performance of SecMC-Bound. The performance of SecMC-NI is significantly higher than BTA and close to BP. SecMC-Bound achieves 70% of the insecure baseline performance, outperforming BTA, SecMC-NI and even BP. The performance benefit shows that SecMC-Bound allows more flexible memory scheduling than completely secure schemes while providing a theoretic bound on information leakage.

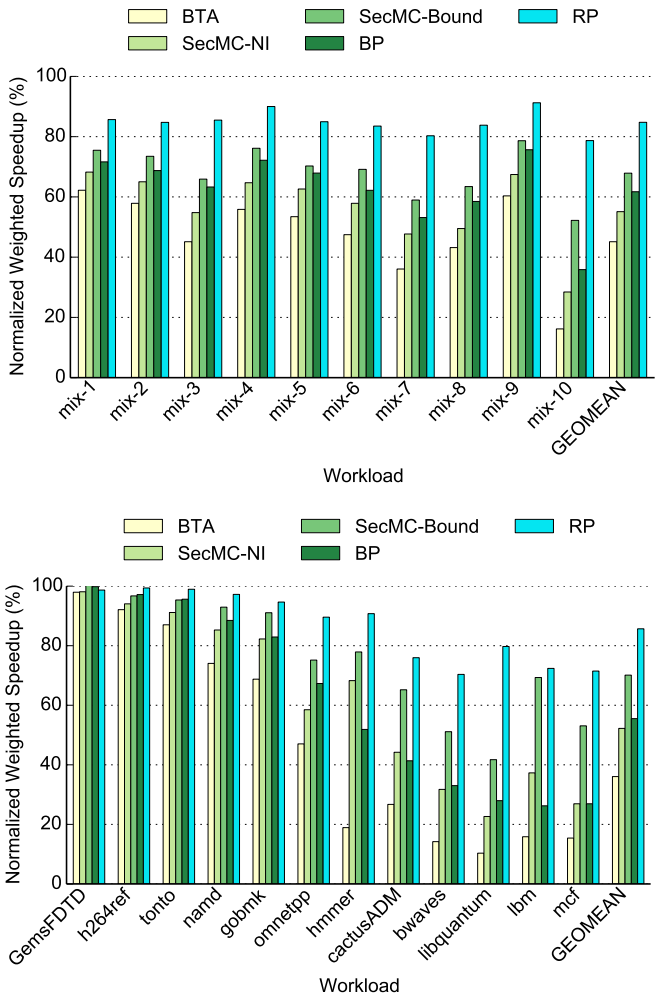


Figure 4.38: Performance comparison of different schemes.

Summary

Figure 4.39 shows the summary of the secure memory controller design space using the performance for the mixed workloads. SecMC schemes do not require spatial partitioning and outperform BTA, which represent the best previous scheme without spatial partitioning. SecMC-NI achieves similar performance as bank partitioning. SecMC-Bound's performance spans from 56% to 87% of the baseline, depending on the values chosen for b and d, enabling a trade-off

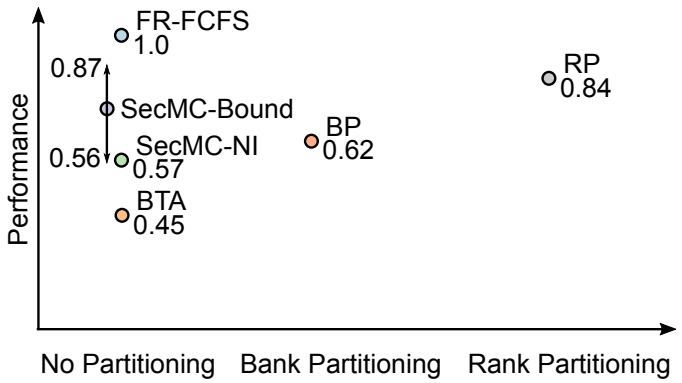


Figure 4.39: Design space summary (mixed workloads).

between performance and security with a bounded information leakage. While not shown here, we found that SecMC-Bound can also be combined with spatial partitioning to further improve their performance.

CHAPTER 5

RTL VERIFICATION

In the previous chapters, we propose several timing channel protection schemes for hardware components in a multi-core processor. While we believe our protection schemes are secure, an actual hardware implementation may be insecure either because there is a flaw in a protection scheme or because a hardware designer implements a secure scheme incorrectly. Therefore, it is important to verify the actual hardware design is secure in a low level abstraction—RTL (Register Transfer Level). In this chapter, we use a tool called SecVerilog [ZWSM15] to verify secure processors in RTL at compile time. Section 5.1 introduces some background knowledge about SecVerilog. Section 5.2 and Section 5.3 describe our experience of designing a single-core MIPS processor and a multi-core PARC-tiny processor using SecVerilog.

5.1 Background: SecVerilog

SecVerilog is a new hardware description language (HDL) that adds a security type system to Verilog so that hardware-level information flows can be checked statically. SecVerilog compiler is implemented based on Icarus Verilog [ive]. It extends Verilog with the ability to give each signal a *label* that specifies the security level of the signal. SecVerilog then verifies that the information flows between signals conform to a specified security policy using the security labels. All forms of information flow are tracked, including implicit flows and timing channels.

```
1  reg[18:0]       tag0[256],tag1[256];
2  reg[18:0]       tag2[256],tag3[256];
3  wire[7:0]       index;
4  wire[1:0]       way;
5  wire[18:0]      tag_in;
6  wire            write_enable;
7
8  always @(posedge clock) begin
9    if (write_enable) begin
10     case (way)
11     0: begin tag0[index]=tag_in; end
12     1: begin tag1[index]=tag_in; end
13     2: begin tag2[index]=tag_in; end
14     3: begin tag3[index]=tag_in; end
15     endcase
16   end
17 end
```

Figure 5.1: A simple Verilog code example.

5.1.1 SecVerilog Examples

To see how SecVerilog works, consider a simple Verilog code example that updates the tags of a 4-way cache, as shown in Figure 5.1.

We use this code to design a secure partitioned cache described in Section 3.6.5, in which way 0 and 1 (*tag*0 and *tag*1) are used as the *L* partition, and way 2 and 3 (*tag*2 and *tag*3) are used as the *H* partition. The code writes a new cache tag to a way specified by *way* when *write_enable* is asserted.

The secure partitioned cache has several requirements on how the cache can be used. First, *tag_in* must not contain confidential information when *way* is 0 or 1, to prevent the *H* partition from affecting the state of *L* partition. Second, *write_enable*, which controls whether a write occurs, cannot be influenced by

```
1  reg[18:0]   {L} tag0[256],tag1[256];
2  reg[18:0]   {H} tag2[256],tag3[256];
3  wire[7:0]   {L} index;
4  //Par(0)=Par(1)=L   Par(2)=Par(3)=H
5  wire[1:0]   {Par(way)} way;
6  wire[18:0]  {Par(way)} tag_in;
7  wire        {Par(way)} write_enable;
8
9  always @(posedge clock) begin
10   if (write_enable) begin
11     case (way)
12       0: begin tag0[index]=tag_in; end
13       1: begin tag1[index]=tag_in; end
14       2: begin tag2[index]=tag_in; end
15       3: begin tag3[index]=tag_in; end
16     endcase
17   end
18 end
```

Figure 5.2: Labeled code example.

confidential information when *way* is 0 or 1. Verifying these restrictions is tricky since signals such as *tag_in* and *write_enable* are shared by *L* and *H* partitions.

SecVerilog solves this problem by giving each signal a security label and verifying information flows are legal according to a security policy. In this example, the security policy is that information is not allowed to flow from *H* to *L*. Using the same example, the labeled code is shown in Figure 5.2.

The functionality of this Verilog code example does not change. The only modification is the added security labels to each signal declaration. For example, cache way 0 and 1 are labeled as *L* to indicate they are the *L* partition. With the security labels, SecVerilog is able to check every assignment statement in the code and report any illegal statement. For example, if a signal with *H* label is assigned to another signal with *L* label, it will be considered as an illegal as-

```
1  wire {L} in1;
2  wire {H} in2;
3  wire {L} sel;
4  reg  {L} out;
5
6  always @(*) begin
7    if (sel == 1'b0)
8      out = in1;
9    else
10     out = in2;
11 end
```

Figure 5.3: An insecure two-input mux.

signment because it violates the security policy that the information flow from H to L is disallowed. However, in the code example above, the signal *tag_in* can be assigned to any cache way based on the value of *way*, it will be insecure to assign either H or L to *tag_in*.

A novel feature of SecVerilog is the support of dependent types, which allows a signal's security label to change based on the value of another signal. In the code example above, the label function *Par*(*way*) utilizes this feature. *Par* denotes a type-level function that maps 0 and 1 to L, and 2 and 3 to H. When *way* is 0 or 1, the security label of *tag_in* becomes L. Since *tag_in* is being assigned to *tag*0 or *tag*1, which also has a L label, the assignment is legal. Same applies to the case when *way* is 2 or 3.

As another example, consider an insecure mux design shown in Figure 5.3. The security label of the mux output is L, while one of the input signals has a H label. When checking this code, SecVerilog reports the assignment at line 10 as illegal, because the assignment directly passes the value from a H signal to a L

```
1  reg[7:0] {H}       secret;
2  reg[7:0] {L}       public, x;
3  reg[7:0] {LH(x)} y; // LH(0)=L LH(1)=H
4  always@(posedge clock) begin
5    if (x==1) begin
6      y <= secret;
7    else begin
8      public <= y;
9    end
10 end
```

Figure 5.4: An example of implicit declassification.

signal. With this automatic information flow tracking in SecVerilog, hardware designers can easily identify and fix security vulnerabilities in RTL design.

5.1.2 Implicit Declassification

Whenever the value of a variable changes, the meaning of any security label that depends on it also changes. To be secure, SecVerilog needs to prevent such changes from implicitly declassifying information. Consider the example in Figure 5.4. This code is clearly insecure since it copies *secret* into *public* when x changes from 1 to 0.

At the assignment to y in the first branch, y's level is H because x is 1. But at the assignment to *public*, the level of y is L because x is 0. The insecurity arises from the change to the label of y during the execution, while its content remains the same. In other words, if x changes from 1 to 0, the label of y cannot protect its content.

SecVerilog relies on a dynamic mechanism to erase register contents when the security level of a register changes and information is not allowed to flow from the old level to the new level. Code to dynamically zero out registers is automatically inserted as part of the translation to Verilog.

While this dynamic mechanism may affect the functionality of the original hardware design, it may not be a major issue in practice. In the design of a secure MIPS processor (see Section 5.2), registers that have dependent security labels need to be cleared when the security label changes from H to L. However, this clearing is rare and indeed necessary for security. In fact, the new value of the register need not to be zero—any constant value suffices. Hardware designers can explicitly specify the new value of a register when the clearing happens so that the functionality of the hardware is not affected. Although automatically inserting the right constant values is not currently supported in SecVerilog, the type checker can notify a designer when automatic clearing is generated to help the designer specify the clearing values.

5.2 A Single-Core MIPS Processor

We used SecVerilog to design and verify a secure MIPS processor. We sketch the processor design, and show how SecVerilog helps avoid security vulnerabilities. We then provide results on the overhead of timing channel protection. Overall, we found that the capability to statically control information flow at a fine granularity (by labeling the software with language-based protection framework [ZAM12]) enables efficient secure hardware designs.

Module Name	LOC
Fetch	60
Decode + Register File	465
Execute + ALU	218
FPU	N/A
Memory + Cache	537
Write Back	20
Control Logic + Forwarding + Stalling	419
Total w/o FPU	1719

Table 5.1: Lines of Code (LOC) for each processor component.

5.2.1 Processor Design

We designed a complete MIPS processor that enforces the software label contract discussed in Section 3.6.2. Specifically, we implement the DirtyCache design (see Section 3.6.5), which marks every cache line dirty for security. Our processor is based on a classic 5-stage in-order pipeline with separate instruction and data caches, both of which are 32kB and 4-way associative. Our processor also includes a floating point unit (FPU) that we constructed using the Synopsys DesignWare library.

The Verilog code for our processor has more than 1700 LOC excluding the FPU, as shown in Table 5.1. LOC for the FPU is not reported because the source code for the DesignWare library component is not available. Table 5.2 summarizes the processor's ISA, which is rich enough that we can compile a recent OpenSSL release with an off-the-shelf gcc compiler. The ISA is at least comparable to the ISAs of prior processors with formally verified security (e.g., [LKO+14]). New instructions "setr" and "setw" are used to set security labels.

Our secure processor design supports fine-grained sharing of hardware resources between different security levels. For example, the design allows both

Instruction type	Instructions
Additive Arithmetic	add, addi, addiu, addu, sub, subu
Binary Arithmetic	and, or, xor, nor, srl, sra, sll, sllv srlv, srav, slt, sltu, slti, sltiu, andi, ori, xori
Multiply/divide	mult, multu, div, divu
Floating point	add.s, sub.s, mul.s, div.s, neg.s, abs.s mov.s, cvt.s.w, cvt.w.s, c.lt.s, c.le.s
Branch and jump	bne, beq, blez, bgtz, jr, jalr, j, jal
Memory operation	lw, lhu, lh, lbu, lb, sw, sh, sb, swc1, lwc1
Others	mfhi, mflo, lui, mtc1, mfc1 syscall, break
Security-related	setr, setw

Table 5.2: Complete ISA of our MIPS processor.

high and low cache partitions to be securely used by a single program, while a coarse-grained approach may only allow a program to use either the high partition or the low partition.

To implement such a rich policy, we divide a 4-way cache into a low partition and a high partition. When the security label of an instruction is H, both low and high partitions can be used securely. When the security label of an instruction is L, both low and high partitions are still searched. However, to ensure that timing can be affected only by the low cache partition, a cache access is treated as a miss even when there is a hit in the high partition. To avoid data duplication, the cache line moves from the high partition to the low partition when the data arrives from memory, achieving functional correctness without violating the timing constraint.

The pipeline, on the other hand, is dynamically partitioned using the security label of an instruction. When the security label changes, the pipeline is drained to avoid leaking information. A pipeline that interleaves instructions of

different security labels without flushing is indeed insecure, since the instructions with higher security labels may stall the ones with lower security labels.

I found that implementing such a complex policy securely would be difficult without using SecVerilog. For example, the SecVerilog type checker caught a security flaw not foreseen by me: the dirty bit copied from the high partition to the low partition created a potential timing channel. This was the motivation for me to develop the DirtyCache design (see Section 3.6.5).

Another security issue caught by the SecVerilog type checker is a stall at the instruction fetch stage can affect the memory stage. In our pipeline implementation, a load miss in an instruction could stall instructions in later pipeline stages because instruction cache and data cache could not be accessed simultaneously. Thus, when the security label changes, an instruction with security label H in the fetch stage can stall another instruction with label L in the memory stage, breaking the security label contract. To make the design type-check, the pipeline is drained when the new instruction has a different security label from instructions that are already in the pipeline.

5.2.2 Overhead of Timing Channel Protection

The timing channel protection mechanisms in the processor adds some overheads compared to the unmodified baseline that has no protection because of some restrictions on how the cache can be used. In this section, we evaluate the overhead of our secure processor design.

	Baseline	Secure
Delay w/ FPU (ns)	4.20	4.20
Delay w/o FPU (ns)	1.64	1.66
Area (μm^2)	399400	402079
Power (mw)	575.5	575.6

Table 5.3: Comparing processor designs.

Delay, Area and Power

We synthesized the processor designs with the Synopsys synthesis flow, using the 90nm *saed*90*nm_max* digital standard cell library. For both designs, we increased the frequency of the processors to the maximum achievable to see what overhead the secure design adds to the critical path. The synthesis results are shown in Table 5.3. When an FPU is included, we found that the critical path delays are identical for both our secure design and the baseline, as shown in Table 5.3. This is because the critical path of the processor lies in the FPU, which is largely unmodified. To more meaningfully evaluate the impact of secure design, we also measured the maximum achievable frequency without an FPU. Nevertheless, the delay overhead is still only 1.22%. The area overhead of 0.67% was also quite low, and power overhead is almost negligible. Because SecVerilog allows hardware resources to be shared across security levels while properly restricting their allocations, timing channel protection mostly does not require duplicating or adding hardware.

Performance

Our benchmarks include three security programs (blowfish, rijndael, SHA-1) from MiBench, a popular embedded benchmark suite for architectural de-

signs[1] [GRE+01], as well as ciphers and hash functions in a recent release (version 1.0.1g) of OpenSSL, a widely used open-source SSL library.

Thanks to the rich ISA of our MIPS processor, compiling and running these benchmarks require only modest effort. We use an off-the-shelf gcc compiler to cross-compile the benchmarks to the MIPS 1 platform. We use Cadence NCVerilog to simulate our processor design running these binaries. Because we lack an operating system on the processor, system calls (e.g., open, read, close, time) are emulated by Programming Language Interface (PLI) routines. Dynamic memory allocation is implemented by simple code using preallocated static memory.

Most test programs in these benchmarks were used as is. The only exceptions are a few tests in OpenSSL that take a long time to simulate. To make evaluation feasible on these tests, we replace long inputs with shorter ones.

We evaluate two security policies: *nomix*, a coarse-grained policy where the entire program is labeled H, corresponding to the security policy targeted by previous secure hardware design methods, and *mixed*, a fine-grained policy allowing mixed H and L instructions, enabled by the new features of SecVerilog. In the latter case, we use a simple policy to decide timing labels: for ciphers (e.g., AES, RSA), the encryption and decryption functions are marked as H; for secure hash functions (e.g., MD4, SHA512), we pretend part of the input is secret, and mark the hash functions on these inputs as H. Due to the coarse-grained labeling, *nomix* policy can only use the H partition of a cache while *mixed* policy can use both the L and H partitions. Performance results are shown in Figure 5.5.

From the MiBench suite, only rijndael shows a noticeable performance overhead of 19.6%. Overhead is reduced to 12.2% when the fine-grained model with

[1]The only benchmark omitted is PGP, which requires a full-featured OS.

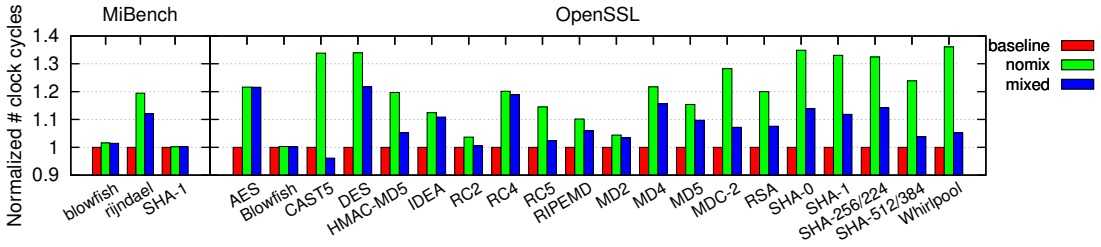


Figure 5.5: Performance overhead of timing channel protection.

mixed labels is used. Overhead on OpenSSL ranges from 0.3% (Blowfish) to 34.9% (SHA-0) for the coarse-grained model, with an average of 21.0%. For the fine-grained model, the overhead on OpenSSL ranges from −3.9% (CAST5) to 21.7% (DES), with an average of 8.8%. CAST5 runs faster with the partitioned cache compared to the baseline because *H* instructions cannot evict frequently used data in the *L* partition.

The results clearly show the benefit of fine-grained information flow control within a single application. Most slowdown comes from the restriction that *H* instructions cannot write to the *L* cache partition. Since *H* instructions are only a subset of program instructions, allowing mixed *H* and *L* instructions in a single program still improves the overall performance.

5.3 A Multi-Core PARC-tiny Processor

We implement a secure multi-core processor that allows mutually distrusting security domains to run concurrently and share the hardware resources. We use SecVerilog to verify the security of our processor, and the results suggest that our timing channel protection mechanisms indeed remove timing channels.

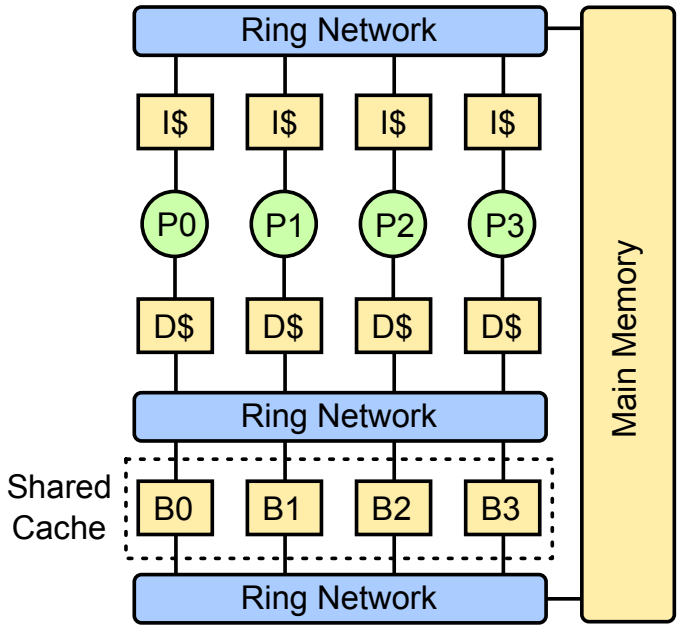


Figure 5.6: Architecture of the multi-core processor.

5.3.1 Baseline Architecture

Figure 5.6 shows the architecture of the multi-core processor. The processor has four cores ($P0$ - $P3$), and each core has a private $L1$ instruction cache and data cache. The private caches are 2-way set associative. The four cores also share a $L2$ cache, which is divided into four cache banks ($B0$ - $B3$). Ring networks are used to connect different levels of caches as well as the main memory. Each ring network has four routers, which use Elastic Buffer (EB) [MBD09] flow control algorithm to forward packets. The routing algorithm uses deterministic greedy routing. Memory controllers are not implemented in this processor.

The processor uses an ISA called PARC-tiny, which is a subset of the PARCv2 instruction set developed by Batten et al. [PAR]. The list of instructions that constitute PARC-tiny are listed in Table 5.4.

Instruction type	Instructions
Additive Arithmetic	addiu, addu, subu
Binary Arithmetic	and, or, slt, ori, sra, sll
Multiply/divide	mul
Branch and jump	bne, beq, jr, j, jal
Memory operation	lw, sw
Others	mfc0, mtc0, lui
Security-related	setr, setw

Table 5.4: Complete ISA of the PARC-tiny processor.

5.3.2 Protection Mechanisms

We assume two mutually distrusting security domains are sharing this multi-core processor, hence the security goal is to prevent any information flow between these two domains. Security Domain 0 ($SD0$) run on core 0 and core 1, while Security Domain 1 ($SD1$) run on core 2 and core 3. We assume the two security domains can share read-only data (e.g., program code) but not data that can be written. In this baseline architecture, the two security domains can launch timing channel attacks by exploiting the interference in the ring networks or the shared caches. Indeed, when we checked the RTL design of this baseline architecture using SecVerilog, many illegal information flows were identified by SecVerilog. To remove these illegal information flows, we implemented the protection mechanisms described below.

Shared Cache

The shared cache is vulnerable to timing channel attacks due to the interference between security domains. We adopt the partitioning scheme to remove cache timing channels. Since the shared cache is already banked, we partition the

cache at the granularity of cache banks. Specifically, $SD0$ only uses bank $B0$ and $B1$ while $SD1$ only uses bank $B2$ and $B3$. We encode the security domain ID into the destination field of network packets to implement this partitioning scheme. Since we allow shared read-only data between the two security domains, we duplicate these data if both domains fetch the shared data into the cache. Hence, there could be two copies of the same data in different cache banks. However, there is no coherence issue since these shared data are read-only.

Ring Network

We made two major modifications to the ring network. First, we use separate input buffers for each security domain. This avoids the head-of-line blocking interference between security domains. Second, we use temporal partitioning to divide the bandwidth among security domains at a per-cycle basis. Specifically, $SD0$ can use the routers and links only at even cycles while $SD1$ can only use resources at odd cycles. The two modifications completely remove the network interference between security domains.

Cache Coherence

We also implement a simple cache coherence protocol in this multi-core processor. The protocol, MI, is a directory-based protocol which only has two stable states. M stands for "Modified", which indicates one of the private caches owns the data. I stands for "Invalid", which indicates none of the private caches owns the data. MI protocol is restrictive in that it does not allow a clean data to exist in

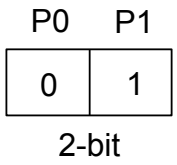


Figure 5.7: A directory entry in bank 0.

multiple private caches. We use this protocol because it is simple to implement and demonstrates the effectiveness of our protection mechanism.

Each bank of the shared cache has a directory that tracks the coherence state of every cache entry in the bank. An entry of the directory has two bits, one for each core that share this cache bank. As an example, Figure 5.7 shows a directory entry in $B0$. Since $B0$ is shared by $P0$ and $P1$, each bit in the entry represents the presence of the corresponding cache entry in the private cache of $P0$ or $P1$. In this example, the cache entry exists in $P1$'s private cache. Note that at most one of the two bits can be 1 because of MI protocol.

When a cache request arrives at the $L2$ cache bank, the cache controller first checks the directory entry that corresponds to the address of the request. If the data exists in another private cache, the cache controller sends an eviction request to that private cache to take ownership of the data. After it receives the response from the private cache, it sends the data to the cache that initiates the original request and updates the directory entry. When an $L2$ eviction happens, the $L2$ cache controller also sends an eviction request to the private cache that owns the data so that the most updated data is written back to memory and the cache bank remains inclusive.

Separate ring networks are used to send coherence requests and responses. Similar to the ring network protection, we use temporal partitioning to elimi-

Module Name	LOC
Pipeline	1472
Multiplier	470
L1 Cache	884
L2 Cache	929
Router	589
Ring Network	210
Memory to Network Adapter	128

Table 5.5: Lines of Code (LOC) for each processor component.

nate interference between security domains in the coherence networks. Since private caches and L2 cache banks from different security domains do not communicate with each other in our protocol, there is no interference between security domains at any cache controller.

5.3.3 SecVerilog Verification

The aforementioned protection mechanisms were implemented in this multi-core processor. The components and LOC for each component are shown in Table 5.5. To use SecVerilog to check the information flows of this multi-core processor, we need to mark every signal with a security label. However, we ran into an issue due to the lack of bit-level label support in current version of SecVerilog. In SecVerilog, all the bits in a multi-bit signal share the same security label. Similarly, all the elements in an array also share a single security label. This limitation makes it impossible for a signal to combine multiple signals from different security domains and later decouple into separate signals. As a result, separate signals need to be declared for each security domain, which significantly increases the programming effort.

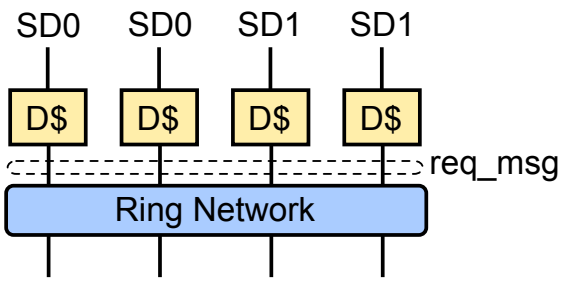


Figure 5.8: Illustrative example for bit-level label support.

As an illustrative example, consider the interface between caches and ring networks in Figure 5.8. In a normal design without SecVerilog, the four cache signals to the ring network can be combined into a single signal, *req_msg*, before entering the ring network module. As a benefit, the ring network module only needs to declare a single input signal. Inside the ring network module, each cache signal can be derived by picking different ranges of the input signal. It is also possible to write *for* loops to instantiate modules in a batch, as shown in the code example below:

```
1 genvar i;
2 generate
3 for (i = 0; i < 4 ; i = i + 1) begin:
4     new_module(
5         .input  (req_msg[i]),
6         .output (out_msg[i])
7     );
8 end
9 endgenerate
```

However, if SecVerilog is used, the four cache signals cannot be combined because there is no way to specify the security label of *req_msg* without bit-level

Module Name	LOC
Ring Network	120/210
MemNet	267/406
ProcCacheNet	783/1181

Table 5.6: Extra lines due to lack of bit-level label support.

label support. Hence, four input signals are required for the ring network, and *for* loops using the *generate* block is no longer possible. The lack of bit-level label support increases the LOC of some modules significantly, as we show in Table 5.6. The *MemNet* module instantiates a ring network module and two memory to network adapter modules. The *ProcCacheNet* module is the top module that instantiates the entire multi-core processor. The first number is the extra lines due to the lack of bit-level label support, while the second number is the total number of lines in the module. As the results suggest, the extra programming effort results from this limitation is significant, and support for bit-level label is a needed feature in future versions of SecVerilog.

With the protection mechanisms implemented, the multi-core processor passes SecVerilog verification, which proves the effectiveness of these protection schemes against timing channels. Currently, the multi-core processor does not functionally work because I haven not finished the functional verification of the cache coherence protocol due to its complexity. However, the interference between security domains resulting from the *MI* protocol all happens in the ring network, which SecVerilog did verify. The functional verification of the cache coherence protocol remains to be the future work.

CHAPTER 6

RELATED WORK

This chapter describes previous work that is related to this thesis research. Section 6.1 discusses previous work on possible timing channel attacks in both software and hardware. Section 6.2 summarizes related work on timing channel protection mechanisms. Finally, Section 6.3 describes previous work on verifiable hardware information flow control.

6.1 Timing Channel Attacks

6.1.1 Software Timing Channels

Software timing side channel attacks usually exploits security vulnerabilities in the software to extract secret information through timing. Kocher [Koc96a] designed a timing attack to extract secret keys of Diffie-Hellman [DH06] and RSA [RSA78] cryptographic algorithms. The attack exploits the fact that the execution time of modular reduction steps depends directly on the secret key. The attack was believed to be defeated if Montgomery multiplication [Mon85] and Chinese Remainder Theorem are used to implement Diffie-Hellman or RSA. However, later work by Brumley et al. [BB03] proves that this enhanced implementation is still vulnerable to timing channel attacks by demonstrating a successful timing attack against OpenSSL, which uses RSA.

Several projects have looked into timing channel attacks in web applications. Felten et al. [FS00] shows the timing variations of loading a webpage due to web caching can reveal a user's visit history, hence compromising users' privacy. A

similar timing attack was performed by exploiting CSS filters [KPJJ13]. Bortz et al. [BB07] shows the response times of a website can reveal a user's login status or the size of a online shopping cart. Song et al. [SWT01] demonstrates that a statistical analysis of keystroke timing derived from SSH can help predict users' passwords.

Software covert channel attacks have also been studied, most in the context of TCP/IP protocol. Griffin et al. [GGLT03] proposes a covert channel attack by delaying packets and thus encoding information in the packet timestamp. Cabuk et al. [CBS04] develops another covert channel attack by synchronizing adversaries and encoding bits into the number of packets sent in each time interval. Later, Sellke et al. [SWBS09] proposes another attack that uses packet inter-transmission times to convey information, which achieves 2 to 5 times data rate compared to Cabuk's attack.

6.1.2 Hardware Timing Channels

Hardware timing channel attacks usually exploit the timing variations in hardware resources as well as the interference between security domains that share a hardware resource. For example, cache hits take much less time than cache misses, and a program can evict the cache lines of another program in a shared cache. We discuss related work on timing channel attacks on each hardware component.

Caches

Timing channel attacks on caches have been studied extensively in the literature. In prime-probe attacks [Per05, OST06, LYG+15], the attacker loads the cache with its cache lines and measure which cache lines have been evicted by the victim process that shares the cache. The attacker is able to extract secret key from the victim's access pattern. A similar attack that exploits interference between security domains is evict-time attack [OST06]. The attacker evicts the victim's cache lines, and then triggers the victim's cryptographic operations. The attacker measure the execution time of the victim and can figure out whether the evicted cache lines have been accessed by the victim. Interference within a program can also leads to timing channels. A program's cache lines can evict each other, hence resulting in different execution times given different inputs. Bernstein [Ber05] presents such an attack in experiments, which was further improved and realized in real-world settings by Onur et al. [AScKK07]. All these attacks are based on cache contention.

Another type of attacks relies on reuse of cache lines. Basically, if a data block has been accessed before, it will be cached and the following accesses to the same data block hit in the cache. One such attack is called flush-reload attack [GBK11, YF14]. In this attack, an attacker and a victim may share some data such as a common library. The attacker first flushes the shared data out of a shared cache, then waits for the victim to run. After that, the attacker reloads the shared data and can figure out which part of the shared data has been accessed by the victim since accesses to these data result in cache hits. Another example of reuse-based attacks is the cache collision attack [BM06], in which the victim

reuses some of its cache lines based on the input, hence resulting in different execution times given different inputs.

Besides attacks on data caches, previous work also demonstrates possible attacks on instruction caches. Aciiçmez [Aci07] proposes a proof-of-concept timing attack on an instruction cache that allows the attacker to recover the execution flow of a victim program. This attack was further refined in [ABG10] and carried out in real-world settings.

Previous work also demonstrates the feasibility of cache timing channel attacks in cloud computing environment. Ristenpart et al. [RTSS09] proposes an attack that can reveal the timing of keystrokes typed in a console from a virtual machine in Amazon EC2 servers. Later, Zhang et al. [ZJRR12] develops a more fine-grained cross-VM attack that allows an attacker to extract cryptographic keys from a co-resident VM.

On-Chip Networks

Our work [WS12] identifies possible timing channels in on-chip networks. We demonstrate a simple RSA attack example that exploits interference between security domains in the shared on-chip network (see Section 2.1.2).

Memory

Several projects have looked into timing channels in main memory. Wu et al. [WXW12] discover the memory bus locking due to atomic instructions on x86 platforms can be exploited to create covert channels across the system. Our

work [WFS14] identifies possible timing channels in a shared memory controller. We demonstrate a side-channel attack example on RSA that exploits the interference in memory scheduling among security domains. By manipulating the number of memory accesses, a covert channel attack is also possible to communicate unauthorized information between colluding adversaries. Timing channels have also been explored in the context of ORAM [GO96]. Fletchery et al. [FRY+14] points out that when memory accesses are issued may also leak information in ORAM.

Another line of work on memory timing channel attacks exploits memory deduplication, which is a technique to reduce memory footprint by combining identical memory pages. Write access to these deduplicated pages leads to a page fault, which takes much longer to process than write access to normal pages. Suzaki et al. [SIYA11] proposed an attack that can determine whether specific applications are running in a co-located virtual machine in the cloud. Owen et al. [OW11] demonstrated that it is possible to efficiently fingerprint operating systems of co-resident virtual machines using memory deduplication attacks. Attackers can use OS fingerprinting to learn the type and version of an operating system so that they can launch attacks that are specific to the target OS. Gruss et al. [GBM15] presented a page deduplication attack in sandboxed JavaScript, which allows a remote attacker to collect private information such as whether a program or website is currently opened by a user.

Others

Timing channel attacks have been studied in other microarchitectural components in modern processors. Wang et al. [WL06] discovers that processor ar-

chitecture features such as simultaneous multithreading (SMT) can be used to create covert channels. For example, a thread can manipulate the use of function units to delay another thread and encode the information using the delay. Onur et al. [AKS06, AKS07] proposes timing channel attacks that can extract the secret key of a RSA process by analyzing the CPU's branch predictor states.

6.2 Timing Channel Protection

6.2.1 Software Approaches

Many software solutions rely on rewriting software to defend against known timing channel attacks. For example, to defend against timing channel attacks against AES, previous work [OST06, BGNS06] proposes using mathematical operations to replace AES table lookups. One problem with this approach is that the solution is ad hoc, i.e., the software is rewritten to defend against a specific attack. Hence, new attacks may be possible after the software is rewritten. For instance, RSA implementation using Montgomery multiplication and Chinese Remainder Theorem was believed to be secure against timing channel attacks, but was proven wrong by later work [BB03]. Furthermore, the performance of the new software implementation can be very high (2X to 4X slower than the original implementation [BGNS06]).

Another software approach to timing channel mitigation focuses on manipulating the time that an attacker can observe. One such scheme [GH06] is to add random delays to the execution time of various operations, which reduces the bandwidth of timing channel attacks. However, random noise does not

eliminate timing channels, since it can be removed by large number of samples. Askarov [AZM10] proposes to use predictive mitigation mechanisms to bound the information leakage of timing channels. The response time observed by the attacker increases exponentially if it fail to meet an expected response time. Later, Zhang [ZAM11] generalizes predictive mitigation to take advantage of knowledge about the system being protected, significantly improving the tradeoff between security and performance.

Several projects have looked into applying language-based techniques to timing channel protection. Agat [Aga00] proposes a type system that detects timing channels and uses program transformations to remove timing variations. However, this work sacrifices language expressiveness and does not consider timing channels that is caused by hardware features such as a cache. Zhang et al. [ZAM12] proposes a language-based technique to control and mitigate timing channels. In this approach, each instruction of the software is marked with a timing label and a write access label. This approach improves the language expressiveness and deals with the underlying hardware features through the labels. As another defense mechanism, Coppens et al. [CVBS09] proposes to use program transformations in a compiler backend to remove key-dependent control flows. They demonstrate that their approach is effective in defending against timing side-channel attacks on modern x86 processors.

Other software protection schemes involve modifications to an OS or a hypervisor to prevent some known attacks. For example, Zhang et al. [SXZ13] proposes a hypervisor-based solution that restricts the use of atomic instructions and defeats the covert channel attacks through memory bus locking [WXW12].

6.2.2 Hardware Approaches

Hardware solutions remove timing channels by making microarchitecture changes that eliminate interference between security domains or obfuscate information being observed by an attacker. We discuss related work on hardware protection schemes for each hardware component.

Caches

Protections for caches can be generally categorized into two approaches, partitioning approach and randomization approach. Page [Pag05] proposes to use static cache partitioning to eliminate cache interference between different security domains. However, static cache partitioning often incurs high performance overhead. Our work, SecDCP [WFZ+16], utilizes the asymmetric property of a security policy to enable dynamic cache partitioning while still maintaining the security guarantee. Another work that follows the partitioning approach is Partition-Locked cache (PLcache) [WL07], which locks the security-critical data in the cache and disallows other data to evict them. However, as pointed out by later work [KASZ08, KASZ09], PLcache still has security vulnerabilities unless the security-critical data is preloaded into the cache without being noticed by the attacker.

Wang et al. [WL07, WL08] proposes Random Permutation cache (RPcache) that tries to eliminate interference through randomization. The basic idea is to randomize the mapping between addresses and cache lines so that an attacker does not know which address of the victim program causes the eviction of the attacker's cache lines. Unfortunately, it does not defend against a covert

channel attack, which encodes information using the number of cache misses. RPcache also cannot defend against cache collision attacks [KASZ08, KASZ09], which are based on the reuse of cache lines. To defend against cache collision attacks, Liu et al. [LL14] proposes random fill cache architecture. Random fill cache removes the demand fetch feature of conventional cache and replace it with randomized fetch. Random fill cache requires security-critical data to be in continuous memory regions for security. Random fill cache can also incur significant performance overhead because many unuseful cache lines may be fetched into the cache.

On-Chip Networks

Our work [WS12] proposes to use Temporal Network Partitioning (see Section 2.2) to eliminate the interference between different security domains. A priority-based protection, RPSL (see Section 2.3), was also developed to improve the performance by providing uni-directional protection. Later, SurfNoC [WGO+13] was proposed to improve the performance of TNP by more efficient network scheduling that avoids unnecessary waiting delays in each router.

Memory Controllers

Timing channel protections for memory controllers are more difficult than on-chip networks because of the complexity of memory scheduling. We propose Temporal Partitioning (TP, see Section 4.2) to remove timing channels in a shared memory controller. TP introduces the "dead time" to completely remove

the interference between security domains, but dead time also introduces significant performance overhead. Shafiee et al. [SGS+15] proposes to improve the performance of TP by using spatial partitioning such as rank partitioning. Mapping security domains to different ranks significantly reduces the dead time, hence improving the system performance. However, spatial partitioning limits the number of security domains that can be supported simultaneously and leads to high memory fragmentation. To improve the performance of TP without spatial partitioning, we propose SecMC-NI (see Section 4.3), which interleaves requests from different banks and ranks to construct an efficient schedule. We then propose SecMC-Bound (see Section 4.4) that further improves performance by trading off security. Previous work also looked into performance improvement by providing uni-directional protection. Ferraiuolo et al. [FWZ+16] propose Lattice Priority Scheduling (LPS) scheme that enables dynamic scheduling of memory requests without breaking the security guarantees.

Fuzzy Time

Previous work has also looked into timing channel protection by obfuscating an attacker's notion of time. Hu [Hu92] proposes to reduce the bandwidths of covert timing channels by making all clocks available to a process noisy, although the noise may be filtered by large number of samples. Martin et al. [MDS12] propose to add random offset and delay to the RDTSC instruction in x86, which returns the current value of the timestamp counter (TSC) register. Their method reduces the resolution of an attacker's timing measurement so that the attacker cannot distinguish microarchitectural events (e.g., cache hits

vs cache misses). However, this approach can still be vulnerable to covert channel attacks, which uses the number of cache misses to encode information.

6.3 Verifiable Hardware Information Flow Control

Several projects have looked into verifying the information flow in hardware. GLIFT [TWM+09] tracks the information flow at the logic-gate level by inserting extra tracking logic into the hardware. However, GLIFT only detects information flow violations at run time. Hence, the hardware is already fabricated when a violation is detected, resulting in high cost. Because of the extra tracking logic, it also incurs high overheads in area, power and performance. Execution Lease [TLW+09] enforces information flow policies by restricting the time and address space for a region of execution. The design is implemented with GLIFT, but turns out to be very restrictive since it does not naturally support hardware features that have timing variabilities (e.g., caches, pipelining). Tiwari et al. [TOL+11] improves Execution Lease by designing an architectural skeleton that can govern the information flow of the entire system. In their design, information flow is statically checked by enumerating all possible states of their $star - logic$ through gate-level simulations, an approach that is unlikely to scale to large designs.

Previous work has also looked into language techniques to verify hardware information flow. Caisson [LTO+11] is a Hardware Description Language (HDL) that can statically verify the information flow of hardware. However, since Caisson uses a purely static type system, all registers must be duplicated for different security domains, leading to high area and performance overhead. It

also prevents fine-grained sharing of wires even if the sharing is secure. Sapper [LKO+14] is another language that automatically generates and inserts dynamic security checks into the Verilog hardware design. These checks ensure that security violations will be revealed as functional bugs during testing so that designers can fix any illegal information flow. However, security vulnerabilities that are not caught during the testing phase can cause functional problem in hardware deployment. SecVerilog [ZWSM15] verifies the hardware at design time by marking every signal with security labels. The dependent types introduced in SecVerilog allow more flexible resource sharing between different security domains. However, to handle the implicit declassification feature of SecVerilog, hardware designers need to specify the value of a register explicitly when the label of the register changes from a higher security label to a lower one.

CHAPTER 7

CONCLUSION

7.1 Summary

This thesis explored and addressed some of the timing channel attacks in modern multi-core processors. Possible timing channel attacks were identified in shared hardware resources including on-chip networks and memory controllers. Secure hardware designs were proposed to defeat some of the known timing channel attacks. Finally, we verified that the timing channels have been indeed eliminated in RTL by our protection schemes using SecVerilog.

In Chapter 2, we described our discovery of timing channel attacks on the shared on-chip networks. Temporal Network Partitioning (TNP) was proposed to defeat the attacks. Furthermore, a priority-based protection scheme called RPSL was developed to improve the performance by providing uni-directional protection. In Chapter 3, we summarized previous timing channel attacks on caches and also pointed out the number of cache accesses can be used as a covert channel. We proposed SecDCP, a scheme that allows secure dynamic cache partitioning to defend against cache timing channel attacks, while significantly improving the performance of static partitioning. To support more security domains, another protection scheme based on a high-associativity cache (ZCache) was proposed. One observation was that even cache line relocations between security domains can lead to new timing channel attacks. We also came up with a couple of secure cache designs that support the language-based protection. One of the designs (DirtyCache) was implemented in RTL and verified by SecVerilog in Chapter 5. In Chapter 4, we described our findings on possible

timing channel attacks on a shared memory controller. Temporal Partitioning (TP) was proposed as an initial protection scheme that eliminates interference between different security domains by static scheduling. To reduce the performance overhead, we proposed another secure memory controller, SecMC-NI, which constructs an efficient schedule of memory requests by interleaving requests from different memory banks and ranks. To further improve the performance, we designed SecMC-Bound, which allows users to trade off security for higher performance. Security analysis on SecMC-Bound provides a bound on information leakage rate. Finally in Chapter 5, we built two secure processors in RTL and used SecVerilog to verify that both of them are free of timing channels.

7.2 Future Directions

7.2.1 Efficient Timing Channel Protections

Although many timing channel protection schemes have been proposed in recent years, these protections usually come with either high performance cost or inflexibility. For example, SecMC-NI only achieves 57% performance of an insecure baseline. Rank partitioning can reach 80% of the baseline's performance, but puts restrictions on the use of DRAM memory. To summarize, there is still a significant performance gap between timing channel protection schemes and insecure designs, and this gap will only get larger as we scale the number of security domains in future computing systems.

Hence, one interesting future direction is to close this performance gap with more efficient timing channel protections. To this end, we already explored

some techniques that improve performance by providing uni-directional protections. However, there is still more work to be done. For example, we proposed a ZCache-based protection scheme (see Section 3.5) to support more security domains, but only allowing cache line relocations within the same security domain is very restrictive and hurts performance.

One approach to improve the efficiency of timing channel protections is trading off security for performance, as we did in SecMC-Bound (see Section 4.4). One challenge in this approach is deriving a tight information leakage bound that is useful in practice. Theoretical information leakage analysis in SecMC-Bound is still quite conservative and derives an leakage rate of several hundred bits per second, which is much higher than the threshold suggested by the Trusted Computer System Evaluation Criteria (the orange book [oD85]). Once a tight information leakage bound is found, another interesting research direction would be calculating the leakage bound at run time efficiently in hardware and enforcing the leakage to be lower than some user-specified limit.

7.2.2 QoS Support with Timing Channel Protections

Many QoS techniques [DK14, DK13] have been proposed to provide certain performance guarantees (e.g., target IPC) for concurrently running programs that share hardware resources such as caches and memory. However, most QoS techniques do not consider the threat of timing channels. As a result, current QoS techniques are vulnerable to timing channel attacks since the resource allocation decisions can reveal confidential information about a program. One interesting research direction is then to design schemes that provides both QoS guarantees

and timing channel protection. For example, in SecDCP (see Section 3.4), the cache can reserve a minimum number of cache ways for each security domain to fulfill their QoS requirements while dynamically and securely allocating the rest to achieve higher performance. A challenge in this research direction is how to meet the QoS requirements without breaking the security guarantee. Since subtle interference may lead to timing channel attacks, care must be taken when deciding resource allocation to meet QoS requirements.

7.2.3 Verifiable Information Flow Control in Hardware

Although several tools have been developed to verify the information flow control in hardware, these tools still have their limitations. Thanks to the dependent type system, SecVerilog allows flexible and verifiable hardware designs. However, the implicit declassification in SecVerilog requires the designer to explicitly specify the value of a register when its label changes from a higher security label to a lower one. Moreover, current SecVerilog does not support fine-grained labels, which means a vector of bits or an array of signals only has a single label. To assign different labels to different elements in an array, the array must be decoupled into multiple signals, which is inconvenient for hardware designers. Type inference is currently unsupported in SecVerilog as well. To summarize, a few desired features are still to be added into current tools of hardware information flow verification to make them more flexible and usable.

BIBLIOGRAPHY

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, 2010.

[Aci07] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, 2007.

[Aga00] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[AKS06] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, 2007.

[ALOJ13] Jung Ho Ahn, Sheng Li, Seongil O, and Norman P. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013.

[AScKK07] Onur Aciiçmez, Werner Schindler, and Çetin K. Koç. Cache based remote timing attack on the AES. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2007.

[AZM10] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium*, 2003.

[BB07] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, 2007.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[BGNS06] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, 2006.

[CBS04] Serdar Cabuk, Carla E. Brodley, and Clay Shields. Ip covert timing channels: Design and detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[CVBS09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[DH06] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 2006.

[DK13] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[DK14] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[DT04] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. 2004.

[EG85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, 1985.

[FRY+14] Christopher W. Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[FS00] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000.

[FWZ+16] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[GBM15] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *20th European Symposium on Research in Computer Security*, 2015.

[GGLT03] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert messaging through tcp timestamps. In *Proceedings of the 2Nd International Conference on Privacy Enhancing Technologies*, 2003.

[GH06] James Giles and Bruce Hajek. An information-theoretic and game-theoretic study of timing channels. *IEEE Transations on Information Theory*, September 2006.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of ACM*, May 1996.

[GRE+01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Workshop on Workload Characterization*, 2001.

[HKR+15] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[Hu92] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, May 1992.

[ive] Icarus Verilog. http://iverilog.icarus.com/.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. 2007.

[KASZ08] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2Nd ACM Workshop on Computer Security Architectures*, 2008.

[KASZ09] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[Koc96a] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, 1996.

[Koc96b] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology CRYPTO '96*, 1996.

[KPJJ13] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using css filters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, 2013.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[LKO+14] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[LL14] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[LSC+10] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Pengju Ren, Omer Khan, and Srinivas Devadas. DARSIM: A Parallel Cycle-level NoC Simulator. In *MoBS 2010 - Sixth Annual Workshop on Modeling, Benchmarking and Simulation*, 2010.

[LTO+11] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[MBD09] George Michelogiannakis, James Balfour, and William J. Dally. Elastic-buffer flow control for on-chip networks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[MDS12] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985.

[oD85] Department of Defense. Tcsec: Trusted computer system evaluation criteria. Technical report, 1985.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[OW11] Rodney Owens and Weichao Wang. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, 2011.

[Pag05] Daniel Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Eprint archive*, 2005.

[PAR] PARC Instruction Set Architecture. http://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-parc-isa.txt.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[QP06] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[RCBJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[RSA78] Ron L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[SDR02] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[SGS+15] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[SIYA11] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, 2011.

[SK10] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[SK11a] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[SK11b] S. Subashini and V. Kavitha. Review: A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 2011.

[SWBS09] Sarah H. Sellke, Chih-Chun Wang, Saurabh Bagchi, and Ness Shroff. Tcp/ip timing channels: Theory to implementation. In *Proceedings of the 28th IEEE International Conference on Computer Communications*, 2009.

[SWT01] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium*, 2001.

[SXZ13] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. BusMonitor: A hypervisor-based solution for memory bus covert channels. In *Proceedings of 6th European Workshop on Systems Security (EuroSec)*, 2013.

[TLW+09] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[TOL+11] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[TWM+09] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[WFS14] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[WFZ+16] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[WGO+13] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[WL06] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceeding of the 22 nd Annual Computer Security Applications Conference (ACSAC*, 2006.

[WL07] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[WL08] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[WS12] Yao Wang and G. Edward Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, 2012.

[WWS17] Yao Wang, Benjamin Wu, and G. Edward Suh. Secure dynamic memory scheduling against timing channel attacks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[XL09] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[ZAM11] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[ZAM12] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[ZF05] Yongbin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[ZR97] William K. Zuravleff and Timothy Robinson. Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order, May 1997. US Patent 5,630,096.

[ZWSM15] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.