Obfuscating Malware through Cache Memory Architecture Features

Çağlar SAYIN

Masteroppgave
Master i Teknologi - Medieteknikk
30 ECTS
Avdeling for informatikk og medieteknikk
Høgskolen i Gjøvik, 2014

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Obfuscating Malware through Cache Memory Architecture Features

Çağlar SAYIN

2014/06/01

Abstract

There is no doubt that malicious software (malware) is one of the most important threat in computer security. With increasing of the information systems and computer network usage in the industrial and governmental infrastructures, their economy and impact over our society are increasing. According to Symantec's report in 2008[1], "The release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications." The worst of all, malware design is not as simple as how it was before. A few years ago, we saw countries who developed malware as a professorial weapon for their political benefits, and it would not be surprising if one of these weapons were seen in the corporate world soon. This malware was utilized with many camouflaging techniques (e.g. polymorphism, metamorphism, etc.) against the malware detection system.

Basically, the most of the camouflaging techniques obfuscate and hide the signatures to be stored safely in a non-volatile memory or disk, and before they started to run on the main memory, they deobfuscate the whole code to execute. Consequently, the detection systems have simply started to search the signatures in the main memory. In this thesis, we designed a way to raise the bar from "from disk to memory obfuscation" to "from disk to cache obfuscation". More specifically, we designed theoretical malware obfuscation methods for tightly coupled multi-processor systems which utilize caches as a private memory to evade main memory observer systems as well as other conventional static data analysis. In order to achieve this goal, we anticipated cache behaviours and exploited them as well as cache efficiency optimizations. With increasing deployment of multi-processor computing and other parallel processing devices, the implementation of local memories like NUMA and hierarchical caches are increasing in order to increase efficiency and performance and decrease power consumption, and this can be even the only reason which highlight our studies. Additionally, this thesis discusses implementation issues arising from interactions between cache coherence mechanisms as well as from Harvard architecture implementations

Keywords

Security, Malware Design, Cache Oriented Polymorphism, Cache Coherency, Malware Evasion, Code Obfuscation

Preface

I would like to express my gratitude to my supervisor Prof. Stephen D. Wolthusen for the useful comments, remarks and engagement through the learning process of this master thesis. I can simply say that every sentence which he built was enlightening not only during this this process, but also in his courses. Furthermore, I would like to thank Emre Tınaztepe for sharing his fabulous practical knowledge with me to the topic as well for the support on the way. Also, I like to thank the team in Stanford University who develops BookSim Simulation tool and willingly share experience. I would like to thank my family. Words cannot express how grateful I am to my mother for all of the sacrifices that you've made on my behalf. I would also like to thank all of my friends who supported me in writing, and incited me to strive towards my goal.

Finally, I would like to thank to all free software volunteers.

Çağlar Sayın 2014/06/01

Ethical and legal considerations

The content of this document could be used for malicious purpose, but any matter or information could be misused in the life. The risky threat is not the information in this thesis, but to be ignorant about them. For this purpose, this thesis aims to enlighten security specialist and system developers against the recent methods of the possible attacks.

However, in order to act ethical responsibility, we tried to eliminate practice of tools and piece of codes which could leads malicious usage. In any case, there is no doubt that it is critical to discover and publish vulnerabilities which could cause deep impact before malicious people discover and abuse them.

> "Virus don't harm, but ignorance does."
> - VxHeaven

Contents

List of Figures

List of Tables

1 Introduction

The purpose of this chapter is to give introduction of the subject and challenge in question, as well as justification and motivation of its importance. The chapter also propose research question to guide the thesis.The purpose of this chapter is to give introduction of the subject and challenge

1.1 Topic covered by the project

This thesis is mainly about a novel approach to specially malware obfuscation methods although software obfuscation concept is the more broad topic. Malware is the short malicious software and can cover any program or script that is harmful to a system or its user. One of the biggest discussion is a definition of harmful on computer security. It is quite hard to define what harmful is and what malicious is. Because of the reason why we cannot classify software easily, whether malicious or not, the most efficient and suitable way to detect them is a black or white listing. Basically, there are predefined list of software which claims they are good or evil. This black lists are comprised with signatures of the software which are presumed malicious; then, we are looking for these signatures in our computers to detect malware.

Even if these lists include every possible signature, malware can evade signature detection with obfuscating itself over and over again. For each attempt, it reproduces itself, and thus, it conceals the signature. However, there are many countermeasures against them[6][7]. For example, memory dumping and scanning later is recently one of the trendy detection methods[1]. It is quite reasonable because while the malware is running over memory, it is naked and vulnerable.

In this thesis, we are seeking possibilities to enhance "Disk-memory level" to cache level. It is a pretty novel contribution to the computer security field because there have been a few works, so far, about what we are searching for.

1.2 Problem description

The conventional PC architecture is now which security researchers, developers and malware authors familiar with. However, the density of development spreads around other architecture instead of common PC arch.. For example, mobile devices recently have much higher profile at this point comparing with our conventional desktop systems.Besides, hardware platforms especially for mobile devices are increasing in complexity and sophistication with the use of multi-processor systems becoming common place as well as the use of many layer memory architectures, such as NUMA, caches for performance and efficiency(esp. on power consumption).This opens new opportunities for obfuscation and concealment of malware.

Apart from the basic signature-based methods known to be ineffective against unknown threats, there are two mainstream techniques to detect malicious code which are called static and dynamic analysis, but indeed, the main stream method is still signature-

[1] Especially, after boot level rootkits

based ones. Static analysis identifies malware mainly with code flow graph and data flow graph obtained from static information, while dynamic analysis is taking account dynamic execution features.

There are number of techniques for obfuscating, some relying on the precise properties of the hardware and its behaviours, that can render static analysis ineffective as well as signature analysis. Some of the key techniques deployed by malware authors for obfuscation do include race condition and exploitation of memory uses. The detection algorithms and techniques have been adequately worked so far because of the simplicity of our conventional architectures and using generic PC architecture despite these systems also started to become more heterogeneous(e.g. Intel Haswell). However, with increasing deployment of multi-processor computing and other parallel processing devices, the implementation of local memories like NUMA and caches are increasing in order to increase efficiently and performance and decrease power consumption. It is obviously clear that these features will also be exploited by malware authors at some point. The project seeks to investigate the feasibility of such malware both from a theoretical viewpoint and through the development of a proof-of-concept.

1.3 Justification, motivation and benefits

If malware designing is superficially considered, you could fall in the usual fallacy that it is not beneficial, and maybe it is malicious. However, if we can design it, there is always a more skillful malicious author who might already abuse this vulnerability on the black side of the moon. The duty we are actually obligated to discover these vulnerabilities and design countermeasure against them. In this way, our blessed motivation is finding possible vulnerabilities, and mitigate or eliminate their risk. Otherwise; if we are lucky, we might detect these zero time vulnerability attacks, yet it could be too late to fix and analyze them. Besides, for example, some of the most sensational and beneficial papers([6],[8],[7]) are criticizing malware with designing them as like as we do, and their values over computer security are undoubted today.

In short, we are building brakes. Sometimes they are that thing slow us down, or sometimes they even stop us in our tracks. However, they are actually there to enable us to go faster and secure.

1.3.1 Research questions

1. How can an obfuscation method, which exploits caches to conceal information from the observers, be designed for tightly coupled and multiprocessor systems?
2. How can the efficiency optimisations of common cache coherency protocols found in tightly coupled multiprocessor systems be exploited for probabilistically hiding and obfuscating malware?
3. How can we execute deobfuscated code on the Harvard Architecture without leakage to upper memories?

1.4 Choice of methods

In order to solve our problems, we followed particular research methodologies for each question.

In the first question, we limited the question with tightly coupled and multi-processor systems, and we chose the design science methodology to follow during our work and we

supported it with literature review methodology. First, we started to seek cache memory architecture to understand how they work, interact with each other and affect performance. Then, we classified them in a proper way, and we noted all remarkable points in background studies chapter section 1. Besides, we try to seek practical implementation on new ARM based boards. We obtained ARM little big multi-processor based Samsung Exynos 5410 and 5420 systems on the chip boards to explore their cache memory implementation. Next, we proposed an attack vector under a constant isolated theoretical system which has the most basic tightly coupled, multiprocessor and Von Neumann architecture with write back cache policy and without cache coherency. At the end of the solution, we clearly stated possible implementation problems which we can encounter during implementation and gave references to the next questions.

In the second question, we used literature which we have already researched, and we inspected the architectural detail of modern cache coherency techniques, especially snoopy cache coherency and multi-processor interconnector design in order to describe a model to exploit the efficiency optimization of common cache coherency protocols found in multiprocessor systems. When we used design science method in this chapter, we used a quantitative approach to measure possible interconnection network latency. This approach is prepared with a simulation experiment whose simulation tools have already been acknowledged by many processor producers as accurate on its latency cycle calculation. We totally designed 4 experiments which are crowded and small topologies with congested and silent traffics and present our results in Appendix B as well as Chapter 5. At the end of the chapter, we built and designed a method to exploit the efficiency optimization of lazy cache coherency found in multiprocessor systems.

For the third question, at the end of the literature overview, we found a similar cases[9][10], which can empower our solution. The problem which we try to solve is not what they are trying to do, but it simultaneously solved our problem. We built our model and concluded interpretation approach which has been seen in previous studies with our own model. When we designed our own model, we used our attack vector which we proposed in Chapter 4 and merge it with a interpreter to execute the code in DCache, which we can access, load and store, but we cannot execute. At the end of the studies, we discussed and criticized practical issues which we can fall during the implementation of our theoretical model.

1.5 Thesis Outline

This section provide a brief summary listing of the content presented in this thesis. The listing is based on chapters, where each chapters and its content is described. First the related works and background studies are presented. Then, our designs is presented as well as further work and conclusion in a sequence. At the end of the thesis, we attached the simulation code which we wrote and the experiments results by Booksim v2.0 to appendix.

- Chapter 2 is related works chapter, which presents the literature related to malware, code obfuscation and countermeasures against malware. Firstly, we give the malware self defense methods' literature, and then, we conclude it with malware analysis and detection methods' literature.

- Chapter 3 is background studies chapter, which presents the background knowledge to design the cache oriented obfuscation system. Our studies are highly related to computer architecture and organization knowledges. Firstly of all, we start with cache related studies. We explain the motivation of cache, how they are working. Then, we conclude them with details about caches. Secondly, we mention about cache coherency and consistency and this section involve with an explanation of snoopy cache coherency protocols. At the end of the chapter, we describe computer interconnection network's basics. This section highly involves with their design features and their effects on the interconnector performance.
- Chapter 4 is called "Cache Oriented Obfuscation". In short, this chapter defines the basics of our thesis. In the chapter, we propose a method to exploit tightly coupled multi-processor systems and explain details and the attack vector which we propose. At the end of the studies, we discuss the pitfalls, limitations and fallacies.
- Chapter 5 is the chapter which we try to solve our second research question. We propose a probabilistic attack method to hide or obfuscate malware. We also define several formulas in this chapter to measure coherency latencies. It also includes experiment results which we designed with a interconnection simulation tool, Booksim. At the end of the studies, we discuss the pitfalls, limitations and fallacies.
- Chapter 6 includes the answer of our third research question and proposes a solution for the Harvard Computer Architecture implementation issue. It includes some elementary studies about virtual machine designing and interpretation. At the end of the studies, we discuss the pitfalls, limitations and fallacies.
- Chapter 7 concludes and summarizes the most important findings in this thesis and add presents a range of topics that should be further work to better understanding of countermeasures and to implementing the theoretical studies which we proposed.
- Appendix A is attached with the python code of simple cache simulation. We used this code to enhance our intuition over cache behaviors. It could be useful to proof our studies further.
- Appendix B includes the experiment results, which are very detailed, and configuration files to replicate it. It could be useful to proof our latency experiments and proposals further.

2 Related Works

2.1 Malware Self-Defense

This section will give an overview of researches about Malware‘s self-defense technique, the methods to analyze them, and their application on concurrent architecture. This section will try to give the literature about malware evasion techniques. These techniques are generally antonym solution which by malware authors, however, there are enough surveys about known technique. We classified all these methods in six categories which are code obfuscation, code reuse, anti debugging, anti emulator and visualization and covert channel over network traffic. This taxonomy is well defined by Jonathan A.P Marpaung, et al [11], yet malware authors used them to protect their own properties.

Code obfuscation was originally founded for protecting intellectual property[12], but it aims to puzzle code‘s binary against merely static analysis and disassembling[13]. The first known obfuscation method used encryption in order to hide its content. It was called Cascade which is seen first 1986[14]. This simple architecture of the obfuscation is called packing[15]. It involves with two parts of binary which are slub part, in order to decipher and encipher.[11]. Cascade was using simple XOR encryption and that was increasing performance.

In early of the 1990s oligomorphism and polimorphism have started to show up[14]. The main idea behind them is basically transforming their slub part in each attempt of encryption process[13]. Today, there are two types of polymorphic approach to generate different variants of slub.[16]

- Rewriting the code each time from pseudo-code so it differs code synthetically, which is actually transformation based obfuscation.
- Self-cipher itself different, order of these ciphers and using different keys.

One of the other important milestone of polymorphic malware is Mutation Engine(MtE) is written by a Bulgarian virus Author, called The Dark Avanger. It was automated obfuscation tool which actually considered impossible in those times.[17]

There are also several methods to prevent unpacking process. These methods are collected carefully by Peter Ferrie [18]. These methods are especially obstacle for automated analysis.

Compare with polymorphic methods, metamorphic approach is more complicated. It is a transformation based method instead of encryption approach.[19] Fundamentally, it produces different codes which doing same blue printed semantic. That just mitigates the detection possibility because of lack of static code.

Network traffic, by malware generally Achilles heel to detect them because they are generally adequately unique traffics to be identified[11]. They usually cover their overt malicious traffic with covert channel methods.[20]

Code reuse attacks are strong attacks because they do not inject any code in them as obfuscation methods did. They aim to use legitimate software to evade them. There are three well known applied version which are return-into-libc, return oriented program-

ming and Frankenstein.

Return into libc attacks were demonstrated by the solar designer in 1997 as a method of bypassing non executable stack to executable libc libraries[21]. Its object is to change the "ret" infrastructure argument to the known address, possibly libc library (stdio, system, etc). However, this attack is limited with libc libraries, which we improved with return oriented programming.

Return oriented programming is a more flexiable version of retur-into-libc attack, which is introduced by Shacham in 2007[22]. Return oriented programming purpose a programming language with small gadgets(instruction bound) which involve all ability of Turing's machine[23]. Frankenstein is one of the novel application of return oriented programming by Vishwath Mohan and Kevin W. Hamlen[24].

Anti debugging and anti emulator methods are really usual for today's malware. The survey of Chen Xu et al. showed us in 2008, a majority of 6900 on-the-air malware could evade their self with exhibiting benign behavior in sandboxes, debuggers, and virtual machines.[25]. VM and debuggers are most important element of dynamic analysis techniques in autonomous sector, because it must run the file just before it touch the working environment. Yet, it is not that knotty to determine whether working environment is virtual or not. Fuzzing cpu bechmarks and comparing results entropy is a good way to determine virtual machines.[26]

Rootkits are the piece of malicious code which aims to crack integrity of the system state. The idea of the remaining invisible to the system state is traced back to one of the oldest virus "Brain"[27]. It was changing the boot process and activate the virus during booting. "Tequila" and "1689" viruses followed "Brain" in 1991 and 1993[28]. There are NTRootkit and HackerDeffender rootkits today. The proper classification of the rootkit by Adnan Abdakka[29].

On the contrary with all studies presented above, we found several directly similar cases[9][10], which can empower our Harvard implementation solution. These studies include interpreters to obfuscate malware and they use these interpretters as virtual computer over a computer. We metion about it with details in Chapter 6, section flying over interpreter.

2.2 Malware analysis methods

Malware analysis methods could be considered two dimensional plane which are "Anomaly" and "Signature based" detection techniques and "Statistic" and "Dynamic" and analysis methods. In addition to these two dimensional aspect, we could add also one more dimension with "Manual" and "Autonomous" methods, yet the figure 1 does not include this third dimension.

There are also several applied techniques, which combine terminology above. We tried to enumerate common techniques used by main frame detection algorithms below.

N-gram It is a anomaly based heuristic detection method algorithm. It is a bit costly process and not practical for client side analysis. It could be fit for honey pot analysis [30] [31] [32]. They are capable against to zero time malwares and that could makes it futures malware detection system.

Sequential approach on system and funtion call This approach is anomaly based dynamic analysis and observing and recording the flow graph of systems and function

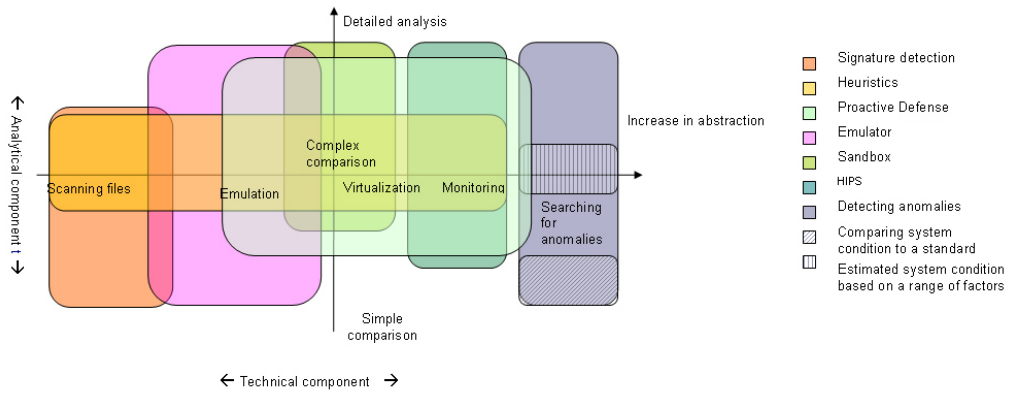

Figure 1: Detection models [2]

calls and try to analysis anomaly behaviors.[33]

Taint It is also called data flow analysis or data flow graph. It is basic tracking marked data values during execution.[34][34][35]

Control Flow Graph They are one of the most important arm of commercial autonomous malware detection tools[36] [37] [38]. After the invention of the polymorphic and metamorphic, syntactic analysis could not bear with them. Then we moved to upper layer of information, semantic layer. Semantic can be representation of code flow, and the routes of the code are adequate to produce signature to identify malware. This methods are a member of static analysis and disassembling and source code analysis job.

Network Monitoring Malware intention of the communication over network actually big clue to detect them. They generally use unique hostname, ip adress or specific protocol with particular way [11].

3 Background Studies

3.1 Caches

Solely, a cache is a small, fast, array of memory which is placed between lower level memory and higher one. It store a special block of information, in order to increase performance of computer systems. It is like a buffer area which has some logic to exploit locality features of programming logic. Today, with increasing of processing ability of computer systems, memory access is bottle neck.

The "cache" is originally french rooted with meaning "concealed place for storage."[5] We can move this definition basically to the computer science. The cache's design is definitely isolated from software layer, however; if you know your caches feature and how caches working you could program a lot more efficient codes easily.

3.1.1 Motivation of Caches and Principle of Locality

The main motivation of caches is indisputably performance. As we mentioned, Performance of high-speed computers is usually limited by memory bandwidth and latency. In order to increase, and turn around that, we use an small array of memory which is located close to the processors. The location of chip is important and there are many design decision (e.g On chip, out of chip), but more crucial properties of caches are their designs (e.g. Naive Capacitor , SRAM, DRAM) and their logic complexity[39].Due to physical constrains, the size of the memory is limited which we can locate close to memory. On the other hand, these design choices are decisive factor about prices of memories. Because of all these reasons, Multi-Layer Memory Hierarchy with several caches between processor core and main memory is well-known option in order to improve performance. Nevertheless, In multilayer memory hierarchy, it is hard to know where the particular data reside in, and whether it is coherent or not. It also add many layer between memory and processor and in some cases it even decrease system performance,especially because of logical complexity of the line.

The idea all the caches logic depending on is Principle of Locality. Principle of locality is actually a concern of information theory[40]. It a conjecture of data distribution and processing order. The phenomenon assume that the the same data and related document will be accessed more frequently than other data[41]. Today, it is the one of the corner stone of computer science. It was first developed with Atlas System with purpose to develop virtual memory systems work well[42]. Then, it spread from search engines optimization to hardware caches.

There are mainly two type of locality of reference:

Spacial Locality Spacial locality propose if there is a particular of memory which is accessed on memory, then it is more likely to accessing memory locations around of it in near feature. Especially arrays and instructions are exploiting this locality. Arrays, formed structure and instructions on memory are laid out lineally over memory. On figure 2, we can see spacial locality simply. For example, during instruction fetches part on the figure, n loop iterations accesses same memory loca-

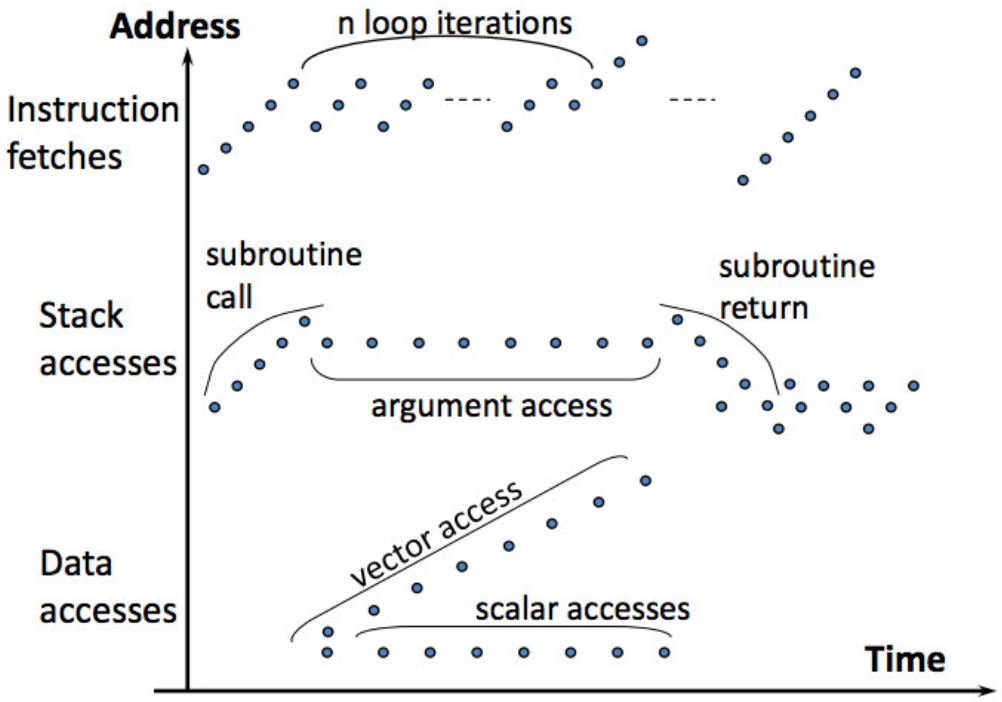

Figure 2: Principle of Locality
[43]

tions for many times. There are also subclass of spacial localities like Branch Locality and Equidistant Locality. They are designed locality types of indeterministic feature of program structure. Branch prediction and Special compiler designs aims to exploits this kind of locality more efficiently.

Temporal Locality Temporal locality propose if there is a particular of memory location which is accessed recently, it will be accessed again more likely than any other location. Especially, variables, subroutines of stacks or other calls exploits this feature of locality. On figure 2, it is obviously seen that the values accessed once is possible to accessed again.

3.1.2 The basic logic of caches

As we said in previous section, the basic logic behind caches is moving arranging caches with local data. In order to provide this feature as smooth as possible, we use a logic circuit called "Cache Controller". It does basic logic comparison and wiring the request and response into the right path. Thus, it intercept the write and read request from processor, replace its memory array with right scheduling method, and evict it safely and coherently. It processes with diving address of th request into three fields which are Index set field, tag field, and block field. In figure 3, these fields showed.

At the beginning of cache process after it divided address fields, It first request right cache line which is shown in figure 2. So if we have M byte memory and N byte cache line, we must have $M/N = cacheline$, then we can represent it with p when $cacheline = 2^p$ Thus, cache controller just wire corresponding line with given set index.

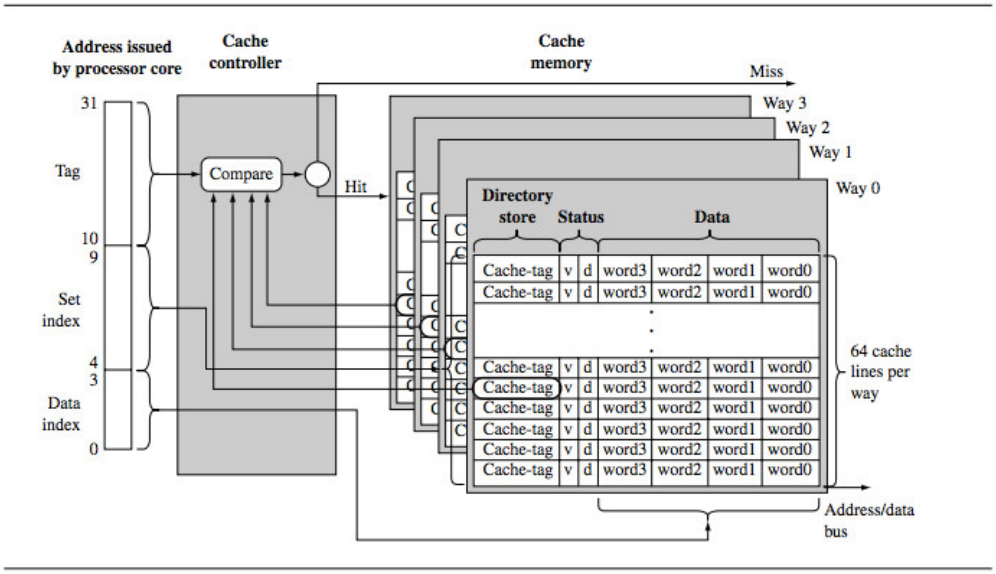

Figure 3: 4 KB 4-way set associative cache with 256 cache lines
[5]

In traditional cache convention, first field belongs to tag id. Tag id is determined depending on other field i.e. the remaining part after index field and block field calculated is tag id length. Tag id is using to verify the stored line is actually belongs to right location of memory. The cache controller has comparison circuit(XOR) and compare the requested address and the address which is in the pointed line by set index field. If they are matched with each other, then it check valid byte and hit or miss. There is a simple AND circuit between tag comparison.

Final field is called data index or block index field. It will point in the cache line the smallest addressable memory location. Therefore, when processor want to read a value, cache fetch the whole block, and that makes cache to exploit spacial locality linearly. However, it will limit the access speed remarkable, if we increase block size. The optimum block size is about 64 byte for many system. As we mention before each cache line includes cache-tag field, valid bit, dirty bit, and some coherency bits in some special systems. The length of the data index field is equal to r if $wordsize = 2^r$.

When we increase the set index count it increase basically temporal locality, but not always. The cache conflict could happened when two memory location which uses same cache line could be used concurrently or twisted. Highly trashing can reduce cache performance. For this reason, associative caches are developed. Set associative caches are represented by their way number e.g 3 way associative caches or full associative caches, and there are group of cache arrays corresponding to the same set index. So that decrease the set index count but increase the performance during conflict miss in some cases. However, because of the complexity of the comparison circuit, it must be carefully chosen the number of ways. The associative caches are showed in figure 3.

The computer architecture we uses today actually first formulated by John Von Neumann [44]. On the first design of computer it was a single cycle instruction machine without any pipeline or superscalar idea. Then Hardward Mark I machine is designed

with proposing two type of caches which are one for instruction, and one for data. Icache and Dcache are specified for their own purpose, because data and instruction on memories have different deterministic properties. Instruction are more tent to be linearly accessed by memory and they has branch locality which can be predict earlier. Icache also could be located more close to decode and fetch parts of processors when Dcache are instead closer to memory fetch parts. Yet, the most significant benefit of Harward design is concurrently usage both caches during pipelined architectures.

3.1.3 Allocation, Write and Replacement Policies

There are three policy type determine a cache behaviours. They are write policy, read policy and reallocate policy. System's performance, coherency, and designs are determined depending on these rules.

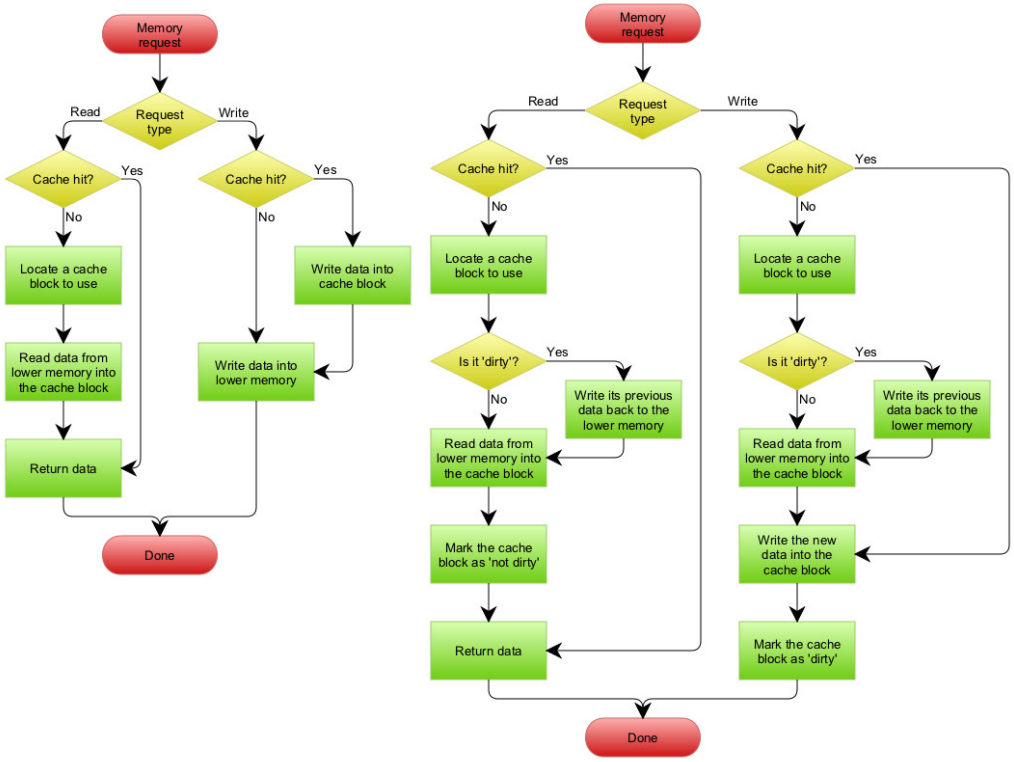

Figure 4: A. A Write-Through cache with No-Write Allocation B. A Write-Back cache with Write Allocation

[45]

Write Policies

Write Through When the cache controller designed based on *writethrough* policy, it write the values into the memory and caches simultaneously, when the write request is arrived from processors. It does not depend on *writemiss* or *writehit*. It will reduce write performance, because writing data on memory is a lot slower, but it stay coherent all the times. It is performance could be increased a bit with write buffer memories between memory and cache.

Write Back The systems with that policies does not have same values in memory and corresponding cache line, so the coherency between memory and caches are provided by a trashing algorithm. Cache line always store more recent data, but if there is more than one cache it is hard to decide which one is more valid or whether there is a valid coherent one. However, it effects performance quite remarkable (e.g. in ARM 15 cpu writing cycle to memory is around 200 cycle, but caches is about 4.) . The system with limited register numbers can overflow to the memory to store loop variables and that could increase write memory usage. *WriteBack* policy makes this kind of systems really effective. The dirty bit are stand for *WriteBack* policy. If you write some value on any cache line, dirty bit must be set for eviction. During trashing process, you must first move dirt block back to memory.

Replacement Policies

Random Random policies are designed to evict a random line in the associative caches. It is not really random on implementation, but enough random to work with it. It sounds to weak and primitive approach but actually it could be really effective on highly associative caches.

Least Recently Used Least recently used replacement policies are actually implemented in two types. Fully most recently used and Not most recently used random. It is probably the most efficient algorithm to replace cache index sets, but it is really hard to implement on highly associative caches. You must record history of schedule and update it each attempt of access. It could be most effective and easy method on 2 way associative caches and it just need one bit to record who used last. It actually increase temporal locality, because it offers the most recently used one is more likely to be used again. The most recently used but random is a hybrid solution of least recently used and random policies. It just record who accessed last and replace one random set except most recently one.

First In First Out It is also known Round robin. It is also mostly using with highly associative caches. In its implementation, it has one one tail pointer of stack and in each attempt of access it evict tail pointers set, and increment the the tail pointer to next set.

Allocation Policies

Write Allocate *WriteAllocate* policy is also known as *ReadWriteAllocate* policy. It refers that during write miss process, cache controller allocates the cache line with related address, as like as normal read miss process. It is mostly using with *WriteBack* policies, because it assume it is more likely to access same data which you write before.

No-Write Allocate $No - WriteAllocate$ policy is also known *ReadAllocate*. It is an exotic implementation of caches. It is generally seen with *WriteThough* policy. This systems can be special to read privileged and they do not hope to read or write subsequent write(or even read after write.)

3.1.4 Miss Type and Advance Cache Optimization Methods

Miss Type

Cold Misses Cold misses are sometimes referred to as compulsory misses . If you never invoke related memory address and if you calling it first time, You will encounter with that misses. It is natural misses, and really hard to mitigate them. Spacial locality is the one of the method to avoid this misses. As we mention before, when we increase the size of block, it will increase spacial locality.Also before initializing memory, pre-fetching algorithms and branch prediction algorithms can be useful to eliminate this kind of misses. In addition to this, usage of large amount of caches will naturally reduce this misses, but it is side effect of it.

Conflict Misses Those misses are the one we are able to avoid. Conflict happens in systems set with lower associativity esp. with direct map systems. To reduce this you should increase associativity. In full associative caches, it all conflict misses are avoided. The change of conflict miss is $tagsize/memorysize$.

Capacity Misses They are also natural misses related with size of the caches. We can not store every information in memory into cache. Those misses are based by definition of caches. You can't solve it even with perfect replacement algorithm, but maybe you could decrease the rate of capacity miss with pre-fetching.

Advance Cache Optimization Methods

Pipelined Caches As we did in processors, we could divide cache organization in two separate stage which are decode and data. It will increase the writing efficiency because it will increase the bandwidth during subsequent requests. However the clock mechanism will decrease to hit time.

Write Buffers Write buffers are small fully associated buffer memories between caches and memories. They effects cache performance because the time between writing values to memory from cache, cache memories must lock if we do not use cache memories. Thus, Cache memories store values to buffer buffer will responsible with writing it. Buffer size is important, when consecutive write operation requested. When buffers is full, it will makes cache lock to get empty.

Multilayer Cache Multilayer caches are game changer optimization decisions, because when we have level 2 caches, then we could have faster level 1 caches, because it could be smaller and simpler. Namely, we are adding systems higher level caches, in order to, decrease lower level caches miss time penalty and increase the hit response time, but it will decrease lower level caches hit rate. Level 2 or higher caches could be also on-chip (i.e fast as possible) and SRAM, yet lower level caches must always be faster closer and simpler.

Victim Caches Victim caches are really useful and simple idea for decreasing miss penalty time. It is a buffer memory, fully associative and mostly 4 to 16 cache line. It stores recently evicted lines in it. It means it increase the associativity of recently used lines on other small buffer with cheap and flexible design.

Hardware Prefetching There are many theoretical pre-fetching method, but there are a few example implemented. The most well known is prefetch the most recently

values incremental block line. That targets to increase most recently used ones spacial locality. It is really efficient to applying it, because increasing block depth is expensive job for caches and increase hit time. If you implement one buffer memory, which prefetch next block of block you need, it automatically increase spacial locality. Also compiler based branch prediction methods are good example of instruction prefetching, however, generally, prefethers for instruction caches load all branches to decrease miss rate.

3.2 Cache Coherence and Consistency

Many modern computer systems with parallel processing ability have support of shared memory in hardware. Shared memory has lots of advantage over message based memory systems. Each processor could access same address space, read and write them simultaneously with using their own caches. This features has lots of benefit such as; low power consumption, higher performance and lower prices. However, without consistency between processors, parallel processing can not use many advantage of parallel programming. It could be also insecure to use a system without consistency between processors.

To provide better understanding of shared memory correctness, we defined it in two separate them in two definition, which are consistency and coherency. Consistency provide a definition of memory access rules and how they will act around computer system with store and load operations. When we compare it with coherency model, it must be more simple and easy to understand it. Therefore, it define a correct behaviours of the memory accesses of multiple threads by allowing or disallowing executions. On the other hand Coherency is a way of implementing a control protocol between memories and processors to support and provide consistency. Correct coherency provide a system which programmer or operator of the system can never determine behaviours (misbehaviours or correct behaviours) of caches[46].

As mentioned, Mention Consistency is try to define to correct shared memory behaviour between many processor in term of loads and stores. It does not have to concern specific hardware issues, such as hardware level pipelines, write buffers, caches, Out-of-order processing schemes etc. However, in the market, there is no hardware provide consistency perfectly, because the reordering store and load operations is regular optimization techniques in out-of-order processors. In addition to out-of-order processors, the multi layer memory architecture makes consistency vague and subtle. Yet, most of the programmers assume memories are completely consistent. There are several level between inconsistent and sequentially consistent memory.

Memory Coherency (a.k.a. Cache Coherency) is actually to impose a protocol between caches to provide a specific consistency model on shared memory systems. Unlikely consistency, it also concern hardware uncertainness and subtle part such as write buffer, pre-fetcher. Typical consistency protocol has features which include instruction caches, multiple-level caches, virtual-physical address transaction, and coherent direct memory access. However, it is not enough to ensure consistency(depending on consistency model) by itself. It tries to makes caches synchronization in shared memory systems invisible from software developer. However there are timing techniques to analysis cache architecture and coherency model of system.

In figure 5 and 6, the consistency issues on multi layer memory systems. Assume there

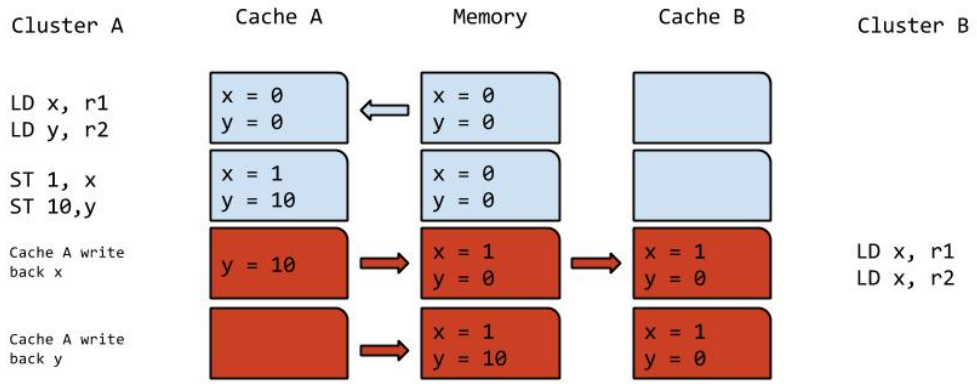

Figure 5: Write-back Policy Cache Memory Inconsistency

is two cluster which has ability to process values with given instruction codes. LD and ST instruction refer to memory load and store request. In figure 5, there is a system with two caches which belongs each cluster and one shared memory block. x and y is represents a particular memory address. Contrast with figure 6, figure 5 uses write-back policy. In step 1, cluster A loaded x and y to the processor(it could be also pre-fetcher who load them to the cache block). In second step, somehow clusters stored 1 in memory location x and 10 in memory location y. In this step, memory is not consistent with memory but it is not hazard because they are not shared with cluster B. In step 3, caches evicted the block which include address x and later address x and y were loaded into the cache B. After this moment, they will never share the values which other cluster is actually using. Y was 10 at the end in the memory but it can't be seen by cluster B, even if it try to read it a million times.

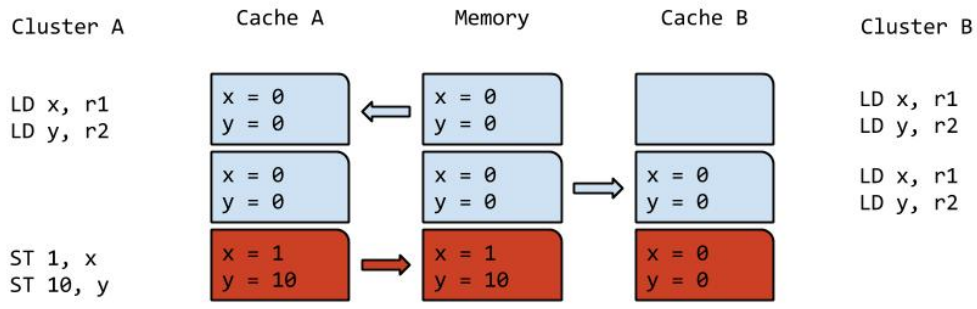

Figure 6: Write-through Policy Cache Memory Inconsistency

Write-through cache policy is intuitively perceived as solution of this problem, because it just write every values directly to the memory and it will be always synchronized with memory, yet it is not. In figure 6, write-through cache policy inconsistency showed. The problem with write-through policy, clusters use values which in their cache instead of memory, so even if memory is consistent with clusters, it is not consistent with each caches. In step 1 of figure 6, cluster A loaded x and y addresses into the its registers. Then, in step 2, cluster B loaded values of x and y addresses. In step 3, system got in

inconsistent state, because cluster A write values through memory, but Cluster B uses the old values, and it will never reach never values, even if it try to load many times. For this reason, many of the large systems which has more than 64 core use this type of cache coherence.

In order to solve this problems, there are several coherency mechanisms and their protocols. Depending on the case and the number of cluster or processor in the system, system could use Snooping and Directory based mechanism. These each protocol have their own benefits and drawbacks. Snooping protocol is tent to use a lot of bandwidth, however, it is faster and more synchronous. Its logic is to broadcast each state to every node on the system. However, directory based mechanism work with request and response. There is interconnector to forward message to the right address and it makes directory based mechanism slower because of the increased latency, lighter because of the decreased bandwidth.

3.2.1 Snooping Coherence Protocols

Snooping coherence (a.k.a. Bus Sniffing) is a technique to have caches to watch other processors caches and provide consistency depending on specified protocol. It basically implemented with external port to the system bus. Therefore, it implemented over cache controller which has feature to watch bus. It makes cache controller bigger and waste more power, so lower layer caches could use less complex coherency protocols and vice versa. There are many snoopy cache coherency protocol also depending on consistency model, but we can categorize them in two class which are Write update and write invalidate.

In this both protocol, we try to get rid of stall data which are in different caches, but it is provided with different logics. Write-update protocol is a broadcast write protocol that in every write attempt, it will write the values into the corresponding cache block but also it broadcast the write message to the every caches on the connected bus. Thus, everyone on the bus which has the ability of interpreting the message of write-update protocol will update stall values with new ones.

Secondly, Write-invalidate is whenever you write, you invalidate other cache copies and reduce to possibility usage of stall data. Instead of sending whole data block, it just send the tag number and state of the tag. It could effectively be successful, if you have limited bandwidth and power source. Most processor with coherency is today using write-invalidate protocol. However, it is efficient if there a few writer and many reader clusters or processors. Comparing with write-update protocol, if there is many writer, it could be less efficient because of invalidation process validate-invalidate-forward hops.[43]

There are many protocols for both write-invalidate and write-update to maintain coherence, such as MSI, MESI (aka Illinois), MOSI, MOESI, MESIF, write-once, and Synapse, Berkeley, Firefly and Dragon protocol. In this thesis, we will just focus on write-invalidate protocols because of their popularity, but basic principles are same as each other.

MSI - Basic

Basic write-invalidate snoopy cache control protocol is MSI (a.k.a Modified-Shared-Invalid protocol). In this model, each cache block has cache tag, and two status bit as same as standard caches, but instead of dirt and valid status bit, MSI cache line has state bits to

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Clean	No	No	Invalid State
Modified	Dirty	Yes	Yes	-

Table 1: MSI states' properties

refer in which state it is. MSI has three state in state machine and they are "Modified", "Shared", "Invalid". two bits can represent four state, so definitely represent three state. The main idea behind this protocol is that one writer and many reader states provide always consistent memory sharing. Therefore, every cache in the system has different responsibilities when they read or write.

Invalid Invalid state is exactly same state with standard caches' invalid state. When cache need to access a invalid block, it must act as cache miss, and be fetched this block again.

Shared When there is no writer processor on this line, and if a processor request this line with purpose of read it, it will be in shared state. It is read-only cache block, and processors are not allowed write without transforming state. The processor also can evict it without writing back to the upper layer memory, because that is for sure, it is clean block.

Modified It is modified and also modifiable cache block. In a memory coherent system there can be at most one modified cache and all other cache must be invalidated. It is responsible with writing back cache to the upper layer memory.

In figure 7, MSI protocol's state diagram is showed. Cache memory launch with invalid cache block, and when a read miss is comprised, cache controller will request memory block from memory. Then, the snooping control bus will broadcast the request of read. If there is a modified copy on the bus, it will abort request of memory block from memory. It will evict its line to memory, and change its state to shared state. Then, memory responds source of the request. After the fetching cache block to the source cache it, it sets the state as shared state. If there is a shared stated copies in the system, It does not matter who responds the request. In any case, It will fetch the memory block, and sets the state bits to shared.

When write miss is compromised in invalid state or shared, It will fetch the data as same as read miss cases, but the difference is it will invalidate other case's corresponding block which are shared or modified. Modified stated block must evict blocks properly. At the end, source cache block fetches the block.

Write hit can be compromise in modified state, and read hit can be compromise in shared state.

MESI - Exclusive

The MESI protocol (a.k.a Illinois protocol due to its development at the University of Illinois at Urbana-Champaign) is a widely used cache coherency protocol[47]. The idea behind the MESI is to use forth state we can use with 2 bits. In order to increase efficiency *exclusive* state is developed by JH. Patel et. al. in 1984[47]. As showed in MSI protocol, there is modified, shared and invalid states but also we have exclusive state.

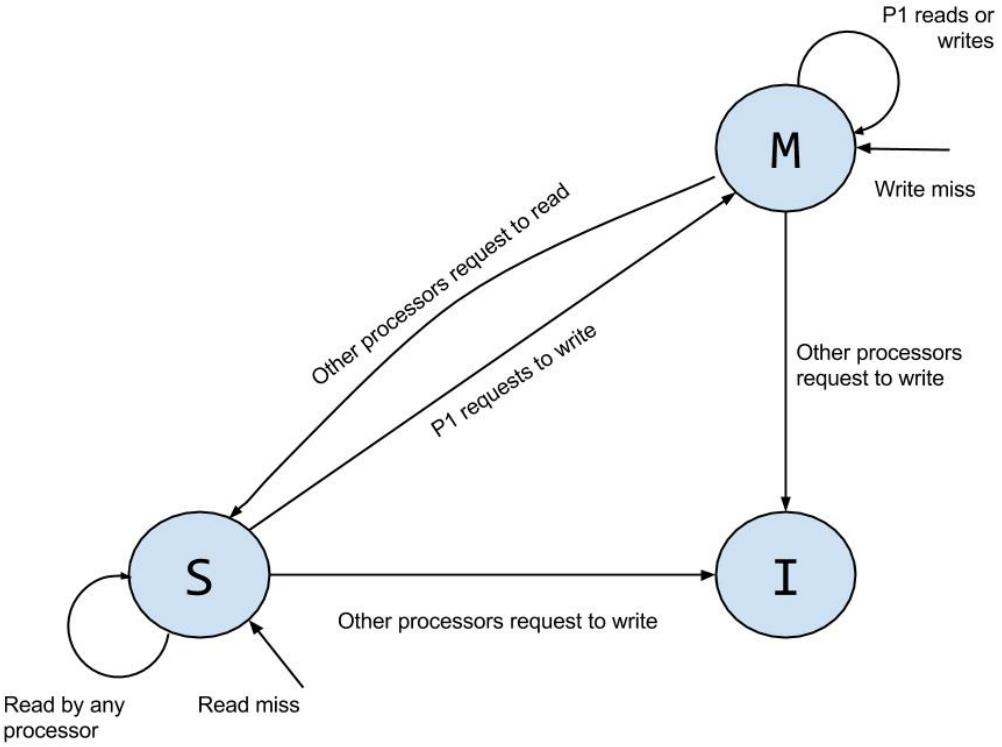

Figure 7: MSI State Diagram for processor P1

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Clean	No	No	Invalid State
Excursive	Clean	No	Yes	Shared Modified Exclusive States
Modified	Dirty	Yes	Yes	-

Table 2: MESI states' properties

This exclusive states also known unmodified exclusive state, if we refer modified state as modified exclusive. This is very similar to the shared state in MSI, and in fact, Shared state is split in two different states. That is because of reducing the communication on the bus and increasing efficiency. In this case, there is exclusive cache blocks which are in read mode and they are unique i.e there is no other cache controller on the system has this cache block.

Exclusive The cache line is only present in current cache memory, and it has not modified yet. It is not a state to provide coherency, but it is state for increasing efficiency of bus bandwidth usage. When a cache line in exclusive state the cache controller can decide the transaction of the line without communicating with other caches. When a cache is requested with load operation, it is loaded in exclusive state, if there is no other cache controller has the cache block.

In figure 8, state transactions are showed. Bus usage is bottle neck, low performance behavior in cache coherency. Silent state transactions are transactions in cache controller without communicating with other caches. For example, there is no need to broadcast

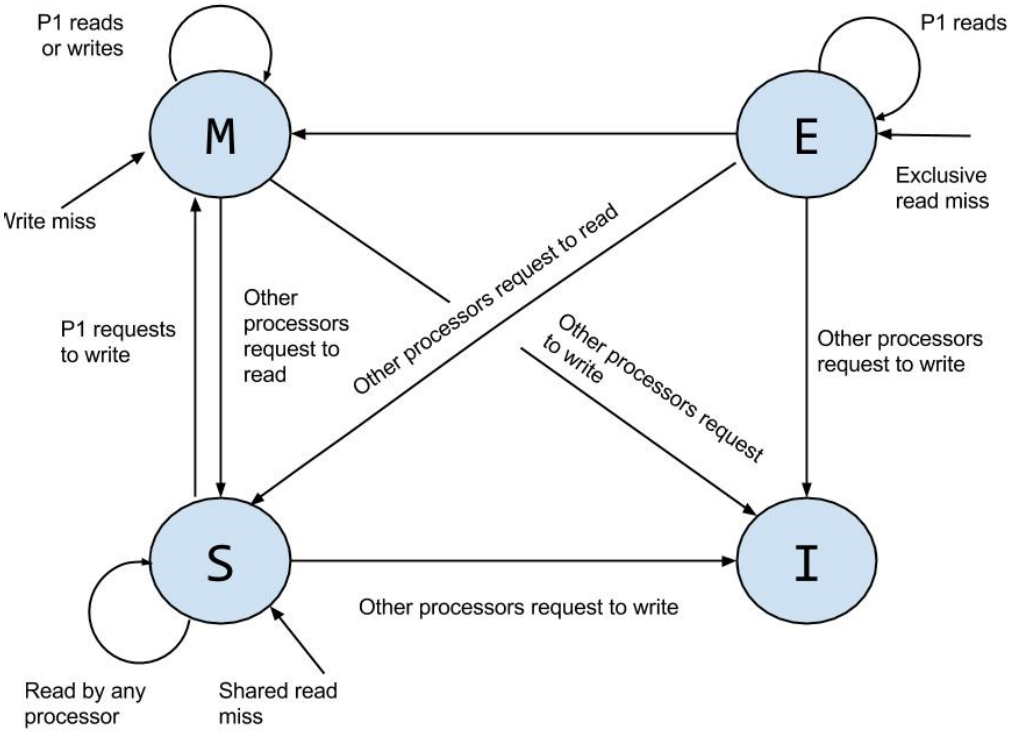

Figure 8: MESI State Diagram for processor P1

and occupy bus for invalidating shared state in MSI protocol. If a cache controller is in shared state, the other cache controller can be shared or invalidate in MSI, so there is no dependency in the system transaction from shared to invalidate. Exclusive state is to exploit the salient transactions. When a load request arrive to cache controller from a processor, it request the line from upper level memory controller and other child caches controller. If any child controller send a shared state broadcast message, it load it in shared state. If there is a exclusive cache controller on the bus, it will degrade its state to shared and broadcast it. If there is no other shared state on the bus. It load the cache line in excluded state. Then, in case of store operation from processor, it will transact its state from exclusive to modified. It does not need to broadcast it, because we know it is unique in system. Contrast with modified state, due to be cleanness of the line, it does not need to evict line to upper memory, it can just invalidate it silently. The weakness of this protocol comparing with MSI, if there is many processor with the corresponding cache line, when it count the copies to test uniqueness, it occupy shared bus more in some cases. If there is n cache controller with corresponding cache line, it will send n broadcast message with this message, however, instead of sending whole line to upper memory it is mostly efficient to send this message.

MOESI - Owned Exclusive

Such processor producers AMD Opteron and Arm Cortex A are using MOESI protocol for cache sharing. In addition to the four states in MESI, a fifth state "Owned" appears here representing data that is both modified and shared. Using MOESI, instead of writing modified data back to main memory, it directly forward the dirty value from cache to

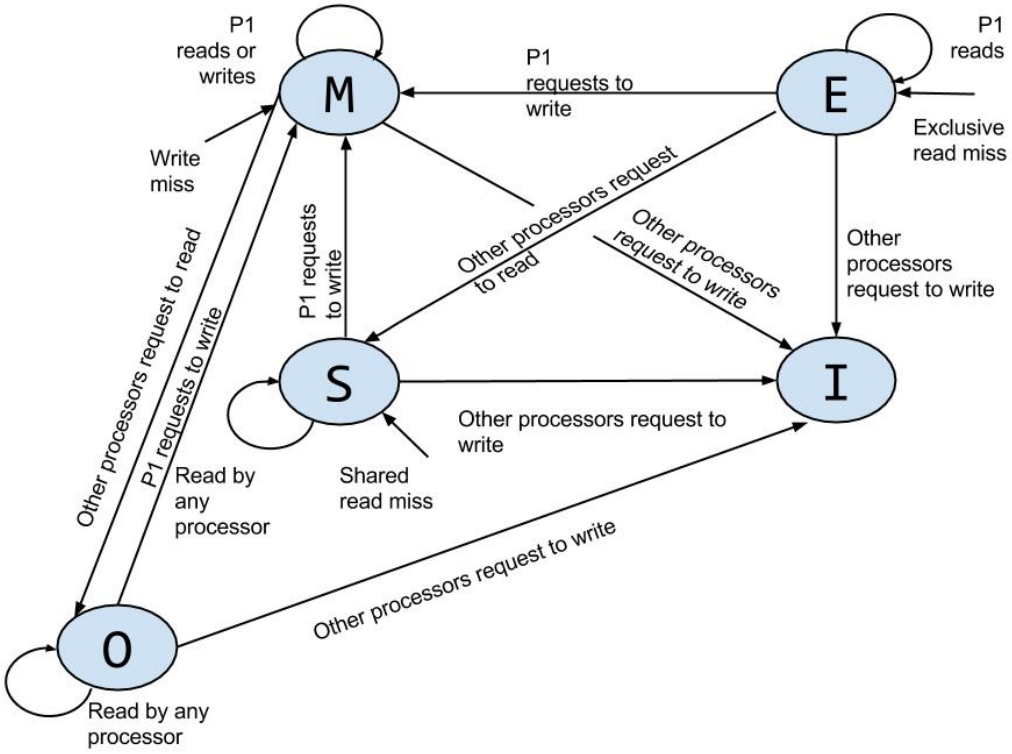

Figure 9: MOESI State Diagram for processor P1

•	Clean/Dirtiy	Write?	Unique?	Silent Transition to
Invalid	Clean	No	No	-
Shared	Either	No	No	Invalid State
Excursive	Clean	No	Yes	Shared Modified Exclusive States
Owned	Dirty	No	Yes	-
Modified	Dirty	Yes	Yes	Owned

Table 3: MEOSI states' properties

cache before being shared, which could save bandwidth and gain much faster access to users to the cache.

Owned Owned state is a state if and only if a cache line can transact in it, when a read request message snooped from another processors when the cache line is in modified state. It allows dirty line sharing between caches, and reduce the latency which is arisen due to the communication between memories and processors. The line is read only by all processors, when it is owned state.

In figure 9, state transactions of MOESI protocol are showed. The relationships of states are almost same with MESI, but there is a state which supplants upper level memory with its own cache line. Hence, it is responsible with evicting lines and cleaning state. The cache line may be changed to the Modified state after invalidating all shared copies, or changed to the Shared state by writing the modifications back to main memory. If could increase efficiency sharply, if the line between upper memory and itself is long and bandwidth is limited. Mostly the L1 and L2 caches are located on-the-chip,

and memory are located somewhere outside, the buses' bandwidth between in side and outside of chips are game changer. It can be efficient to use a chip as a forwarder in many system. However, in the MOESI protocol, it is not possible to forward the cache line which is not dirty but present on the chip. If there is a shared cache line in a cache, and if any other cache controller request to load the same cache line, it fetches it from memory.

3.3 Inter-connector Design

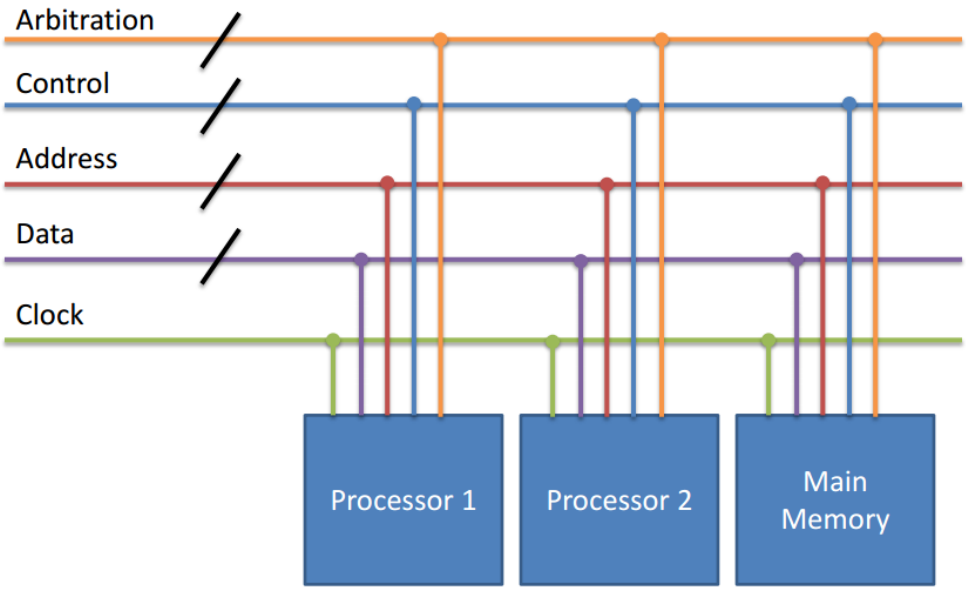

Figure 10: Primitive Multi-Drop Memory Bus
[43]

Computer Bus which is the primitive version of the inter-connection network was designed to transfers data between components inside a computer, or between computers. They are defined to include all computer hardware components and software, included with communication protocols, in order to communicate devices. Devices is generally called as node or end node in taxonomy. However, this definition is quite broad and it covers from today's Internet network to cloud computing network and evolved in many aspect to different direction.

In figure 10, There is an early multi-drop bus example. Multi-drop bus term is used for a bus line with many element on a line (not a ring), and there is an arbitration mechanism, so it is normal computer buses which is used in interconnection taxonomy. The multi-drop buses includes 5 separated wires which is distinguished by their purpose. Arbitration line decide actually how has right to speak, request. There is a logic devices to determine the arbitration and it is one of the most crucial research area in computer architecture and especially interconnector design[39]. Control wire is actually determine the purpose of the node. Generally, they are `store` and `load` operations. The address wire determine the requested address from corresponding place, in this case there is no cache controller so directly memory. Data wire carries the data which is stored or load, so the

communication is synchronous, with consecutively request and reply. Lastly, clock wire provides a fixed, constant frequency to carry values.[39]

On recent systems the communication mechanism between nodes are quite more complicated comparing with given primitive example. The pace in the development on parallel systems makes correlation and communication between notes chaotic. Systems comprise with many nodes and requires high bandwidths to overcome and increase their bottleneck. Intercommunication is still the slowest part of mainframe and personal computers. On the other hand, with multi layer memory aspect, communication between nodes and parallel computing gets more and more complicated. It makes every cache controllers a member of interconnector and perhaps more. Today, there are some coherent interconnector which are also responsible with traffic management (i.e. QoS), barriers between devices and memories, and coherency[48].

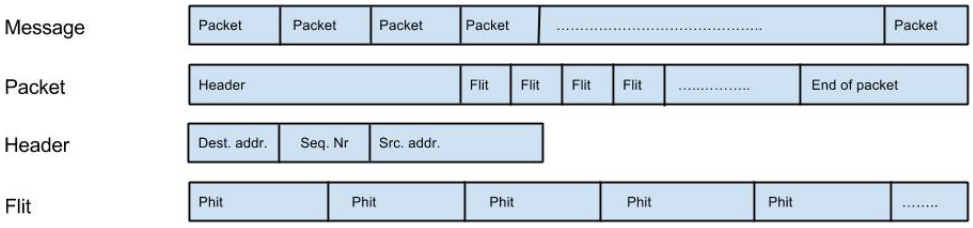

Figure 11: An example of interconnector message anatomy

There are two main category of computer interconnectors which are host based On-Chip and System/Storage area network and remote over LAN and WAN networks interconnectors. On-Chip networks purpose to mitigate the on-flight latency and chip-crossing wire delay problems related with increased technology scaling and transistor integration. Nevertheless, there is not enough space in a single chip to fill many cores. It is a good design for interconnecting ALU, registers caches, compute tiles, and perhaps several cores and memory. System/Storage area networks are the most used interconnection systems between multi-processors, multi-computer, multi-thread systems and memory system interconnection between this cores. Because of physically constrains such as distance and density, it is usual the interconnector between systems and their I/O extensions (e.g DMA chips). LAN and WAN based systems are actually designed to connect enormous number of node together. This kind of networks distributed several locations and interconnecting PCs cluster of computers. Cloud computing is actually one of the good example to show the ability of this species. On of the other advantage of remote interconnectors is that they are generally build on well-known protocols which are tested and acknowledged protocols e.g. Ethernet, GSM, IP, TCP, UDP. All routing issues are tested for many years and solved properly.[39].

Modern interconnections with advance switching and routing mechanism are using message protocols. In figure 11, an example of simple interconnector message anatomy is shown. Alternatively, the bus anatomy we mention above, the message based protocols are packetized. However, this packetizing process has some overhead as latency[43]. Message anatomy of interconnectors comprises several layers[4].

Message The message is the unit of information which must be transmitted with a pro-

pose. If it is about cache coherency, It could be whole line of the cache to provide coherency.

Packet Packets are the fixed maximum sized smallest unit of information which include routing information in its header section. It can also include sequence number for flow control protocol. Its size is depending on the arbitration mechanism on the router or switches. It comprises with data flits which actually part of information in message.

Flit The small unit of link layer is called flit. Flits size are depending on the switching algorithms. In circuit switching flit size are whole packed. They are typically 4 byte to 16 byte.

Phit It is the unit of the physical layer in the interconnectors design. Its size is depending on the clock cycle of the interconnector. On the primitive bus example, the clock mechanism determine the phit size when it tick. They are around 8 bit to 32 bit.

In order to characterize interconnector device we will use several feature of networks which are switching mechanism, switching mechanism, routing algorithms, topology, and flow control of networks. These feature are determine depending on application domain and defuse all character of network. Across the designs, performance with latency and bandwidth parameters and queuing theory is the valuable analysis tools to define network and its classification[39].

3.3.1 Topology

Topology is a mathematical study of shapes and the points and their relationships in the environment. Network topology is actually determining the path and shape of the network. The shapes which topology concerns depending on the dimension they are build on. Electronic circuit are generally build on two or tree dimensional space. The wire and nodes are the basic element of interconnector topologies, but also router is the switch element which can decide the path. The node could be grouped to regulate communication and bandwidth e.g. there could be two group as memories and processors. Also, there are two main type of network topologies, that are In-directly connected distributed and directly connected centralized. The root of "central" word in telecommunication comes from this switches, but they are too vast topics to discuss in this thesis. The basic idea of the centralizing topology is to use a central switching fabric between nodes. The switching fabrics is actually external subsystem or combination of systems, e.g omega and crossbar network topologies[4].

The general assumption of topologies is wire are faster then logical routers and transactions.-Today, there is in chip transaction devices which are faster then wires[39].- In order to design efficient topology, optimum cost and measure its quality, we have several parameters, which are diameter of tomography, routing distance, minimum bisection bandwidth and degree of a node.

Routing Distance It is any given two points distance by mean of number of links or hops

Diameter It is maximum routing distance between any two point of the network. In figure 12 and example A, it is from the first node of the bus to last node, so it is 5. It could be sometimes not that obvious, but it is the most far two nodes' distances.

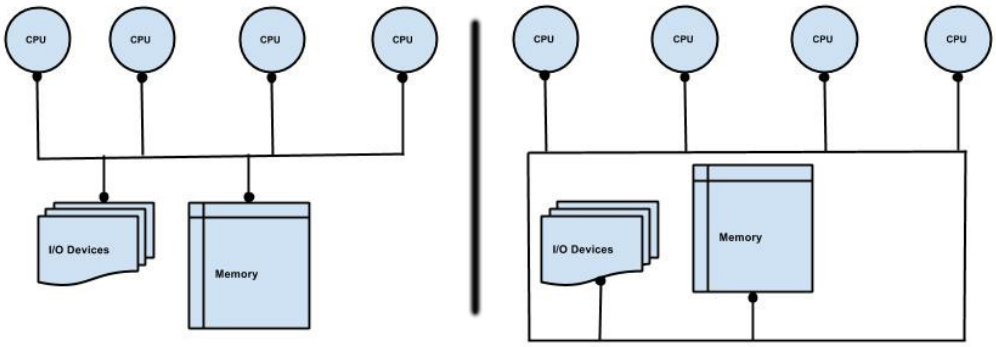

Figure 12: A) Bus topology example B) Ring topology example

Average Distance Average Distance is $TotalDistance/NumberofNodes$. It is one of the value which using in average latency calculation. Generally performance values are compared with each other by average latency.

Minimum Bisection Bandwidth If network is segmented in two equal part, and if the bandwidth of these two segments is as minimum as possible, it is called minimum bisection bandwidth. Typically, this bisection will be the most occupied lines, and the bandwidth of bisection will effect total performance sharply. To embody it better, it is like a bridge between two island. Most of the traffic caused by the occupation on the bridge in ordinary traffic networks. The inner island bandwidth is futile to effect overall traffic.

Degree of a Node It is the properties of each node to imply how many nodes it is directly connected to. The node with the highest degree is applied as degree of the network.

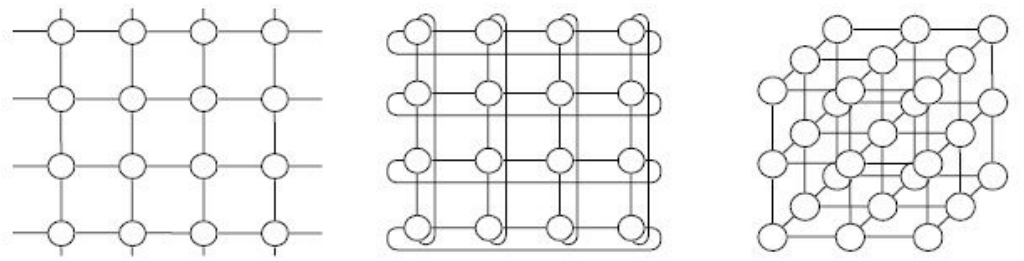

Figure 13: A) Mesh topology example B) Torus topology example C) 3D mesh topology example[3]

The number of topologies designed in literature can be too many to count, but the idea behind them has always same logic. Topology directly effect performance, but also greatly impact the cost of systems. Physical constrains such as chips' pin-out, light speed, dimension count on the board and etc. determine topologies properties. Generally chips are reducing their inner bandwidth, when they are communicating with out side of chip, due to the restriction of building pins[39].

3.3.2 Topologies

Buses and Rings Buses and rings are the first dimensional primitive and basic type of interconnector topologies. They are both directly connected nodes in in sequence as shown in figure 12, but rings are end around buses. Namely, ring topology aims to reduce the longest link which is actually diameter of the network. If N is the total number of node, node i is directly connected to node $i+1$, and node $i-1$ except element 0 and element $N-1$ in buses, and in rings, every node i is directly connected to every node $i+1$ and $i-1$ in mod N. In bus topology, diameter is $N-1$, in ring topology, diameter is $(N-1)/2$. In segmented and pipelined networks, rings are also increasing bandwidth, because when 2 closer nodes communicate with each other, other nodes can connect over all around the line. For example, When 4 and 5 is communicating, in buses, there is no way from 1, to 6 but in rings there is another links goes around the network to connect 6 and 1, so bisection bandwidth is 2 in rings and 1 in buses Rings looks more efficient and logical to use it, however, in practice it could be hard to implement because of the physical constraints, but they are vast topics to discuss in this section.

Two Dimensional Networks: Meshes, Tori Meshes and Tori are the idealized structure of two dimensional interconnectors, because every best two dimensional forms to connect m^2 nodes. while mesh topology is derivative of bus topology, tori are derivative of rings-as showed in figure 13, so terminologies uses meshes end around term instead of torus. The bisection bandwidths are $2\sqrt{N}$ for meshes and $4\sqrt{N}$ for tori. The diameters are $2\sqrt{N}-2$ for meshes and $\sqrt{N}-1$. Degree of the network is 4(5 in some terminologies), and every nodes degree is same in the torus network which can be seen in figure 13.[43]

Multiple Dimensional Networks They are excessive versions of the mesh and torus networks which are influenced by the chips packaging technology. Multi dimensional system over 2D space is still good to obtain higher bandwidth and balance the traffic but more complicated example are seen with Storage/System, local and wide area networks. Because they are instead of chips real 3D systems and surely, 3D networks works better in 3D systems. On an N dimensional system, if you want to build more dimensional systems such as $N+1$ or $N+i$, it increase the wire length exponentiation when you increase i. Wire length is related with flight time of the data, and it effects directly bandwidth. One of the idealized for of the multi dimensional networks are cubes. Cubes are the three dimensional topologies which every nodes are 3Th degree and all nodes are equally close to each other. In figure 13, there is an example of three dimension mesh. CM1 and Thinking machine is the examples of their practice today in market. They actually connects thousands of computers in hyper cube topologies. They connects the mesh dimensions, which are two dimensional with each others.

Fully Connected Star Fully connected stars are directly connected star topologies which actually known in some terminologies as mesh networks. This topology is the best possible topology which have been designed so far, because every nodes are directly connected with each other within 1 routing distance. If there is N number of node that means the bisection bandwidth is exactly 1024, because each node has for

every other nodes another links with an other bandwidth, it makes it bottleneck-less topology.

Omega and Fat Tree Omega and fat tree topologies are examples of centralized systems. They are improvements of crossbar switches. Crossbar switches are expensive designs because its complexity increase quadratically with the number of ports. Instead of the increasing the design complexity, it increase the stages thus, permutation[39]. With N number of node and with kxk switches, $log_k N$ stages each of which contains N/k is required in omega network. However, the reduction of the implementation cost has some negative sides, which are lower bandwidth, dropped packet, more latency.

3.3.3 Switching

Switching name root originally comes from circuits. It is complex combination of many circuits switch which actually connects conductors together. It determines in interconnection networks how data is allocated for data transmission, i.e. how and when the input channel will be connected to the output channel. Buffer states, channel flow, and surely routing algorithms effects the switching designs as we know so far from computer network design. To sum up, it is actually model how to connect different locations and nodes together[43].

Circuit Switching Circuit switching is the oldest type of communication model. Most of the telecommunication networks still using its advantages. Circuit switching reserve whole line and its whole its bandwidth during communication. It could be advantage, if message is infrequent and long, e.g. analogue sound transfer. In addition to this, message could be whole part instead of fixed sized packets, phits, and flits. On its implementation, there are two state which are circuit establishment phase and message transmission phase[4]. The physical path between point A and point B is reserved before it started to transfer message. The line is reserved through *routingprobe* message. It is generally one phit and one fhit long message. It discover the route which is provided by router algorithms and invoke routers to preserve corresponding line to the packet destination. When *routingprobe* reached to the destination note, it means all intermediate router(switches) reserved for connection, then point B will sent back acknowledgment message and finish the establishment. This establishment is tricky point in circuit switching, because it requires setup delay which actually comprises with logical calculation time and time of flight. This time is called as RTD(Round trip delay) or RTT(Round trip time), and it is one time delay. There is no routing decision and switching delay. That is good for long message, but disadvantage for short message. At the end of the communication, it must be ended with *resetprobe*.

Packet Store and Forward Switching This Switching method is packetizing type of communication model. It divides each message into the fixed length packets and phits and flits. Each channel(output and input) in every switches has its reserved buffer memory as large as packet size. It stores fits and plits in this buffers until they are totally loaded, and switch them to the right output channel. The most important think of packet switching is every packets are switched(routed) individually from the source to the destination. In addition to this, every packets could be routed

through different switches and routers depending on routing algorithms. If there are parallel routers between source and destination nodes, it could makes calculation of the chaotic, so they are not preferred by real-time systems' network. Also, implementation of buffers and switching each packets again and again continuously makes it slower and expensive. However, on the other side, it has many advantages. When we need to communicate with short and frequently packets, we can not reserve whole line for two nodes. It makes the same line provide many nodes with sending small packets and also dynamic routing is what makes our Internet works today. Theoretically, It does not have establishment state, but piratically it is so usual to implement it upon link layer.

Packet Cut Through It is also called as wormhole networks. It is a hybrid combination of circuit and packet store switching. It allows simple, small, cheap, and relatively fast switches. In theory, it does not need any buffer and instead of waiting the tail of the message, it just start routing the packet from the head. In implementation input channels could use buffer memories, because of busy output channels, and it whole message could get stalled if there is no buffer to store it, then it need to demand this message again, or worse, it could perhaps lose whole packets forever. In implementation, it divides packets flits and phits and intercept header of packets and then continuously switch or route all phits or flits.

3.3.4 Routing

Routing of the networks is actually provided by predetermined set of rules which draw the path of the packets they will follow on the network. Namely, routing algorithms determine the path over network topology with using switching mechanisms. If we look at the smaller picture, each intermediate router on the network determines which input ports are to connect to which output port. Some of the routing algorithms can complexly adapt their algorithms depending on the network condition i.e. they can monitor bandwidth and latency differences, compare them with demands and route depending on latest traffic, and some of the can simply choose random available path. However, there are two significant criteria of ideal routing: They should be deadlock-free, and they should give the shortest path.

Deadlock is more difficult to handle, and there are two common strategies are used in practice: avoidance and recovery[39]. Avoidance strategy determine deadlock stages and keep packets out of the path which include those stages or vice versa i.e. determining the clean states and routing over them. Instead of avoidance strategies, recovery strategies accept existence of the dead lock stages and have mechanism to detect likely existence of deadlock situation. In many cases avoidance and recovery strategies are using together.

We can simply categorize routing algorithms in two title: obvious and adaptive routing algorithms. Obvious routing algorithms are responding obvious rules repeated, yet adaptive one could change their decisions depending on the network condition. Obvious algorithms generally simple, and consequently, they are generally desirably quick and safe. On the other side, adaptive algorithms can check up the network overall and route packets depending on the current situation. Adaptive algorithms are preferable in huge networks like Internet[4], because adapting the path in response to non-uniform network traffics spread and normalize network bandwidth, but it is to much processing latency and power consumption for on-chip networks. Nevertheless, adaptive algorithms

are what the market focused today esp. for wide area networks.

If an obvious algorithms are always giving the same path for each source-destination couple, it is called deterministic routing. Deterministic routing algorithms are simple way to avoid deadlock patters, but they consumes bandwidth without using many possible routes to aggregate traffic and reduce the contention. There are also non-deterministic algorithms to propagate network bandwidth all around the network, they just always use random way to route packets from source to destination. Also, there are algorithms to flood packets all around the networks to find fastest path to reach destination. It is called flooding routing.

3.3.5 Flow Control

Flow control internals define protocol for synchronization of sender and receiver nodes' communication. with other words, it tries to prevent packet loss. It generally works on packet level, but there are some exotic examples of flit level flow control mechanisms. There are several situation can lead packet loss which are full buffers, busy output links, bandwidth issues, and faults, deadlocks, issues on link isolation, etc. The proper protocol must avoid packet loss and also recover lost packets on line. There are mainly two methods to synchronize flows in on-chip networks and each of them has advantages and disadvantages. They are On/Off with stall signal and credit based flow control mechanisms. Signalling mechanism could be useless because when receiver send a stall signal to sender, it could be too late to stop the flow for further transmissions, because when the receiver send stall message for implying that buffer is full, the further packets would be already sent. To solve this problem extra nodes and buffer could be used but there is no guaranty. On the other side, Credit-base systems needs more preconfigured knowledge between pairs. They should know their buffer space before start communication to use proper credit. If they use lower credit then proper, It will reduce bandwidth, if they use higher, it will increase packet loss. If there are more then one sender for receiver, credit base systems turns chaotic systems to implement. The pairs also compromise on the rate of packets in a period of time to reduce lost rate. For example, it could be problematic issue in heterogeneous systems which have many different processors with different processing abilities and frequencies.

4 Cache Oriented Obfuscation

In this chapter of the thesis, we proposed a method to design a reliable and efficient obfuscation technique for tightly coupled multi-processing systems by exploiting the feature of cache oriented programming. Besides, we emphasized a number of points to characterize proper attack vector. After we had elaborated our obfuscation methods and its primitives, we discussed what it is and what it is not on pitfalls and fallacies section. Finally, on the implementation section, we drew a picture of possibilities for practical applications. However, this chapter and this thesis do not concern specific and deeper studies on obfuscation and memory protection(TLB, MMU, MPU). They are correlated with our thesis, but it is in the upper layer which is like TCP and IP layers of the network. On the whole and in the brief, this chapter gives an isolated workspace for obfuscation techniques.

As we mentioned previous chapters, malware detection tools needs sensors to analysis running code. Regular sensors observes real-time systems with monitoring shared memory, it means a lot for operating systems, because OS, computer architecture and computer conventions assume everything on the memory and ultimate and consistent. With this assumption, dumping memory and analyzing snapshots are practically an efficient and convenient way of malware monitoring and detecting. There are actually a number of sensor type for real time memory observation: external monitors, internal monitors, and exotically virtual monitoring. External monitors in contrast with internal monitors are the systems which is comprised with external hardware devices and its software component. They could be implemented on external PCI or GPU devices, FPGA co-processor or on-board chip[49]. They are efficient because they are pre-installed, omnipresent systems which does not require OS and other middleware platform which constrains their limits. Also, They are efficient because they are the hardwares which can be designed with the purpose. However, they are expensive to implement comparing to internal monitors. Internal monitors are using regular devices on the computer architecture such as CPU, and they are working under the operating system's kernel, which could be easy to deceive [29], [20]. Our methods is actually not depending on the monitoring type, since it is related to where they are monitoring. Tightly coupled multi-processing systems have many processors with their own caches and one shared memory[50]. Cache memories are not developed with the same purpose with memories, but they are developed for performance reason which we discussed in background studies. If we exploit them and design our program properly, cache memories could behave as another layer of memories, and on the monitoring side, even if it scan perfectly memory(s) and detect malicious codes, cache memories are still out of the box.

The required systems for our technique in this chapter is tightly coupled multi-processing systems without cache coherence interconnector. Also, we need simple Harvard computer architecture instead of Von Neumann. Incoherent systems are surprisingly popular because of the implementation errors i.e. Samsung's mainstream CPU Exynos 5410 which sold millions, and also due to costs. Many hardware designers also believe programming

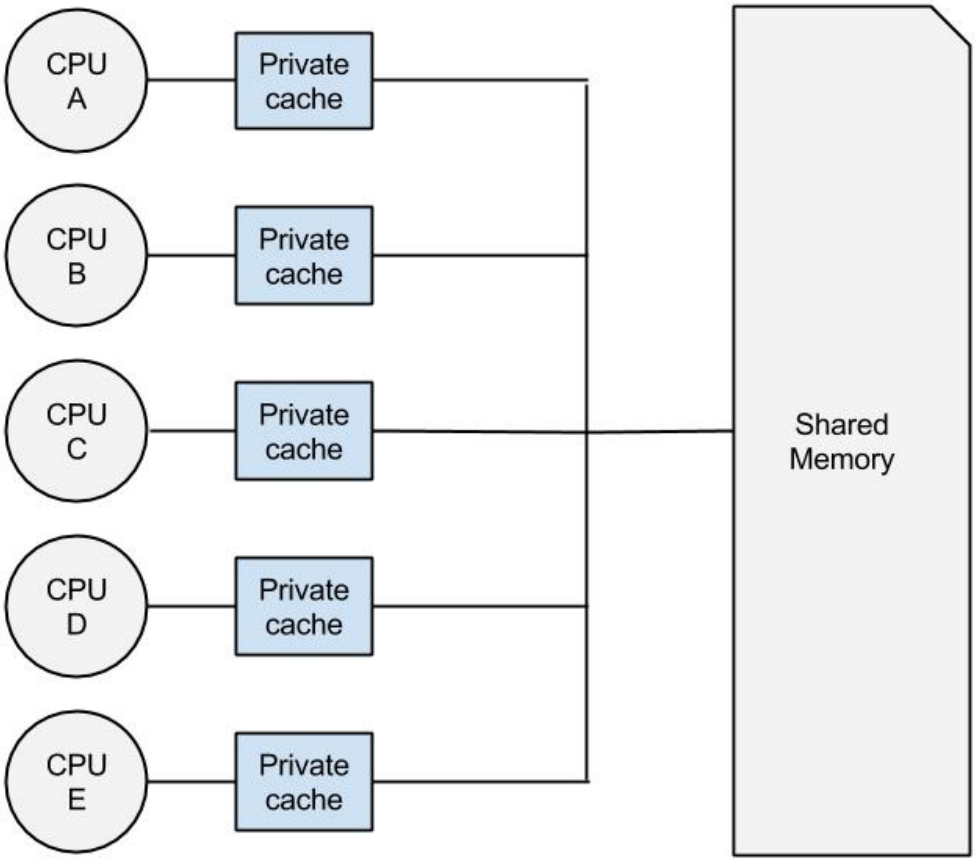

Figure 14: An Example of Tightly Coupled Multi-Processing Systems

with shared memory is not appropriate way and platforms such as Android already using message based communication on multi-threaded application, and they implement clustered processors which limits programs instead of implementing expensive coherency interconnector. Yet, they have not considered security approaches.

4.1 Exploiting Tightly Coupled Multi-Processing Systems

In this section, we will formulate our attack vector one by one, and we will show implementation and theoretical obstacles. Essence of the thesis is exploiting caches to hide values from upper layer memories but also it concern some subtle studies to run a comprehensive code from beginning to end. In figure 15, our attack vector flow chart showed. There is no doubt that most important part of our attack is designing and reconnaissance part, because after we produce cache oriented malware, it is a kind of system depended malware and can't work on the every systems, but can be useful as targeted attacks (e.g Stuxnet). Targeted attacks could be really efficient because of the facts that, all anti malware tools aims massive market, and if malware aims sneakily small portion of the market, they could be successful forever. Setting system up and loading memory second step of the attack vector and first step of attack loop. In this section, we will show cache tricks and explain why it is efficient. In the last chapter we will emphases obfuscation

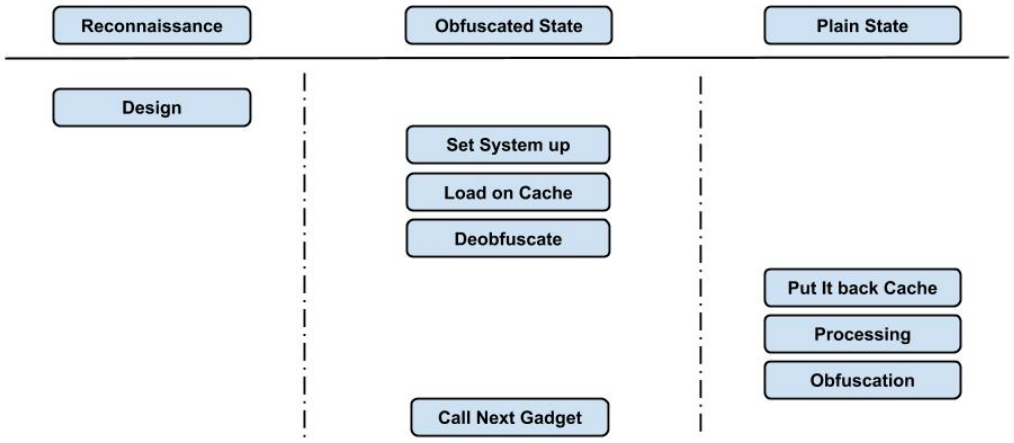

Figure 15: Attack vector flow chart

and deobfuscation methods briefly, and we will discuss some control flow issues. As you can see, when code or data is deobfuscated, it is on the cache memory. When we call the next independent part of the code, it must definitely be obfuscated back, because we do not want to evict cache blocks to memory plain.

4.1.1 Reconnaissance and Design

Design and reconnaissance is the most crucial part of our attack approach, because the size of the cache memory is the basic dependence. If you try to fill more data than its actual size to cache memory, it evicts a number of cache blocks depending on the replacement policy. It is what really we do not want, if the evicted block are not obfuscated. Our first design approach to prevent leakage of plain data upper layer of the cache memory. As we have seen in background chapter, cache are a kind of memories but it also comprise with logical circuit to replace its block autonomously, because they are incomparably small and depending on the main memory to store and load informations. If we claimed they are invisible layer between memory and processors, it would not be totally wrong, and also many programmers today do not concern about their performance feature when they design their program's loops. On the other side, if you consider cache properties while designing your programs, It is hard to determine the base system. For example if we have 32 byte cache block, we could consider when we load an address it will load 32 byte into the cache and in the most frequently working look could progress on this values e.g. arrays are continues bytes and they are generally loading all together. However, cache sizes are extremely various today on the market and if you design a cache oriented program for 32 byte cache block it will not work properly on the cache which has 20 cache block size. Beside that fact, cache size is also important constraint and worst, they have more various then cache block size on the market.

Therefore, we propose to generate a specific piece of code to fit for each cache corresponding to cache size, cache characteristic and also target system arch and security level. Target system architecture may imply cache architecture, and we could design more complex and more flex gadget with multi layer cache arch. if many processors can reach a cache and share values, we could easily broke control flow graph[51], and also on the higher level caches we should use more memory space comparing with lower layer, but if

and only if there is no malware sensor which is monitoring memory, we should use upper layer of memory. When we are using a CPU, it is quite certain that it is the only one who is accessing corresponding cache. In figure 16, illustration of a gadget is showed. It has two sections which are stub and body. This body section is loaded with actual gadget code and tail section. Tail section is responsible with stitching gadget properly. So every gadgets knows before they started to run, which gadget they are followed by. It is a bit problematic and tricky, see Pitfalls and Fallacies section for details.

When we determine the gadgets size, we should consider the cache properties, such as replacement policy, block size, set associativity, line size and total size of cache. The size of gadget must not be more than the total size of cache, and also must be designed with no eviction before obfuscation principles. Hence, we must know who will be evicted next always. On random eviction model, we should not consider eviction during processing of gadget. During running a gadget code, also we must be sure there is memory load or store operation to memory except gadget memory space. Also all gadget must be adjusted to settle in memory continuously in order to utilize spacial locality.

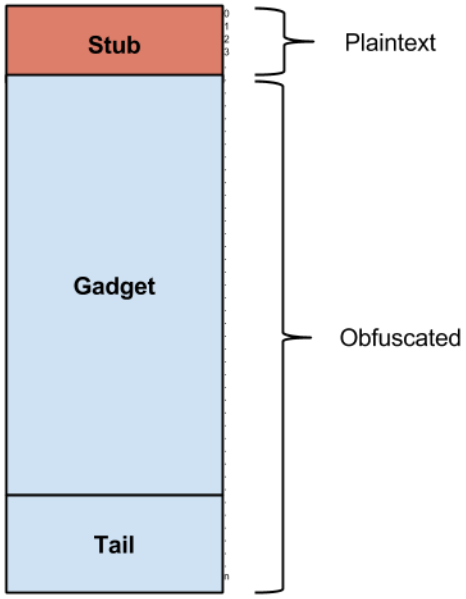

Figure 16: Gadget Sections

4.1.2 Setting System up and Loading Cache Memory

The second step of our attack is to settle on target system. It was the first step on target machine. In order to load all gadgets into the memory, we should use a kind of middle-ware to carry them. In conventional computer architecture and operating system internal, there is not many way to do this. Since volatility of main memory, code must be keep under a kind of I/O devices. This devices are generally connected to DMA devices which provide direct access of I/O devices to main memory. It is a short cut to avoid flooding CPU for each request[52]. Reading this devices from scratch is not simple task, because CPU instruction and memory maps allow you to read sectors or blocks on the disk, and this could be meaningless without proper structure e.g. FAT, LDM, Raid.

Instead, we recommend to use OS or boot-loader features to read files. In listing 4.1, we showed an example for loading a gadget from disk 0 to memory and set instruction pointer to first memory block of gadget on U-BOOT[1]. It is a method as if a rootkit could use, because it load itself just before OS is loaded. In this case, Keep in mind that you should initialize OS kernel on the other processor later. On the other hand, On OS implementation, you gadget should be covered with executable structures such as ELF or PE. The most important point here is when you define your structures you should define your gadget part as executable, writeable and readable; therefore, you can deobfuscate executable code.

Listing 4.1: The code example for loading a gadget from disk to memory and run it on U-Boot

```
mmc dev 0;       #Device Number
fatload mmc 0:1 ${addr_r} gadget_file;
go ${addr_r};
```

One of the other attentive point is choosing appropriate address for loading first byte of gadget. It is really crucial, for all gadgets are designed to size of cache as we mention in previous section. If you start to fetch a gadget to the wrong block in memory, then the gadget overflows cache size, although gadget size is right. As we mention in background studies chapter, cache memories have some internal organization. Therefore, before gadgets are fetching from disk to memory, the first address of each gadgets must be predetermined as fit for the first block of any cache line. In other words, the first bit of the gadget must be loaded in a memory address which is first block of any cache line. If it starts in the middle and passes x block, it overflow x block at the end and when it is requested by CPU, it will evict a set in first cache line depending on replacement policy. To calculate first block addresses, we can use formula below.

$$S = \{address | 0 \equiv address \bmod cachelinesize\} \quad \forall x \in S := firstcacheblock \tag{4.1}$$

Listing 4.2: The code example for loading a section of memory to cache

```
        mov R2, 0x012345 #address
        mov R3, 0x013456 #end of gadget
fetch:
        mov R1, [R2] #Load address from memory
        addi R2, 4 # add size of word to address
        cmp R2,R3 # compare R2 and R3
        jl fetch
```

The code which is showed above in listing 4.2 shows the primitive method for fetching memory section into cache memory. Beside of this, you can enhance this code with adding steps between memory location it fetched, because you do not need to fetch each word as showed above. Instead, you can skip $cachelinesize - wordsize$ byte. The cache controller autonomously fetch the following blocks to exploit spacial locality as we mention in previous chapter. Although every further attempt to fetch word will be loaded from cache memory and it is faster then memory access, stepping approach is always better. On this X86 similar assembly code example, on the first two line, it move the first

[1]U-Boot is the one of the most common ARM boot-loader by Denx Engineering

and last address of the gadget into R2 register. Then, it starts loop and fetches incrementally location in the memory, and when it reach to the ending address it jumps out of loop.

4.1.3 Obfuscating, Running and Deobfuscating Gadget

After we load the gadget into the cache, it is time to deobfuscate it to run it. There are many buzz term for obfuscation methods and techniques such as polymorphism, metamorphism, omnimorphism and homomorphism[13][16][11][12]. However, in this paper, we don't mention the details of these techniques, but instead we will emphases highlights. The most obvious key is that it is not disk to memory obfuscation, it has quite little place to use in cache, so we can not use complicated obfuscation algorithms.

Most known methods we have known so far are XOR and NOT operations. NOT operation does not require key to deobfuscate it, though XOR need. Key could be useful against advance malware detection methods because it is quite common that anti malware software are checking signatures and also their NOT state. Even though XOR needs key and can change it after every obfuscation deobfuscation operation, there are techniques to deobfuscate XOR patters simply since key is stored plainly somewhere in stub part, and it is also reason why we call it obfuscation instead of encryption. Sequential XOR keys are also well known attack against generic XOR decryption which is influenced by enigma and solved like enigma i.e. entropy analysis of assembly instruction. Custom obfuscation algorithms are also another method to obfuscate, but the vital point to design obfuscation algorithms, well known and commonly used algorithms should be preferred because the deobfuscation algorithm and key are in stub part. It is pretty easy to detect, identify and signature unique algorithms, so with manual analysis, signature could be obtain and used to detect our stub part. To solve this, well known encryption algorithms such as DES or a specific part of them can be used.

Listing 4.3: The code example of obfuscation and deobfuscation routine

```
1         mov R2, 0x012344          #start address of body
2         mov R3, 0x013456          #end of gadget
3         mov R4,0
4 obfuscate:
5         mov R1, [R2]              #Load address from cache
6         xor R1, 12345             #key
7         mov [R2], R1
8         addi R2, 4                #add size of word to address
9         cmp R2,R3                 #compare R2 and R3
10        jne obfuscate
11        cmp R4,0
12        jne R4                    #jump next gadget
13 start_addr_body:
14        ...
15        ...
16        ...
17        ...
18 tail_addr:
19        mov R4, 0x234568          #next gadget
20        mov R2, 0x012344
21        jmp obfuscate
22 end_addr:
```

On the figure 16, a typical gadget compartments are showed. It has stub and body parts. Stub part consists deobfuscation algorithm and key(s). It must be usual code which is using frequently, so it can not be identical. Body part consist two subsection which are called gadget body and tail. Gadget body is actual code which is a part of malware. Tail part is a part for maintaining obfuscation process as like as stub part. In tail part, it can regenerate stub part in order to obtain uniqueness on every attempt, but it is baroque job to fit in cache size. Instead, we can change key for each attempt. After we regenerate stub part, then we should obfuscate body part.

On the listing 4.3, a primitive code sample showed for obfuscation and deobfuscation routine. The compartments of the gadget can be seen roughly on the code example. From beginning of the chapter to $\texttt{start}_\texttt{a}\texttt{ddr}_\texttt{b}\texttt{ody}$, it is `stub` part, and rest of it is body. Tail of the body part start from $\texttt{tail}_\texttt{a}\texttt{ddr}$ to end of code. As you can see in this example, tail code is not obfuscated state. The area with ellipsis belongs to gadget body which is actual malware. The first three line is to initiate loop. As we fetched the gadget to the cache on previous section, we started to work on cache in this section. from line 5 to 8, it is actual obfuscation and deobfuscation part. We used here static key but it is easy to use dynamic key with a register. First jump instruction on line 10 is checking for loop variables, and second jump on line 12 is checking for whether it is obfuscation or deobfuscation state. In this example, we used common code to obfuscate and deobfuscate. It is quite efficient with symmetric obfuscation techniques. After it complete deobfuscation of whole body part, it naturally goes in body part, then at the end, it jumps to obfuscation section after it sets R4 register with start address of next gadget's `stub`. At the end of the gadget, it runs instruction on line 12 which jumps new gadget, therefore; it evicts cachelines and fetches its own gadget and do all these routines. We consider that all these gadgets works as a loop of gadgets, so it does not need to have exception or interrupt between gadgets transactions.

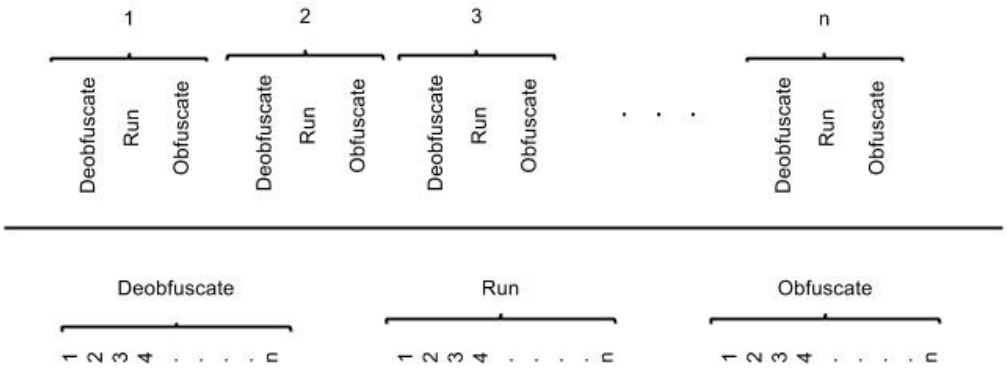

Figure 17: Control Flow Illustration A) Stepped Control Flow B) All at Once Control Flow

We will also propose two type of control flow method for obfuscation, running and deobfuscation routine. In figure 17, these two approach are illustrated which are on the upside "Stepped Control Flow" and on the downside "All at Once Control Flow". Stepped control flow is running instruction one by one in contrast to all at once. Thus even if it leaks some information in the cache memory, it is just an instruction. However, it is barque to implement this control flow due to many reason. It must be perfectly designed

before it run, because there must be register convention between stub and gadget and they should not use same registers in the same time. In addition to this should put a nop or junk instruction in a loop and should change the memory of the concerned instruction with new one. In listing 4.4, the code try to express what it means. The instruction on the memory location 120 is actually junk instruction which is there to be replaced with deobfuscated code, and the instruction on the memory location 116 is to move plain instruction to the junk's place. However, it is not that easy as showed on code, because of the fact that branches. We should consider branches and fetch them in the loop before they are called and it could be made with an interpreter similar code. It is not utopia, but still hard to implement. It is beneficial to emphases one more time that the register in used by obfuscation algorithm must not be used with gadget or stored in a conventional places in case of stepped algorithm usage. This kind of routines are also really common in stack based assembly programming (e.g. calling convention). Instead of stepped control flow approach, all at once approach is easier, faster but less obfuscated. The code in listing 4.3 is one of example of all at once method.

Listing 4.4: The code example of stepped control flow approach

```
100             mov R2, 0x012344        #start address of body
104             mov R3, 0x013456        #end of gadget
        loop:
108             mov R1, [R2]            #Load address from cache
112             xor R1, 12345           #key
116             mov [120], R1           #move ins to junk line
120             nop                     #junk line
124             addi R2, 4              #add size of word to address
128             cmp R2,R3               #compare R2 and R3
132             jne obfuscate
```

4.2 Pitfalls, Limitations and Fallacies

The things we mention so far was explained for actually a simple test system which does not have many features of real systems. That makes it simpler and applicable. However, It is not that simple, when implementing on real systems. The systems today we use even in our daily PC are chaotic and baroque. They have many units to make them feasible for our daily usage. In this chapter, we ignored some issues with memory management unit and memory protection units, cache coherency interconnectors, Harvard Architecture, write through cache policy, out of order processors, multi-threading and context switching, and designed our attack for primitive and theoretical computer system, and it could be quite difficult and knotty to implement it on a real system.

One of the most important issues which we ignore is definitely interconnectors which provide cache coherency. With increasing of multi processor system usage, it is getting common to implement an interconnector between processors to provide cache coherency. It converts caches to devices which can communicate with each other. All cache memories are assumed coherent, so if a malware work on a CPU and its cache can't conceal its content from another CPU. However, in next chapters we will propose an attack to exploit coherency latency. In any case, it certainly constrains our attack abilities.

Secondly, on Harvard architecture, There are two different caches which are separate for instruction and data. It breaks our approach throughout. We can work on upper layer cache, e.g L2, L3, but then they generally share the cache with several CPU. We will also

propose a novel method to bypass this issue as well.

Write through caches aren't also perfectly suitable for our attack, but the stepped approach could solve this issue. As it does not need to deobfuscate all code at once, It could reduce the possibility of obtaining signature with leaking a word of memory per time. One word which is about 32 bit size is mostly not adequate to identify a malware. On stepped method, one of the most important obstacle we can encounter is out-of-order and pipelined processors. In stepped approach, all the instructions are nested, and before it write new instruction to the junk instruction place, it might start to decode it or worse might have been already run. Memory barriers and junk delay instructions could be used against this.

It is a bit tricky to design gadgets, and it constrains functional programming because it is really usual for functions to be independent from the rest of the part, it makes them a suitable candidate to be gadgets. In contrast with functions, gadgets cannot be called again and again. Therefore, we must write another gadget for each use. However, if we convert them to functions with inputs and outputs, it could be an efficient way of designing feasible programs. In addition to functional input and output, they should get the address of gadget which they will stitch with. In order to protect eviction on cache memory, we can use some architectures cache lock. Cache locks can prevent a cache line to evict, in case of fetching its identical pair from memory. If a CPU fetch a memory location, and its corresponding cache line is locked, then the CPU fetches it directly from the memory.

5 Probabilistic Timing Attack against to Snoopy Cache Coherency

In this chapter of the thesis, we proposed a probabilistic attack to increase evasion probability of malware against to snoopy cache coherence protocol on tightly coupled systems with write back cache policy. We briefly explained the issue we can encounter during implementation of the cache oriented obfuscation method with the snoopy coherent systems. The snoopy cache coherency protocol's internals have already been mentioned in background studies chapter; however, we assumed through whole section that all coherence operations are atomic, and contention between processes are not subject. Yet, They are not close to be atomic; and moreover, the latency is sometimes enough long to process and to complete whole gadget or whole malware. In order to exploit this contention, we methodized a probabilistic race condition attack.

As a brief and in other words, instead of giving an absolute obfuscation, we proposed a method which probably obfuscates malware, and this probability depends on the systems design and gadgets' processing overhead. On the other hand, this value gives us a quantitative rate, but if our concern is signature based detection methods, signatures' value can be measured with qualitative approaches rather than quantitative ones, e.g. some signature like port number or IP address are treasury. Yet, the quantity of signature is certainly another value to measure efficiency, especially when we relate binary codes and signature detection.

5.1 The Issue

Cache coherency is a term and discipline arisen from the incoherent states of caches due to parallel computing. It does not need multi processor environment. For incidence, sometimes, DMA devices can be enough to emerge it. In tightly coupled systems, because of the usage of caches to increase performance, it is highly possible to falling into an incoherency state. We also mentioned much more details in background studies chapter. in this chapter, the term of "stale data" is used to describe globally[1] the data which is not reflected for most current or synchronized value in the system(included with other caches and memories). In order to synchronize stale data and provide coherency between caches, cache coherency protocols and policies are used between caches. As we mentioned, most of protocols and policies need networks between each other and logical operator per cache controller. There are many methods to provide coherency between caches and one of the most known and capable one is snoop mechanism.

Snoopy cache coherency mechanisms provide synchronization with a bus watch mechanism in the system bus. These mechanisms imply that cache must first request the data from any other caches, before it request from the memory. The implementation of the snoop mechanism is vice versa, but works as well, as same.Cache generally watches the bus and record activities, arise exception in case of coherency problem(when it is likely to fall incoherent states).[50] There is generally an interconnector to organize snoopy

[1] In local frame, it concerns the relationship between the CPU and cache values rather than between caches.

cache protocols, and filter useless communication.

Moreover, the nature of the cache organization which we use commonly is lazy. The mean of lazy is they do not update the modified block and values until they evict and replace it with other cache block. The reason why they are lazy is obvious, the efficiency optimisations. If they write every modification directly memory as well as cache[2], it will consume most of the available bandwidth. The main essence how we keep our data in a cache as like as private memory is arisen from this laziness, because we anticipate and controls the cache line evictions and replacements. However, we cannot simply say write-through caches are perfectly coherent because of the reasons explained in Chapter 3. For example, $CacheA$ read address x, and then, $CacheB$ read address x. If $CacheA$ write address x with Write-Thought policy, the value in $CacheB$ is still stale.

With perfect coherent caches, we can not exploit private caches to use as private memory (see also. NUMA); hence, we can't evade anything from one CPU to another as we did in the previous chapter. Namely, there is no difference between disk to memory and disk to cache obfuscation. Even though the highest workspace which you deobfuscate your code is cache, cache are synchronizing each other. This coherence gives ability to anti-malware scanning other caches and detecting signature.

5.2 Solution

Let's assume we have a tightly coupled multi processor test bed system, and it has one CPU reserved to scan memory for malware detection, and another CPU is occupied by malware itself. Their caches are snoopy coherent with an interconnection network. They could use any of protocols which we mentioned in background studies e.g. MOESI, MESI, MSI. Let's also assume that the malware which present in the second CPU's cache is designed as we defined in previous chapter. It has prewarmed cache as we described and started to deobfuscate and run the code. The presumption is the CACHE2's cache blocks is accessible by CPU1 as well as any other CPUs. However, the access and synchronization of any stale data is not that simple. It is presumed as atomic, but it is not, and worst of all, tightly coupled systems are heterogeneous with cache coherency, because the distance to memories are not equal and systems are not homogeneous. This heterogeneousness comes with different access times to the memories. Especially MOESI protocol is much more heterogeneous because it works as semi-NUMA(Non uniform Memory Architecture) type cache i.e. a modified cache block can be moved around various caches without updating main memory. Secondly, cache controllers rule are not really elastic. When an address is touched by a CPU, it fetches whole block in order to exploit spacial locality and increase performance[3]. We actually exploited throughout whole chapter two weaknesses which are horizontal directional cache fetching attribute and synchronization latency and heterogeneous access time of snoopy caches.

5.2.1 Horizontal Directional Cache Fetching

In computer architecture conventions, we arrange instructions into memory space, incrementally, and then, we can prefetch them before they run. Also, we are tent to use the space around we recently accessed. It is called spacial locality and we mentioned about it more deeply in background studies. For this reason, we cache mechanisms works hori-

[2] It is salso mentioned in background studies as write through policy

[3] Sometimes they wait for feeding CPU until fetch it all, sometimes feed CPU as soon as possible depending on algorithm.

zontally. It fetches a particular size of memory in the same time, and put it a cache block. Besides, cache blocks are the smallest addressable memory spaces; therefore, it makes caches more simpler, faster[4]and cheaper. Accordingly, Fetching and eviction operations are handled as line-based namely horizontally. In figure 18, we showed data fetching

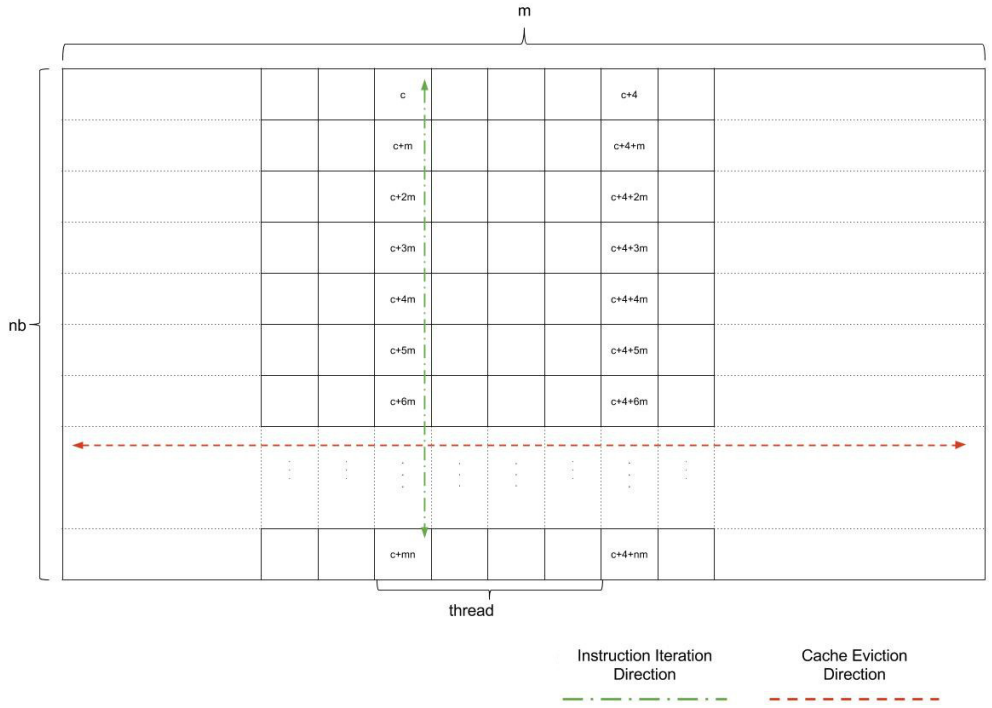

Figure 18: Directional Exploitation

direction and its contrast with our instruction iteration direction approach. As we mentioned many times, but in order to emphases it one more time, instruction sequence normally increments one by one. Yet, in our approach it iterate as in equation 5.1. m is the cache block size, and n is the number of cache block line in the whole cache. nb is number of line which is allocated to body of our gadget as mentioned in previous chapter, so this figure is a frame of cache in which obfuscated part of our malware allocated. Let's say c is the initialization point of our malware. When i is the number of instruction on the queue, $I(i)$ gives us the location of instruction in the memory; thereby, in the cache.

$$I(i) = m * (i \bmod (nb)) + ((\lfloor i/nb \rfloor * thread) + c) \bmod m \tag{5.1}$$

$thread$ value in the cache is step number between the vertical blocks. In order to generate a function which onto(bijective) the cache frame, equation 5.2 must be provided, because after it overflow mod function, it will uses just the one next block which previously used and go on until end. It is obvious that $tread$ should be smaller than m, yet it is already in mod and i must be smaller than total size.

$$\forall\, m \bmod thread = 1 \;:\; I() \text{ is } bijective \text{ } function \tag{5.2}$$

Yet, Why do we iterate institution sequence vertically? Indeed, we assumed that anti-malware scans horizontally from CPU1 above, because it is faster. For example, our code

[4]After an amount of memory, it could decrease performance.[43]

starts from c and our second instruction in $c + m$, with purpose of scanning from CPU1 in this order, it should fetch first the line of c into Cache2, and then, it should fetches the line of $c + m$, and so forth. In the perfect world, with atomic instruction to fetches whole cache line and without latency, it could be race condition free approach, but in real systems, it is not. We will combine this attack with latency problem in the next sections.

5.2.2 Synchronization Latency of Snoopy Caches

One of the famous myth on the computer science and design is that tightly coupled parallel architectures are mostly considered as symmetric systems, but if we look more closer, the caches usage and moreover cache usage with coherency protocols and network makes them quite asymmetric and preforce them to be heterogeneous. Processors could have same properties and be arranged in symmetrical, but if they can't give same throughput in an time interval, we can't call them homogeneous[5].

Latency Calculation

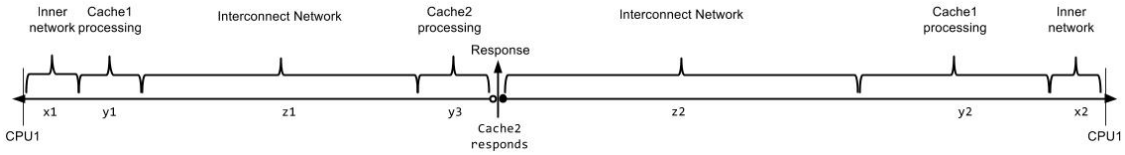

Figure 19: The Time Line of the Fetching Cache Line which is Used by Another Cache

In figure 19, we showed a representational illustration of cache line requesting process which is used in another cache, and labelled latency types with x for inner network latency, y for cache processing latency, z for overall interconnection latency. As seen in figure 19, the throughout latency for synchronization process with another cache is showed in equation 5.3. However, the latency of direct reaching to the cache is showed in equation 5.4 which is obviously shorter[6].

$$L_s = x_1 + y_1 + z_1 + y_3 + z_2 + y_2 + x_2 \tag{5.3}$$

$$L_d = x_1 + y_d + x_2 \tag{5.4}$$

All this latency variables is depending on many different factors which we can calculate them in theory, but which is difficult to estimate in practice. x variables are the latencies between CPU and cache. It is tent to be really short, because L1 cache and CPU should be designed so close. Mostly $x_1 = x_2$, yet it is not certain. The reason is write buffer and pipelined CPU makes it case depended. Generally, most of the latency comes from the queue of cache which also related to writers block and pipelined architecture as we mentioned in background studies. y variables are cache's logical response latencies. As we mentioned, It depends on cache size, associativity, line size and logical operations complexity. In this example, we have one layer cache, but most systems use multi layer caches. For each layer, it adds more overhead. In brief, caches are as fast as how simple

[5] In order to prevent this, there are systems which avoid to use caches, when they are sharing data. Because of the symmetry of cache processor relationship, they keeps their symmetric design themselves.

[6] This formulas are valid for the simple systems showed in figure 14

they are.[7] However, there is no doubt that most important and game changer latency is z namely, interconnection network latency. Mostly, lack of bandwidth is considered as

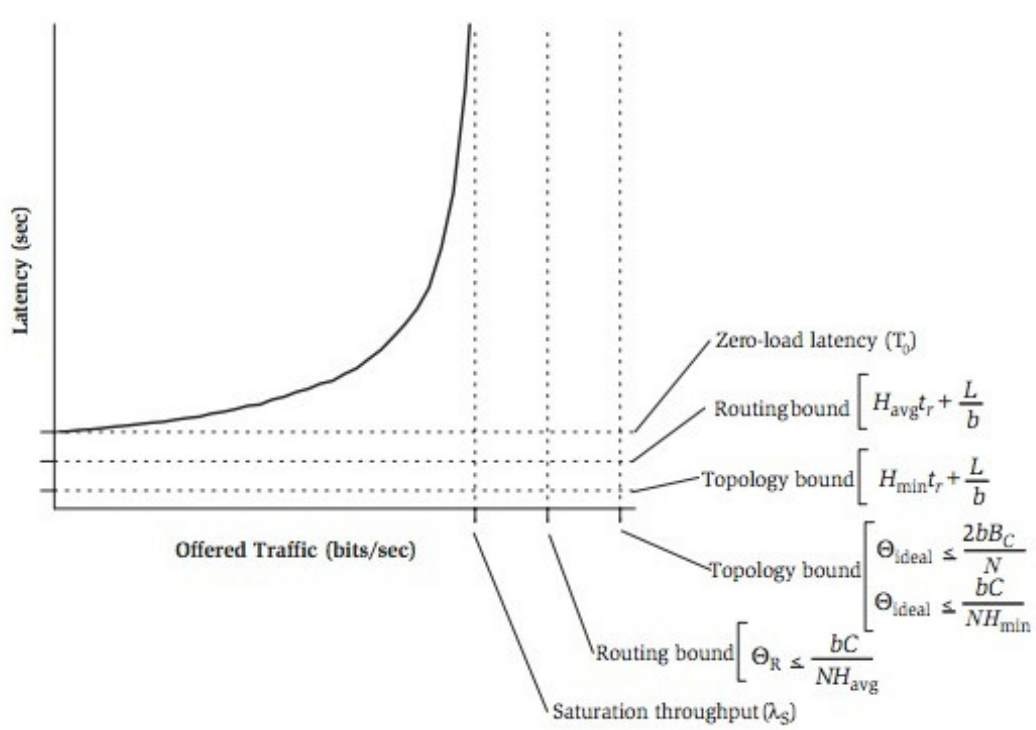

Figure 20: Interconnector Latency Versus Offered Traffic [4]

the only reason of network latency, whereas there are many other factor and problems. Nevertheless, it is one of the most important source of latency. Bandwidth is the rate of the data can transmitted from point a to point b in a given time, but originally, it was the number of wire in width of buses. This definition ignores the clock speed contrast with current definition because the speed of wire is related with speed of light and resistance of the wire, but instead, the things which limit it is the performance of source and receiver. The bandwidth could be formulated as $b = n * f$ where n is width of the channel and f is clock speed. Surprisingly, if our message is smaller than channel width, bandwidth can not effect latency because if we have 100 bit bandwidth, then it carries 20 and 100 bit in the same time. Notable the interconnection networks' costs are routing, serialization/deserialization, link traversal latencies[4]. We already have mentioned about their details; still, it is good to shortly recall that serialization and deserialization are processes to converting messages to given channel bandwidth and could be shown as $sd = L/b$ where L is length of message and b is bandwidth. Therefore, overall latency can be measured with formula 5.5.

$$T_0 = \sum_{k=1}^{minr} tr_k + \sum_{k}^{minc} D_k/v_k + L/b \tag{5.5}$$

In this equation, $minr$ and $minc$ denote the minimum number of router and channel

[7]Their design affect performance a lot, but they are mostly SDRAM instead of DRAM or switch based registers

between point a and b. tr denotes time consumed during routing process, while D/v denotes distance divided by velocity. The easy way to calculate T_0 for each flit is $T_0 = T_{head} + L/b$ because $\sum_{k=1}^{minr} tr_k + \sum_{k}^{minc} D_k/v_k$ is calculated one and only one time in pipelined networks as have been mentioned. If it is not pipelined, the latency could be measured by formula 5.6 or if it was store and forward flow control instead of cut-through as shown in equation 5.5, the latency could be measured by formula 5.6 (see more details [4]).

$$T_0 = (\sum_{k=1}^{minr} tr_k + \sum_{k}^{minc} D_k/v_k) * L/b \tag{5.6}$$

$$T_0 = (\sum_{k=1}^{minr} tr_k + \sum_{k}^{minc} D_k/v_k) + (\sum_{k=1}^{minr} *L/b) \tag{5.7}$$

However, T_0 is not a general latency function, and it is very special status of networks which is also called zero-load latency. Zero-load latency is the lowest bound of the latency where there is no contention between packets. In figure 20, a generic latency vs offered traffic curve is showed. Although they are the most accurate way to measure and determine ultimate performance, and we are using discrete event simulation to draw them. Theoretic latency bounds, which are topological and routing, and their corresponding throughputs are showed in figure. In formula 5.4, 5.5 and 5.6, zero latency values are all shown with minimum routing hops which means topology bounded zero latency, but actual zero-load latency, T_0 in figure, incorporates the constraints of topology along with actual performance, routing, flow control and line traversal latency[4]. As have seen obviously, if you increase the contention between packets through increasing offered traffic, the latency grows about exponentially. It is one of the most important factor support our proposes and encourage us to implement because of the fact that roughly loading memory and storing back to the hard disk[8] produce remarkable traffic which can lead considerable latency.

Simulation Result

An accurate simulation is definitely one of the most important tools for analyzing interconnection networks and exploring design tradeoffs. The reason why we are using simulations instead of real systems is a bit similar with theoretic physic. There are many noises on information we gathered from real systems and they are extremely expensive and difficult to implement. Even one of the most acknowledged books about interconnection network, which is called "Principles and Practices of Interconnection Networks" repeatedly emphasizes that designer's intuition is the most important factor to design better performance on interconnection networks[4]. This book has a simulation tool freely available at http://cva.standford.edu/[53]. Even though, every processor company invests their years and efforts to design better simulator, this tool is quite simple and totally free licensed. However, it is not designed for coherency purpose, so its model designed in flit level and includes many topologies and routing algorithms. Indeed, it can give us some important cues and ideas.

We designed it with an example topology which we use in multiprocessing systems. It is the inheritor topology of Fat tree. In figure 21, the topology is shown which has

[8] It is basically dumping memory

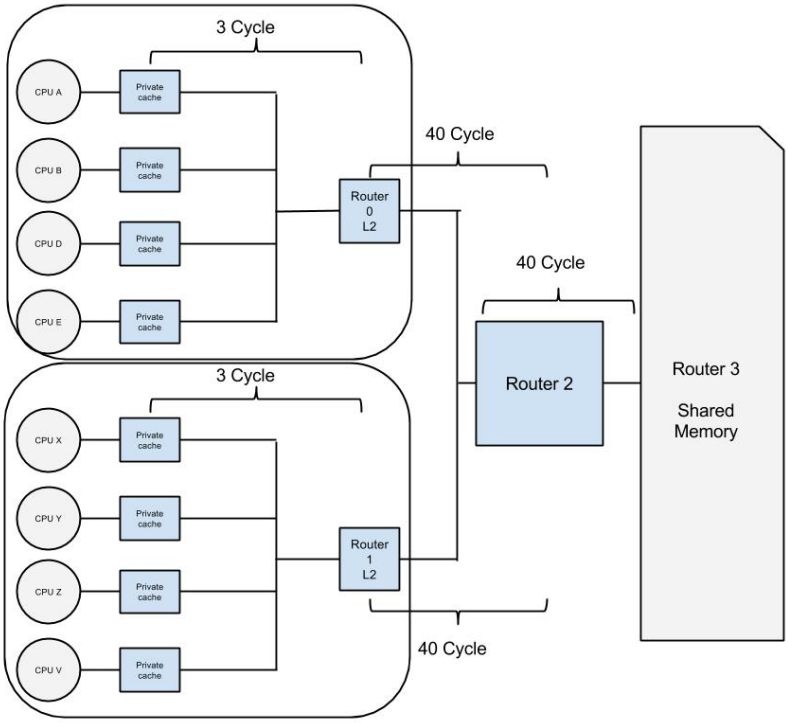

Figure 21: Our Simulation Topology[4]

two processors, 3 routers[9] and shared memory. This designed is influenced by one of the most known ARM chips Samsung EXYNOS 5420. This latency values are rounded numbers, but they are close values which we obtained the concerned chip[10]. One way channel length is about 3 cycles from L1 cache controller to L2 cache controller, 40 cycles from L2 cache controller to L4 cache controller. This contrast could be because of the difference between in-chip out-chip communication and also the complexity of caches internal operation. We assumed the routing delay is about 2 cycles, even though it could be a bit more. We used 1 cycle credit delay for flow control.

The injection rate of simulation is the number of packets which it injects every cycle time. The book claims average rate is 0.15, but we consider that it is enormously high for our experiment. As we have said, injection rates could be really high during memory dumping; however, it could be decreased, if scanner avoids bad programming practice and exploit cache performance feature, it could be around one store and one load operation request for each cache block from upper layer cache. If we take this into consideration, we assumed the injection rate 2 and 4 for each 100 cycles. It is also good to assume it lower for any cases.

One of the other important value about our experiment is the character of our generated traffic. We used a uniform traffic generator which means every nodes talk with each other. However, it is not actually appropriate for our attack. In our example, scanner CPUA talks a lot, and other CPUs talks nominal, but CPUX attacker CPU does not load or store, after it initializes. We can implement our own traffic generator for this simulator

[9]Though we said 3 routers, we used in figure and topology design file 4 router. That is due to lack of simulation tools. We tried to give a latency between router 3 and shared memory

[10]we used CPU-Z program to measure cache latencies.

in the future. This simulator is also highly adaptive and extensible with plugins.

In implementation of our topology, we decided to design two different modes to emphasize contention over network, which are small topology and crowed topology. In the small topology experiment, we assumed there are just two active nodes, three active routers and one shared memory in the network and they are the ones which connected to CPUA and CPUX. In this example, latency is mostly based on distance and pure logical complexity, rather than contention of routing or bandwidth. In the second experiment, we designed the crowded topology which has four nodes connected to router0 four nodes connected to router1, and they are all connected to router2, and also we presumed there are two "slave nodes" which might be DMA devices and one GPU connected to router2.

	Packet latency Min/Avg/Max	Network latency Min/Avg/Max	Flit latency Min/Avg/Max	Hops Avarage
Crowded 2%	164/68045/171627	13/417/1487	13/353/1487	2.04442
Small 2%	8/72/216	8/72/216	8/72/216	2.30611
Crowded 4%	15698/178598/458108	13/410/1736	13/353/1736	2.04591
Small 4%	8/322/1273	8/129/376	8/129/376	2.33081

Table 4: Simulation Results Comparison

The result of the simulation is added in appendix B, and briefly showed in table 4. We made four experiment with two different topology and two different injection rate as have been mentioned. For more details, you can check appendix B. We present in this table three different kind of latencies. Flit latency is a simple latency time required from the beginning of flit till end. Latency of a packet is measured from the time its head flit is generated by the source to the time its tail flit is consumed by the destination. Obviously, flit latency is the delay time for transferring a flit from one node to next node which routers applied as node in this case. The contrast with network latency and flit latency is tricky. Network latency is flit latency plus extra serialization and routing costs. Hops average is the average number of hops every flits traversed (except source hop) during experiment. It could give an idea about distribution of message. As you can see, the average hop number is lower on crowed topology because nodes under the same router can talk with each other i.e. CPUA can send a message to CPUB as well as CPUX does CPUY. The value which we concern is packet latency, because packets are smallest meaningful communication object. To provide coherency, cache controllers communicate with packets between each other.

As you can see, on an typical crowded network, average latency varies around 68000 cycle to 178000 cycles. If we assume clock cycle is around 1GHZ, 100000 cycle is about 0.1 second absolute time. On the other hand, it is around 72 to 322 on small sized topology. However, if we concern the communication between CPUA and CPUX it is at least 40 cycles + 40 cycles + (3 * routing costs) + 3 cycles + 3 cycles = 92$cycles$ because of the distance. It means that the latencies we need to concern is above average and close to upper bound because CPUA and CPUX is one of the farthest couple.

It is hard to point an exact average latency value for any system, even if we are exactly sure about target system. The contention is the most important factor for latency. In order to increase the chance of our attack we could use noise production methods from other nodes, but because of its character and highly detectability, it is not recommended, still

the scanning and storing processes produces enough latency due to high dependency of memory load store operation.

5.2.3 Overall Explanation of the Timing Attack

										REQUEST TIME	RESPONSE TIME	FETCHING TIME
0 300	5000 7000	10000 15000	1000 1020	7000 8000	10000 15000	1500 3000	9000 9100	15000 20000	5000 7000	0	400	1000
0 300	5000 7000	10000 15000	1020 1500	7000 8000	10000 15000	3000 4000	9100 9200	15000 20000	5000 7000	1000	1400	2000
0 300	5000 7000	10000 15000	1500 3000	7000 8000	10000 15000	3000 4000	9200 9250	15000 20000	5000 7000	2000	2400	3000
0 300	5000 7000	10000 15000	1500 3000	7000 8000	10000 15000	3000 4000	9250 9450	15000 20000	5000 7000	3000	3400	4000
300 1000	5000 7000	10000 15000	1500 3000	7000 8000	10000 15000	3000 4000	9450 10000	15000 20000	5000 7000	4000	4400	5000
300 1000	5000 7000	10000 15000	1500 3000	7000 8000	10000 15000	4000 5000	9450 10000	15000 20000	5000 7000	5000	5400	6000
300 1000	5000 7000	10000 15000	1500 3000	8000 9000	10000 15000	4000 5000	9450 10000	15000 20000	5000 7000	6000	6400	7000
300 1000	5000 7000	10000 15000	1500 3000	8000 9000	10000 15000	4000 5000	9450 10000	15000 20000	5000 7000	7000	7400	8000
300 1000	5000 7000	10000 15000	1500 3000	8000 9000	10000 15000	4000 5000	9450 10000	15000 20000	5000 7000	8000	8400	9000
300 1000	5000 7000	10000 15000	1500 3000	8000 9000	10000 15000	4000 5000	9450 10000	15000 20000	5000 7000	9000	9400	10000

Figure 22: The illustration of cache and time Interaction is showed with leaked portion of obfuscated data

Let's presume again we have a malware which is designed as we have proposed in chapter 4 and we have the same system which we have designed in figure 21. Then, the malicious malware is loaded from non-volatile memory e.g. hard-disk to memory obfuscated and it loaded to the cache which belongs CPUX. It arranged the structure of cache as have been mentioned in previous proposal. It have been designed as same as it was so far. The point in this situation is we have a memory scanner and dumper application which works in another cache (in this example it is the cache belongs CPUA.), and we assume CPUA has coherent cache with $cache_X$. This coherency can be provided with snoopy cache coherency through one of the protocols of MEOSI, MESI, MSI. After malware loads the obfuscated code from the memory to the cache, it is allocated as shared state in MSI or exclusive state in MOESI and MESI. It is good to know that it is not going to transmit to modified or owned state until it deobfuscate it. The things which we don't want to share is plain data, and lets call this action as leakage. When we de obfuscate the code, its block in the cache transmits to modified state. Then, if an scanner cpu, CPUA, request the block line which we deobfuscate, the long adventure of synchronization starts.

Our stepped control flow approach has already be proposed as a solution for coherency because instead of whole cache block line leakage, it will lose one step during even theoretical atomic snapshot, and strongly probably it is not going to be meaningful to be

signature; however, this stepped approach is not completely good because of its complexity and the effect of that complexity over stub which increase the chance of detectability. The attack we have proposed in this chapter is starting with arranging the steps(more flexible steps, more likely to be smaller gadgets) to the cache vertically as shown in figure 18, with given formulas 5.3, 5.4.

The example which we illustrated is shown in figure 22. The time zero on this example is representing the time of the first moment which actually scanner reach the memory location of malware, while the malware's process is flowing. We have another latency assumption here the latency of $x1 + y1 + z1 + y3$ in figure 19 is equal to 400 cycle time and $z2 + zy2 + x3$ is equals to 600 cycle time. The second one is longer because first one which request the cache line is a packet with a header flit, and tail flit. On the other hand response packet comes with whole cache line[11]. The response time, which we showed in figure 22 and mentioned in figure 19, is the moment, after it received load request from CPUA and responded it with the cache line. In this moment, cache line is not in modified state, since it is transmitted to the owned or shared state. The interesting point is here that if CPUX manages to write back obfuscated value in place of plain value, then the cache controller of can synchronize this value before scanner detect or store it[12].

If you look at figure 22 again, the time intervals which they are deobfuscated and plain showed in the boxes. These intervals are the vulnerable moments for leakage. However, they are ordered vertically, so they can not be fetched in the some time from another cache. The leaked boxes are showed also in the figure. They are the cache block whose line is synchronized, when they are deobfuscated. For example; the first red block on the second line is sent to $cacheA$ at 1400 cycle time, and it was using from 1020 till 1500 cycle time. Therefore it is leaked. However, there are just 8 boxes over 100 boxes overall.

Overall Obfuscation Rate Calculation

In order to calculate overall obfuscation rate, we can use formula below.

$$\frac{totalblock - leakedblock}{totalblock} \tag{5.8}$$

However, it gives you an quantitative rate which does not represent whether it is evasion or not. Evasion is more depended on qualitative approaches rather than the percent age of obfuscated block because the signature which we uses for detection is generally some strings such as IP addresses or domain names. Of course, the instruction structure is important especially with control flow graph detection, still majority is string searching. The leakage from these strings could be fatal for evasion, even though their size are relatively smaller. Nevertheless, it can imply a general perception about how successful obfuscation we have. It is not going to be same for even particular system and latency, but the mean of many attempts is reliable. So lets formulate it for better accuracy like below:

$$\frac{\sum_1^n \frac{totalblock - leaked}{totalblock}}{n} \tag{5.9}$$

In order to increase obfuscation complexity, the exotic kind of settlement structure can be proposed as vertical settlement have been proposed e.g. crosswise, curved so forth.

[11] the experiments we made with Booksim 2.0 showed this values as nominal network latency for the topology we designed

[12] this synchronization could be quite fast with MOESI protocol because $cacheX$ probably allocated it with owned state

Complexity is not good security practice, but can be useful and necessary for obfuscation. This attack, which we proposed throughout chapter, will absolutely work with some degree of evasion, but the thing which determines its success is latency of interconnetion network and so, their designs and their logical complexity.

5.3 Pitfalls, Limitations and Fallacies

Even though our methods are novel and theoretically possible, we skip many details in this chapter. For example, the deadlock and live locks in interconnection networks are one of the most common problems. In order to break coherency, they could be useful to exploit, however, it is another attack and vast area. However, it is theoretical likely to appear, if there are to node which share same resource in the same time.

We also did not mention so much about coherency protocols in this chapter, but they could be useful or problematic depending on case. For example, while you are running around malware cache, if there is another CPU who try to scan you cache, it cat not simply modify cache. Instead, it will demand right of modify for cache line, then other cache controller invalidate the line and then respond to the cache which want to write. It require a lot of time[13], and it is the thing can be resulted with deadlock. In addition, this latency could be useful to detect a process of detection. Namely, if there is a difference between write latency, it could be signature of coherence request, but it can't be useful for prevention against scanning.

In the example we showed in figure 22, the malware's processing flow spread around the whole cache equivalently, but the most of common processing flow of regular malware are normally comprised with long period of time delay algorithms or watch dog instructions. In these cases, the only thing scanner can obtain is useless instruction which can not be useful to identify malware.

We did not try to calculate process overhead in this chapter, but it is all about contention and race between latency and process overhead. The reason why we did not is that In modern computer architecture, it is quite difficult to calculate it because of the pipelined and out-of-ordered processors. They can process many instruction in the same time concurrently and simultaneously. We should be aware of this, when we design malware. Arranging different kind of instructions sequentially makes it more faster and concurrent, even if we have one core. You can see more details of this architecture in figure 23[14].

Memory management and protection units are also the subjects which we never mention in this chapter, yet they are making this attack a bit more possible because they produce extra traffic and extra logical complexity and worst of all, they increase traffic exponentially, however, today, there are interconnector chips for multi processing system which handle memory management itself i.e. Arm CCI 400, 500.

[13] As we mentioned in background studies chapter it is a lot more flexible

[14] Computer architecture: The quantitative approach book is the one of the best resource about them[43]

6 Implementation on Harvard Computer Architecture

In this chapter of the thesis, we proposed a method to solve implementation issues on Harvard architecture or equivalent designs. We explained the Harvard computer architecture internals and connect semantic relation between our cache oriented obfuscation technique in order to emphasize details. After we had elaborated the issue with Harvard model, we proposed a theoretical novel solution. Then, we showed several implementation techniques with FORTH programming language. Lastly, we discussed the pitfalls and fallacies in the final section.

6.1 The Issue

Harvard model is a one of the most famous computer architecture model which contrast with other models with its memory pathway implementation. Basically, there are two acknowledged computer architecture model which are Harvard and Von Neumann architectures. Von Neumann model is an of the basic definition of computer architecture which is first designed in 1945 by Von Neumann[54]. He describe one of the earliest electronic and digital computation machine and divide processing units in several subdivision consisting of an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms. Harvard architecture is an evolved version of Von Neumann Model rather than being complete new approach. Harvard architecture first appeared with "Harvard Mark I" relay based computer, because it was a primitive modern computer which stored instructions on punched tape (24 bits wide) and data in electro-mechanical counters, whereas it's usage purpose is totally different.

At first glance, having separated memories could sound absurd i.e. Flexibility of one unified memory increases the programmer's ability, but if you look at deeper, there are many reasons to use separated memories. They are increasing performance, especially on pipelined processors, provide wider bandwidth and natural routing mechanisms to spread bandwidth equally, and reduce power consumptions. However, there are also middle ways between these two architectures. If Harvard model implemented on cache layer, and unified memory is preferred as a main memory, it is called Modified Harvard architecture. Most modern computers that are documented as Harvard architecture are, in fact, Modified Harvard architecture.

Figure 23 is one of the best diagram to show Harvard architecture's ability. It is a block diagram of ARM Cortex A15 processor. A15 is one of the implementation of pipelined, out of order, Modified Harvard architecture[5]. As seen in the figure, they are not just two different cache memories, but also they are implement functionally different. Beside, they can also have different characteristics: block lines, sizes, policies, etc[50], because they stores different data characteristic. It is really like to use next instruction in instruction cache, but temporal locality can be more useful for data caches. Another important thing is that their physical places are optimised to response faster. It is really common for

instruction fetching and load/store units to be placed opposite directions, because of the fact that they are the last and first elements of CPU unit diagram.

In addition to all this logical arguments, there is a game changer reason to implement it. On the pipelined processors, every unit such as instruction fetch, load/store working concurrently, in deed in parallel. Rather then processing each instruction sequentially, it divides them steps for the units[43]. While the instruction fetching unit is processing for instruction x, the load/store unit can work on instruction $x - n$ concurrently. It means our new bottle neck is cache memory, because they both uses memory: one for fetching instruction, another one for fetching or storing data, and they are obviously independent region on memory can work in parallel. Harvard architecture leaps their performance remarkably.

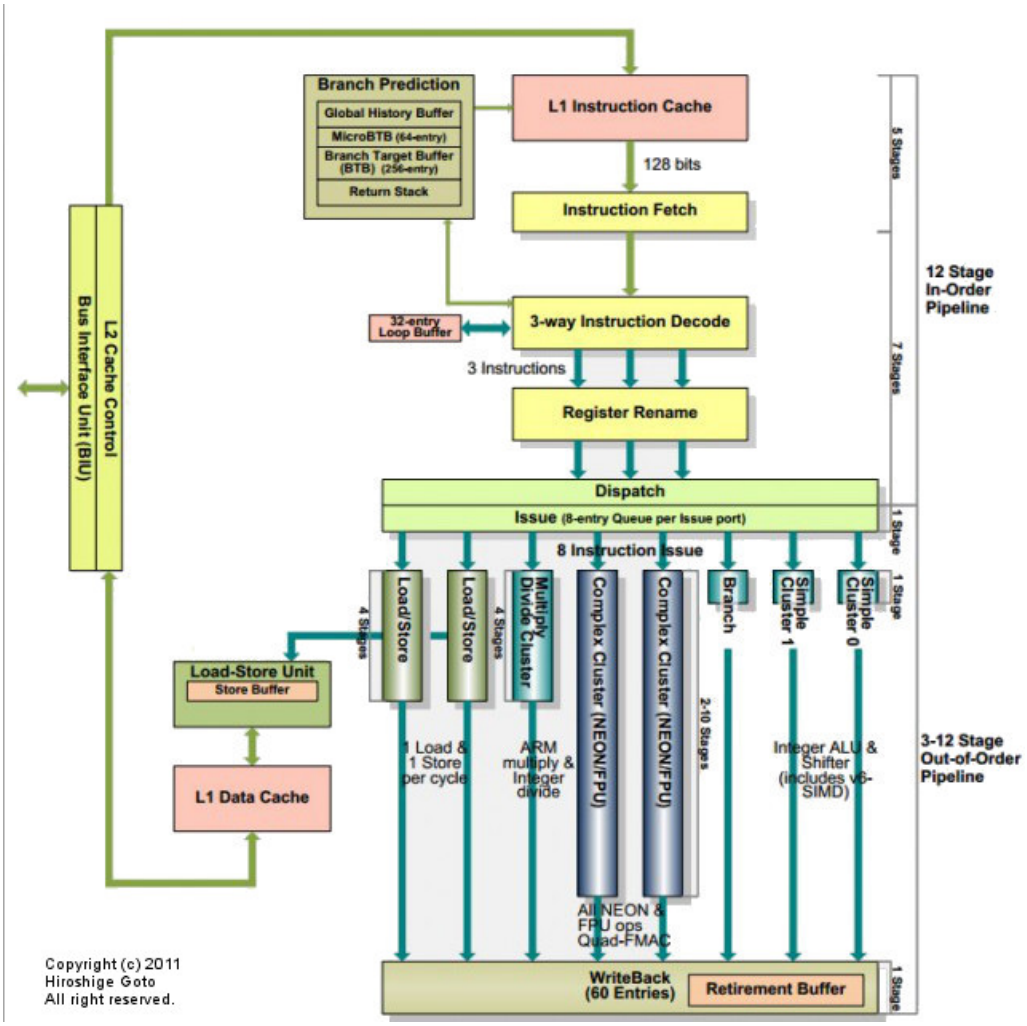

Figure 23: Cortex A15 Block Diagram [5]

In our cache oriented obfuscation method, we have assumed the system which we worked on is Von Neumann so far. In our implementation, we showed deobfuscation code routine in listing4.3. The reason why we can not simply implement this model is, it is working on the instruction section which actually stored in instruction cache, but in

Modified Harvard architecture, there is no direct access to write instruction caches. As seen in figure 23, after fetching obfuscated instruction code into data cache and deobfuscation and storing it back to the cache, it does not move directly to instruction cache. It has to be evicted back to main memory and be fetched into instruction cache instead. Therefore, if we implement same code on Modified Harvard architecture, we encounter with incoherent cache issue which obstructs deobfuscation of instruction or we cause to leak our plain code to upper layer memory. If there is one more layer cache such as L2 cache, instruction and data cache communicate over those; however, it reduces our attack's stealthiness in any case.

6.2 Solution

In brief and on the whole of mentioned issue in the previous section is the obstacles of modifying instruction cache's values. to sum up our issue, we have two caches which are "dcache" which we can read and write but can't not execute and "icache" which we can execute but can't read[1] and write, thereby we can not deobfuscated and run obfuscated code in the same cache. The question we answered in this section is whether it is possible to redesign a running program's control flow in support of data which it is working on or not. It is only possible, when we design instructions and statements in the program depending on the branches, namely values that branches depending on. In computer science convention, some methods of interpretation is equivalent to our definition and requirement, and it is shoved that the ability of the interpreter can be equal to ability of the machine it is working on[55][56]. Even though it is possible to building our own interpreter, it is not recommended, because it will be embedded in "Icache" which is plain text and possible to identifying and obtaining signatures. Therefore, we propose to use well known interpreter; then, it could be legalized on signature detection phase.

6.2.1 Flying over Interpreter

Instead of running a machine code on the target system, we proposes interpreting byte codes or structured abstraction (and therefore it is not bound to any specific machine code but the interpreter itself) for the Harvard Architecture system. It makes malware architecture independent (On the other hand, it is still cache dependent malware) except stub part. Although Interpretation and compilation are two different methods, they are tightly coupled terms in computer programming. Hence, they are not mutually exclusive on implementation of high level language. Interpreted or compiled languages mean the canonical implementation of the language is a compiler or interpreter based[55].

There are typically several variants of interpreters in a spectrum between compiling and interpreting[55]. Byte code interpreting is a method for parsing code directly from interpreter language to byte code. The byte code is not machine code for a specific architecture, but machine code for the interpreter's virtual machine. The byte code representation of the corresponding code is generally highly compressed and optimized because they use all compilation techniques and backgrounds, but efficiency depending on the interpreter and architecture relationship[55]. The second variant of the interpreters is using an intermediate representation of interpreter's source code. Concrete and abstract syntax tree interpreters are the two of the most known and acknowledged examples. It has more overhead than byte code representation, but it is easier to interpret

[1] we can't read it directly, but we can observe its results and estimate it.

and preform better analysis during runtime because of better structure and relation representation between statements of the code[55]. The one of another variant has been just-in-time compilation. They are not pure interpreters or compilers. These techniques compile byte code or an intermediate representation into the native code at runtime. Therefore, it is not appropriate for our methods to use because it is still compiling and running native code directly.

In figure 24, our approach is illustrated. In this example, there is one shared memory, one "Icache", one "Dcache" and one CPU. It is a part of the whole system, and we assume there are more nodes connected to the same shared memory, but only one of the nodes showed in the figure. The stub section of the Icache inherited from the previous chapters. Basically, it deobfuscates body of the malware which is stored in dcache. As we can load, process and store values to dcache, stub implementation and internals are same. However, when we need to run the deobfuscated code on dcache, we use an interpreter to bypass the obstacle, in contrast to previous proposals. Feeding interpreter from memory has already been supported by most of the interpreters for embedded devices e.g. Python, Forth, Basic interpreters. The only thing they need generally starting address. In addition to previous proposals, after running body of the malware, obfuscation could be handled by interpreter at the end of the gadget, or a tail section must be added, after the interpreter code in Icache.

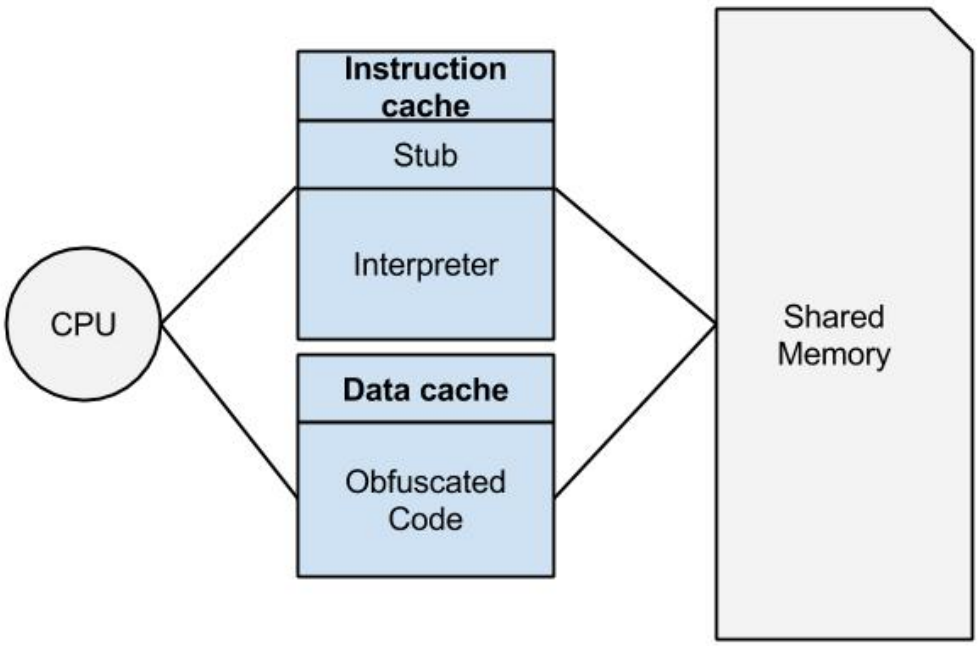

Figure 24: Illustration of Our Approach

6.2.2 Forth Interpreter Language

Forth programmers traditionally pay importance and emphases complete understanding and control over the machine and its architecture. Therefore, first principle of Forth is simplicity and openness, and it is completely documented and clearly easy designed.

Koopman says in 1993 "Type checking, macro preprocessing, common subexpression elimination, and other traditional compiler services are feasible, but usually not included in Forth compilers and this simplicity allows Forth development systems to be small enough to fit in the on-chip ROM of an 8-bit microcontroller. On the other hand, Forth's extensibility allows "full-featured" systems to consume over 100K bytes and provide comprehensive window-based programming environments."[57], and it is more than fit for recent systems. This simplicity is one of the most important reason why we choose it for our implementation, but the most important reason, it gives you full control between compilation and interpretation. Beside, it is also designed for the embedded systems which use ROM memory instead of ICache, but similarly with our approach, it is read only memory, so it is treasury for our approach.

Because there is no explicit Forth parser and no formal grammar, it does not have many concrete standard. There are a few words with ANS Fort which are compiled in assembly, so it is quite simple and small. Moreover, Fort interpreter is very elastic to implement assembly words or remove them, but the standard forth interpreters are better options in term of stealthiness.

There are two level of interpretation in Forth, which are text and address interpreting[57]. While text interpreting accepts keyboards or file inputs, address interpreting accepts memory inputs. Text interpreter mode parse white-space separate character string and determine their role in the process. First, it checks whether extracted string is word or not. if it is a word; then, do interpret word's statements, else it checks whether number or not. if it is number then push it to stack, else return exception. It is main control flow of the Forth text interpretation[58]. On the other hand, address interpretation is a bit more complex. It used to execute Forth words which are actually dictionary structures. They all stitched each other with address pointers, and ultimately they point assembly word which is mentioned above and embedded in Forth interpreter. Therefore, there are a few root words which are written in assembly and children words which point parents and ultimately roots.

6.3 Pitfalls, Limitations and Fallacies

In this chapter, we do not concern the difficulties arise from cache coherency mechanism and we worked on incoherent caches. It is probable to implement "flying over interpreter" method with timing attack approach, but there is much more complexity on implementation phases. However, it theoretically has same principles. During implementation of stepped control flow, there could be more overhead related to interpretation. When we need to calculate interpretation latency, we should consider algorithmic overheads. The code which is processed by interpreters frequently uses threaded code. Threaded code refers to implementation of call routine and their consistence. We should also consider the latency which is arising out of this threaded code overhead, and prefer proper algorithms to decide interpreter.

Throughout this chapter, we never modified the values in instruction chapter, which is the head of our gadget and involve with an interpreter and stub parts. In order to prevent misunderstanding, it is better to underline that we can modify the values in the instruction cache, but if we move values from the data cache to instruction cache, it will lose its stealthiness, but not vice verse. Therefore, we can move instruction cache values in the data cache and modify them freely. It gives us an ability to regenerate stub part

again and again.

Abstract syntax tree method as an intermediate interpretation language is pretty compact and compressed representation of byte codes[59], but produce more overhead[60], so it is better to use it systems without cache coherency. Also, code density can vary from compilation method to method. Because we have limited space, it is better to fit as much as code in minimum space. Due to the division of data and instruction caches, Harvard systems have half sized cache for data.

7 Conclusion and Further Works

This study is set out to explore the concept of code obfuscation through cache memory architecture features. The reason and motivation of our studies is that the increase in deployment of private local memories like NUMA and hierarchical caches is accelerated by the development of modern computer architecture models, which is especially used by the mobile systems, in order to enhance performance and efficiency, but also decrease the power consumption. Because of possible opportunities and also vulnerabilities, we have focused on these architecture. We have considered through whole study that the usage of the private caches and memories like NUMA can be vulnerable and be exploited to evade an observer in the interconnection network of tightly coupled multiprocessor systems. Therefore, we have proposed three different solution for our tree excellent research question.

Firstly, we have designed an obfuscation method, which exploits the private caches to conceal information from the observer devices or CPUs for tightly coupled and multiprocessor systems with write-back cache policy. The main essence of our method is exploiting the laziness of cache memories, whose laziness is actually arisen with performance optimization. At the end of the study, we gave a theoretical product which can obfuscate deobfuscate code, works within cache boundaries, and take cache eviction and replacement into account. We have called it "Cache Oriented Obfuscation" for. However, this attack has been designed only for the concerned theoretical system which has the tightly coupled, multiprocessor and Von Neumann architecture with write back cache policy and without cache coherency.

For the second question, we have proposed a probabilistic attack to the systems which we concerned in the previous question and with snoopy coherent cache. The attack we have proposed involves exploitation of cache fetching vector direction, the coherency latency between caches and the laziness of cache with write-back policy. We have support ourself with latency simulation experiments. We have theoretically showed how possible it is to exploit these systems to obfuscate and hide malware. In order to calculate the overall obfuscation rate, we have given a formula; however, it gave a quantitative rate, which might not represent evasion possibility.

For the last question, we have introduced a method to solve implementation issues on Harvard architecture or equivalent designs. The solution which we have proposed to combine our attack with interpretation the information, which is stored in the data cache, from the code which is located in the executable cache. In other words, we used a interpreter as a virtual machine over the instruction cache in order to execute our gadgets. Instead of writing our own interpreter, the use of legitimate and known interpreters have been considered more suitable because they cannot be valid signature to be detectable. FORTH is one of the most adequate interpreter languages for our implementation and we have discussed implementation issues with FORTH. Thus, this question has been solved with novel approach as well as previous question.

Last but not least,The reason why security analysts and researchers must be stay tuned with these vulnerabilities has been proved in this thesis with elaborated prepared designs

and experiments. We have made a significant progress in documenting and analyzing the theoretical foundation of obfuscation methods with support of cache memories as also possible as other private local memories like non uniform architecture. On the whole, we have proposed three theoretical attack which actually need to be blend during the implementation in the same attack, and then, it turned really strong obfuscation technique against today's detection mechanisms. It is strongly probable that the concerned modern architectures will be more and more popular day by day due to their advantages over performance and efficiency. Without a doubt, these features that are exploited by us is going to be also exploited by malicious authors, although there have been no sign whether they are abused or not yet.

The Feature Works

Because it is first and only one contribution of the cache based obfuscation techniques, the future works could be many and broad. However, to sum up, we will present several opinion in this section. The whole studies could be easily interpreted and documented with the non-uniform memory architecture, and we strongly believe it might efficient to show some NUMA specific features and limitations.

Most importantly, we should research the possibilities to detect this kind of attacks autonomously. These studies will be complimentary to this thesis because the one of the main reasons of the thesis is to develop the proper defense mechanisms against them. Our opinion is the behavioral analysis on a interconnection network could be helpful. We also strongly believe some particular dynamic analysis methods could be efficient, as well as machine learning based deobfuscation methods on the static side of defense.

The implementation issues with out of order CPU does not concerned in this thesis, but we could encounter several issues with Out of ordering. There are commonly four types of out of them which are I3O, 2I2O, I3O, IOIO. They could affect our control flow designs and they must be analyzed in standalone research.

The consistency models are a bit mentioned in background studies, but also some models can give coherency between caches. The effect of all consistency models over the cache oriented obfuscation must be analyzed and documented. There could be several exploration opportunities or tune up for implementation with these architectures.

For the cache coherency network, we could implement noise generator to increase latency, but the important point here is that they could be detectable one. However, we have already illustrated how the latency is important. If we could produce latency with another obfuscated malware or gadget from another node, so the noise generator is really interesting and powerful plug-in for our cache coherency attack in order to increase the attack probability.

Also, as we have mentioned before, Booksim 2.0 simulation tool can be improved with particular features to measure more specialized latency for our experiment. The traffic in our interconnection network is quite characteristic because the observer node is the one which produce regular scanning noise which is quite high, malicious node is the one quite silent and other nodes produce nominal noise. Booksim v2.0 gives you ability to program your own plug-ins to generate your own traffic. This could give better understanding and more valid results in term of latency.

One of the most important usage of the code obfuscation in term of security is to provide an enduring platform for software against crackers. Software developers use

same principles with malicious authors to protect their authentication, validation or critical scoring values. We strongly believe that our cache oriented obfuscation can be a good candidate to protect their software frameworks, too.

For next stage of our advance attack, concurrency between gadgets could be implemented. Because of the undeterministic features of parallel computing, the parallel working gadgets could raise the bar one more level against the behavioural identification.

Bibliography

[1] Turner, D., Fossi, M., Johnson, E., Mack, T., Blackbird, J., Entwisle, S., Low, M. K., McKinney, D., & Wueest, C. 2008. Symantec global internet security threat report–trends for july-december 07. *Symantec Enterprise Security*, 13, 1–36.

[2] Shevchenko, A. 2007. The evolution of technologies used to detect malicious code.

[3] HPC. 2013. High performance computing valued opinion.

[4] Dally, W. J. & Towles, B. P. 2004. *Principles and Practices of Interconnection Networks (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann.

[5] Sloss, A., Symes, D., & Wright, C. 2004. *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann.

[6] Moser, A., Kruegel, C., & Kirda, E. 2007. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 421–430. IEEE.

[7] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2), 6.

[8] Cavallaro, L., Saxena, P., & Sekar, R. 2008. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 143–163. Springer.

[9] Sparks, S. & Butler, J. 2005. Shadow walker": Raising the bar for rootkit detection. *Black Hat Japan*, 504–533.

[10] Yan, W., Zhang, Z., & Ansari, N. 2008. Revealing packed malware. *Security & Privacy, IEEE*, 6(5), 65–69.

[11] Marpaung, J. A., Sain, M., & Lee, H.-J. 2012. Survey on malware evasion techniques: state of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, 744–749. IEEE.

[12] Balakrishnan, A. & Schulze, C. 2005. Code obfuscation literature survey. *CS701 Construction of Compilers*, 19.

[13] Nachenberg, C. 1996. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers*, 30, 16.

[14] You, I. & Yim, K. 2010. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, 297–300. IEEE.

[15] Team, I. S. Bypassing anti-virus scanners. *Packet Storm Security*.

[16] Li, X., Loh, P. K., & Tan, F. 2011. Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, 149–154. IEEE.

[17] anonymous. Polymorphic generators. In *VxHeaven*.

[18] Ferrie, P. 2008. Anti-unpacker tricks. In *Amsterdam: CARO Workshop*.

[19] Konstantinou, E. & Wolthusen, S. 2008. Metamorphic virus: Analysis and detection. *Retrieved on February*, 22, 2011.

[20] Rutkowska, J. 2006. Rootkits vs. stealth by design malware. *Black Hat, Europe*.

[21] Designer, S. 1997. Getting around non-executable stack (and fix).

[22] Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 552–561. ACM.

[23] Roemer, R., Buchanan, E., Shacham, H., & Savage, S. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2.

[24] Mohan, V. & Hamlen, K. W. 2012. Frankenstein: Stitching malware from benign binaries. In *WOOT*, 77–84.

[25] Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, 177–186. IEEE.

[26] Franklin, J., Luk, M., McCune, J. M., Seshadri, A., Perrig, A., & Van Doorn, L. 2008. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review*, 42(3), 83–92.

[27] Chris, M. 2008. What is rootkit and how to detect and protect from rootkits.

[28] Ducklin, P. 1991. Tequila.

[29] Abdalla, A. 2011. Rootkits classification and their countermeasures.

[30] Reddy, D. K. S. & Pujari, A. K. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2(3), 231–239.

[31] Abou-Assaleh, T., Cercone, N., Keselj, V., & Sweidan, R. 2004. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, 41–42. IEEE.

[32] Abou-Assaleh, T., Cercone, N., Keselj, V., & Sweidan, R. 2004. Detection of new malicious code using n-grams signatures. In *PST*, 193–196.

[33] Kendall, K. & McMillan, C. 2007. Practical malware analysis. In *Black Hat Conference, USA*.

[34] Saxena, P., Sekar, R., & Puranik, V. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 74–83. ACM.

[35] Smith, G. 2007. Principles of secure information flow analysis. In *Malware Detection*, 291–307. Springer.

[36] Lee, J., Jeong, K., & Lee, H. 2010. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM symposium on applied computing*, 1970–1977. ACM.

[37] Christodorescu, M. & Jha, S. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.

[38] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., & Bryant, R. E. 2005. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, 32–46. IEEE.

[39] Hennessy, J. L. & Patterson, D. A. 2012. *Computer architecture: a quantitative approach*. Elsevier.

[40] Shannon, C. E. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1), 3–55.

[41] Denning, P. J. 2005. The locality principle. *Communications of the ACM*, 48(7), 19–24.

[42] Kilburn, T., Edwards, D. B., Lanigan, M., & Sumner, F. H. 1962. One-level storage system. *Electronic Computers, IRE Transactions on*, (2), 223–235.

[43] Wentzlaff, D. 2013. Computer architecture.

[44] Von Neumann, J. 1961. Collected works. *Oxford: Pergamon, 1961, edited by Taub, AH*, 1.

[45] wikipedia. 2014. Cache(computing).

[46] Sorin, D. J., Hill, M. D., & Wood, D. A. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 1–212.

[47] Papamarcos, M. S. & Patel, J. H. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. 12(3), 348–354.

[48] Development, A. & Department, R. 2013. Corelink cci-400 cache coherent interconnect.

[49] Tsopokis, C. V. 2013. Concurrent memory inspection for intrusion detection.

[50] Handy, J. 2007. *the Cache Memory Book, second edition*. Academic Press.

[51] Ramilli, M., Bishop, M., & Sun, S. 2011. Multiprocess malware. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, 8–13. IEEE.

[52] Rubini, A. & Corbet, J. 2001. *"Memory Mapping and DMA." Linux Device Drivers. Sebastopol:*. O'Reilly & Associates.

[53] Jiang, N., Becker, D. U., Michelogiannakis, G., Balfour, J., Towles, B., Shaw, D., Kim, J., & Dally, W. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 86–96. IEEE.

[54] Von Neumann, J. 1993. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4), 27–75.

[55] Mak, R. 2011. *Writing compilers and interpreters: a software engineering approach*. John Wiley & Sons.

[56] Abelson, H. 1996. *Structure and interpretation of computer programs*. Paul Muljadi.

[57] Koopman, P. 1993. A brief introduction to forth.

[58] Pelc, S. 2011. Programming forth. *Microprocessor Engineering Limited*.

[59] Kistler, T. & Franz, M. 1999. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1), 21–33.

[60] Garen, G. 2008. Announcing squirrelfish. *Surfin'Safari*.

A Cache Memory Simulation

```
__author__ = 'caglar'
import random


class Memory(object):
    def __init__(self):
        self.memory = {}

    def __getitem__(self, item):
        try:
            return self.memory[item]
        except KeyError:
            return 0

    def __setitem__(self, key, value):
        self.memory[key] = value


class Cache(object):
    def __init__(self, size=32768, block_size=64, sets=2):
        self.size = size
        self.block_size = block_size
        self.sets = sets
        self.addr_size = 32
        self.cache = list()
        self.line = size / (sets * block_size)
        self.block_addr_len =\
         self.__calculate_addr_len(self.block_size)
        self.line_addr_len = \
        self.__calculate_addr_len(self.line)
        self.tag_addr_len = self.addr_size -\
         self.block_addr_len - self.line_addr_len
        self.super = None
        self.__build()

    def __repr__(self):
        return str(self.cache.__len__())

    def __getitem__(self, address):
        query = self.toquery(address)
        return self.read_request(query)

    def __setitem__(self, address, value):
        self.write_request(address, value)

    def setsuper(self,super):
        self.super = super
```

```
def __build(self):
    for i in range(self.line):
        self.cache.append(CacheLine(i, self.block_size, self.sets))

def __calculate_addr_len(self, size):
    i = 0
    result = 1
    while size > result:
        result <<= 1
        i += 1
    return i

def toquery(self, address):
    mask = (1 << 32) - 1
    query = {"address": address,
             "tag": address >> (self.addr_size - self.tag_addr_len),
             "index": (address >> (self.block_addr_len)) <<\
              ((self.addr_size - self.line_addr_len) & mask) >> (
                 self.addr_size - self.line_addr_len),
             "block": ((address << (self.addr_size - self.block_addr_len))\
              & mask) >> (
                 self.addr_size - self.block_addr_len)}
    return query

def isvalid(self, query):
    for i in range(self.cache[query["index"]].sets):
        if (self.cache[query["index"]].lines[i]["tag"] is query["tag"]) and\
         (self.cache[query["index"]].lines[i]['valid'] is True):
            self.cache[query["index"]].used = i
            return self.cache[query["index"]].lines[i]
    return False

def read_request(self, query):
    line = self.isvalid(query)
    if line:
        return self.__read_hit(line, query)
    else:
        return self.__read_miss(query)

def __read_hit(self, line, query):
    return line["data"][query["block"]]

def __read_miss(self, query):
    self.request_line(query) #proof it later.
    return self.read_request(query)


def write_request(self, address, value):
    query = self.toquery(address)
    line = self.isvalid(query)
    if line is not -1:
        return self.__write_hit(line, query, value)
```

```
        else:
            return self.__write_miss(query)

    def __write_hit(self, line, query, value):
        line["data"][query["block"]] = value
        line["dirty"] = True

    def __write_miss(self, query):
        pass

    def request_line(self, query):
        line = query["address"] - query["block"]
        rline = {"tag": query["tag"], "data": bytearray(self.block_size)}
        for i in range(self.block_size):
            rline["data"][i] = self.super[line+i]
        self.cache[query["index"]].fill_line(rline)


class CacheLine(object):
    #CacheLine is a Class for defining each cache blocks in cache hierarchy
    #During initialization it build a static block of given
    #sets sized array with given block size as byte
    #It has only one method to fill remote block in.
    def __init__(self, index, size, sets):
        self.size = size
        self.sets = sets
        self.index = index
        self.lines = list()
        self.__build()
        self.used = 0

    def __repr__(self):
        return repr(self.lines)

    def __build(self):
        #Initialize the first view of the cache.
        #It will invoke build_line() for each line of the cache.
        for i in range(self.sets):
            self.lines.append(self.__build_line())

    def __build_line(self):
        #Build a line with default variables.
        #data is a byte array which is given length with size
        data = bytearray(self.size)
        return {"tag": 0, "valid": False, "dirty":\
         False, "used": False, "data": data}

    def mark_used(self, line):
        self.used = line

    def give_replaceable(self):
        if self.sets == 1:
            return 0
```

```
        else:
            line = random.randint(0, self.sets - 1)
            if self.used != line:
                return line
            else:
                return (line + 1) % self.sets

    def fill_line(self, rline):
        # It fills the corresponding block from the upper layer into the line .
        # Input "rline" is abbreviation of remote line formed as {tag, data}
        # As default, it uses Not recently used, random replacement Policy
        i = self.give_replaceable()
        self.lines[i]["tag"] = rline["tag"]
        self.lines[i]["valid"] = True
        self.lines[i]["dirty"] = False
        self.lines[i]["data"] = rline["data"]
```

B Real Systems Cache Coherency Latency Simulation Results

B.1 Small Topology

B.1.1 Small Topology Simulation with Two Percent Injection Rate Configuration File and Results

```
topology = anynet;


routing_function = min;
network_file = anynet_file_small;
traffic = uniform;

use_read_write = 0;

sample_period = 10000;
injection_rate = 0.02;
credit_delay = 1;
routing_delay = 2;

vc_allocator = separable_input_first;
sw_allocator = separable_input_first;
alloc_iters  = 1;

traffic = uniform;

num_vcs     = 1;
vc_buf_size = 3;

END Configuration File: ./anynet_config
OVERRIDE Parameter: latency_thres=500000.0
========================Network File Parsed================
******************node listing**********************
Node 0  Router 0
Node 1  Router 1
Node 2  Router 3

****************router to node listing*************
Router 0
         Node 0 lat 1
Router 1
         Node 1 lat 1
Router 2
Router 3
         Node 2 lat 1

*****************router to router listing************
Router 0
```

```
        Router 2 lat 40
Router 1
        Router 2 lat 40
Router 2
        Router 0 lat 40
        Router 1 lat 40
        Router 3 lat 40
Router 3
        Router 2 lat 40
==========================Node to Router ====================
router 0 radix 2
        connected to node 0 at outport 0 lat 1
router 1 radix 2
        connected to node 1 at outport 0 lat 1
router 2 radix 3
router 3 radix 2
        connected to node 2 at outport 0 lat 1
==========================Router to Router ====================
router 0
        connected to router 2 using link 0 at outport 1 lat 40
router 1
        connected to router 2 using link 1 at outport 1 lat 40
router 2
        connected to router 0 using link 2 at outport 0 lat 40
        connected to router 1 using link 3 at outport 1 lat 40
        connected to router 3 using link 4 at outport 2 lat 40
router 3
        connected to router 2 using link 5 at outport 1 lat 40
========================== Routing table  ====================
Class 0:
Packet latency average = 72.9545
        minimum = 8
        maximum = 210
Network latency average = 72.3687
        minimum = 8
        maximum = 201
Slowest packet = 12
Flit latency average = 72.3687
        minimum = 8
        maximum = 201
Slowest flit = 164
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0198667
        minimum = 0.0189 (at node 1)
        maximum = 0.0204 (at node 2)
Accepted packet rate average = 0.0198
        minimum = 0.0194 (at node 0)
        maximum = 0.02 (at node 1)
Injected flit rate average = 0.0198667
        minimum = 0.0189 (at node 1)
        maximum = 0.0204 (at node 2)
```

```
Accepted flit rate average= 0.0198
        minimum = 0.0194 (at node 0)
        maximum = 0.02 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 2 (0 measured)
latency change    = 1
throughput change = 1
Class 0:
Packet latency average = 73.9654
        minimum = 8
        maximum = 214
Network latency average = 73.6232
        minimum = 8
        maximum = 214
Slowest packet = 998
Flit latency average = 73.6232
        minimum = 8
        maximum = 214
Slowest flit = 998
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0202833
        minimum = 0.01975 (at node 0)
        maximum = 0.0212 (at node 2)
Accepted packet rate average = 0.0202167
        minimum = 0.0197 (at node 0)
        maximum = 0.02075 (at node 1)
Injected flit rate average = 0.0202833
        minimum = 0.01975 (at node 0)
        maximum = 0.0212 (at node 2)
Accepted flit rate average= 0.0202167
        minimum = 0.0197 (at node 0)
        maximum = 0.02075 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 4 (0 measured)
latency change    = 0.0136663
throughput change = 0.0206101
Class 0:
Packet latency average = 74.9832
        minimum = 8
        maximum = 203
Network latency average = 74.9446
        minimum = 8
        maximum = 203
Slowest packet = 1378
Flit latency average = 74.9446
        minimum = 8
        maximum = 203
Slowest flit = 1378
Fragmentation average = 0
```

```
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0198667
        minimum = 0.0189 (at node 0)
        maximum = 0.0211 (at node 1)
Accepted packet rate average = 0.0198667
        minimum = 0.0198 (at node 0)
        maximum = 0.0199 (at node 1)
Injected flit rate average = 0.0198667
        minimum = 0.0189 (at node 0)
        maximum = 0.0211 (at node 1)
Accepted flit rate average= 0.0198667
        minimum = 0.0198 (at node 0)
        maximum = 0.0199 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 4 (0 measured)
latency change    = 0.0135743
throughput change = 0.0176174
Warmed up ...Time used is 30000 cycles
Class 0:
Packet latency average = 69.2534
        minimum = 8
        maximum = 202
Network latency average = 69.1351
        minimum = 8
        maximum = 174
Slowest packet = 2114
Flit latency average = 69.3289
        minimum = 8
        maximum = 174
Slowest flit = 2114
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0198
        minimum = 0.0182 (at node 1)
        maximum = 0.0221 (at node 0)
Accepted packet rate average = 0.0198667
        minimum = 0.0177 (at node 0)
        maximum = 0.0218 (at node 2)
Injected flit rate average = 0.0198
        minimum = 0.0182 (at node 1)
        maximum = 0.0221 (at node 0)
Accepted flit rate average= 0.0198667
        minimum = 0.0177 (at node 0)
        maximum = 0.0218 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 2 (2 measured)
latency change    = 0.0827374
throughput change = 0
Class 0:
```

```
Packet latency average = 72.145
        minimum = 8
        maximum = 202
Network latency average = 72.0387
        minimum = 8
        maximum = 183
Slowest packet = 2114
Flit latency average = 72.124
        minimum = 8
        maximum = 183
Slowest flit = 2425
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.02025
        minimum = 0.01885 (at node 2)
        maximum = 0.02165 (at node 0)
Accepted packet rate average = 0.0203
        minimum = 0.01905 (at node 0)
        maximum = 0.02095 (at node 2)
Injected flit rate average = 0.02025
        minimum = 0.01885 (at node 2)
        maximum = 0.02165 (at node 0)
Accepted flit rate average= 0.0203
        minimum = 0.01905 (at node 0)
        maximum = 0.02095 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 1 (1 measured)
latency change    = 0.0400804
throughput change = 0.0213465
Class 0:
Packet latency average = 71.4124
        minimum = 8
        maximum = 216
Network latency average = 71.3402
        minimum = 8
        maximum = 216
Slowest packet = 3080
Flit latency average = 71.3998
        minimum = 8
        maximum = 216
Slowest flit = 3080
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0198889
        minimum = 0.0191667 (at node 2)
        maximum = 0.0207667 (at node 0)
Accepted packet rate average = 0.0199
        minimum = 0.0186 (at node 0)
        maximum = 0.0208667 (at node 1)
Injected flit rate average = 0.0198889
```

```
        minimum = 0.0191667 (at node 2)
        maximum = 0.0207667 (at node 0)
Accepted flit rate average= 0.0199
        minimum = 0.0186 (at node 0)
        maximum = 0.0208667 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 3 (3 measured)
latency change    = 0.0102581
throughput change = 0.0201005
Class 0:
Packet latency average = 71.295
        minimum = 8
        maximum = 216
Network latency average = 71.2135
        minimum = 8
        maximum = 216
Slowest packet = 3080
Flit latency average = 71.2577
        minimum = 8
        maximum = 216
Slowest flit = 3080
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0202
        minimum = 0.0193 (at node 2)
        maximum = 0.021325 (at node 0)
Accepted packet rate average = 0.020175
        minimum = 0.019725 (at node 0)
        maximum = 0.0208 (at node 2)
Injected flit rate average = 0.0202
        minimum = 0.0193 (at node 2)
        maximum = 0.021325 (at node 0)
Accepted flit rate average= 0.020175
        minimum = 0.019725 (at node 0)
        maximum = 0.0208 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 7 (7 measured)
latency change    = 0.00164709
throughput change = 0.0136307
Draining all recorded packets ...
Draining remaining packets ...
Time taken is 70306 cycles
====== Overall Traffic Statistics ======
====== Traffic class 0 ======
Packet latency average = 71.4146 (1 samples)
        minimum = 8 (1 samples)
        maximum = 216 (1 samples)
Network latency average = 71.3333 (1 samples)
        minimum = 8 (1 samples)
        maximum = 216 (1 samples)
```

```
Flit latency average = 71.4756 (1 samples)
        minimum = 8 (1 samples)
        maximum = 216 (1 samples)
Fragmentation average = 0 (1 samples)
        minimum = 0 (1 samples)
        maximum = 0 (1 samples)
Injected packet rate average = 0.0202 (1 samples)
        minimum = 0.0193 (1 samples)
        maximum = 0.021325 (1 samples)
Accepted packet rate average = 0.020175 (1 samples)
        minimum = 0.019725 (1 samples)
        maximum = 0.0208 (1 samples)
Injected flit rate average = 0.0202 (1 samples)
        minimum = 0.0193 (1 samples)
        maximum = 0.021325 (1 samples)
Accepted flit rate average = 0.020175 (1 samples)
        minimum = 0.019725 (1 samples)
        maximum = 0.0208 (1 samples)
Injected packet size average = 1 (1 samples)
Accepted packet size average = 1 (1 samples)
Hops average = 2.30611 (1 samples)
Total run time 0.128789
```

B.1.2 Small Topology Simulation with Four Percent Injection Rate Configuration File and Results

```
topology = anynet;


routing_function = min;
network_file = anynet_file_small;
traffic = uniform;

use_read_write = 0;

sample_period = 10000;
injection_rate = 0.04;
credit_delay = 1;
routing_delay = 2;

vc_allocator = separable_input_first;
sw_allocator = separable_input_first;
alloc_iters  = 1;

traffic = uniform;

num_vcs     = 1;
vc_buf_size = 3;

END Configuration File: ./anynet_config
OVERRIDE Parameter: latency_thres=500000.0
========================Network File Parsed================
********************node listing**********************
Node 0  Router 0
```

```
Node 1  Router 1
Node 2  Router 3

**********router to node listing**********
Router 0
        Node 0 lat 1
Router 1
        Node 1 lat 1
Router 2
Router 3
        Node 2 lat 1

**********router to router listing**********
Router 0
        Router 2 lat 40
Router 1
        Router 2 lat 40
Router 2
        Router 0 lat 40
        Router 1 lat 40
        Router 3 lat 40
Router 3
        Router 2 lat 40
==========Node to Router ==========
router 0 radix 2
        connected to node 0 at outport 0 lat 1
router 1 radix 2
        connected to node 1 at outport 0 lat 1
router 2 radix 3
router 3 radix 2
        connected to node 2 at outport 0 lat 1
==========Router to Router ==========
router 0
        connected to router 2 using link 0 at outport 1 lat 40
router 1
        connected to router 2 using link 1 at outport 1 lat 40
router 2
        connected to router 0 using link 2 at outport 0 lat 40
        connected to router 1 using link 3 at outport 1 lat 40
        connected to router 3 using link 4 at outport 2 lat 40
router 3
        connected to router 2 using link 5 at outport 1 lat 40
========== Routing table  ==========
Class 0:
Packet latency average = 308.783
        minimum = 8
        maximum = 710
Network latency average = 132.524
        minimum = 8
        maximum = 332
Slowest packet = 20
Flit latency average = 132.524
        minimum = 8
```

```
        maximum = 332
Slowest flit = 251
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0395667
        minimum = 0.037 (at node 1)
        maximum = 0.0419 (at node 0)
Accepted packet rate average = 0.0389333
        minimum = 0.038 (at node 0)
        maximum = 0.0394 (at node 1)
Injected flit rate average = 0.0395667
        minimum = 0.037 (at node 1)
        maximum = 0.0419 (at node 0)
Accepted flit rate average= 0.0389333
        minimum = 0.038 (at node 0)
        maximum = 0.0394 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 22 (0 measured)
latency change    = 1
throughput change = 1
Class 0:
Packet latency average = 306.502
        minimum = 8
        maximum = 753
Network latency average = 133.564
        minimum = 8
        maximum = 343
Slowest packet = 20
Flit latency average = 133.564
        minimum = 8
        maximum = 343
Slowest flit = 1535
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0401333
        minimum = 0.03815 (at node 1)
        maximum = 0.0415 (at node 0)
Accepted packet rate average = 0.0399833
        minimum = 0.0384 (at node 0)
        maximum = 0.04085 (at node 2)
Injected flit rate average = 0.0401333
        minimum = 0.03815 (at node 1)
        maximum = 0.0415 (at node 0)
Accepted flit rate average= 0.0399833
        minimum = 0.0384 (at node 0)
        maximum = 0.04085 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 10 (0 measured)
latency change    = 0.00744372
```

```
throughput change = 0.0262609
Class 0:
Packet latency average = 188.926
        minimum = 8
        maximum = 985
Network latency average = 112.832
        minimum = 8
        maximum = 352
Slowest packet = 2565
Flit latency average = 112.832
        minimum = 8
        maximum = 352
Slowest flit = 3086
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0384
        minimum = 0.0376 (at node 2)
        maximum = 0.039 (at node 0)
Accepted packet rate average = 0.0382333
        minimum = 0.0365 (at node 0)
        maximum = 0.0403 (at node 1)
Injected flit rate average = 0.0384
        minimum = 0.0376 (at node 2)
        maximum = 0.039 (at node 0)
Accepted flit rate average= 0.0382333
        minimum = 0.0365 (at node 0)
        maximum = 0.0403 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 16 (0 measured)
latency change    = 0.622339
throughput change = 0.0457716
Warmed up ...Time used is 30000 cycles
Class 0:
Packet latency average = 551.94
        minimum = 8
        maximum = 1273
Network latency average = 142.715
        minimum = 8
        maximum = 327
Slowest packet = 3568
Flit latency average = 142.685
        minimum = 8
        maximum = 327
Slowest flit = 3918
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0408333
        minimum = 0.0387 (at node 1)
        maximum = 0.0432 (at node 0)
Accepted packet rate average = 0.0406667
```

```
        minimum = 0.0379 (at node 1)
        maximum = 0.0424 (at node 2)
Injected flit rate average = 0.0408333
        minimum = 0.0387 (at node 1)
        maximum = 0.0432 (at node 0)
Accepted flit rate average= 0.0406667
        minimum = 0.0379 (at node 1)
        maximum = 0.0424 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 22 (22 measured)
latency change    = 0.657706
throughput change = 0.0598361
Class 0:
Packet latency average = 460.257
        minimum = 8
        maximum = 1273
Network latency average = 137.793
        minimum = 8
        maximum = 339
Slowest packet = 3568
Flit latency average = 137.81
        minimum = 8
        maximum = 339
Slowest flit = 5156
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0409
        minimum = 0.03965 (at node 2)
        maximum = 0.04255 (at node 0)
Accepted packet rate average = 0.0408333
        minimum = 0.04035 (at node 1)
        maximum = 0.04125 (at node 0)
Injected flit rate average = 0.0409
        minimum = 0.03965 (at node 2)
        maximum = 0.04255 (at node 0)
Accepted flit rate average= 0.0408333
        minimum = 0.04035 (at node 1)
        maximum = 0.04125 (at node 0)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 21 (21 measured)
latency change    = 0.199201
throughput change = 0.00408163
Class 0:
Packet latency average = 367.683
        minimum = 8
        maximum = 1273
Network latency average = 131.212
        minimum = 8
        maximum = 339
Slowest packet = 3568
```

```
Flit latency average = 131.253
        minimum = 8
        maximum = 339
Slowest flit = 5156
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0401444
        minimum = 0.0388 (at node 2)
        maximum = 0.0414333 (at node 0)
Accepted packet rate average = 0.0401444
        minimum = 0.0393 (at node 1)
        maximum = 0.0409667 (at node 2)
Injected flit rate average = 0.0401444
        minimum = 0.0388 (at node 2)
        maximum = 0.0414333 (at node 0)
Accepted flit rate average= 0.0401444
        minimum = 0.0393 (at node 1)
        maximum = 0.0409667 (at node 2)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 16 (16 measured)
latency change    = 0.251776
throughput change = 0.0171603
Class 0:
Packet latency average = 329.038
        minimum = 8
        maximum = 1273
Network latency average = 127.882
        minimum = 8
        maximum = 339
Slowest packet = 3568
Flit latency average = 127.923
        minimum = 8
        maximum = 339
Slowest flit = 5156
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.04045
        minimum = 0.03945 (at node 2)
        maximum = 0.041525 (at node 0)
Accepted packet rate average = 0.0404083
        minimum = 0.039525 (at node 1)
        maximum = 0.041075 (at node 0)
Injected flit rate average = 0.04045
        minimum = 0.03945 (at node 2)
        maximum = 0.041525 (at node 0)
Accepted flit rate average= 0.0404083
        minimum = 0.039525 (at node 1)
        maximum = 0.041075 (at node 0)
Injected packet length average = 1
Accepted packet length average = 1
```

```
Total in-flight flits = 22 (22 measured)
latency change    = 0.117447
throughput change = 0.00653056
Class 0:
Packet latency average = 333.177
        minimum = 8
        maximum = 1273
Network latency average = 129.5
        minimum = 8
        maximum = 376
Slowest packet = 3568
Flit latency average = 129.529
        minimum = 8
        maximum = 376
Slowest flit = 8574
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.04046
        minimum = 0.03966 (at node 2)
        maximum = 0.04158 (at node 0)
Accepted packet rate average = 0.0404667
        minimum = 0.04008 (at node 1)
        maximum = 0.04124 (at node 0)
Injected flit rate average = 0.04046
        minimum = 0.03966 (at node 2)
        maximum = 0.04158 (at node 0)
Accepted flit rate average= 0.0404667
        minimum = 0.04008 (at node 1)
        maximum = 0.04124 (at node 0)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 14 (14 measured)
latency change    = 0.0124201
throughput change = 0.00144152
Class 0:
Packet latency average = 327.389
        minimum = 8
        maximum = 1273
Network latency average = 129.176
        minimum = 8
        maximum = 376
Slowest packet = 3568
Flit latency average = 129.2
        minimum = 8
        maximum = 376
Slowest flit = 8574
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0402778
        minimum = 0.0398333 (at node 2)
        maximum = 0.0408833 (at node 0)
```

```
Accepted packet rate average = 0.04025
        minimum = 0.0396667 (at node 2)
        maximum = 0.04085 (at node 0)
Injected flit rate average = 0.0402778
        minimum = 0.0398333 (at node 2)
        maximum = 0.0408833 (at node 0)
Accepted flit rate average= 0.04025
        minimum = 0.0396667 (at node 2)
        maximum = 0.04085 (at node 0)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 20 (20 measured)
latency change    = 0.0176767
throughput change = 0.00538302
Class 0:
Packet latency average = 320.373
        minimum = 8
        maximum = 1273
Network latency average = 128.734
        minimum = 8
        maximum = 376
Slowest packet = 3568
Flit latency average = 128.756
        minimum = 8
        maximum = 376
Slowest flit = 8574
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0401429
        minimum = 0.0398571 (at node 2)
        maximum = 0.0406143 (at node 0)
Accepted packet rate average = 0.0401238
        minimum = 0.0396857 (at node 2)
        maximum = 0.0408857 (at node 0)
Injected flit rate average = 0.0401429
        minimum = 0.0398571 (at node 2)
        maximum = 0.0406143 (at node 0)
Accepted flit rate average= 0.0401238
        minimum = 0.0396857 (at node 2)
        maximum = 0.0408857 (at node 0)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 21 (21 measured)
latency change    = 0.0219011
throughput change = 0.00314503
Draining all recorded packets ...
Draining remaining packets ...
Time taken is 100954 cycles
====== Overall Traffic Statistics ======
====== Traffic class 0 ======
Packet latency average = 321.091 (1 samples)
        minimum = 8 (1 samples)
```

```
        maximum = 1273 (1 samples)
Network latency average = 128.916 (1 samples)
        minimum = 8 (1 samples)
        maximum = 376 (1 samples)
Flit latency average = 129.072 (1 samples)
        minimum = 8 (1 samples)
        maximum = 376 (1 samples)
Fragmentation average = 0 (1 samples)
        minimum = 0 (1 samples)
        maximum = 0 (1 samples)
Injected packet rate average = 0.0401429 (1 samples)
        minimum = 0.0398571 (1 samples)
        maximum = 0.0406143 (1 samples)
Accepted packet rate average = 0.0401238 (1 samples)
        minimum = 0.0396857 (1 samples)
        maximum = 0.0408857 (1 samples)
Injected flit rate average = 0.0401429 (1 samples)
        minimum = 0.0398571 (1 samples)
        maximum = 0.0406143 (1 samples)
Accepted flit rate average = 0.0401238 (1 samples)
        minimum = 0.0396857 (1 samples)
        maximum = 0.0408857 (1 samples)
Injected packet size average = 1 (1 samples)
Accepted packet size average = 1 (1 samples)
Hops average = 2.33081 (1 samples)
Total run time 0.298342
```

B.2 Crowded Topology

B.2.1 Crowded Topology Simulation with Two Percent Injection Rate Configuration File and Results

```
topology = anynet;


routing_function = min;
network_file = anynet_file;
traffic = uniform;

use_read_write = 0;

sample_period = 10000;
injection_rate = 0.02;
credit_delay = 1;
routing_delay = 2;

vc_allocator = separable_input_first;
sw_allocator = separable_input_first;
alloc_iters  = 1;

traffic = uniform;

num_vcs     = 1;
vc_buf_size = 3;
```

```
END Configuration File: ./anynet_config
OVERRIDE Parameter: latency_thres=500000.0
=========================Network File Parsed==================
******************node listing**********************
Node 0  Router 0
Node 1  Router 0
Node 2  Router 0
Node 3  Router 0
Node 4  Router 1
Node 5  Router 1
Node 6  Router 1
Node 7  Router 1
Node 8  Router 2
Node 9  Router 2
Node 10 Router 2
Node 11 Router 3

****************router to node listing*************
Router 0
         Node 0 lat 1
         Node 1 lat 1
         Node 2 lat 1
         Node 3 lat 1
Router 1
         Node 4 lat 1
         Node 5 lat 1
         Node 6 lat 1
         Node 7 lat 1
Router 2
         Node 8 lat 1
         Node 9 lat 1
         Node 10 lat 1
Router 3
         Node 11 lat 1

*****************router to router listing************
Router 0
         Router 2 lat 40
Router 1
         Router 2 lat 40
Router 2
         Router 0 lat 40
         Router 1 lat 40
         Router 3 lat 40
Router 3
         Router 2 lat 40
=========================Node to Router ======================
router 0 radix 5
         connected to node 0 at outport 0 lat 1
         connected to node 1 at outport 1 lat 1
         connected to node 2 at outport 2 lat 1
         connected to node 3 at outport 3 lat 1
```

```
router 1 radix 5
         connected to node 4 at outport 0 lat 1
         connected to node 5 at outport 1 lat 1
         connected to node 6 at outport 2 lat 1
         connected to node 7 at outport 3 lat 1
router 2 radix 6
         connected to node 8 at outport 0 lat 1
         connected to node 9 at outport 1 lat 1
         connected to node 10 at outport 2 lat 1
router 3 radix 2
         connected to node 11 at outport 0 lat 1
============================Router to Router ======================
router 0
         connected to router 2 using link 0 at outport 4 lat 40
router 1
         connected to router 2 using link 1 at outport 4 lat 40
router 2
         connected to router 0 using link 2 at outport 3 lat 40
         connected to router 1 using link 3 at outport 4 lat 40
         connected to router 3 using link 4 at outport 5 lat 40
router 3
         connected to router 2 using link 5 at outport 1 lat 40
============================ Routing table  ======================
Class 0:
Packet latency average = 1657.81
        minimum = 8
        maximum = 6271
Network latency average = 312.254
        minimum = 8
        maximum = 1208
Slowest packet = 93
Flit latency average = 312.254
        minimum = 8
        maximum = 1208
Slowest flit = 806
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.011975
        minimum = 0.0073 (at node 2)
        maximum = 0.0205 (at node 8)
Accepted packet rate average = 0.011625
        minimum = 0.01 (at node 5)
        maximum = 0.0139 (at node 8)
Injected flit rate average = 0.011975
        minimum = 0.0073 (at node 2)
        maximum = 0.0205 (at node 8)
Accepted flit rate average= 0.011625
        minimum = 0.01 (at node 5)
        maximum = 0.0139 (at node 8)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 53 (0 measured)
```

```
latency change    = 1
throughput change = 1
Class 0:
Packet latency average = 3193
        minimum = 8
        maximum = 13321
Network latency average = 334.875
        minimum = 8
        maximum = 1293
Slowest packet = 93
Flit latency average = 334.875
        minimum = 8
        maximum = 1293
Slowest flit = 2291
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0112958
        minimum = 0.00695 (at node 2)
        maximum = 0.019 (at node 8)
Accepted packet rate average = 0.0111042
        minimum = 0.01005 (at node 6)
        maximum = 0.01185 (at node 8)
Injected flit rate average = 0.0112958
        minimum = 0.00695 (at node 2)
        maximum = 0.019 (at node 8)
Accepted flit rate average= 0.0111042
        minimum = 0.01005 (at node 6)
        maximum = 0.01185 (at node 8)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 58 (0 measured)
latency change    = 0.480798
throughput change = 0.0469043
Class 0:
Packet latency average = 8449.87
        minimum = 114
        maximum = 19618
Network latency average = 342.531
        minimum = 13
        maximum = 1135
Slowest packet = 2677
Flit latency average = 342.531
        minimum = 13
        maximum = 1135
Slowest flit = 3713
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0112417
        minimum = 0.0068 (at node 2)
        maximum = 0.0206 (at node 8)
Accepted packet rate average = 0.0112417
```

```
        minimum = 0.0099 (at node 7)
        maximum = 0.0137 (at node 10)
Injected flit rate average = 0.0112417
        minimum = 0.0068 (at node 2)
        maximum = 0.0206 (at node 8)
Accepted flit rate average= 0.0112417
        minimum = 0.0099 (at node 7)
        maximum = 0.0137 (at node 10)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (0 measured)
latency change    = 0.622125
throughput change = 0.0122313
Warmed up ...Time used is 30000 cycles
Class 0:
Packet latency average = 12343.5
        minimum = 164
        maximum = 26760
Network latency average = 349.241
        minimum = 13
        maximum = 1297
Slowest packet = 4074
Flit latency average = 354.648
        minimum = 13
        maximum = 1297
Slowest flit = 4733
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0108667
        minimum = 0.0063 (at node 3)
        maximum = 0.018 (at node 8)
Accepted packet rate average = 0.010875
        minimum = 0.0096 (at node 0)
        maximum = 0.0139 (at node 7)
Injected flit rate average = 0.0108667
        minimum = 0.0063 (at node 3)
        maximum = 0.018 (at node 8)
Accepted flit rate average= 0.010875
        minimum = 0.0096 (at node 0)
        maximum = 0.0139 (at node 7)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.315437
throughput change = 0.0337165
Class 0:
Packet latency average = 13883.6
        minimum = 164
        maximum = 31189
Network latency average = 346.06
        minimum = 13
        maximum = 1432
```

```
Slowest packet = 4074
Flit latency average = 348.781
        minimum = 13
        maximum = 1432
Slowest flit = 5791
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0110875
        minimum = 0.00695 (at node 0)
        maximum = 0.01905 (at node 8)
Accepted packet rate average = 0.0110833
        minimum = 0.0102 (at node 10)
        maximum = 0.0117 (at node 1)
Injected flit rate average = 0.0110875
        minimum = 0.00695 (at node 0)
        maximum = 0.01905 (at node 8)
Accepted flit rate average= 0.0110833
        minimum = 0.0102 (at node 10)
        maximum = 0.0117 (at node 1)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 59 (59 measured)
latency change    = 0.11093
throughput change = 0.018797
Class 0:
Packet latency average = 15672.2
        minimum = 164
        maximum = 37429
Network latency average = 344.413
        minimum = 13
        maximum = 1432
Slowest packet = 4074
Flit latency average = 346.243
        minimum = 13
        maximum = 1432
Slowest flit = 5791
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.011125
        minimum = 0.0072 (at node 7)
        maximum = 0.0184333 (at node 8)
Accepted packet rate average = 0.011125
        minimum = 0.0105667 (at node 2)
        maximum = 0.0123 (at node 3)
Injected flit rate average = 0.011125
        minimum = 0.0072 (at node 7)
        maximum = 0.0184333 (at node 8)
Accepted flit rate average= 0.011125
        minimum = 0.0105667 (at node 2)
        maximum = 0.0123 (at node 3)
Injected packet length average = 1
```

```
Accepted packet length average = 1
Total in-flight flits = 58 (58 measured)
latency change    = 0.114131
throughput change = 0.00374532
Class 0:
Packet latency average = 17194.6
        minimum = 164
        maximum = 43793
Network latency average = 348.875
        minimum = 13
        maximum = 1432
Slowest packet = 4074
Flit latency average = 350.215
        minimum = 13
        maximum = 1432
Slowest flit = 5791
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0110042
        minimum = 0.00715 (at node 0)
        maximum = 0.0181 (at node 8)
Accepted packet rate average = 0.0110042
        minimum = 0.010225 (at node 0)
        maximum = 0.011975 (at node 3)
Injected flit rate average = 0.0110042
        minimum = 0.00715 (at node 0)
        maximum = 0.0181 (at node 8)
Accepted flit rate average= 0.0110042
        minimum = 0.010225 (at node 0)
        maximum = 0.011975 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.088534
throughput change = 0.0109807
Class 0:
Packet latency average = 18819.7
        minimum = 164
        maximum = 49891
Network latency average = 350.226
        minimum = 13
        maximum = 1432
Slowest packet = 4074
Flit latency average = 351.289
        minimum = 13
        maximum = 1432
Slowest flit = 5791
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0109783
        minimum = 0.00706 (at node 0)
```

```
        maximum = 0.018 (at node 8)
Accepted packet rate average = 0.01098
        minimum = 0.01012 (at node 0)
        maximum = 0.01198 (at node 3)
Injected flit rate average = 0.0109783
        minimum = 0.00706 (at node 0)
        maximum = 0.018 (at node 8)
Accepted flit rate average= 0.01098
        minimum = 0.01012 (at node 0)
        maximum = 0.01198 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.0863534
throughput change = 0.00220097
Class 0:
Packet latency average = 20473.2
        minimum = 164
        maximum = 56702
Network latency average = 350.566
        minimum = 13
        maximum = 1484
Slowest packet = 4074
Flit latency average = 351.451
        minimum = 13
        maximum = 1484
Slowest flit = 11185
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0109583
        minimum = 0.00715 (at node 0)
        maximum = 0.0181333 (at node 8)
Accepted packet rate average = 0.0109569
        minimum = 0.0102667 (at node 5)
        maximum = 0.0118833 (at node 3)
Injected flit rate average = 0.0109583
        minimum = 0.00715 (at node 0)
        maximum = 0.0181333 (at node 8)
Accepted flit rate average= 0.0109569
        minimum = 0.0102667 (at node 5)
        maximum = 0.0118833 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 59 (59 measured)
latency change    = 0.0807633
throughput change = 0.0021042
Class 0:
Packet latency average = 22190
        minimum = 164
        maximum = 63533
Network latency average = 350.325
        minimum = 13
```

```
        maximum = 1484
Slowest packet = 4074
Flit latency average = 351.085
        minimum = 13
        maximum = 1484
Slowest flit = 11185
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0109619
        minimum = 0.00714286 (at node 4)
        maximum = 0.0181571 (at node 9)
Accepted packet rate average = 0.0109619
        minimum = 0.0102429 (at node 0)
        maximum = 0.0115714 (at node 6)
Injected flit rate average = 0.0109619
        minimum = 0.00714286 (at node 4)
        maximum = 0.0181571 (at node 9)
Accepted flit rate average= 0.0109619
        minimum = 0.0102429 (at node 0)
        maximum = 0.0115714 (at node 6)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 58 (58 measured)
latency change    = 0.0773679
throughput change = 0.000452505
Draining all recorded packets ...
Class 0:
.
.
.
Draining remaining packets ...
Time taken is 272185 cycles
====== Overall Traffic Statistics ======
====== Traffic class 0 ======
Packet latency average = 68044.5 (1 samples)
        minimum = 164 (1 samples)
        maximum = 171627 (1 samples)
Network latency average = 416.403 (1 samples)
        minimum = 13 (1 samples)
        maximum = 1487 (1 samples)
Flit latency average = 352.451 (1 samples)
        minimum = 13 (1 samples)
        maximum = 1487 (1 samples)
Fragmentation average = 0 (1 samples)
        minimum = 0 (1 samples)
        maximum = 0 (1 samples)
Injected packet rate average = 0.0109619 (1 samples)
        minimum = 0.00714286 (1 samples)
        maximum = 0.0181571 (1 samples)
Accepted packet rate average = 0.0109619 (1 samples)
        minimum = 0.0102429 (1 samples)
        maximum = 0.0115714 (1 samples)
```

```
Injected flit rate average = 0.0109619 (1 samples)
        minimum = 0.00714286 (1 samples)
        maximum = 0.0181571 (1 samples)
Accepted flit rate average = 0.0109619 (1 samples)
        minimum = 0.0102429 (1 samples)
        maximum = 0.0115714 (1 samples)
Injected packet size average = 1 (1 samples)
Accepted packet size average = 1 (1 samples)
Hops average = 2.04442 (1 samples)
Total run time 2.05016
```

B.2.2 Crowded Topology Simulation with Four Percent Injection Rate Configuration File and Results

```
topology = anynet;


routing_function = min;
network_file = anynet_file;
traffic = uniform;

use_read_write = 0;

sample_period = 10000;
injection_rate = 0.04;
credit_delay = 1;
routing_delay = 2;

vc_allocator = separable_input_first;
sw_allocator = separable_input_first;
alloc_iters  = 1;

traffic = uniform;

num_vcs     = 1;
vc_buf_size = 3;

END Configuration File: ./anynet_config
OVERRIDE Parameter: latency_thres=500000.0
========================Network File Parsed================
******************node listing**********************
Node 0  Router 0
Node 1  Router 0
Node 2  Router 0
Node 3  Router 0
Node 4  Router 1
Node 5  Router 1
Node 6  Router 1
Node 7  Router 1
Node 8  Router 2
Node 9  Router 2
Node 10 Router 2
Node 11 Router 3
```

```
***************router to node listing************
Router 0
         Node 0 lat 1
         Node 1 lat 1
         Node 2 lat 1
         Node 3 lat 1
Router 1
         Node 4 lat 1
         Node 5 lat 1
         Node 6 lat 1
         Node 7 lat 1
Router 2
         Node 8 lat 1
         Node 9 lat 1
         Node 10 lat 1
Router 3
         Node 11 lat 1

*****************router to router listing************
Router 0
         Router 2 lat 40
Router 1
         Router 2 lat 40
Router 2
         Router 0 lat 40
         Router 1 lat 40
         Router 3 lat 40
Router 3
         Router 2 lat 40
=========================Node to Router ====================
router 0 radix 5
         connected to node 0 at outport 0 lat 1
         connected to node 1 at outport 1 lat 1
         connected to node 2 at outport 2 lat 1
         connected to node 3 at outport 3 lat 1
router 1 radix 5
         connected to node 4 at outport 0 lat 1
         connected to node 5 at outport 1 lat 1
         connected to node 6 at outport 2 lat 1
         connected to node 7 at outport 3 lat 1
router 2 radix 6
         connected to node 8 at outport 0 lat 1
         connected to node 9 at outport 1 lat 1
         connected to node 10 at outport 2 lat 1
router 3 radix 2
         connected to node 11 at outport 0 lat 1
=========================Router to Router ====================
router 0
         connected to router 2 using link 0 at outport 4 lat 40
router 1
         connected to router 2 using link 1 at outport 4 lat 40
router 2
         connected to router 0 using link 2 at outport 3 lat 40
```

```
         connected to router 1 using link 3 at outport 4 lat 40
         connected to router 3 using link 4 at outport 5 lat 40
router 3
         connected to router 2 using link 5 at outport 1 lat 40
========================= Routing table =====================
Class 0:
Packet latency average = 3366.49
        minimum = 8
        maximum = 8206
Network latency average = 334.299
        minimum = 8
        maximum = 1309
Slowest packet = 61
Flit latency average = 334.299
        minimum = 8
        maximum = 1309
Slowest flit = 236
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0114333
        minimum = 0.0067 (at node 6)
        maximum = 0.0208 (at node 10)
Accepted packet rate average = 0.0110583
        minimum = 0.0095 (at node 5)
        maximum = 0.0129 (at node 8)
Injected flit rate average = 0.0114333
        minimum = 0.0067 (at node 6)
        maximum = 0.0208 (at node 10)
Accepted flit rate average= 0.0110583
        minimum = 0.0095 (at node 5)
        maximum = 0.0129 (at node 8)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (0 measured)
latency change    = 1
throughput change = 1
Class 0:
Packet latency average = 6731.19
        minimum = 8
        maximum = 16414
Network latency average = 344.165
        minimum = 8
        maximum = 1474
Slowest packet = 61
Flit latency average = 344.165
        minimum = 8
        maximum = 1474
Slowest flit = 2330
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0111958
```

```
        minimum = 0.00705 (at node 6)
        maximum = 0.0185 (at node 8)
Accepted packet rate average = 0.011
        minimum = 0.0099 (at node 6)
        maximum = 0.01175 (at node 8)
Injected flit rate average = 0.0111958
        minimum = 0.00705 (at node 6)
        maximum = 0.0185 (at node 8)
Accepted flit rate average= 0.011
        minimum = 0.0099 (at node 6)
        maximum = 0.01175 (at node 8)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 59 (0 measured)
latency change    = 0.499866
throughput change = 0.00530303
Class 0:
Packet latency average = 16598.2
        minimum = 9990
        maximum = 24475
Network latency average = 337.129
        minimum = 13
        maximum = 1289
Slowest packet = 2665
Flit latency average = 337.129
        minimum = 13
        maximum = 1289
Slowest flit = 2948
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0114083
        minimum = 0.0069 (at node 7)
        maximum = 0.0204 (at node 8)
Accepted packet rate average = 0.0114
        minimum = 0.0099 (at node 7)
        maximum = 0.0144 (at node 10)
Injected flit rate average = 0.0114083
        minimum = 0.0069 (at node 7)
        maximum = 0.0204 (at node 8)
Accepted flit rate average= 0.0114
        minimum = 0.0099 (at node 7)
        maximum = 0.0144 (at node 10)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 60 (0 measured)
latency change    = 0.594464
throughput change = 0.0350877
Warmed up ...Time used is 30000 cycles
Class 0:
Packet latency average = 23636.4
        minimum = 15698
        maximum = 33231
```

```
Network latency average = 359.878
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 369.162
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0105083
        minimum = 0.0046 (at node 2)
        maximum = 0.017 (at node 8)
Accepted packet rate average = 0.0105333
        minimum = 0.0092 (at node 10)
        maximum = 0.0133 (at node 7)
Injected flit rate average = 0.0105083
        minimum = 0.0046 (at node 2)
        maximum = 0.017 (at node 8)
Accepted flit rate average= 0.0105333
        minimum = 0.0092 (at node 10)
        maximum = 0.0133 (at node 7)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.297769
throughput change = 0.0822785
Class 0:
Packet latency average = 27112
        minimum = 15698
        maximum = 41195
Network latency average = 351.766
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 356.494
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0107583
        minimum = 0.0064 (at node 2)
        maximum = 0.01875 (at node 8)
Accepted packet rate average = 0.0107708
        minimum = 0.00985 (at node 0)
        maximum = 0.01145 (at node 7)
Injected flit rate average = 0.0107583
        minimum = 0.0064 (at node 2)
        maximum = 0.01875 (at node 8)
Accepted flit rate average= 0.0107708
```

```
        minimum = 0.00985 (at node 0)
        maximum = 0.01145 (at node 7)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.128192
throughput change = 0.0220503
Class 0:
Packet latency average = 30548.4
        minimum = 15698
        maximum = 49549
Network latency average = 351.466
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 354.582
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0109028
        minimum = 0.00706667 (at node 6)
        maximum = 0.0190667 (at node 8)
Accepted packet rate average = 0.0109111
        minimum = 0.0103667 (at node 2)
        maximum = 0.012 (at node 3)
Injected flit rate average = 0.0109028
        minimum = 0.00706667 (at node 6)
        maximum = 0.0190667 (at node 8)
Accepted flit rate average= 0.0109111
        minimum = 0.0103667 (at node 2)
        maximum = 0.012 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 57 (57 measured)
latency change    = 0.11249
throughput change = 0.0128564
Class 0:
Packet latency average = 33902.2
        minimum = 15698
        maximum = 57859
Network latency average = 350.421
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 352.766
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
```

```
        maximum = 0
Injected packet rate average = 0.0109292
        minimum = 0.00705 (at node 2)
        maximum = 0.019175 (at node 8)
Accepted packet rate average = 0.0109292
        minimum = 0.01015 (at node 10)
        maximum = 0.0119 (at node 3)
Injected flit rate average = 0.0109292
        minimum = 0.00705 (at node 2)
        maximum = 0.019175 (at node 8)
Accepted flit rate average= 0.0109292
        minimum = 0.01015 (at node 10)
        maximum = 0.0119 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 60 (60 measured)
latency change    = 0.0989266
throughput change = 0.00165205
Class 0:
Packet latency average = 37085.7
        minimum = 15698
        maximum = 66072
Network latency average = 352.369
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 354.233
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.010895
        minimum = 0.00688 (at node 2)
        maximum = 0.01922 (at node 8)
Accepted packet rate average = 0.0108983
        minimum = 0.01014 (at node 0)
        maximum = 0.01184 (at node 3)
Injected flit rate average = 0.010895
        minimum = 0.00688 (at node 2)
        maximum = 0.01922 (at node 8)
Accepted flit rate average= 0.0108983
        minimum = 0.01014 (at node 0)
        maximum = 0.01184 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 58 (58 measured)
latency change    = 0.0858419
throughput change = 0.00282918
Class 0:
Packet latency average = 40625.6
        minimum = 15698
```

```
        maximum = 74448
Network latency average = 349.688
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 351.25
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.010975
        minimum = 0.00711667 (at node 2)
        maximum = 0.0190167 (at node 8)
Accepted packet rate average = 0.0109778
        minimum = 0.0103 (at node 5)
        maximum = 0.0118833 (at node 3)
Injected flit rate average = 0.010975
        minimum = 0.00711667 (at node 2)
        maximum = 0.0190167 (at node 8)
Accepted flit rate average= 0.0109778
        minimum = 0.0103 (at node 5)
        maximum = 0.0118833 (at node 3)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 58 (58 measured)
latency change    = 0.0871346
throughput change = 0.00723684
Class 0:
Packet latency average = 43887.5
        minimum = 15698
        maximum = 82865
Network latency average = 350.253
        minimum = 13
        maximum = 1736
Slowest packet = 4073
Flit latency average = 351.591
        minimum = 13
        maximum = 1736
Slowest flit = 4722
Fragmentation average = 0
        minimum = 0
        maximum = 0
Injected packet rate average = 0.0109571
        minimum = 0.00705714 (at node 2)
        maximum = 0.0191857 (at node 8)
Accepted packet rate average = 0.0109583
        minimum = 0.0102571 (at node 0)
        maximum = 0.0115571 (at node 6)
Injected flit rate average = 0.0109571
        minimum = 0.00705714 (at node 2)
        maximum = 0.0191857 (at node 8)
```

```
Accepted flit rate average= 0.0109583
        minimum = 0.0102571 (at node 0)
        maximum = 0.0115571 (at node 6)
Injected packet length average = 1
Accepted packet length average = 1
Total in-flight flits = 59 (59 measured)
latency change    = 0.0743237
throughput change = 0.0017744
Draining all recorded packets ...
Class 0:
.
.
.
Measured flits: 73097 (1 flits)
Draining remaining packets ...
Time taken is 558770 cycles
====== Overall Traffic Statistics ======
====== Traffic class 0 ======
Packet latency average = 178598 (1 samples)
        minimum = 15698 (1 samples)
        maximum = 458108 (1 samples)
Network latency average = 409.409 (1 samples)
        minimum = 13 (1 samples)
        maximum = 1736 (1 samples)
Flit latency average = 352.997 (1 samples)
        minimum = 13 (1 samples)
        maximum = 1736 (1 samples)
Fragmentation average = 0 (1 samples)
        minimum = 0 (1 samples)
        maximum = 0 (1 samples)
Injected packet rate average = 0.0109571 (1 samples)
        minimum = 0.00705714 (1 samples)
        maximum = 0.0191857 (1 samples)
Accepted packet rate average = 0.0109583 (1 samples)
        minimum = 0.0102571 (1 samples)
        maximum = 0.0115571 (1 samples)
Injected flit rate average = 0.0109571 (1 samples)
        minimum = 0.00705714 (1 samples)
        maximum = 0.0191857 (1 samples)
Accepted flit rate average = 0.0109583 (1 samples)
        minimum = 0.0102571 (1 samples)
        maximum = 0.0115571 (1 samples)
Injected packet size average = 1 (1 samples)
Accepted packet size average = 1 (1 samples)
Hops average = 2.04591 (1 samples)
Total run time 4.10308
```