2010

Architectural Support For Improving Computer Security

Jingfei Kong
University of Central Florida

Part of the Computer Sciences Commons, and the Engineering Commons

Find similar works at: https://stars.library.ucf.edu/etd
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
https://stars.library.ucf.edu/etd/4294
ARCHITECTURAL SUPPORT FOR IMPROVING COMPUTER SECURITY

by

JINGFEI KONG
B.E. Southeast University, 1999
M.S. University of Central Florida, 2005

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2010

Major Professor. Huiyang Zhou
© 2010 Jingfei Kong
ABSTRACT

Computer security and privacy are becoming extremely important nowadays. The task of protecting computer systems from malicious attacks and potential subsequent catastrophic losses is, however, challenged by the ever increasing complexity and size of modern hardware and software design.

We propose several methods to improve computer security and privacy from architectural point of view. They provide strong protection as well as performance efficiency. In our first approach, we propose a new dynamic information flow method to protect systems from popular software attacks such as buffer overflow and format string attacks. In our second approach, we propose to deploy encryption schemes to protect the privacy of an emerging non-volatile main memory technology – phase change memory (PCM). The negative impact of the encryption schemes on PCM lifetime is evaluated and new methods including a new encryption counter scheme and an efficient error correct code (ECC) management are proposed to improve PCM lifetime. In our third approach, we deconstruct two previously proposed secure cache designs against software data-cache-based side channel attacks and demonstrate their weaknesses. We propose three hardware-software integrated approaches as secure protections against those data cache attacks. Also we propose to apply them to protect instruction caches from similar threats. Furthermore, we propose a simple change to the update policy of Branch Target Buffer (BTB) to defend against BTB attacks. Our experiments show that our proposed schemes are both security effective and performance efficient.
ACKNOWLEDGMENTS

This dissertation would not have been possible without the help and support of a number of people. First and foremost, I would like to express my deepest gratitude and greatest regard to my advisor, Dr. Huiyang Zhou. I feel very fortunate to work with him and certainly have learned a lot during the process. His inspiring and constructive supervision has always been a constant source of encouragement for my study. I thank him for his generous support, insightful guidance and all his help extended to me throughout all these years.

Next, I wish to thank the other members of my dissertation committee (Dr. Mark Heinrich, Dr. Cliff Zou, and Dr. Liqiang Ni) for flexibility in scheduling my final oral exam and for providing me with insightful comments to constantly improve this dissertation. Also I would like to thank Michael Mantor, Dr. Onur Acıçmez and Dr. Jean-Pierre Seifert for their sound advice, excellent insight and strong encouragement. I wish to bear their passion for knowledge and research in my future work. I would like to thank my group members: Hongliang Gao, Martin Dimitrov, Yi Ma, Yi Yang, Lin Cao, Ping Xiang, Zhaoshi Zheng and Jacob Staples. I have enjoyed every moment that we have worked together.

I dedicate this dissertation to my loving and supportive wife, Xiaoyi Jin, and my parents and parents-in-law. Without their positive attitudes and unwavering support I never would have reached this point.

Last but not least, I would also like to extend my thanks to those who have cared and helped, in one way or another, in making this dissertation possible.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................................................ xii

LIST OF TABLES ........................................................................................................................................... xv

CHAPTER 1.  INTRODUCTION .......................................................................................................................... 1

1.1.  Hardware/Software challenges to building secure systems ............................................................... 1

1.2.  Our solutions to improve computer security .................................................................................... 1

1.3.  Contributions ........................................................................................................................................... 3

CHAPTER 2.  IMPROVING SOFTWARE SECURITY VIA RUNTIME INSTRUCTION-LEVEL TAINT CHECKING ... ............................................................................................................................ 4

2.1.  Motivation and Related Work .............................................................................................................. 6

2.1.1.  Low-level Vulnerabilities ................................................................................................................... 6

2.1.2.  Taint Tracking and Checking ........................................................................................................... 7

2.1.3.  Non-Control Data Attack Examples ................................................................................................. 9

2.3.  Experiments .......................................................................................................................................... 13

2.3.1.  Experimental Methodology ............................................................................................................... 13

2.3.2.  Preliminary Results from SPEC CPU2000 ...................................................................................... 13

2.4.  Application Examples ........................................................................................................................... 15
2.4.1. Buffer Overflow ...................................................................................................... 15
2.4.2. Format String .......................................................................................................... 17
2.5. Discussion and Conclusion ............................................................................................ 18
2.5.1. Overhead to Taint Tracking Architecture ............................................................... 18
2.5.2. Collection of Taintless-Instructions Profile ............................................................ 19
2.5.3. Vulnerability and Shortcoming ............................................................................... 19
2.5.4. Conclusion .............................................................................................................. 20
List of References ............................................................................................................... 21

CHAPTER 3. IMPROVING PRIVACY AND LIFETIME OF PCM-BASED MAIN MEMORY

3.1. Background and Related Work ...................................................................................... 25
3.1.1. Privacy .................................................................................................................... 25
3.1.2. Wear-leveling Techniques ...................................................................................... 27
3.1.3. Reliability, Lifetime and ECC ............................................................................... 28
3.2. Privacy Protection for PRAM ........................................................................................ 29
3.2.1. The Impact of Encryption on PRAM Wear-leveling Techniques ......................... 29
3.2.2. A New Encryption Counter Scheme to Mitigate the Encryption Impact ............... 31
3.3. Adaptive ECC Management ........................................................................................... 32
3.3.1. The ECC Space and Address Mapping................................................................. 33
3.3.2. Leveraging Encryption Counters as Write-access (Age) Counters .................. 36

3.4. Overall Architecture.............................................................................................. 37

3.5. Experimental results............................................................................................. 39

3.5.1. Methodology....................................................................................................... 39

3.5.2. Impact of Encryption on Wear-leveling Techniques....................................... 41

3.5.3. Effect of the New Proposed Encryption Counter Scheme on Partial Writes..... 43

3.5.4. Impact of Memory Compression on Wear-leveling Techniques..................... 44

3.5.5. PRAM Lifetime Comparison............................................................................. 45

3.5.6. ECC Space Overhead and Wear Out............................................................... 47

3.5.7. Performance Overhead of Encryption and ECC............................................. 49

3.5.8. Summary........................................................................................................... 49

3.6. Conclusion............................................................................................................. 50

List of References......................................................................................................... 51

CHAPTER 4. DECONSTRUCTING NEW CACHE DESIGNS FOR THWARTING SOFTWARE CACHE-BASED SIDE CHANNEL ATTACKS....................................................... 54

4.1. Background ........................................................................................................... 56

4.1.1. AES and Its Software Implementation............................................................. 56

4.1.2. Software Cache-based Side-Channel Attacks............................................... 58

4.2. Recently Proposed New Cache Designs for Thwarting Software Cache-based Side Channel Attacks.......................................................... 65
4.2.1. Partition-Locked Cache .......................................................................................... 66
4.2.2. Random-Permutation Cache ................................................................................... 68
4.3. Security Analysis of The New Cache Designs ............................................................... 69
   4.3.1. Analysis of PLcache ............................................................................................... 69
   4.3.2. Analysis of RPcache ............................................................................................... 72
4.4. Potential Solutions .......................................................................................................... 73
   4.4.1. Securing the PLcache .............................................................................................. 73
   4.4.2. Securing the RPcache ............................................................................................. 74
4.5. Conclusions .................................................................................................................... 75

List of References ..................................................................................................................... 77

CHAPTER 5. ARCHITECTURAL APPROACHES TO DEFEND AGAINST SOFTWARE
CACHE-BASED SIDE CHANNEL ATTACKS ........................................................................ 81

5.1. Threat Model and Attacks .............................................................................................. 84
   5.1.1. The Advanced Encryption Standard (AES) and Data Cache Attacks ................. 85
      5.1.1.1. Access-driven Attacks against AES ................................................................. 85
      5.1.1.2. Time-driven Attacks against AES ................................................................. 86
      5.1.1.3. Source of Information Leakage in Data Cache Attacks ................................. 88
   5.1.2. RSA and BTB/I-cache Attacks .............................................................................. 88
      5.1.2.1. BTB Attacks .................................................................................................. 89
      5.1.2.2. I-cache Attacks ............................................................................................. 90
5.3.1.2. Proposed Implementation ................................................................. 101

5.3.2. Securing the RPcache with Informing Loads ........................................... 103

5.3.2.1. The Idea........................................................................................... 103

5.3.2.2. Proposed Implementation ................................................................. 105

5.3.3. Securing Regular Caches with Informing Loads .................................. 106

5.3.3.1. The Idea........................................................................................... 106

5.3.3.2. Proposed Implementation ................................................................. 107

5.4. Defending against BTB Attacks and Instruction Cache Attacks .................... 109

5.4.1. An Always BTB Update Policy as a Countermeasure against BTB Attacks ...... 109

5.4.2. Using Hardware Countermeasures for Data Cache Attacks against I-cache Attacks .................................................................................................................. 111

5.5. Experiments............................................................................................... 112

5.5.1. Methodology......................................................................................... 112

5.5.2. Security Analysis .................................................................................... 114

5.5.2.1. Microarchitectural Effects on Cache Collision Time-driven Attacks ............. 114

5.5.2.2. Security Evaluation of the Proposed Schemes........................................ 116

5.5.3. Performance Evaluation.......................................................................... 119

5.5.3.1. Performance Impact ........................................................................... 119

5.5.3.2. Performance Impact on an SMT Processor.......................................... 123

5.5.4. Summary............................................................................................... 127
LIST OF FIGURES

Figure 1. Current taint tracking and checking scheme ................................................................. 8
Figure 2. Examples of non-control data attacks ........................................................................... 9
Figure 3. Our Taintless-Instructions-enhanced processor-memory model ................................. 11
Figure 4. The percentages of Taintless-Instructions and Tainted-Instructions in SPEC CPU2000
CINT ............................................................................................................................................. 14
Figure 5. Buffer overflow attack detection .................................................................................... 15
Figure 6. Format string attack detection ....................................................................................... 18
Figure 7. Counter-mode encryption for secure processors ............................................................ 26
Figure 8. The impact of ECC correctability on bit error rates (based on data block size of 64
bytes) ............................................................................................................................................. 28
Figure 9. Avalanche effect caused by encryption ........................................................................ 30
Figure 10. Dynamic requirement for ECC to meet UBER of $10^{-19}$ as PRAM gets worn out .... 32
Figure 11. The design of our proposed scheme for dynamic ECC management ......................... 34
Figure 12. The address mapping of ECC protection bits ............................................................... 36
Figure 13. The architecture of the proposed scheme for privacy protection and lifetime
improvement .................................................................................................................................... 37
Figure 14. Impact of encryption on redundant bit-write removal .................................................. 42
Figure 15. Impact of encryption on partial writes ........................................................................ 43
Figure 16. Impact of our new encryption counter scheme on partial writes ............................... 44
Figure 17. Impact of Frequent Pattern Compression on redundant bit-write removal ...................... 45
Figure 18. PRAM lifetime with various schemes (assuming uniform writes) ................................. 46
Figure 19. Impact of encryption on redundant bit-write behavior of ECC .................................... 48
Figure 20. Performance overhead of encryption and ECC .......................................................... 49
Figure 21. Round computations in AES ...................................................................................... 57
Figure 22. An example of access-driven attacks: prime and probe caches .................................. 61
Figure 23. The relationship between the number of collisions in the final round and the encryption time ......................................................................................................................... 63
Figure 24. The relationship between the mean encryption time and $x_0 \oplus x_1$ given $k_0 \oplus k_1$ being 254 ........................................................................................................................................... 65
Figure 25. The new cache line of the PLcache ............................................................................. 66
Figure 26. One logical view of the RPcache ................................................................................. 68
Figure 27. An example of using access-driven attacks to reveal cache line usage in the PLcache .... 71
Figure 28. Experimental results of running last round cache-collision attack on RPcache .......... 72
Figure 29. Vulnerable table lookup operations in AES ................................................................. 85
Figure 30. The relationship between the number of collisions in the last round of AES and the encryption time on one Pentium 4 machine .................................................................................. 87
Figure 31. Modular exponentiation using the binary version of the Square-and-Multiply algorithm ........................................................................................................................................... 88
Figure 32. Various approaches to remove key-dependent control flow ........................................ 97
Figure 33. Code of the informing load exception handler ........................................................... 105
Figure 34. Converting a one-dimension table into a two-dimension array .................................. 108
Figure 35. Address permutation by swapping both the pointers and the data ......................... 108
Figure 36. Pseudo code of the exception handler for informing loads, which prefetches all critical data to the cache and randomly permutes the cache lines ......................................................... 109

Figure 37. The relationship between the number of collisions in the final round and the normalized encryption time on various processor configurations ........................................... 115

Figure 38. The effect of our informing loads approaches on the relationship between the number of collisions in the final round and the normalized encryption time ........................................... 117

Figure 39. Performance impacts of different protection schemes over various data cache configurations ......................................................................................................................... 120

Figure 40. Performance impacts of various protection schemes over various data cache configurations in two-way SMT processors ................................................................. 124

Figure 41. Performance impacts of the protection scheme over various BTB cache configurations in two-way SMT processors ................................................................................. 126

Figure 42. Performance impacts of the protection schemes over various instruction cache configurations in two-way SMT processors ........................................................................ 127
LIST OF TABLES

Table 1. Default processor configuration ........................................................................ 40
Table 2. Default processor configuration ........................................................................ 70
Table 3. A summary of recent software cache-based side channel attacks .................. 91
Table 4. Default processor configuration ...................................................................... 112
Table 5. Different processor configurations to evaluate cache collision effects .......... 114
Table 6. Required numbers of samples for key recovery on various processor configurations. (based on 25 random keys with random input plaintext) .......................................................... 116
Table 7. A comparison between various protection schemes ...................................... 129
CHAPTER 1. INTRODUCTION

1.1. Hardware/Software challenges to building secure systems

Computer security and privacy are becoming extremely important nowadays. The task of protecting computer systems from malicious attacks and potential subsequent catastrophic losses is, however, challenged by the ever increasing complexity and size of hardware and software design. On one hand, thorough consideration on security at every stage of product design and development is necessary and important. On the other hand, performance optimization, cost control and time-to-market pressure often leave security holes which are hard to plug effectively and efficiently.

1.2. Our solutions to improve computer security

We propose several methods to improve computer security and privacy from architectural point of view. They provide strong protection as well as performance efficiency. In our first approach, we analyze new non-control data attacks, which exploits popular vulnerabilities such as buffer overflow and format string. Previously proposed dynamic information flow schemes (also called tainting architectures) cannot detect such attacks as they only cover control data. We propose to classify instructions into two different types based on tainted or non-tainted data that they handle. This enables instruction-level taint checking to cover some non-control data attacks. Such architectural solution incurs small changes to hardware while introducing small performance overhead.
In our second approach, we investigate the issue of using counter-mode encryption to protect the privacy of a new emerging non-volatile main memory technology – phase change memory (PCM). We show that encryption has negative impact on some of the previously proposed wear leveling techniques for PCM. To mitigate such impact on PCM lifetime, we propose a new encryption counter scheme. Also we propose to leverage existing encryption counters to efficiently manage error correct code (ECC) to extend PCM lifetime.

In our third approach, we first deconstruct two previously proposed secure cache designs (PLcache and RPeache) for security against software data-cache-based side channel attacks. We show that those secure cache designs, although effective against two particular attacks with small hardware cost and performance overhead, fail to address the root causes of software data-cache-based side channel attacks. We identify that cache misses of critical data, whose addresses are dependent on secret keys, are the source of information leakage. We then propose three hardware-software integrated approaches to protect PLcache, RPeache and regular caches from data cache attacks. A comparison of various protection schemes on tradeoff between security and efficiency is also presented.

Software cache-based side channel attacks not only exploit data caches, but also other hardware caches such as branch target buffers (BTB attacks) and instruction caches (I-cache attacks). We identify that in BTB attacks, updates of critical conditional branch target addresses in BTB, which are determined by secret keys, are the source of information leakage. We then propose a simple change to BTB update policy to defend against such attacks. For I-cache attacks, we identify that cache misses of instructions, whose execution depends on secret keys, are the source of information leakage. As I-cache attacks work in a similar way as data cache
attacks, we propose to apply similar defense schemes for data caches to protect instruction caches.

1.3. Contributions

The contributions of this dissertation are summarized as follows:

1. We propose an architectural support for the protection of data integrity through fine-grain instruction-level taint checking. Besides, our taint checking architecture requires minor changes to the original taint tracking architecture and it can complement other taint checking techniques such as pointer taintedness checking to provide a higher degree of security protection.

2. Our work on PCM, to our knowledge, is the first work to investigate the impact of encryption on wear-leveling techniques for PCM-based main memory. To mitigate the negative impact of encryption, we propose a new encryption counter scheme to revive partial writes to reduce write traffic. We also propose to leverage existing encryption counters as age counters and design an efficient way for ECC management to extend the PRAM lifetime.

3. We analyze two secure cache designs and show that they fall short on avoiding some data cache attacks. We identify the source of information leakage for data cache attacks and propose three hardware-software integrated approaches as protection schemes. We also identify the sources of information leakage for the other two software cache-based attacks and propose effective and efficient solutions as defense mechanisms.
CHAPTER 2. IMPROVING SOFTWARE SECURITY VIA RUNTIME INSTRUCTION-LEVEL TAINT CHECKING

The increasing size and complexity of modern software systems lead to an increasing number of security vulnerabilities. Well-known examples include buffer overflow, heap corruption, format string, integer overflow, etc. By carefully exploiting these vulnerabilities, attackers may cause severe damages to the running process or even ultimately gain the control of victim computers.

During the past decade, numerous schemes have been developed against different kinds of security attacks. Among them, mitigation techniques are one very important defending category since it is always difficult to discover and fix program flaws in advance. As a way of providing additional protection to unsafe systems, mitigation techniques usually try to mitigate the consequence of an attack by stopping the malicious behavior from happening upon attack detection. Although there are a great many of mitigation techniques being developed and deployed, no solution yet can defeat all currently-known exploits [12].

Recently dynamic taint tracking and checking [17], [6] has been widely accepted as a promising mitigation scheme. Based on the observation that attacks are always launched from suspicious I/O channels such as files or network sockets, it seeks to capture the essence of attacks. It treats data from those suspicious input channels as tainted data and keeps track of the tainted data propagation as they may directly or indirectly affect other data values in the program. In order to do so, the processor-memory model is enhanced to keep track of taint information. Based on tainted data tracking, certain taint checking is performed during the
runtime execution. It is not until checking fails that the system is to be alarmed. Given the fact that current attacks usually seek to change the control flow of the victim program, one commonly adopted taint checking rule is that control data shall never be tainted [17], [6].

Although control-data attacks are currently dominant, in this chapter we focus on non-control data attacks. The reality and applicability of these attacks have already been demonstrated in previous work [2]. It is foreseeable that when control data protection techniques are widely deployed, attackers may have the incentive to bypass those techniques by using non-control data attacks. Previous work from [3] has used taint checking on pointers against those attacks. In other words, if tainted data is used as an address, an alarm is raised. However it has false negative scenarios (See Section 2.1). Annotating important data structures that should never be tainted could be another approach [3] and it may involve very complex program analysis.

In this chapter, we present a generic instruction-level runtime taint checking architecture to prevent non-control data attacks. Under our architecture, instructions are classified as either Taintless-Instructions or Tainted-Instructions prior to program execution. An instruction is called a Tainted-Instruction if it is supposed to deal with tainted data. Otherwise it is called a Taintless-Instruction. A security alert is alarmed whenever a Taintless-Instruction encounters tainted data at runtime.

The contribution of our work is the architectural support for the protection of data integrity through fine-grain instruction-level taint checking. Besides, our taint checking architecture requires minor changes to the original taint tracking architecture and it can complement other taint checking techniques such as pointer taintedness checking [3] to provide a higher degree of security protection. Our preliminary statistical results from experiments on SPEC CPU2000 benchmarks show that there are a significant amount of instructions which are
not supposed to deal with tainted data. We also demonstrate the effective usage of our architecture for buffer overflow and format string attack detections on an enhanced SimpleScalar simulator through manual annotation.

2.1. Motivation and Related Work

2.1.1 Low-level Vulnerabilities

Low-level software vulnerabilities such as buffer overflow, heap corruption, format string and integer overflow account for the largest portion of CERT advisories [3]. Malicious users may exploit these vulnerabilities to write to arbitrary memory locations with any specified values. For example, an attacker may overwrite values such as function return addresses, jump targets or function pointers. Once control transfer instructions use those values, the control flow may change to the code which the program should not execute. Ultimately an attacker may compromise a host by doing so.

**Buffer overflow.** It results from writing to a buffer without bounds checking. Once data is stored beyond the boundary of a fixed length buffer, adjacent memory data may be corrupted. It is the most common and exploitable one among all vulnerabilities [11]. There have been enormous efforts put into buffer overflow detection and prevention, including static analysis and dynamic runtime monitoring. A good summary and evaluation of dynamic buffer overflow prevention can be found in [18]. Static analysis may generate too many false warnings or miss errors in the code [15] while dynamic mitigation techniques also have limitations, as highlighted in [12]. Those exploit-focused dynamic mitigation techniques are not completely effective against buffer overflow; those defect-focused dynamic mitigation techniques such as runtime
bounds checking are not broadly deployed because of the high performance costs and application compatibility problems.

**Heap corruption.** Heap data structure for dynamic memory allocation contains user data and heap management data. For performance reasons, user data and management data are normally mixed together in memory. There are certain operations on management data involved with each malloc()/free() call. Heap vulnerabilities, such as heap buffer overflow and double free, allow malicious users to corrupt these management data to cause arbitrary memory write.

**Format string.** The use of user input as the format string parameter in certain C functions, such as printf(), may incur security problems. A malicious user may use %s and %x to read sensitive memory content (i.e., information leaking) or even use %n to write arbitrary data to arbitrary memory locations.

In summary, attackers often exploit software flaws to compromise software systems. Low-level vulnerabilities such as those above are all effective examples, which until now people are still struggling to defeat [12]. Although cryptographic methods can be used to protect data confidentiality and integrity, they cannot eliminate software vulnerabilities inside modern systems (For example, the integer overflow as pointed out in [16]).

### 2.1.2 Taint Tracking and Checking

Program defects exist and it is always hard to stop attackers from exploiting them. It is also well-known that most current attacks are control data attacks, namely hijacking the control flow of victim programs. It is this final step that attackers follow to break systems and also the final line people can defend their systems from being compromised. For a successful break-in, certain value to be assigned to the intended control data needs to be provided by outside
attackers. Based on this observation, *Dynamic Information Flow Tracking* [17] and *Minos* [6] propose using taint tracking and checking technique to stop the final step. Figure 1(a) shows the normal attack procedure and Figure 1(b) shows their basic taint tracking and checking protection mechanism. Here a data value becomes tainted if it is arithmetically derived or simply copied from tainted data. Their scheme is effective because normally the program counter data value is not supposed to be tainted. After their work, there have been several taint-related hardware and software proposals adopting the similar taint tracking and checking idea [10], [4], [3], [9] and [13]. In addition, Chen et al [3] proposes to add pointer taintedness checking mechanism, which makes sure that the address value for a memory access is not tainted. Their scheme is based on the observation that non-control data attacks using data pointers are as effective as the well-known control data attacks.
Our proposed architecture is closely related to TaintCheck [10] and Vigilante [4]. In their work, taint analysis code is added into programs at a fine-grain level by binary instrumentation tools. Through program instrumentation, taint tracking mechanism is implemented and taint checking is performed on the program counter, function call parameters, etc. In comparison, our scheme aims to exploit instruction-level taint behavior and provides a generic fine-grain hardware infrastructure for preventing non-control data attacks. Flexible taint checking policies can be built upon it with minor performance overhead (see Section 2.5).

2.1.3 Non-Control Data Attack Examples

As we can see from Figure 1, current taint checking mechanism is only for control data protection. However there are many cases that non-control data are also important for software

```c
void do_auth(char* passwd)
{
  char buf[40];
  int auth;

  if (!strcmp("encrypted_passwd", passwd))
    auth = 1;
  else auth = 0;
  scanf("%s", buf);
  printf(buf); //format string!
  A: if (auth)
    access_granted();
}

void information_leakage()
{
  unsigned int limit = 50;
  char buf[20];
  int p[50];
  unsigned int i;

  scanf("%s", buf); //buffer overflow!
  B: for (i=0; i<limit; i++)
    printf(" %x ", *p+i));
}
```

Figure 2. Examples of non-control data attacks
security. Figure 2 shows two synthetic cases where certain degree of damage can be done by corrupting non-control data. In Figure 2(a), the program is in danger if the critical variable “auth” is overwritten by a format string attack. In Figure 2(b), unexpected extra program data will be revealed out if the counter value “limit” is changed by a buffer overflow attack. Since neither of them involves control data change, attacks in both examples cannot be detected by the original taint checking scheme. Pointer taintedness checking [3] may detect the format string attack in example (a) if the attack embeds the target address in the format string itself. However, some format string attacks may use untainted data as the target address [7]. Neither such format string attack nor the buffer overflow in example (b) can be detected by pointer taintedness checking [3] since there are no tainted pointers.

Using our architecture, instructions corresponding to the statements at line A and line B are all Taintless-Instructions since variables “auth”, “limit” and “i” are not supposed to be tainted. Attacks in the examples cause “auth” and “limit” become tainted because of taint tracking. Thus they will be detected when the processor executes the instructions corresponding to the line A or the line B. If a format string attack uses untainted data as both the target address and the target value, we resort to taintless instructions in the function vfprintf (See Section 2.4).

2.2. Design

Figure 3 shows our Taintless-Instructions-Profile-Enhanced processor-memory model. The data taintedness tracking procedures and associated hardware units are similar to the ones in the original taint tracking scheme [17],[6]. The operating system sets taint tag bits to suspicious input data and forward the data along with tags into program space. The processor fetches,
computes and stores data along with taint tags back and forth from the memory. The special T-ALU unit is used for data taintedness tag computation. To support instruction-level taint checking, however, shaded structures have been added. Meanwhile the taint tag (T) for instructions has a different meaning from the taint tag (T) for data, although in the main memory there is still just single extra tag bit for every memory storage. The tag for data indicates whether a value is derived from suspicious outside data. Setting the tag bit to 1 indicates that the value is tainted. The tag for instructions indicates whether an instruction is a Taintless-Instruction. An instruction is called a Tainted-Instruction if it is supposed to deal with tainted data. Otherwise it is called a Taintless-Instruction. Setting the tag to 1 indicates that the instruction is a Taintless-Instruction. During runtime execution, the taint tags of instructions are buffered inside the processor core, e.g., in the re-order buffer (ROB). Finally, when a committing Taintless-Instruction meets some tainted operand, an exception will be generated through the AND gate.
Whether an instruction is a Taintless-Instruction or Tainted-Instruction must be determined before program runs. It can be collected either through manual annotation, static analysis or dynamic training. A programmer may mark special instructions manually if the data involved is deemed to be very important and also not supposed to be tainted. This is very useful when there is not enough static or dynamic information available. Although its coverage is not as broad as those automatic methods, very important program execution points can be protected. In this chapter, we show the effective applications of our architecture mainly through manual annotation. Static analysis can help to identify taint program variables and therefore those corresponding instructions. Dynamic training can help to expand coverage when program source code is not available or cannot provide enough information.

During runtime execution taint checking is carried out in the following four steps:

1. load the collected Taintless-Instructions profile along with program code into the main memory.
2. tag data from suspicious input channels as tainted.
3. track taintedness propagation through execution.
4. raise an alarm when a Taintless-Instruction encounters some tainted operand.

In summary, our proposed scheme defines a new generic instruction-level taint checking architecture for protecting software systems from outside attacks. The instruction-level taint checking can be used to protect against a wide range of attacks, e.g., non-control data attacks as in the examples in Figure 2. It requires minor changes to the original taint tracking architecture.
2.3. Experiments

2.3.1 Experimental Methodology

We implement our scheme on the SimpleScalar processor simulator [1] with the PISA instruction set to study the instruction-level taint behavior. Since our scheme is concerned with protection on both control data and non-control data, it is very important to maintain accurate taint tracking in order to achieve good data coverage. We choose to implement per byte tagging by adding one additional taint bit to each byte. The byte granularity fits into the behaviors of most applications dealing with outside data if not all. It can provide enough accuracy without sacrificing much performance. During program load, taintedness bits corresponding to program code are initialized according to the collected profile and taintedness bits for program data are cleared. Those data coming from certain I/O system calls like READ, RECV, etc. are tagged as tainted. As for taintedness propagation, the common logic is that the destination taintedness bit is the bitwise OR of the corresponding taintedness bits from source operands. In addition, special attention has been paid to certain ALU operations. Shift instructions cause taintedness bit to shift correspondingly. Any untainted byte of value 0 with AND instructions causes taint bit 0 for the outcome byte. Any untainted byte of value “0xFF” with OR instructions causes taint bit 0 for the outcome byte. These rules are borrowed from [3]. Finally a security exception is raised whenever a Taintless-Instruction meets some tainted operand.

2.3.2 Preliminary Results from SPEC CPU2000

Intuitively, in most programs, there is a large portion of program code dealing with untainted data. Our architecture will be of limited use if the majority of program instructions are
Tainted-Instructions. To validate our observation, we have used the SPEC CPU2000 CINT benchmarks for experiments and run them to completion using the reference inputs. The instructions are classified as either Taintless-Instructions or Tainted-Instructions. If there are multiple reference inputs, we combine the multiple classifications together. If an instruction is identified as a Tainted-Instruction in one run while a Taintless-Instruction in another, the combined result is a Tainted-Instruction. Figure 4 shows the final statistical results. Because of limited number of reference inputs, many instructions of the programs have not been exercised. The statistical results may not be complete. However it shows that there are a significant amount of Taintless-Instructions (average approximately 46% Taintless-Instructions against 11% Tainted-Instructions).

Figure 4. The percentages of Taintless-Instructions and Tainted-Instructions in SPEC CPU2000 CINT
2.4. **Application Examples**

By augmenting program code with extra taint information, our design provides an instruction-level taint checking architecture. It can be used to detect various security attacks at different levels. In this section, we demonstrate the usage of our architecture on detecting buffer overflow and format string attacks through manual annotation.

2.4.1 **Buffer Overflow**

Figure 5 shows the way to use our architecture to detect buffer overflow. One padding data “pad_begin” is added right before the buffer and one padding data “pad_end” is added right after the buffer. Then the simple operation of “++” is inserted to access those padding data after the suspicious buffer operation (“memcpy” at 0x00401cb8). The instructions for “++” are from

```
void vuln_stack_function_ptr(int choice){
    volatile unsigned int pad_begin = 0; // padding data
    volatile unsigned int pad_end = 0; // padding data
    void (*stack_function_pointer)(void);
}

00401cb8 memcpy(stack_buffer, overflow_buffer, overflow+4);
00401c0 pad_begin++;
    // access padding data
| pad_end++;
00401cf8 /* function call using the function pointer */
00401d0 (void) (*stack_function_pointer)();
00401d38
```

```
(a)

00401cb0 <vuln_stack_function_ptr+2a0> jal 00404c00 <memcpy>
00401c0 <vuln_stack_function_ptr+2b0> lw $v0[2],32($sp[29])
00401c8 <vuln_stack_function_ptr+2b8> addiu $v0[2],$v0[2],1
00401d0 <vuln_stack_function_ptr+2c0> sw $v0[2],32($sp[29])
00401d8 <vuln_stack_function_ptr+2c8> lw $v0[2],104($sp[29])
00401ec8 <vuln_stack_function_ptr+2d8> addiu $v0[2],$v0[2],1
00401ef0 <vuln_stack_function_ptr+2eo> sw $v0[2],104($sp[29])
00401f8 <vuln_stack_function_ptr+2e8> lw $v0[2],104($sp[29])
00401f0 <vuln_stack_function_ptr+2f0> lw $v0[2],104($sp[29])
00401f8 <vuln_stack_function_ptr+2f8> jalr $ra[31],$v0[2]
```

(b)

Figure 5. Buffer overflow attack detection
0x00401cc0 till 0x00401cf8 and they are all Taintless-Instructions. Once there is a buffer overflow in “memcpy”, tainted input data will be copied across through “pad_begin”, “pad_end” and cause them to be tainted. Therefore an exception is raised to signal the danger. Note that the statements “pad_begin++” and “pad_end++” are used to illustrate the padding data accesses at the source code level. The compiler-generated padding data accessing instruction may simply be a single load instruction rather than the instruction sequence presented in Figure 5b.

The example code shown in Figure 5a is manually modified from the testbed of twenty buffer overflow attacks developed by John Wilander [18]. Besides padding data and corresponding access code, we construct the “overflow_buffer” through I/O instead of the original array copy so that “overflow_buffer” data are tainted. “jmp_buf” related code is also changed since the SimpleScalar package provides the old version of “jmp_buf” data structure. Using our Taintless-Instructions-enhanced SimpleScalar simulator, we have tried all attack forms in the testbed using similar modifications and successfully identified all the attacks.

Besides protecting buffers in the stack and data segment, heap buffer overflow can be effectively detected in a similar way by modifying the structure of memory chunks and the management routines. The in-band heap management data is enclosed by two padding data. Any double linked list manipulations which take place during the heap management process are prefaced with Taintless-Instructions accessing those padding data. This idea is similar to the ones in [14]. To use our architecture to effectively detect all buffer overflow though, it is important to access the correct padding data right after suspicious buffer operations.

Compared to other canary protection schemes such as Stack Guard [5], ProPolice[8], the above scheme based on our architecture has both performance and security advantages. It is known that Stack Guard and ProPolice are based on the guard canary and the checking code.
With the compiler help, our scheme requires a much fewer number of checking instructions to be executed. Instead of the need for loading the canary, computing values for complex canary schemes, comparing the values, our scheme will only require one load instruction. Our scheme is more secure in the fact that the padding data are not required to be secret. In comparison, the canary must remain secret for other canary schemes to be effective. Otherwise attackers can overwrite the canary with a carefully-chosen value without being detected. For example, if the original canary is a random number, attackers can perform buffer overflow successfully by overwriting the canary with the same random number. Attacks such as above are possible given the existence of vulnerabilities which may lead to information leakage. Under our architecture, the padding data would be checked by Taintless-Instructions. Even when it is overwritten with a carefully-chosen value by tainted data, the attack will still be detected.

In summary, our architecture can provide better support for canary schemes against buffer overflow attacks. In addition, if combined with control data taint checking and pointer taintedness checking, we expect that it is very difficult to bypass the final checking code without being caught. Thus even some buffer overflow attack just corrupts non-control and non-data-pointer values, it can still be detected.

2.4.2 Format String

Format string vulnerability may occur when user inputs are used as format arguments in a printing function. The examples include `sprintf`, `snprintf`, `fprintf`, `vprintf`, `vsnprintf`, `vfprintf`, `syslog`, and `vsyslog`, etc. Among them, `vfprintf` is the basic function on which the other wrapper functions are based. Figure 6(a) shows a part of source code in `vfprintf` from the C library in the SimpleScalar simulator package. Pointer “f” points to the format string and is used
to interpret all kinds of supported specifiers including the famous “%n” in the “while” loop. To detect the attack, the instruction at “00402f68” is marked as a Taintless-Instruction. In other words, user inputs are not allowed to be used as format strings. Each time a tainted input is used as a format string, it will raise an alarm. Similar actions can also be taken to single out the “%n” write attacks. In summary, our architecture enables taint checking on arguments of critical functions such as `vfprintf`, system calls to provide stronger security protection.

2.5. Discussion and Conclusion

2.5.1 Overhead to Taint Tracking Architecture

Since our scheme is based on the original taint tracking architecture, there is no change to the memory system and the processor pipeline dealing with program data and corresponding taint tags. The overhead is related to the taintedness bits associated with instructions, which are the shaded structures and related data paths in Figure 3. All additional computation and
propagation happen in parallel, not on critical path and should not affect the number of pipeline stages and the cycle time. The additional software processing will only occur during the load of program code. Thus our scheme would introduce minor overhead to the original taint tracking architecture, while it can be used to provide a higher degree of security protection.

2.5.2 Collection of Taintless-Instructions Profile

The Taintless-Instructions profile is essential to provide security protections based upon our architecture. We have already shown in this chapter that a manually-annotated profile can be used to protect some security critical data structures. However, the security coverage provided by this method is limited. The preliminary results from experiments on SPEC CPU2000 benchmarks in Section 2.3 have shown that there are a significant amount of Taintless-Instructions. Once those Taintless-Instructions are identified, higher security coverage can be achieved.

2.5.3 Vulnerability and Shortcoming

Under our architecture, each instruction is either a Taintless-Instruction or a Tainted-Instruction. However a Tainted-Instruction does not always deal with tainted data due to the dynamic feature of program executions. Thus our scheme may fail to provide protection to certain untainted data which is handled by Tainted-Instructions. Also the Taintless-Instructions profile does not provide information about the pointer taintedness. Even combined with pointer taintedness checking scheme [3], our architecture still faces some challenges as highlighted in [7]: code overwrite attacks for writable memory region and vulnerabilities coming out of
translation tables. Although the first challenge can be solved through instruction taintedness checking, our current Taintless-Instructions profile is not applicable in that case.

2.5.4 Conclusion

This chapter proposes a new generic instruction-level runtime taint checking architecture. It requires minor changes to the original taint tracking architecture and it can complement other taint checking techniques to provide a higher degree of security protection. The effective usages of our architecture for buffer overflow and format string attack detections are presented as examples in the chapter.
List of References


CHAPTER 3. IMPROVING PRIVACY AND LIFETIME OF PCM-BASED MAIN MEMORY

Phase change memory (PCM) is an emerging memory technology, which features low access latency, byte-addressability, high integration density, and low leakage energy consumption. Recently, there have been strong interests in using PCM as main memory (PRAM) in computer systems [31][38][45].

Despite of having the aforementioned strengths, PCM raises new challenges for computer system design. One key property of PCM is its non-volatility: the information stored in PRAM is persistent without power supply and may last for more than 10 years [36]. Although non-volatility is a fundamental reason for power efficiency, it makes PRAM much more vulnerable to malicious security attacks than volatile DRAM. Another important issue of PCM is its unreliability, particularly due to its limited write endurance[36], and various wear-leveling techniques have been proposed to extend the PRAM lifetime [22][31][37][38][43][44][45].

In this chapter, we adopt a hardware encryption scheme, counter-mode encryption, to protect the privacy of PRAM, given its proven security and high performance. However, due to the diffusion characteristics of encryption algorithms, values in encrypted data blocks are randomized. This negates the effectiveness of some previously proposed wear-leveling techniques, redundant bit-write removal [45] and partial writes [31] in particular.

To mitigate the encryption impact upon wear-leveling techniques, we propose two new schemes. First, we propose simple yet effective extensions to the encryption scheme to revive partial writes. Second, we propose to use encryption counters as age counters and to dynamically
adjust error protection strengths. In this work, we use error correction code (ECC) and design an efficient way to manage ECC protection. Our experiments using a cycle-accurate timing simulator show that the performance overhead introduced by encryption and ECC is small given the long latency of PRAM accesses.

Our main contributions include (1) to our knowledge, this is the first work to investigate the impact of encryption on wear-leveling techniques for PRAM; (2) we propose a new encryption counter scheme to revive partial writes to reduce write traffic; (3) we propose to leverage encryption counters as age counters and design an efficient way for ECC management to extend PRAM lifetime.

3.1. Background and Related Work

In PCM, a memory cell can be transformed between a low-resistivity state and a high-resistivity state by atomic arrangements. Compare to FLASH memory which has the block-erase requirement, PCM is typically byte-addressable. Besides, PCM has better write endurance than FLASH and is projected to be more scalable, more cost-effective and of higher performance [38][45]. In this section, we discuss the challenges for using PCM as main memory (PRAM) and some can be applied to the FLASH technology as well.

3.1.1 Privacy

The non-volatility nature of PRAM aggravates the privacy concerns over the contents residing in main memory. It was reported that secret keys for disk encryption can still be retrieved from volatile DRAM even minutes after a computer is powered off [28]. Attackers may simply dump the plaintext image of PRAM to extract critical information. Such one-time storage
dump is also a major security concern for disk storage systems and disk encryption is often used for security protection [26]. In a more strict security model, it is assumed that more complex security attacks exist and attackers have the abilities to monitor the dynamic data read from and stored to main memory. Protection against such advanced security attacks is addressed with secure processor architectures [32]. The security of those secure architectures relies on the assumption that processor cores are unbreakable. Secret keys involved are generated or sealed inside processors. Given the privacy issues with PRAM, we opt to use the counter mode encryption for its proved security and high performance [21] and assume that the secret keys are inside processors.

Figure 7 shows the counter-mode encryption proposed for secure processors [21]. The counter data block is composed of a counter, two address offsets and a logical page identifier (LPID). The counter is associated with a cache line and is incremented by one every time when the cache line is written back to main memory. One address offset is the cache line offset within a page. The other is the plaintext data block offset within the cache line when the cache line size is larger than the size of an encryption/plaintext data block. The LPID is assigned uniquely for each allocated page in the main memory. The security strength of the counter-mode encryption

![Counter-mode encryption for secure processors](image)

**Figure 7.** Counter-mode encryption for secure processors
comes from the fact that the value of the counter data block for each plaintext data block is unique both spatially and temporally. The LPID and address offsets provide spatial uniqueness while the counter ensures temporal uniqueness. For resource efficiency, counters of limited sizes are used. As a result, when a counter overflows, a new unique LPID is generated and the whole page would be re-encrypted to ensure uniqueness of counter data blocks [21]. The performance advantage of the counter-mode encryption is that the block cipher (i.e., encryption) latency can be overlapped with the latency of fetching the encrypted data block.

3.1.2 Wear-leveling Techniques

Various wear-leveling techniques have been proposed to address the limited endurance of PRAM to extend its lifetime. The granularity of the approaches can be at the segment/page level, the cache-line level, and the individual bit level. They can be classified into two categories. One is using swapping/shifting/rotation to even the wear-out among hot (more write accessed) and cold (less write accessed) spots. For example, segment swapping [45] happens when one segment becomes so hot that it has to be swapped with some cold segment to even out wear-out. Similarly, in the start-gap scheme [37], each cache line would be rotated through the whole memory with the support from one spare cache line. The wear-leveling techniques for FLASH devices by manipulating the mapping between logical blocks and physical FLASH memory blocks also fall into this category. The other category is to reduce write traffic to PRAM. For example, line-level write back [38] only writes dirty cache lines instead of a whole page to PRAM. Similarly, partial writes [31] add dirty bits to track which words of a cache line are dirty. When the cache line is replaced, only the dirty words in the cache line have to be written back to PRAM. Redundant bit-write removal [45] first reads out and compares the existing content to the
new content. Only those bits that differ are written back actually. Since many of the above approaches are orthogonal to each other, various schemes are often combined to achieve better lifetime improvement. The wear-leveling schemes based on reducing write traffic suit better for PRAM due to its byte-addressability than FLASH, in which the block-erase requirement makes it difficult to exploit bit-level or cache-line-level redundancy.

3.1.3 Reliability, Lifetime and ECC

To store information, PCM material is heated to change state under electrical pulses. The repeated heat stress, however, could render PCM material unstable and unreliable. Data retention, endurance, program and read disturbs are some of the basic reliability aspects that have been investigated recently [30][36]. In this chapter we assume that there exist failing mechanisms or process variations causing some PRAM cells fail earlier than others [27].

Since PCM may have failures at some point, we need a scheme to detect and correct errors in PRAM. Error Correcting Code (ECC) is a common mechanism for such a purpose [29][34][42][44]. ECC achieves information redundancy by generating redundant bits based on the data to be protected. The more bit errors that need to be corrected, the more ECC

---

![Figure 8. The impact of ECC correctability on bit error rates (based on data block size of 64 bytes)](image-url)
bits are required. Figure 8 shows the bit error rate before (raw bit error rate) and after (uncorrectable bit error rate) ECC protection. The figure shows that ECC protection greatly reduces the bit error rate, often at a few orders of magnitude. The implication on PRAM is that the failure of PRAM would be postponed dramatically. For example, with the data traffic to PRAM at 4GB/second, without ECC protection, a PRAM with the raw bit error rate at $10^{-8}$ would have one bit error after around 3 milliseconds ($=1/(4G*8*10^{-8})$ seconds) of usage. With ECC capable of correcting 1 bit error, the uncorrectable bit error rate becomes $2.6*10^{-14}$, which means that the PRAM would not encounter one bit error until after almost 20 minutes ($=1/(4G*8*2.6*10^{-14})$ seconds) of usage.

3.2. Privacy Protection for PRAM

3.2.1 The Impact of Encryption on PRAM Wear-leveling Techniques

We argue that encryption is necessary for protecting privacy of non-volatile PRAM. However, to our knowledge, no prior work, especially the wear-leveling techniques, have investigated the impact of encryption. To analyze the impact, it is necessary to dissect certain characteristics of modern encryption algorithms.

Between two main classes of encryption algorithms, symmetric and asymmetric-key encryption, symmetric-key encryption is often chosen for its high-speed. Between stream ciphers and block ciphers in symmetric-key encryption, block ciphers are often used in memory/storage encryption. With the basic encryption unit being a data block, block ciphers produce an output block of the same length as an input block. The size of one block for encryption is often smaller than the size of a single cache line/block. For example, Advanced Encryption Standard
(AES) [23] supports the encryption block size of 16 bytes while one last-level cache line/block may have 64 or more bytes. So, a cache line contains 4 or more encryption data blocks. To ensure security, there is one important principle for block cipher design – diffusion [33]. Diffusion tries to disperse the statistical characteristics of plaintexts into long spectrum of ciphertexts. In other words, each plaintext bit would affect many ciphertext bits. The result is that there is no relationship between two ciphertexts even there are just few different bits in their plaintexts.

![Diagram](image)

Figure 9. Avalanche effect caused by encryption

The diffusion characteristics have severe impacts on some previously proposed wear-leveling techniques. Since the encryption block size is less than the size of one cache line/block, encryption mainly affects wear-leveling techniques below the cache line granularity. Schemes on or above the cache line granularity such as segment swapping [45], the start-gap [37] and the line-level write-back [38] are not affected by encryption. For redundant bit-write removal [45], with encryption, the new ciphertext data values would be largely different from the old ciphertext data values. Figure 9 shows such an example with AES encryption. Even if the to-be-
written-back plaintext data block is the same as the old one, the two ciphertext data blocks are largely different from each other due to the encryption counter update. Therefore the effectiveness of the redundant bit-write removal is greatly reduced. For the partial-write scheme [31], the problem with encryption is that the encryption counter for a cache line is incremented for every write back. As a result, the whole cache line needs to be re-encrypted and modified, thereby completely disabling partial writes even if there is only one dirty word in the cache line.

3.2.2 A New Encryption Counter Scheme to Mitigate the Encryption Impact

Based on the analysis in Section 3.2.1, we propose to extend the original encryption counter scheme to revive the partial-write wear leveling. Our extension includes additional counters at the encryption-block granularity. In other words, for each cache line, besides one cache-line-level counter, we add multiple block-level counters. Encryption for each data block is done using the combination of the cache-line-level counter and the block-level counter. Upon a write-back, only the block-level counters corresponding to dirty blocks are incremented and only the dirty blocks are re-encrypted. Other non-dirty blocks within the same cache line can therefore be spared from being written back to PRAM. When any block-level counter overflows, however, all block-level counters in the same cache line are reset to zero and the cache-line-level counter is incremented by one. In this case, the partial write scheme does not work as the whole cache line is re-encrypted. Note that since the basic encryption unit is an encryption block (typically 16 bytes), finer granularities (e.g., word size) for partial writes are not beneficial as the whole data block will be encrypted even if only one word in the block is updated.
In Section 3.5.2, we show the quantitative impact of encryption on the wear-leveling techniques. The effectiveness of our new encryption counter scheme is shown in Section 3.5.3. Note our new scheme revives partial writes to reduce write traffic to PRAM and it alone may not be sufficient to improve lifetime. Some rotation scheme is necessary to distribute write traffic evenly in order to take full advantage of the write traffic reduction.

3.3. Adaptive ECC Management

A straightforward way to extend PRAM lifetime using ECC is allocating enough ECC storage to cover the maximum number of bit errors that are expected. However, during the most of the PRAM lifetime, the number of bit errors is much less than the expected maximum number. Therefore it is wasteful in space and potentially harmful to the performance. The reasons are (a) some memory space are allocated for unnecessary ECC storage and (b) the logic for correcting a high number of bit errors is slower than it for a small number of bit errors.

![Figure 10. Dynamic requirement for ECC to meet UBER of 10^-19 as PRAM gets worn out](image)
In this chapter, we propose to dynamically manage ECC strength according to the reliability/wear-out status of PRAM. To keep track of the wear-out status of PRAM, we use the number of write-accesses to PRAM as a metric. We assume that the raw bit error rate increases as PRAM ages [27]. Therefore, we gradually increase the strength of ECC protection by allocating more ECC bits when PRAM is gradually worn out. Figure 10 shows one such example in which the number of bit errors to be corrected by ECC is increasing as the raw bit error rate is increasing. The target for the uncorrectable bit error rate (UBER) in Figure 10 is $10^{-19}$, which enables a PRAM with data traffic at 4GB/second to have just one bit error after around 10 years ($=1/(4G*8*10^{-19})$) of usage.

For efficient ECC management, two main challenges need to be addressed. The first is where to store the ECC bits as the size varies for different error correction requirements and how to access them accordingly. The second is to obtain the write-access counters as they are necessary for monitoring the wear-out status of PRAM.

3.3.1 The ECC Space and Address Mapping

To accommodate dynamic ECC management, we propose a hardware-software integrated approach, shown in Figure 11. Instead of having separate memory chips for ECC, we propose to store both data and their ECC bits in a unified memory space (i.e., virtual memory space). The operation system (OS) will assist the dynamic allocation of ECC and data pages and maintain information of the ECC pages so that the data pages will not be mapped to the same places. The virtual memory is partitioned into groups and each group contains multiple data pages and the corresponding ECC pages. All the data pages in the same group have the same level of ECC protection. The assumption is that some aforementioned rotation-based wear-leveling techniques
are deployed and they introduce evenly or close to evenly distributed write traffic to PRAM. Our experiments are based on such an assumption. When some pages in a group are getting older and reach a threshold, more ECC space is required. In this case, an exception would happen and OS would intervene to reorganize the group to contain less data pages and more ECC pages. Note that since such exception event is rare, the introduced performance overhead would be negligible. Depending on workload write-traffic characteristics, the effectiveness of rotation-based wear-leveling to distribute the write traffic, and process variations, different groups may have different numbers of ECC pages according to their wear-out status. As shown in Figure 11, group 0 may have a data pages and (N-a) ECC pages while group i may have k data pages and (N-k) ECC pages. Eventually, the ratio of the number of ECC pages against the number of data pages in a group will reach the worst case, where the maximum number of bit errors is expected. In such a case, the group may be marked unreliable and the OS will not use it anymore.

Managing memory in groups simplifies address mapping to access the ECC bits. As shown in Figure 11, the memory controller has a protection-level look up table which contains the information of how many ECC pages exist in each group. Such information is then used to derive the number of ECC bits for a cache-line in the group. The space cost for the protection-
level lookup table is small. For example, if each group contains 1K pages and the page size is 4KB, a 1K-entry table can manage a 4GB-PRAM. The process of determining the physical address of ECC bits of a data access is shown in Figure 12. First, the physical address of the data \((data\_cache\_line\_addr\) in Figure 12 as the unit of data operation is the cache line size\) is used to decide which group contains the data \((ECC\_base\_addr\) in Figure 12\). Then, the offset within the group \((ECC\_offset\_addr\) is calculated to see where the data is located within the group at the cache line granularity. Based on how many ECC bits are allocated for each cache line, the address of the ECC bits is generated. Note that due to the use of the two's power numbers, the multiplication and division in Figure 12 can be implemented using simple shifts and adds. The group size is mainly dependent on the PRAM endurance characteristics. Since all the data in the same group have the same ECC, we essentially assume that the memory in a group will wear out at a similar rate. If the endurance variation is expected to be large, we prefer a small group size to avoid a small region to affect a large amount of memory. Note that although an ECC page is referenced more frequently than data pages (as each ECC page can accommodate multiple data pages), it is unlikely for ECC pages to become most worn out ones. The reasons are two-folds. First, as shown in Section 3.5.6, after encryption, both data and ECC code have similar bit-level redundancy and bit-level wear-leveling techniques have the similar effects. Second, inside an ECC page, each individual ECC block has the same number of writes as the corresponding data block (e.g., a cache line). So the lifetime of an ECC page will be the same as (or very close to) the data page which is most worn out.
Storing ECC bits in virtual memory space is independently proposed in a recent work [41]. An ECC page table is introduced in their scheme to locate the ECC pages.

3.3.2 Leveraging Encryption Counters as Write-access (Age) Counters

There are several ways to track the number of write-accesses to memory pages in PRAM. One is to associate a local counter with each cache line. The maximum among all local counters in a page would be the age of the page. Another way is to have a single counter for one page and it records the total number of write-accesses to the page. The first approach incurs high space overhead (local counter size * number of lines in a page). The second approach has much less space overhead (just one counter) but less accurate since write-accesses to different lines in a page are accumulated, leading to a highly overestimated age. To reduce space overhead without losing much accuracy, we choose to use a two level counters scheme similar to the one used in [40]

In the counter-mode encryption described in [21], there already is a local counter (LC) for each line in a page. In addition, there is a 64-bit logical page identifier (LPID) assigned for each memory page when it is allocated. To account for a high number of write accesses, we add another global counter (GC) for each page. When a local counter overflows, GC would be incremented by one and all the local counters in the same page would be reset to zero. In this case, as discussed in Section 3.1.1, a new unique LPID is generated and the page would be re-

$$
\text{ECC\_addr} = \text{ECC\_base\_addr} + \text{ECC\_offset\_addr} \times \text{ECC\_size\_for\_one\_cache\_line}
$$

$$
\text{ECC\_base\_addr} = \text{data\_cache\_line\_addr} \& \text{group\_size\_mask} \quad \text{/* group\_size\_mask = ~(group\_size-1) */}
$$

$$
\text{ECC\_offset\_addr} = \left(\text{data\_cache\_line\_addr} \& \left(\lnot \text{group\_size\_mask}\right) - \text{Data\_base\_addr\_in\_the\_group}\right) / \text{cache\_line\_size}
$$

$$
\text{ECC\_size\_for\_one\_cache\_line} = \text{the number of ECC bits for the protection\_level\_for\_the\_group}
$$

$$
\text{Data\_base\_addr\_in\_the\_group} = \text{number of ECC pages in the group} \times \text{page\_size}
$$

Figure 12. The address mapping of ECC protection bits
encrypted to ensure security [21]. Such a combined two-level global and local counters are much more accurate than one global counter and have much less space overhead than the local counter scheme. Note as the overflow happens rarely, the associated performance overhead is negligible [40].

3.4. Overall Architecture

The overall architecture for improving privacy and lifetime is shown in Figure 13.

Figure 13. The architecture of the proposed scheme for privacy protection and lifetime improvement
With the proposed architecture, a memory access proceeds as follows. When an encrypted cache line is to be fetched from PRAM, its memory address is used to locate the corresponding counters and generate the seed for the block cipher. As the counters are stored along with the ciphertext data in the PRAM, directly accessing them will postpone the seed generation process and expose the block cipher latency. To overcome this performance issue and overlap this latency with PRAM access latency, a counter cache is included as in the previous work [21]. Here, note that if the counter scheme proposed in Section 3.2.2 is used, each local counter (LC) will be appended with a few small block-level counters. The encrypted data are stored in a data page in PRAM. The ECC bits are stored in an ECC page in the same group as the data page. In our scheme, ECC and data pages have the same organization and the counters in either type of pages are updated when there is a write access. The only difference is that the counters in the ECC page are only used to track the age and not for decryption.

There is an interesting interaction between encryption and ECC generation. Two options exist: the ECC bits can be computed either based on plaintext data or ciphertext data. If ECC is computed using plaintext data, the write access time can be reduced as the ECC computation and encryption can be performed in parallel. On the other hand, the read access latency is affected as the ECC check has to wait after the ciphertext data is decrypted. In comparison, computing ECC bits on ciphertext data reduces the read latency and increases the write latency. Since read operations are more performance critical than writes, we choose to compute ECC based on ciphertext data, as shown in Figure 13.

In an alternative memory organization, which uses PCM as main memory and DRAM as another level of cache [38], the counter cache can reside in the DRAM cache. This DRAM cache
can also be used to store the uncompressed data when memory compression technologies [19][25] are employed. Since encryption makes data less compressible, encryption should happen after the compression stage. In this case, the plaintext data shown in Figure 13 is a compressed data block. The impact of memory compression is evaluated in Section 3.5.4.

The storage overhead for counters, which are used for both encryption and age estimates for ECC management, is small. If we assume the cache line size as 128 bytes in the last level cache, a 4KB page size, a 64-bit LPID, a 2-bit counter per encryption block, a 13-bit LC per line and a 32-bit GC, the overhead is around 3%. The selection of the counter sizes is to make sure that the age counters can record the number of writes that is beyond the endurance of PCM device \((10^8\text{ to } 10^9)\) [37].

3.5. **Experimental results**

3.5.1 **Methodology**

Our experiments are conducted using a cycle-accurate timing simulator developed upon the SimpleScalar toolset [20]. The underlying processor model is MIPS R10000 and the default configuration is listed in Table 1. The L2 cache size is set to 1MB to increase the memory traffic. The PRAM access latency is 1024 cycles [37]. The block cipher engine is a 128-bit AES cipher and the encryption latency is assumed to be 80 cycles [21]. Each cache line in the counter cache stores the counters for one page. It is composed of a 64-bit LPID [21], multiple local counters and one global counter (GC). The error correction code used is a binary cyclic code (BCH) [35]. The BCH latency depends on the number of bit errors that can be corrected and the maximum latency is 120 cycles for correcting 8 bit errors [29][44]. For each message data of \(k\) bits, a BCH
codeword (containing both data and the redundant ECC bits) with a length of n bits can be constructed to correct up to t bit errors out of the entire codeword. The length of the codeword n should satisfy $2^{(m-1)}-1 < n \leq 2^m-1$ and $m*t \leq n-k$, where m is the minimum number of redundant ECC bits required for every bit error correction. In our experiments, 4 BCH codes are interleaved to protect the data at the granularity of the last-level cache line size (256 bytes). For each BCH codeword, $k = 512$ bits (64 bytes) and $m = 10$. Therefore $n-512 \geq 10*t$ must be satisfied. It indicates that each additional bit error correction would need an additional 10 redundant ECC bits.

<table>
<thead>
<tr>
<th>Table 1. Default processor configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Predictor</td>
</tr>
<tr>
<td>Superscalar Core</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Execution Latencies</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Instruction Cache (private)</td>
</tr>
<tr>
<td>L1 Data Cache (private)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>L2 Unified Cache (shared)</td>
</tr>
<tr>
<td>Counter Cache (shared)</td>
</tr>
<tr>
<td>Cipher Engine</td>
</tr>
<tr>
<td>ECC</td>
</tr>
<tr>
<td>PRAM</td>
</tr>
</tbody>
</table>

Memory-intensive benchmarks from SPEC2000 and SPEC2006 with high cache miss rates are used in the experiments. For lifetime analysis, an in-order processor model is used for simulation speed and the benchmarks are simulated to run for a hundred-billion instructions or upon completion.
3.5.2 Impact of Encryption on Wear-leveling Techniques

In this section, we show the impact of encryption on two wear-leveling techniques: redundant bit-write removal and partial writes.

First, we analyze the write traffic of the SPEC benchmarks. We collect the total number of bit-write traffic to PRAM under three scenarios. The baseline is the one without either redundant bit-write removal or encryption. The other two are redundant bit-write removal with and without encryption, respectively. Assuming uniform bit writes across PRAM, Figure 14 shows the impact of encryption. Without encryption, there are lots of bit-write redundancies in the benchmarks. The highest one, mcf, has around 99.9% of its total bit writes redundant. Even for the lowest one, equake, the redundant bit writes are around 68%. In contrast, with encryption, every benchmark has only around 50% of total bit writes redundant. On average using the geometric mean (Gmean), redundant bit-write removal can save around 90% of the bit-write traffic to PRAM. With encryption, however, it would only save half of bit-write traffic to PRAM.
As discussed in Section 3.2.1, encryption completely removes the benefit of partial-write. In this experiment, we first confirm the benefits of partial writes in reducing write traffic when encryption is not used. The traffic reductions normalized to the baseline, in which every replacement of a dirty cache line writes the whole cache line to PRAM, are shown in Figure 15. It can be seen that partial writes at word granularity (4 bytes) or encryption block granularity (16 bytes) can reduce around 45% or 35% of the write traffic on Gmean. Note here we conduct experiments in a conservative way as we do not simulate memory buffer organization. With coalescing effects under memory buffers, the results may be better [31]. With encryption, however, partial writes fail to reduce any write traffic.

Figure 14. Impact of encryption on redundant bit-write removal

![Figure 14](image-url)
3.5.3 Effect of the New Proposed Encryption Counter Scheme on Partial Writes

Figure 16 shows the effect of our proposed new encryption counter scheme (Section 3.2.2) to revive partial writes to reduce write-traffic to PRAM. The baseline is the one in which the whole dirty cache line is written back to PRAM upon replacement. In our experiment, we examined block-level encryption counters of different sizes, 1-bit, 2-bit, 3-bit, and 4-bits. We also compared them to the ideal case when the block-level counters can be arbitrarily large (labeled 'MAX-bit counter' in Figure 16).

Several observations can be made from Figure 16. First, our new encryption counter scheme effectively revives partial writes to reduce write traffic. Second, as the size of block-level encryption counters increases, more write traffic can be reduced. The reason is that large counters reduce the number of overflows, which in turn spares more non-dirty blocks from being written back to PRAM. The increased counter size, however, incurs higher space overhead. Therefore, we choose 2-bit block-level counters, which achieve 26.8% write traffic reduction on average using Gmean.
3.5.4 Impact of Memory Compression on Wear-leveling Techniques

In this section, we examine the impact of memory compression. It can be used as another wear-leveling technique to reduce write traffic. However, it also varies the bit-level redundancy compared to uncompressed data. In our experiments, Frequent Pattern Compression (FPC) [19] and LZSS [39] are evaluated. FPC exploits the observation that certain value patterns such as zero-values are frequent in main memory while LZSS is a dictionary encoding scheme. Data are compressed at the cache line granularity and the compressed data are stored in-place into the original cache line location in main memory. Note this is an optimistic way of evaluating the compression effect on write traffic reduction. Storing the compressed data in compact may result in underflow/overflow when the size of a new compressed cache line is larger or smaller than the size of the old compressed cache line. The underflow/overflow may result in moving some data around, which increases write traffic.
Figure 17 shows the impact of FPC on redundant bit-write removal (LZSS shows similar performance). Such similarity is also reported in [25]. Note that it is not practical to combine compression with partial writes as compression may change the data layout in the compressed data. There are several observations. First, there are two benchmarks which exhibit very low compressibility (lbm and swim) for FPC. Because of that, write traffic reduction from using FPC reaches 3.5% using Gmean and 28.4% on arithmetic mean. Second, compression reduces the bit-level redundancy. Redundant bit-write removal removes 82.3% traffic on Gmean of compressed write-back data, down from 89.5% on uncompressed write-back data. The significant bit-level redundancy under compression is due to abundant non-dirty data as shown in Figure 15 and high bit-level redundancy as shown in Figure 14. Overall, compression combined with bit-write removal achieves similar write-traffic reduction (87.7% on Gmean) to redundant bit-write removal only (89.5% on Gmean).

3.5.5 PRAM Lifetime Comparison

In this experiment, we map write traffic to PRAM lifetime estimate. We assume that write traffic to PRAM is uniformly distributed across the memory footprints of each benchmark.
In other words, we assume that an optimal cache-line-level swapping/rotating/shifting scheme is already in place to extend the PRAM lifetime. With such idealistic assumption, we estimate the upper bound of the PRAM lifetime and the results are shown in Figure 18. The baseline is the one without partial writes, without bit-redundancy removal, encryption or compression.

From the figure, it shows that in the baseline model, the PRAM has relatively short lifetime for benchmarks ammp and art. This is because the benchmarks have high write-traffic density on their memory footprints. On average, the lifetime of the baseline is around 1.3 years using the arithmetic mean (Amean). With redundant bit-write removal, all benchmarks show large improvement on lifetime as a result of the high bit-write redundancy as shown in Figure 14. With this wear-leveling technique, the lifetime of PRAM reaches to around 51.3 years, more than 39x improvement over the baseline. When encryption is used, however, redundant bit-write removal can only achieve 2x improvement over the baseline, reaching the lifetime of around 2.6 years. For partial writes at word granularity, the average lifetime of PRAM across the
benchmarks is around 7.9 years. For partial writes with encryption, the original encryption counter scheme disables partial writes completely, thereby no lifetime improvement over the baseline. For partial writes with our proposed new encryption counter scheme (2-bit block-level encryption counters), the achieved PRAM lifetime is around 2.5 years on average, almost 2x improvement over the baseline. When we combine both wear-leveling techniques and enable encryption protection, we can achieve almost 5.0 years of lifetime, around 4x improvement over the baseline. Since the results in Figure 18 are already based on an idealistic assumption of uniform write traffic distribution, we believe that even with all the wear-leveling techniques, PRAM may fail in a limited time, which necessitates the use of ECC.

For compression, we can see FPC alone improves lifetime from 1.3 years to 2.1 years on average because of write traffic reduction. But it reduces the effect of redundant bit-write removal (lifetime is reduced from 51.3 years to 27.5 years). Such lifetime reduction despite the similar write traffic reduction in Figure 17 is due to the fact that a small difference in write traffic reduction may result in a large difference in lifetime. For mcf, 99.89% and 99.69% traffic reduction results in 299 years and 106 years of lifetime, respectively. With encryption, such bit-write redundancy is dropped to 50%, the same as without compression due to the diffusion effect of encryption algorithms.

3.5.6 ECC Space Overhead and Wear Out

Without dynamic ECC management, the fixed ECC space overhead for correcting up to 8 bit errors per 64 bytes is 10 bytes. In other words, 15.6% of the total PRAM capacity would have to be reserved for ECC. With dynamic ECC management, we leverage the fact that we only need to correct a much less number of bit errors when the PRAM is young (i.e., have not been written
many times). To correct 1 bit error per 64 bytes, we need 10 ECC bits. Therefore, only 2.0% of the total PRAM capacity is necessary for ECC storage. If a group consists of 1024 pages, only 20 ECC pages is required to protect 1004 pages. For 2 bit errors, it becomes 3.9% and so on. This is much more efficient compared to the fixed ECC allocation scheme, which means the space otherwise reserved for ECC can be allocated for program use.

In the next experiment, we examine redundant bit writes in ECC with and without encryption. Two ECC schemes, SEC-DED [42] and BCH [44], are used in this experiment and the results are shown in Figure 19. From the figure, it shows that without encryption, there are a high number of redundant bit writes, which can be eliminated with the redundant bit-write removal technique. However, with encryption, for both SEC-DED and BCH, the ratio of redundant bit writes becomes 50%, indicating the same behavior as the bit-write traffic in regular data.

![Figure 19. Impact of encryption on redundant bit-write behavior of ECC](image-url)
3.5.7 Performance Overhead of Encryption and ECC

As discussed in Section 3.4, we choose to compute ECC based on encrypted data. To examine the overall performance impact, we model the proposed scheme in our timing simulator. For each benchmark, we skip the first one billion instructions and execute the next three-hundred million instructions. The performance results, which are normalized to the baseline without encryption or ECC, are shown in Figure 20.

![Figure 20. Performance overhead of encryption and ECC](image)

From Figure 20, it shows that adding encryption and ECC has very small performance impact (0.3% on average). The benchmark, mcf_06, has the worst performance degradation (1%) due to its high memory traffic. The reason for such small performance overhead is that the PRAM access latency is large enough to dominate the overall performance.

3.5.8 Summary

In summary, without encryption, redundant bit-write removal and partial writes can achieve 51.3 years and 7.9 years of PRAM lifetime, respectively. With encryption, those two combined can only achieve 2.6 years of lifetime. Our proposed new encryption scheme can improve it to around 5.0 years at 1.6% space cost. Redundant bit-write removal combined with
compression can achieve 27.5 years without encryption and 2.6 years with encryption. The encryption counters can be leveraged to monitor PRAM wear out status and the adaptive ECC management can be deployed with an increment of 2.0% space cost and dozen-cycle latency for each additional bit error to be corrected.

3.6. Conclusion

Phase change memory is a promising technology for computer systems. In this chapter, we investigate the largely overlooked privacy issue of PRAM due to its non-volatility. We show that if encryption is used for privacy protection, some of previously proposed wear-leveling schemes will be severely affected. To mitigate the adverse impact of encryption, we propose to extend the counter-mode encryption to revive a wear-leveling technique: partial writes. We also investigate memory compression and show that it reduces memory traffic but hurts bit-level redundancy. Then we study error correction code (ECC) as an essential mechanism for PRAM lifetime extension. We propose a dynamic ECC management scheme to vary ECC protection strength according to the age of PRAM, which is conveniently provided from the encryption counters. Our experimental results show that the performance overhead for achieving privacy protection and lifetime improvement is minimal.
List of References


[25] M. Ekman et. al, A Robust Main Memory Compression Scheme, ISCA 2005


CHAPTER 4. DECONSTRUCTING NEW CACHE DESIGNS FOR THWARTING SOFTWARE CACHE-BASED SIDE CHANNEL ATTACKS

Concrete implementations of theoretically “bullet-proof” cryptographic algorithms may possess certain weaknesses due to physical properties of those implementations. For example, an adversary can observe so-called “side channel” information such as the power consumption of a cryptographic chip or the execution times of cryptographic applications to derive confidential information, particularly secret keys.

Although previous efforts were often focused on embedded systems like smart cards, practical side channel attacks have been demonstrated on modern commodity computer systems [57],[52],[68]. Recent software-based side channel attacks exploiting architectural features such as data caches [51],[55],[56],[61],[64],[67], instruction caches [47], shared functional units ([50]) and branch predictors [48],[49]) are gaining increased attention as these attacks do not require physical device access and only conduct legitimate activities. As a result, they pose serious threats to the security of computer systems. This is also apparent from the fact that widely used cryptographic software, e.g. OpenSSL, have gone through several revisions (e.g. [62],[63]) to strengthen their implementations against side channel cryptanalysis techniques. Furthermore, the recent AES instruction addition to the x86 ISA from Intel explicitly mentions protection against software-based side-channels as one of the justifications for the new AES instructions [60].

Although caches are highly effective in reducing average memory access time and thus widely used in modern processors, their internal functionalities, i.e., hit/miss behaviors, were
shown to leak critical information that puts trusted software implementations in an unforeseen danger. Of course, some mitigation methods were proposed to defend against software cache-based side channel attacks immediately after the realization of cache architectures as a new side channel source for malicious attacks [65],[55],[64],[67]. However, as also stated in [70], we have seen only what we can call “ad-hoc” solutions to these security problems so far. In other words, each different cache-based side-channel vulnerability was analyzed separately and the software mitigations were proposed in a case-by-case basis. Furthermore, one needs to employ all of these countermeasures together, which brings significant performance overhead, in an implementation to achieve a reasonable security level.

Wang et. al. realized the lack of a comprehensive hardware solution to mitigate cache attacks with low performance overhead and tried to address this issue by proposing new cache designs in a recent work [70]. They analyzed some cache-based side-channel attacks, identified their root causes, and proposed two different cache architectures, Partition-Locked cache (PLcache) and Random Permutation cache (RPcache). Unfortunately, the authors of [70] did not consider every known cache attack, especially the type of attacks known as cache-collision attacks. Therefore, their proposals do not provide sufficient security levels as will be shown in later sections.

In this chapter, we analyze these two cache designs and empirically prove that they fall short on avoiding some cache attacks. We simulated a MIPS R10000 processor with these new cache designs using a timing simulator tool, which is built upon the SimpleScalar toolset [58]. We ran several cache attacks on this simulated CPU and used OpenSSL’s AES implementation as our target cryptosystem application. Our results show that these cache architectures are still
vulnerable to cache attacks. Eventually, we will discuss possible solutions and improvements over the original designs to overcome the identified shortcomings of PLcache and RPcache.

4.1. Background

4.1.1 AES and Its Software Implementation

Rijndael [59] is a symmetric block cipher, which was announced as Advanced Encryption Standard (AES) by NIST [53]. AES allows key sizes of 128, 192, and 256 bits and operates on 128-bit blocks. For simplicity, we will describe only the 128-bit version of the algorithm in this chapter.

AES performs operations on a 4x4 byte-matrix, called State, which is the basic data structure of the algorithm. The algorithm is composed of a certain number of rounds depending on the length of the key. When the key is 128 bits long, the encryption algorithm has 10 rounds of computations, all except the last one of which performs the same operations. Each round has different component functions and a round key, which is derived from the original cipherkey. The four component functions are

- SubBytes Transformation
- ShiftRows Transformation
- MixColumns Transformation
- AddRoundKey Operation

The AES encryption algorithm has an initial application of the AddRoundKey operation followed by 9 rounds and a final round. The first 9 rounds use all of these component functions in the given order. The MixColumns Transformation is excluded in the last round. A separate
key scheduling function is used to generate all of the round keys, which are also represented as 4x4 byte-matrices, from the initial key. The details of the algorithm can be found in [59] and [53].

The most widely used software implementation of AES is described in [59] and it is designed especially for 32-bit architectures. To speed up encryption, all of the component functions of AES, except AddRoundKey, are combined into lookup tables and the rounds turn to be composed of table lookups and bitwise exclusive-or (XOR) operations. The five lookup tables T0, T1, T2, T3, T4 employed in this implementation are generated from the actual AES S-box values as the following way:

\[
T_0[x] = (2 \cdot s(x), s(x), s(x), 3 \cdot s(x)), \\
T_1[x] = (3 \cdot s(x), 2 \cdot s(x), s(x), s(x)), \\
T_2[x] = (s(x), 3 \cdot s(x), 2 \cdot s(x), s(x)), \\
T_3[x] = (s(x), s(x), 3 \cdot s(x), 2 \cdot s(x)), \\
T_4[x] = (s(x), s(x), s(x), s(x))
\]

where \( s(x) \) and \( \cdot \) stand for the result of an AES S-box lookup for the input value \( x \) and the finite field multiplication in GF(2^8) as it is realized in AES, respectively.

\[
\begin{align*}
&x_0^{i+1}, x_1^{i+1}, x_2^{i+1}, x_3^{i+1} \\
&y_4^{i+1}, y_5^{i+1}, y_6^{i+1}, y_7^{i+1} \\
&x_8^{i+1}, x_9^{i+1}, x_{10}^{i+1}, x_{11}^{i+1} \\
&y_{12}^{i+1}, y_{13}^{i+1}, y_{14}^{i+1}, y_{15}^{i+1}
\end{align*}
\]

\[
\begin{align*}
&T_0[x_{0}^{i}] \oplus T_1[x_{1}^{i}] \oplus T_2[x_{10}^{i}] \oplus T_3[x_{15}^{i}] \oplus \{k_{0}^{i}, k_{1}^{i}, k_{7}^{i}, k_{4}^{i}\} \\
&T_0[y_{12}^{i}] \oplus T_1[y_{13}^{i}] \oplus T_2[y_{14}^{i}] \oplus T_3[y_{15}^{i}] \oplus \{k_{0}^{i}, k_{1}^{i}, k_{7}^{i}, k_{4}^{i}\} \\
&T_0[x_{0}^{i}] \oplus T_1[x_{13}^{i}] \oplus T_2[x_{14}^{i}] \oplus T_3[x_{15}^{i}] \oplus \{k_{1}^{i}, k_{0}^{i}, k_{11}^{i}, k_{12}^{i}\} \\
&T_0[y_{12}^{i}] \oplus T_1[y_{13}^{i}] \oplus T_2[y_{14}^{i}] \oplus T_3[y_{15}^{i}] \oplus \{k_{1}^{i}, k_{0}^{i}, k_{11}^{i}, k_{12}^{i}\}
\end{align*}
\]

Figure 21. Round computations in AES

As shown in Figure 21, in a typical 128-bit-key 10-round AES implementation, each round has two inputs, 16-byte \( \{x_0^i, ..., x_{15}^i\} \), which is the output from the previous round, and 16-byte round key \( \{k_0^i, ..., k_{15}^i\} \), which are pre-computed from the 16-byte secret key \( \{k_0, ..., k_{15}\} \).
Each of the lookup tables T0, T1, T2, and T3 contains 256 pre-computed 4-byte values. The
initial state \(\{x_0^0, ..., x_{15}^0\}\) is computed by 16-byte plaintext \(\{p_0, ..., p_{15}\}\) XORed with the key \(\{k_0, ..., k_{15}\}\). The last round uses another set of tables (T4, T4, T4, T4) and its output is the ciphertext.

Since the initial state, which is computed by XORing the plaintext and the secret key, is
used as the table indices in the first round; an adversary can recover the key if the plaintext and
the table indices are known. The strength of AES depends on the infeasibility of recovering the
table indices, which is a valid argument in the context of theoretical cryptanalysis, even in the
case when an adversary has an unlimited number of plaintext and ciphertext pairs computed
under the same secret key. Although AES is (still) secure in theory, cache attacks can, in
practice, directly exploit the implementation weaknesses to recover the table indices.

4.1.2 Software Cache-based Side-Channel Attacks

We have seen increased research efforts on the side-channel analysis of commodity PC
platforms for the last few years, especially after the realization of some processor components
causing serious side-channel leakages. These efforts led to the development of the new
Microarchitectural Analysis (MA) research area. Those attacks exploit the microarchitectural
components of a processor to reveal cryptographic keys. The functionality of some processor
components such as cache and branch predictors generates data dependent variations in
execution time and power consumption during the execution of cryptosystems. These variations
either directly gives the key value out during a single cipher execution [48] or leaks information
which can be gathered during many executions and analyzed to compromise the
system [64],[55],[61].
There are currently four main types of MA attacks in the literature: Data Cache, Instruction Cache Attacks, Branch Prediction Analysis, and Shared Functional Unit Attacks. In this chapter, our focus will only be on data cache attacks, which are usually referred to as cache-based side-channel attacks or simply cache attacks in the literature.

Cache attacks exploit the cache behavior of a cryptosystem by obtaining the execution time and/or the power consumption variations generated via cache hits and misses. Cache analysis techniques enable an unprivileged process to attack another process, e.g., a cipher process, running in parallel on the same processor as done in [64],[61],[67]. Furthermore, some of the cache attacks can even be carried out remotely, e.g., over a local network [51].

Theoretical cache attacks were first described by Page in [65]. He characterized two types of cache attacks: trace-driven and time-driven. Later, we saw another type of cache attack, which is now referred as access-driven, in [67],[64].

We will discuss time-driven and access-driven attacks in this chapter and exclude trace-driven attacks due to the fact that there is no instance of a software based trace-driven attack in the literature.

4.1.2.1. **Access-driven Cache Attacks**

Figure 22 illustrates access-driven attacks on a target crypto application ---AES as in our case. A typical access-driven attack consists of the following steps:

1. The attacker occupies the entire cache with his own data (Figure 22a). This can be accomplished by an attack process/thread through loading a large array into the cache.
2. He then invokes an AES execution. During execution, the cipher issues table lookups with the indices depending on the key and the plaintext, and the corresponding parts of the AES tables will be loaded into the cache (Figure 22b). Since some parts of the attacker's array will be evicted from the cache, the cache state after the encryption becomes dependent on the accessed table indices and thus leaks information of the secret key\(^1\).

3. Shortly after the encryption, the attacker reads again its own large array, but this time he also measures the time to read each individual cache line. Since the cache lines that have been already evicted from the cache require a longer time to read compared to the data still inside the cache, he knows which cache lines were replaced by the AES execution. In other words, the attacker can infer which parts of the AES lookup tables have been accessed during the encryption.

\(^1\) The fact that one single cache line contains multiple table elements and multiple rounds share the same set of tables do reduce the leaked information. However, the full key can still be recovered statistically as discussed in [64]. It is also the same case for time-driven attacks.
Neve et al. [61] have demonstrated that an adversary can get fine-grained cache behavior snapshots of an AES process on single-threaded processors by manipulating the OS scheduling. It is also shown there that observing a small number of encryptions under the same key provides sufficient information to recover a full 128-bit AES key. Multi-threaded architectures facilitate this class of attacks since attackers can use a hardware-assisted spy process to monitor the execution of the crypto process on-the-fly during the encryption, as demonstrated by Acıçmez [47], Osvik et. al. [64], and Percival [67]. Although the attacks in [47] and [67] are against RSA, the RSA-vulnerability associated with table lookups is similar to the AES attacks presented in [64]. All these access-driven attacks exploit the same root cause: the shared cache among processes/threads.

4.1.2.2. Time-driven Cache Attacks

The first cache-based timing attack against AES was introduced by Bernstein [55]. His attack exploits the statistical correlations between the AES encryption time variations and inputs to the encryption, i.e., plaintext and cipher key. This attack was analyzed in [70] and cache interferences are identified as the root cause. However, Bernstein's attack is not the only time-driven cache attack. A more recent time-driven attack, named “cache-collision attack”, has
different characteristics. Therefore, the countermeasures designed to defeat Bernstein's attack, including the ones proposed in [70], may not provide sufficient protection against cache-collision attacks, which is the case as shown in this chapter.

An AES encryption operation can be viewed as a sequence of table lookups and some additional computations. A “cache collision” occurs when two lookups refer to the same element in a table. In this case, the second lookup will certainly be a cache hit. If there is no cache collision, the second lookup may be a cache miss. Statistically, a higher number of cache collisions lead to a smaller number of cache misses in an encryption and thus a shorter encryption time compared to encryptions with a smaller number (or zero) of cache collisions. This statement has been experimentally verified in [51],[56],[69]. We have also confirmed it by re-producing the experimental results of the so-called last round AES attack described in [56],[69] on both a real Pentium 4 machine and a simulated processor model (our processor configuration is given in Table 2).
Our results\(^2\) are shown in Figure 23 and they are computed from 16 million encryptions (each encryption starts with a clean cache) of 16-byte random plaintexts. Each point in this figure shows the variation between the mean encryption time based on the runs with the same number of collisions and the overall mean encryption time. One can see that the mean encryption time decreases as the total number of collisions for the last round increases. The non-decreasing effects of the last data point (encryption runs with 6 collisions) are mainly due to an insufficient

---

\(^2\) In this chapter, timing deviation is defined as the difference between the mean encryption time among the samples with the same feature, e.g., the same number of collisions, and the mean encryption time among all the samples.
number of the samples with such a high number of cache collisions, thereby not statistically
significant.

Cache collision attacks work as follows: Assume there are two table lookups \( k_i \oplus x_i \) and
\( k_j \oplus x_j \) where \( k_i, k_j \) are the AES encryption key bytes and \( x_i, x_j \) are plaintext bytes for the first
round. If used for the last round, \( k_i, k_j \) are expanded last round key bytes and \( x_i, x_j \) are ciphertext
bytes. The essence of the attack is to find the correct key difference between \( k_i \) and \( k_j \), i.e., \( k_i \oplus k_j \). If there is a collision between \( k_i \oplus x_i \) and \( k_j \oplus x_j \), the following equation holds:

\[
    k_i \oplus x_i = k_j \oplus x_j \Leftrightarrow k_i \oplus k_j = x_i \oplus x_j \quad \text{Equation (1)}
\]

From a large number of samples, attackers can expect that among all possible 256 values
of \( x_i \oplus x_j \), the one with the smallest mean encryption time implies a cache collision and
therefore corresponds to the correct value of the key byte difference \( k_i \oplus k_j \). Figure 24 shows
one such example for the last round attack. The figure shows the mean encryption time for each
of the 256 values of \( x_0 \oplus x_1 \), where \( x_0, x_1 \) are the first two bytes of the ciphertext. Here, the
mean encryption times are computed from 16 million encryptions under the same key on both a
real Pentium 4 machine and our simulated processor. The actual difference between the first key
byte and the second key byte (i.e. \( k_0 \oplus k_1 \)) is 254. As shown in Figure 24, among all possible
values of \( x_0 \oplus x_1 \), the one with the smallest encryption time correctly reveals the value of \( k_0 \oplus k_1 \). As a result, an attacker only needs to record the encryption time and the ciphertext to derive
the relationship among different key bytes. Then, an attacker can conduct off-line analysis to
recover the complete key. Certain searching algorithms can also be used to dramatically reduce
the number of samples needed to recover the key. Further details of cache collision attacks can
be found in [46],[51],[56],[69].
In summary, current time-driven attacks exploit the fact that access to different levels of the memory hierarchy can introduce observable time differences. If certain aspects of the secret key correlate to the number of cache hits/misses, the cipher is endangered by time-driven attacks. Furthermore, if attackers have the ability to set the initial “clean cache” state, the number of required samples is reduced as it is more likely for cache collisions to lead to reduced encryption time.

4.2. **Recently Proposed New Cache Designs for Thwarting Software Cache-based Side Channel Attacks**

As stated above, we have seen only what we can call “ad-hoc” solutions to the cache side-channel problem so far. In other words, different cache-based side-channel vulnerabilities
were analyzed separately and the software countermeasures were proposed in a case-by-case basis as outlined above. One needs to employ many of these countermeasures together, which may incur significant performance overhead, to achieve a reasonable security level. Therefore, we evidently need comprehensive, robust, and efficient defense techniques. Along this direction, Wang et. al. [70] analyzed the Berstein's time-driven attack [55] and Percival's access-driven attack [67]. They identified cache internal interference (interferences from the same process) and external interference (interferences from different processes) as the root cause of cache-based attacks and proposed two new cache designs to overcome the cache interference effect.

4.2.1 Partition-Locked Cache

The concept of using partitioned caches against software cache-based side channel attacks was introduced in [66]. It is used to isolate the protected processes from others so that cache interference among different processes (i.e., external interferences) becomes infeasible. Besides incurring too much performance cost, this approach does not solve the internal interference within the same process. The Partition-Locked cache (PLcache) solves those problems with a fine-grain locking over the cache lines. With support from PLcache, software applications are able to lock critical data in the cache in the granularity of one cache line size. Once a cache line is locked in the cache, both external and internal interferences cannot evict it. Therefore, an attacker cannot observe the access pattern of the protected cache lines and cannot gain any useful information from the time variance behavior associated with the accesses of those cache lines.

<table>
<thead>
<tr>
<th>L</th>
<th>ID</th>
<th>Original Cache Line</th>
</tr>
</thead>
</table>

Figure 25. The new cache line of the PLcache
The PLcache design mainly includes two parts: hardware support for locking and architecture exposures for software applications. In the PLcache, each cache line has been augmented with an ID field and a lock bit L. As shown in Figure 25, the ID indicates the owner of the cache line, normally a process. The lock bit L indicates the locking state of the cache line. These two bits will help cache replacement to decide whether a cache line should be replaced as usual or not. For the cache lines of the same process, the basic rule is that incoming non-locked cache lines can't replace locked cache lines. For the cache lines of different processes, the basic rule is that no incoming cache line of one process can replace any locked cache line of another process. For the interface to software applications, special instructions (ld_lock, ld_unlock, st_lock, st_unlock) are introduced. They are used to update the lock bit to control which cache lines should be locked or unlocked. This way, software application can use these special instructions to prevent their critical data being replaced by interferences, thereby defeating some cache attacks.

PLcache has several advantages. The design addresses the root cause of two analyzed attacks - cache interferences and thus provides generic countermeasure support against attacks based on that. Because of fine-grain locking control over cache lines, it was also shown to have low performance overhead on AES. In the meanwhile, however, excessive locking could cause unfairness problems and it was proposed to have the operating system to manage the locking requests from processes.
4.2.2 Random-Permutation Cache

Rather than eliminating cache interferences as the PLcache aims to achieve, the Random-Permutation cache (RPcache) takes another perspective. It allows cache sharing and cache interferences but it obfuscates the interferences so that no useful information can be derived. A spy process can observe another process' cache access only if the victim process replaces the spy process' cache lines deterministically. The RCache improves the security by making such replacements random. In other words, the cache line address in the RCache is generated in a random instead of a pre-determined manner.

In the RCache, each cache line is augmented with one protection bit and an ID. As shown in Figure 26, the protection bit P indicates whether the cache line should be protected. The ID indicates the owner of the cache line, normally a process. In a case of interference, when the fetched cache line and the chosen replacement cache line belong to two different processes, the original cache set will not be used. Instead, another cache set is chosen randomly and replacement happens in that set. This process changes the mapping between addresses and cache
sets. To support it, a hardware permutation table is introduced in the RPeache. As shown in Figure 26, typically the protected process will use the hardware random permutation table and thus indirectly address the cache. Other processes will have no permutation tables and therefore no address indirections. It was shown that with such random permutation cache interferences leak no information. Note the P bit is used to defend against the cache attack in [55] and details about the defense can be found in [70].

The novel RPeache design provides the security protection against some analyzed cache attacks, particularly access-driven attacks. In addition, it has the advantage of low performance overhead and requires no changes in the programs.

4.3. Security Analysis of The New Cache Designs

This section discusses the security problems of PLcache and RPeache that we could identify. We have conducted several experiments to support our claims. We implemented both the PLcache and RPeache in our timing simulator. Our simulator is developed upon the Simplescalar toolset [58]. The simulator models MIPS R10000 processor, with the default configuration shown in Table 2.

4.3.1 Analysis of PLcache

The PLcache design has two security deficiencies. The first security vulnerability of the PLcache lies in the initial phases of process execution. Only after all the critical data of a crypto process are loaded into the cache, the PLcache ensures that they are locked and stay in the cache.
However, when this crypto process starts to run, the critical data are gradually loaded into the cache, making it vulnerable to either access-driven or cache-collision attacks.

Table 2. Default processor configuration

<table>
<thead>
<tr>
<th></th>
<th>7-stage pipeline: Fetch/Dispatch/Issue/RegisterRead/EXE/WriteBack/Retire</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch/Dispatch/Issue/MEM issue/Retire Bandwidth: 4</td>
</tr>
<tr>
<td></td>
<td>Fully-symmetric Function Units: 4</td>
</tr>
<tr>
<td></td>
<td>Reorder Buffer size: 64</td>
</tr>
<tr>
<td></td>
<td>Issue Queue Size: 32</td>
</tr>
<tr>
<td></td>
<td>Load Store Queue Size: 32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Address Generation: 1 cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Latencies</td>
<td>Memory Access: 2 cycles (hit in data cache)</td>
</tr>
<tr>
<td></td>
<td>Integer ALU ops: 1 cycle</td>
</tr>
<tr>
<td></td>
<td>Complex ops: MIPS R10000 latency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Cache</th>
<th>32KB 2-way, Block size: 64B 10-cycle miss penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Data Cache</td>
<td>32KB 2-way, Block size: 64B 10-cycle miss penalty</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>2MB 16-way, Block size: 64B 300-cycle miss penalty</td>
</tr>
</tbody>
</table>

To illustrate this problem, we first ran an access-driven attack on AES using our simulated processor model. In this experiment, we started an AES process with a cold cache and ran a spy process to observe the cache usage of the cipher. Our spy routine asked the AES process to encrypt single 16-byte message blocks and observed cache sets after each encryption. The measurements of this spy process indicate which blocks of AES tables had been accessed and locked in the cache, as shown in Figure 27. For simplicity, we only show the part of the cache that holds only one AES table. The dark boxes represents the cache lines (or blocks of this AES table) locked by the crypto process. As shown from this figure, the PLcache leaks the cache usage information of AES, which enables successful access-driven attacks. It is reported that on
average less than 15 clean observations like the one in our experiment are sufficient to completely break a 128-bit AES key [61].

Similarly, for cache-collision attacks, if an adversary can set up a clean cache state before each encryption run, the encryption time correlates to the number of cache collisions. We implemented the PLcache in our timing processor simulator and used the analysis tool from Bonneau's website\(^3\) to conduct last round cache-collision attack. In this attack, each encryption run starts with a clean cache state. The encryption time and the cipher text are collected to derive the relation between the key bytes. With \(2^{14}\) samples, the derived relationship among different key bytes is enough for the tool to successfully recover the complete 128-bit key.

Besides the security vulnerabilities discussed above, the PLcache is also subject to denial-of-service attacks due to its inherent over-locking problem. Even though OS oversight was proposed to properly handle the locking requests or impose limit on the size of locked cache lines for each process, the PLcache design does not support the locked cache lines to be replaced when the owner process is not running, i.e. switched out because of context switches. As a result, processes may still lock too many cache lines, causing fairness problem or even severely degrading other processes' performance.

---

\(^3\) [http://www.jbonneau.com/research.html](http://www.jbonneau.com/research.html)
4.3.2 Analysis of RPcache

The main security issue with the RPcache is its vulnerability to cache-collision attacks. In cache-collision attacks, no interference is explicitly needed since their fundamental cause is that higher number of cache collisions lead to lower encryption time. As the RPcache does not guarantee that all the protected data (e.g., the AES lookup tables) are in the cache, this side channel still exists. We performed an experiment to test whether the RPcache is vulnerable to cache-collision attacks. In this experiment, we ran 16 million encryptions upon random 16-byte data with the same key and a clean cache state is setup before each encryption. Figure 28 shows the timing characteristics collected from our experiment. As shown in this figure, the relationship between the total number of cache collisions in the final round and the mean execution time still

![Figure 28](image_url)

Figure 28. Experimental results of running last round cache-collision attack on RPcache
holds. The mean encryption time of the encryption runs with the correct key-byte difference is still significantly lower than the overall mean encryption time.

To get a solid proof, we implemented the RPcache in our timing processor simulator and conducted an actual cache-collision attack, the last round attack of Bonneau et. al. [56] on AES. Similar to PLcache results, $2^{14}$ samples were sufficient to completely recover 128-bit AES key on RPcache.

4.4. Potential Solutions

As analyzed in [70], the recently proposed secure cache architectures (i.e., the PLcache and the RPcache) have many desirable features including low performance overhead, generic countermeasure support, and transparency to the protected applications in the case of the RPcache. Unfortunately, as shown above, they are still vulnerable to software cache-based side channel attacks. Thus, in this section, we will briefly discuss potential solutions to their security problems.

4.4.1 Securing the PLcache

The main weakness of the PLcache design is that it does not protect the initial loading procedure. Therefore, to secure the PLcache, one possible solution would be a preloading and locking all critical data right before the crypto operations. This way, there is no initial loading process to be exploited for information leakage attacks. Note that this is different from pure software preloading. The reason is that without the PLcache support, pure software reloading cannot guarantee that the critical data will not be replaced after the preloading process.
In terms of the denial-of-service vulnerability, the key is to make sure that when a process is not active, it will not lock its data in the data cache so that other processes will not suffer from the reduced cache capacity. Then, when an inactive process becomes active again, we need to reload and lock all its critical data since some of the data may be replaced by other processes' data. This way, OS oversight is not necessary for managing locking requests in uniprocessors and each process can have locking request up to the whole cache without causing problems to other processes, which reduces the design complexity. However, in multi-threaded processors, there may be multiple active processes at the same time. OS oversight is still required to manage the processes that are competing for the locked cache lines. For example, OS may simply choose to run one single protected process at a time if an additional protected process requires overlapped locked cache lines.

There exist several possible ways to implement the mechanisms to secure the PLcache. One could be mainly software. This way, the protected process performs the preloading and locking before critical operations and performs unlocking once the critical operations are completed. The OS will perform the preloading and locking during context switches of the protected process. The preloading and locking can also be accelerated with hardware support. In this case, new instructions are required to specify the address and length of the critical data and to invoke the preloading and locking. Besides the security enhancement, these solutions need to be evaluated for performance overheads and complexity.

4.4.2 Securing the RPcache

The main weakness of the RPcache design is that it is vulnerable to collision-based time-driven attacks. The RPcache design defeats access-driven attacks by obfuscating cache
interferences. However, it still leaks information due to the variations of the encryption/decryption time, which are dependent on the number of cache misses. Therefore, to defeat time-driven attacks, the ideal approach is to eliminate cache misses so as to remove the variations of the encryption/decryption time. However, since there is no locking mechanism in the RPcache design, there is no guarantee that the accesses to the critical data will be cache hits. As a result, both compulsory misses (i.e., initial access to the critical data) and conflict misses (re-access to the critical data after they are replaced) are possible. One potential mitigation technique would be to eliminate as many cache misses as possible so as to increase the number of required samples to an infeasible level.

To accomplish this idea, we need a way to reload the critical data after it is replaced. One symptom of replaced data is that the re-access will be a cache miss. To capture such events, we need to change the hardware architecture to expose the cache miss event to the software. Once the software is being notified by the critical event, certain countermeasures can be performed. A detailed study of how this approach will help to secure the RPcache design and how much performance overhead will be introduced is part of our future work.

4.5. Conclusions

In this chapter, we analyze the latest advances in the software cache-based side channel attacks. We show that the previously proposed cache designs, the partition locked cache (PLcache) and random permutation cache (RPcache), although effective in defeating information leakage due to interferences, are vulnerable to latest attacks built upon either cache sharing or
cache collisions. We also discuss the potential enhancement to overcome the vulnerabilities of these new cache designs.
List of References


CHAPTER 5. ARCHITECTURAL APPROACHES TO DEFEND AGAINST SOFTWARE CACHE-BASED SIDE CHANNEL ATTACKS

Side channel attacks exploit “side channel” information such as power, heat, electromagnetic radiation, or time to derive confidential information, particularly secret keys used in cryptographic systems. Recently, there are newly developed software-based side channel attacks which exploit shared architectural components of modern commodity processors such as instruction caches [71],[74], data caches [76],[77],[78],[89],[93],[97],[102] and branch target buffers [72],[73]. These attacks do not require physical access to target computers or direct access to the memory space of victim processes and are conducted through legitimate software operations. As a result, they pose serious threats to modern computer systems [85], [104].

Current software cache-based side channel attacks include access-driven attacks [71],[74], [72],[73],[89],[93],[97] and time-driven attacks [76], [78]. Access-driven attacks exploit the correlation between the secret key and the cache usage of a crypto thread/process. Since the cache is shared among multiple processes/threads, an attacker may derive the cache usage of the victim process by controlling a carefully crafted process, which runs together with the victim process. Time-driven attacks measure the execution times of victim processes and exploit the correlation between the secret key and the number of cache misses (which in turn determines the execution time) to infer the key. To defend against software cache-based side channel attacks, various countermeasures have been proposed and many of them involve some modifications upon the software implementation of crypto algorithms [75],[79],[88],[92]. However, these proposals are often application and attack specific.
In order to achieve a high level of security protection, several defense techniques need to be combined, resulting in substantial performance overhead. In a recent work [104], Wang and Lee identify certain features in data caches as the root cause for software cache-based side channel attacks, and propose new cache designs (PLcache and RPcache) to prevent information leakage from data cache usage. Such hardware-based defenses, although effective for their targeted attacks, lack the flexibility to adapt to newly developed attacks [87]. Another approach to defeat cache-based attacks is to dedicate special hardware function units and instructions to a particular crypto algorithm, such as Intel’s AES (Advanced Encryption Standard [83]) instructions [84], so that data cache accesses can be completely eliminated during crypto operations. This approach, however, requires non-trivial hardware and software changes since existing crypto software has to be re-written/recompiled to leverage the new AES instructions. Furthermore, it does not protect crypto algorithms other than AES. What’s more, many currently proposed countermeasures are targeted at data cache attacks and effective and efficient defenses against other cache attacks are still missing.

In this chapter, we review the state-of-art cache attacks including data cache attacks (D-cache attacks), branch target buffer attacks (BTB attacks) and instruction cache attacks (I-cache attacks). For D-cache attacks, we identify that in both access-driven and timing-driven attacks, cache misses of critical data, whose addresses are dependent on secret keys, are the source of information leakage. We then propose three integrated hardware-software mitigation approaches. First, we propose to use preloading to secure the previously proposed PLcache [104] so as to ensure that all accesses to critical data will be cache hits. Second, we propose to use informing loads to protect the RPcache [104]. Informing loads [86] are lightweight architectural support originally proposed for optimizing memory system performance. When an informing load (a
special load instruction) misses in the cache, a user-level exception is raised. With the support for informing loads, we can easily integrate flexible software-based mitigation schemes into exception handlers. Although the RPcache randomizes cache miss addresses through random permutation, it is vulnerable to time-driven attacks (see Section 5.2.2) and our software scheme fixes the vulnerability by re-loading all critical data upon cache miss detection. It ensures that all subsequent cache accesses to those data are cache hits as long as the critical data fits in the cache and thus removes the correlation between the secret key and the number of cache misses. Third, we propose a software permutation scheme assisted by informing loads to replace the random permutation logic in the RPcache. This way, the protection can be extended to regular caches with the relatively minor hardware support for informing loads. Our experiments show that the proposed approaches based on the PLcache and RPcache provide strong protection with low performance overhead. For regular caches, our lightweight informing loads approach not only provides strong security protection without the high hardware cost of the PLcache or RPcache, but also has significantly lower performance overhead compared to existing software-only solutions.

While data cache attacks exploit the key-dependent data access operations, BTB and I-cache attacks exploit key-dependent branches and the associated diverged execution paths. For BTB attacks, we identify that in access-driven attacks, updates of critical conditional branch target addresses in BTB, which are determined by secret keys, are the source of information leakage. We propose a simple change to BTB update policy to defend against such attacks, i.e. BTB update independent of secret keys. This completely removes the side channel because of BTB. Also our experiments show that the proposed approach incurs very small performance overhead. For I-cache attacks, we identify that in access-driven attacks, cache misses of
instructions, whose executions depend on secret keys, are the source of information leakage. We then propose to apply the PLcache with preloading and the RPcache to protect instruction caches.

5.1. Threat Model and Attacks

There exist mainly two types of software cache-based side channel attacks: access-driven and time-driven attacks. In access-driven attacks, the adversary has control over one or multiple spy processes, which share the cache with the victim process. Due to cache sharing, the victim process may evict the spy process’ cache lines/entries when it accesses key-dependent (i.e. critical) cache lines/entries. By measuring the access times of its own cache lines/entries, the spy process can figure out which cache lines/entries are evicted by the victim process. Such cache access behavior of the victim process may leak enough information for the adversary to infer the key. In time-driven attacks, the adversary sends various encryption/decryption requests to the target crypto process. Upon receiving responses the adversary records the encryption times. Since the secret key may correlate to different number of cache misses upon different inputs/outputs, the variations among encryption times may provide sufficient information for the adversary to derive the key. Although time-driven attacks may be much slower than access-driven attacks, we consider both in this chapter.

In this chapter, we use two widely used cryptographic algorithms – the Advanced Encryption Standard (AES) [83] and RSA [98] to illustrate current cache attacks as well as the existing countermeasures and demonstrate the advantages of our proposed schemes. However it should be noted that our proposed schemes may also be applied to cache attacks on other applications.
5.1.1 The Advanced Encryption Standard (AES) and Data Cache Attacks

AES processes a 16-byte input with a secret key of 16, 24 or 32 bytes to produce a 16-byte output. There are multiple identical rounds involved in encryption/decryption and each round performs four types of operations (substitute bytes, shift rows, mix columns and add round key). Among them, the “substitute bytes” operation requires table lookups, in which a 1-byte input is used as an index to a compact S-box table (an 8-bit substitution box) to generate a 1-byte output. For fast software AES implementations, the four operations are combined into 16 XOR operations and 16 table lookups. The tradeoff is that five new lookup tables ($T_0$, $T_1$, $T_2$, $T_3$ and $T_4$) are used with each having 256 4-byte elements, larger than the original S-box table with 256 1-byte elements [83].

![Figure 29.Vulnerable table lookup operations in AES](image)

5.1.1.1 Access-driven Attacks against AES

In AES, two components decide the indices of table lookups. One is the 16-byte input and/or output and the other is the secret key (as shown in Figure 29). As a result, the key can be computed if an adversary obtains both the input/output and the indices of table lookups. Since the output of AES (i.e., the encrypted ciphertext) is not kept private and/or sometimes the adversary may even know the plaintext (i.e., the input to AES), it is reasonable to assume the availability of the input/output. So, the critical step for the key recovery is to obtain the indices
of table lookups. Since the indices determine which cache lines are accessed in the shared cache, the adversary is able to recover the key once the accessed cache lines can be identified. To identify the cache lines accessed by AES table lookups, access-driven attacks require that those cache lines must not reside in the shared cache beforehand so that some of the spy process’ data can be replaced later. In other words, the table lookups shall experience cache misses. This condition is essential for the access-driven attacks otherwise the spy process cannot know the cache usage. Such condition, however, can be satisfied by the spy process, which loads a large data set to effectively flush the cached data of the victim crypto process.

There are some complications regarding the realization of access-driven attacks. For example, one cache line may contain several lookup table elements and thus knowing one accessed cache line does not exactly lead to the corresponding index value. Nevertheless, from multiple samples (as few as 15 [89]) the correct key values can be filtered out statistically. Detailed description of access-driven attacks against AES is reported by Neve [89] and Osvik [93].

5.1.1.2. Time-driven Attacks against AES

As discussed in Section 5.1.1, AES relies heavily on table lookup operations and the indices to lookup tables depend on the key and inputs/outputs. If prior to AES execution the lookup tables are not in the cache, different data inputs may cause different sequences of table lookups, which in turn result in different numbers of cache misses and thus different execution times. Time-driven attacks exploit the relationship between inputs/outputs and execution times to infer the key. Bernstein [76] demonstrated that different inputs can cause various execution times
and thus the key can be inferred. Bonneau [78] presented a cache-collision attack that finds the key using a much smaller number of timing samples.

Figure 30. The relationship between the number of collisions in the last round of AES and the encryption time on one Pentium 4 machine

A cache collision happens when two table lookups refer to the same element, in which case, the second lookup will be a cache hit assuming no conflict misses occurred in between. For a non-cache-collision case, the second lookup may experience a cache miss. For successful cache-collision timing attacks on AES, the basic observation is that a higher number of cache collisions results in a smaller number of cache misses, thus a shorter encryption time [78],[102]. For example, Figure 30 shows the relationship between the mean execution time of one AES encryption and the number of cache collisions in the last round of AES. The results are collected from 16-million timing samples using one Pentium 4 machine running AES with the same random key. As shown in the figure, the execution time (relative to the average of overall timing samples) declines as the number of collisions increases (e.g., +7.82 cycles for 0 collision and -8.89 cycles for 1 collision). Based on this observation, attackers can pick two table lookups and guess cache-collision cases to infer the XORed values between key bytes, which lead to the complete key recovery. In this particular example, $2^{15}$ samples were good enough for revealing the key. Detailed information on cache-collision attacks can be found in [78],[102].
5.1.1.3. Source of Information Leakage in Data Cache Attacks

Although for D-cache attacks access-driven attacks and time-driven attacks are different in the way the attacks are performed, the source of information leakage is the same: cache misses of lookup table data (whose indices are key dependent) are exploited to infer the key. Section 5.2.3 shows how we utilize this observation to defend against both types of D-cache attacks.

5.1.2 RSA and BTB/I-cache Attacks

In addition to having a similar side channel of leaking critical information as AES because of key-dependent table lookup operations [92], RSA has another side channel because of key-dependent branches and the associated diverged execution paths, i.e. the instructions executed are dependent on secret keys.

```plaintext
S = M;
for (i = 1; i < n; i++) {
    S = S * S (mod N);
    if (d_i == 1)
        S = S * M (mod N);
}
return (S);
```

Figure 31. Modular exponentiation using the binary version of the Square-and-Multiply algorithm

In RSA, a widely used public key crypto algorithm, the main computation is a modular exponentiation $P = M^d \pmod{N}$, where $M$ is a message, $d$ is a secret key, and $N$ is a public modulus. To perform the modular exponentiation, one simple way is to use the binary version of the classic Square-and-multiply algorithm, as shown in Figure 31, where the secret key $d$ is an n-bit binary number $(d_0, d_1, \ldots, d_{n-1})_2$. The algorithm performs $n$ iterations for the modular exponentiation. For each iteration $i$, a modular square computation is always performed. In
addition, an extra modular multiplication may be computed depending on the secret key bit \(d_i\) through a conditional branch statement. If the key bit value is 0, the branch is not taken and the modular multiplication is skipped. Otherwise, the branch is taken and the modular multiplication is computed. Therefore, attackers can recover the secret key bit by either knowing whether the conditional branch is taken (BTB attacks) or whether those instructions belonging to the modular multiplication are executed (I-cache attacks).

5.1.2.1. BTB Attacks

Modern microprocessors utilize branch predictors to help to predict taken/not-taken for conditional branches so as to improve instruction fetch efficiency for high performance. A branch target buffer [99] in the branch prediction unit caches branch target addresses for branches. If the target address for a taken branch is missing in BTB, the instruction fetch needs to either be stalled for a while until the target address is calculated or simply fall through as in a not-taken branch case. In both cases, the execution time is longer compared to the case of a cache hit in BTB. Therefore BTB misses can be detected by measuring the execution time.

Similar to access-driven attacks on data caches, attackers may use their own conditional branches in a spy process to interfere with the key-dependent conditional branches in the crypto process in a shared BTB and cause BTB misses for those critical conditional branches. For the crypto process, *BTB is updated only when the key-dependent conditional branch is taken* (so as to reduce BTB interference [96]). Because of that, the spy process' BTB entries are evicted. By measuring the spy process' own execution time, attackers can detect BTB misses of the spy process because of the taken key-dependent branch in the crypto process. Therefore attackers can recover the key values [72],[49]. This BTB update condition is essential for a successful attack.
Otherwise there won't be a correlation between the BTB update and the outcome of the key-
dependent conditional branch.

5.1.2.2. **I-cache Attacks**

Instead of exploiting BTB to infer the outcomes of key-dependent conditional branches, I-cache attacks exploit instruction caches to determine the instruction execution path. Then by knowing whether the instructions of the modular multiplication (critical instructions) are executed or not as in Figure 31, attackers may still know the outcome of the conditional branch and therefore recover the secret key value.

Similar to access-driven attacks on data caches, attackers use their own instructions in their spy process to interfere with the key-dependent critical instructions in the crypto process in a shared instruction cache and evict them out of the instruction cache. The execution of those critical instructions in the crypto process would bring them back into the instruction cache and evict the instructions of the spy process. The cache misses of those instructions will cause slower execution of the spy process. By simply measuring the execution time of their own spy process, attackers may know the cache misses in the spy process and therefore the execution of those critical instructions in the crypto process. Thus they can recover the key values [71]. Besides the direct key-dependent branches as shown in Figure 31, Other indirect correlation between control flow divergence and the secret key values exists, which is also exploited using I-cache attacks in [74].
5.1.2.3. **Source of Information Leakage in BTB/I-cache Attacks**

The source of information leakage in BTB/I-cache attacks lies on the key-dependent branches. In particular, for BTB attacks, the source is *updates of critical conditional branch target addresses in BTB, which are determined by secret keys*. For I-cache attacks, the source is *cache misses of instructions, whose executions depend on secret keys*. Section 5.2.3 discusses previous work on removing key-dependent control flow. Section 5.4 shows how we utilize our observations to defend against these attacks.

5.1.3 **Summary on Software Cache-based Side Channel Attacks**

Table 3 shows our summary on recent software cache-based side channel attacks. AES has a constant execution path and has no key-dependent control flow. Therefore AES is not subject to BTB/I-cache attacks. However, RSA has key-dependent control flow and may be exposed to BTB/I-cache attacks. Both crypto algorithms have implementations with key-dependent table lookup operations which are subject to D-cache attacks. Access-driven attacks work by observing which certain cache lines/entries are accessed during cache usage and work on all three shared cache components in modern microprocessors. Time-driven attacks are based on the correlation between number of cache misses and the secret key because of heavy table lookup/cache access activities. Time-driven attacks are currently demonstrated in D-cache attacks.

**Table 3. A summary of recent software cache-based side channel attacks.**

<table>
<thead>
<tr>
<th></th>
<th>Key-dependent control flow</th>
<th>Key-dependent table lookups</th>
<th>AES</th>
<th>RSA</th>
<th>Access-driven</th>
<th>Time-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Cache Attacks</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Instruction Cache Attacks</td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>
5.2. **Current Countermeasures**

Current countermeasure proposals against data cache attacks are either application-level software solutions or hardware solutions. Among software approaches, each vulnerable application is analyzed and changed against specific attacks. In order to achieve reasonable security protection, software approaches are often combined, which may incur substantial performance overhead. Hardware proposals revise cache architecture to eliminate information leakage exploited in cache attacks. Although hardware schemes are able to provide general protection at small performance cost, they often incur non-trivial hardware changes and suffer from inflexibility to evolve against newly developed attacks.

With regard to side channel attacks based on exploiting key-dependent control flow, existing defense proposals are mostly software approaches and they target at removing the key-dependent control flow. These approaches, however, require source code, which may not always be available. What's more, the tradeoff for the security protection is often a large performance overhead. Some approaches may even fail to remove the key-dependent control flow as they only remove the key-dependent branches not the associated key-dependent control flow divergence.

5.2.1 **Software Countermeasures against Data Cache Attacks**

5.2.1.1 **Access-all against Access-driven Attacks**

Since cache attacks use cache lines to infer the indices to the lookup tables, one way to defeat them is to eliminate the correspondence between cache lines and table indices. In a
released patch [92] of RSA against access-driven attacks [97], each table element is distributed so that accessing one element ends up accessing all the cache lines of the whole table. This approach is effective for RSA (only less than 10% performance overhead) given the heavy computations involved in RSA. However, it is not applicable to AES since touching the entire table for each table lookup operation incurs too much performance overhead given the high number of table lookups in AES.

5.2.1.2. Random Permutation against Access-driven Attacks

Random permutation of lookup tables changes the mapping between table indices and cache lines. It obfuscates attackers’ observation on cache access activities. However fixed permutation can still leak information, as demonstrated in [77] for AES. Although the security offered by random permutation can be increased by frequently updating the permutation, the updating frequency remains an open question for pure software approaches due to the tradeoff between performance and security.

5.2.1.3. Small Tables against Access/time-driven Attacks

Efficient AES implementations typically utilize large lookup tables, which are pre-computed from the original S-box table. One protection scheme is to use the small S-box table instead to trade performance for security [93]. This way, one single cache line contains more elements, complicating the attacks. There are also other similar approaches, which use smaller numbers of lookup tables [93]. However, at the cost of substantial performance overhead, these approaches only increase the number of samples required for access-driven and time-driven attacks [77],[102].
5.2.1.4. **Preloading against Access/time-driven Attacks**

Preloading of all lookup tables before cryptographic operations aims to mask cache access activities to prevent cache attacks [94]. However, preloading still provides no security guarantee against access-driven attacks since the adversary may still use spy processes to evict the lookup tables after the preloading process.

5.2.1.5. **Hybrid Approaches**

In [79], several defense techniques are combined to provide secure and efficient protection for AES.

* **Software version 1 (v1):** All rounds use the compact S-box table without permutation. Preloading is performed before each round. This is a combination of (3) and (4).

* **Software version 2 (v2):** The most vulnerable rounds (first round and last round) use the compact S-box table. The other rounds use large pre-computed lookup tables. All tables are permuted. Preloading is performed before the first round and the last round. It is a combination of (2), (3) and (4) and it trades security for improved performance.

* **Software version 3 (v3):** All rounds use the compact S-box table with permutation. Preloading is performed before each round. This is a combination of (2), (3) and (4).

Among the three, the v3 is most secure as it combines all three mitigation techniques. The v2 is most performance efficient as it uses large pre-computed lookup tables in inner rounds and does not preload them.
5.2.2 Hardware Countermeasures against Data Cache Attacks

Realizing the limitations of software countermeasures, several hardware schemes are proposed to provide comprehensive, efficient, and generic solutions (i.e., not application/attack specific) to defend against cache attacks.

5.2.2.1. Partitioned Cache and Partition-Locked Cache (PLcache)

In partitioned caches [95], a part of the cache is allocated exclusively to the protected process in order to prevent information leakage. This may cause inefficient cache sharing since the cache partition is fixed statically. In a recent work, the partition-locked cache (PLcache) [104] is proposed to address this problem with a fine-grained locking control so that only the cache lines, which contain the critical data, are isolated. The hardware support for the PLcache includes two additional fields in each cache line: an ID field and a lock bit. The ID indicates the owner of the cache line, normally a process and the lock bit indicates the locking status of the cache line. As for control interface, two mechanisms are proposed: ISA extension and segment/page-based protection. The first introduces several new instructions to provide fine-grain locking control of the cache lines. The second involves new OS-level API calls for coarse-grain control of memory regions. Also the cache line replacement policy is changed to support the locking mechanism.

5.2.2.2. Random-permutation Cache (RPcache)

In contrast to partitioned caches, the random-permutation cache (RPcache) [104] allows flexible cache sharing but randomizes the mapping between memory addresses (i.e., table indices) and cache lines to prevent information leakage. In the RPcache, in case of cache
interference, i.e. when the fetched cache line and the chosen replacement cache line belong to two different processes, the original cache set will not be used for replacement. Instead, another cache set is chosen randomly and replacement happens in that set. This changes the mapping between addresses and cache sets. Because of the swapping of cache sets, the cache lines in the original sets are invalidated. The hardware support includes a permutation table and a revised replacement policy.

5.2.2.3. Security Issues with the PLcache and RPcache

The PLcache can still be vulnerable to both types of cache attacks since AES may still experience cache misses over the critical data before all of them are fetched and locked in the cache. While software can be used to pre-load the AES tables, such initial loading of the critical data may still provide enough information leakage for key recovery. Besides, the PLcache does not support locked cache lines to be replaced even when they are not needed (i.e. the owner process is switched out and not active). This may cause excessive locking (unless properly controlled by the OS [104]), and in any case reduces the size of the cache available to other processes. The RPcache defeats access-driven attacks because even if an adversary knows which cache lines are accessed by a crypto operation, the corresponding index cannot be derived due to random permutation. The vulnerability of the RPcache, however, lies in its inability to defend against cache-collision time-driven attacks since random permutation does not eliminate the execution time variances: a high number of collisions still results in lower execution times. More detailed security analysis and examples of successful attacks to the PLcache and RPcache are presented in [87].
5.2.3 Software Approaches for Removing Key-dependent Control Flow

To defend against side channel attacks which exploit key-dependent control flow, existing work focus on two issues: the elimination of key-dependent branches and the removal of associated key-dependent control flow divergence.

![Figure 32. Various approaches to remove key-dependent control flow](image)

5.2.3.1 Heuristic Methods through Source Code Conversion

A source code transformation is proposed to remove key-dependent branches and associated control flow divergence in [88]. For example, as shown in Figure 32(a), the control flow inside the `for` loop can give out the secret bit value. In the converted code shown in Figure 32(b), a temporary variable T is introduced. Depending on the secret key bit value di, -di is either all 1s or all 0s while the least significant bit of ~di is either 0 or 1. This way T is set to be M

```plaintext
S = M;
for (i = 1; i< n; i++) {
    S = S * S (mod N);
    if (di == 1)
        S = S * M (mod N);
}
```

```plaintext
S = M;
for (i = 1; i< n; i++) {
    S = S * S (mod N);
    T = (M & (-di)) | (1 & (~di));
    S = S * T (mod N);
}
```

```plaintext
S = M;
for (i = 1; i< n; i++) {
    S = S * S (mod N);
    T = (&& taken);
    goto *(void *)T;
}
```

```plaintext
S = M;
for (i = 1; i< n; i++) {
    S = S * S (mod N);
    T = (&& not_taken);
    if (di) T = (&& taken);
    goto *(void *)T;
}
```

S = M;
(when \( d_i \) is 1) or 1 (when \( d_i \) is 0). Therefore the converted code is equivalent to what the original code does. The price paid for such conversion is a large performance overhead. The overhead is introduced by the extra non-trivial modular multiplication with 1 when \( d_i \) is 0. To eliminate the extra modular multiplication with 1, checking the value of \( T \) being 1 is necessary. In that case, however, we are going back to the original key-dependent control flow vulnerability.

5.2.3.2. **Predicated or Conditional Execution through Compiler Support**

There are several practical issues for the above discussed heuristic software approach. For example, as indicated in [82], it can be hard to transform complex code with various function calls. The method may not even work properly because of side effects introduced by extra code/data. Compiler optimization may also remove the deliberately transformed code.

To address the aforementioned issues with heuristic methods, a compiler-based approach is proposed in [82]. As shown in Figure 32(c), a temporary result is computed for the extra modular multiplication. However, whether the temporary result is used or not depends on a condition. The \( \text{if} \) condition statement in the code is actually accomplished through predicated execution, which transforms control dependence to data dependence based on hardware support [82]. Other issues such as predicated execution across function calls, side effect eliminations are also addressed in the work. The tradeoff is still a large performance overhead since that modular multiplication is computed all the time regardless of the key value.

5.2.3.3. **Using Indirect Jump against BTB/I-cache Attacks**

Using indirect jump to replace key-dependent conditional branches is proposed in [75]. As shown in Figure 32(d), predicated execution is performed based on \( d_i \) to decide the target
code address. The modular multiplication is only performed when \( d_i \) is 1. Compared to the previous two approaches, using indirect jump avoids unnecessary extra computations and therefore avoids large performance overhead. However, it has one major security issue as the code still has control flow divergence depending on the secret key. Therefore this approach is still vulnerable to I-cache attacks.

To apply indirect jump to defend against I-cache attacks, a possible solution is to use indirect jump to interleave two cache lines from two different diverged control flow paths. In that case, each cache line would contain a part of instructions along the taken path and a part of instructions along the non-taken path. The indirect jump would link two parts of the taken/not-taken path together. This way, both cache lines would be loaded into instruction caches no matter whether the branch is taken or not. However such approach requires balanced execution path for control flow, which is very hard to achieve efficiently if not possible.

5.3. Integrated Hardware-Software Protection Schemes against Data Cache Attacks

In this section, we propose three hardware-software approaches to eliminate the source of information leakage exploited in software data-cache-based attacks. First, we propose to secure the PLcache by pre-loading the critical data (e.g. the AES lookup tables). Second, we advocate using informing loads to protect the RPcache from time-driven attacks. With the support for informing loads, we are able to respond to the source of information leakage – caches misses over the critical data and integrate flexible software-based defenses. Third, we propose informing loads assisted software-based random permutation to replace the permutation logic in the
RPcache so as to provide the security protection to regular caches. These three approaches present different tradeoffs between hardware complexity and performance overhead.

5.3.1 Preloading to Protect Partition-Locked Cache

5.3.1.1 The Idea

Previous works [77], [79], [94] discussed the concept of preloading or cache warming as a possible countermeasure for cache attacks. The basic procedure is to load all security critical data, e.g., the AES lookup tables, into the cache right before the crypto operations. Preloading itself, however, cannot provide sufficient protection against cache attacks simply because an adversary can still manipulate the cache state after the preloading process. As discussed in Section 5.2.2, the PLcache does not provide high security either due to the initial loading process of the critical data. However, combining preloading and PLcache provides a solid protection mechanism against access/time-driven attacks.

The key here is to make sure that before cryptographic operations all the critical data are preloaded and locked in the PLcache. After that, any access to those critical data will result in a cache hit. This way, time-driven attacks are effectively defeated since there is no correlation between the secret key and cache hit/miss patterns. This scheme also defeats access-driven attacks since all a spy process can observe is that the whole set of the critical data are in the cache, thereby leaking no information of which parts of the critical table (or the indices to the tables in AES) are used in a crypto operation. Here, note that the PLcache with preloading is secure only if all the critical data can fit in the cache. This issue is generally not a problem for cryptographic algorithms, of which a key design objective is to keep critical data small [83]. The five critical lookup tables in AES, for example, take 5KB (1KB for each table).
To address the issue of excessive locking associated with the PLcache, we propose to allow the locked cache lines to be replaced if the cryptographic process is switched out (i.e., not active). When the cryptographic process is switched back, those protected cache lines will be reloaded and locked again.

5.3.1.2. Proposed Implementation

Changes to the PLcache logic The change to the PLcache logic is that when a new non-protected cache line is about to replace a locked (protected) cache line, which is always prohibited in the PLcache, the replacement is now allowed if the owner process of the locked cache line is not active. This is done by comparing the ID field of a locked cache line with the active processes' IDs.

Architectural support for preloading One implementation of preloading and reloading is through hardware logic. In this implementation, a new preloading instruction will be used to specify the beginning address and length of the protected data. The preloading state (i.e., whether the preload has been performed or not and the address range of preloading) becomes part of the process context. Then, if the preloading state is set after a context switch, the hardware will reload all critical data. This pure hardware approach may be too costly and introduce extra hardware complexity.

The preloading and reloading can also be implemented through software in an unmodified PLcache system. The protected process performs the preloading and locking before critical operations and performs unlocking once the critical operations are completed. The operating system (OS), however, needs to be changed to perform the preloading and locking during context switches of the protected process. Instead, we propose to use user-level exception
handling to provide efficient preloading and offer flexibility to deploy newly developed software defense mechanisms.

User-level exception handling is introduced by Thekkath et al. [101]. It provides efficient handling of synchronous exceptions by user-level code. In our design, we implement it in the way described in [86]. When some instruction triggers a user-level exception, it works as a conditional branch. The exception only changes the program counter (PC) of the running process. It does not invoke any OS code. Necessary additions include two new instructions (EH-register and EH-jr) and two new registers (an exception handler address register-EHAR and an exception handler return register-EHRR). EH-register will load the entry address of the user-level exception handler to the EHAR. EH-jr is used at the end of the exception handler to jump to the address stored in EHRR to resume program execution. The procedure works as follows. The user-level exception event will trigger a pipeline squash when the offending instruction reaches the head of the Reorder Buffer (ROB). The next PC is saved to EHRR. Then the exception handler whose address is stored in EHAR will take over the execution and run as a regular function. It saves the registers that it will use to the stack at the beginning and restores those registers at the end. When the handling finishes, EH-jr will use EHRR to return to the interrupted process.

Besides the support for the user-level exception handling mechanism, we propose two new instructions (PL-begin and PL-end) and one new control status register (PL-S). PL-begin and PL-end are inserted to enclose the cryptographic operations that need to be protected. The status register PL-S indicates the state of locking and preloading.

The whole procedure works as follows:
1) The user-level exception handling mechanism registers the entry address of the user-level exception handler during program initialization.

2) When the crypto process comes to the cryptographic operation, PL-begin sets the status register (PL-S) to indicate that the PLcache locking mechanism and preloading become effective for this crypto process. In the meanwhile, it triggers the exception handler, which pre-loads and locks all the critical data.

3) During program execution, if a context switch happens, the status register PL-S will be saved along with other states of the process. The PLcache locking mechanism becomes ineffective for the switched-out process, by comparing every cache access with an Active Process register. When the process is switched back, the status register PL-S is examined. If PL-S indicates that the critical data needs to be loaded and locked, a user-level exception is raised and the same exception handler will reload and lock the critical data to the cache.

4) When the protected cryptographic operations are completed, the PL-end instruction resets the status register, indicating that the critical data are no longer needed. The PLcache locking mechanism becomes ineffective for the process.

5.3.2 Securing the RPcache with Informing Loads

5.3.2.1 The Idea

With cache-based attacks taking advantage of cache hit/miss behavior of crypto processes, we argue that the crypto process itself can leverage the same information to defend against the attacks. Informing loads, originally proposed as a lightweight architectural support for memory optimization [86], enable the crypto process to gain the control once an access to critical data misses in the cache. This way, flexible software defense mechanisms can be
deployed. In this section, we show that informing loads can be used to effectively address the security vulnerability of the RPcache. Because of its flexibility, our approach can be further extended to protect regular caches (Section 5.3.3).

As explained in Section 5.2.2, the RPcache defeats access-driven attacks by randomizing cache line mapping. However, it is vulnerable to cache collision attacks since not all critical data are guaranteed to reside in the cache. Therefore, the fundamental assumption behind collision attacks still holds. In other words, a higher number of collisions still results in lower encryption time. To protect the RPcache, we propose to use informing loads to access the critical data. In the case of AES, it means that all the table lookups are implemented using informing loads while other data accesses use regular load instructions. The informing loads detect whether the critical data is in the cache. If not, they will redirect the PC to a user-level exception handler. Note that preloading alone can’t provide sufficient security support against cache collision attacks as preloaded data can still be replaced. Such a case could be that during the execution, internal data of the victim process may replace the preloaded data, which is identified as internal interference in [104].

In this chapter, we devise an exception handler to load all the critical data (or the tables T₀-T₄, in AES) into the cache. The objective is that after the exception handling, subsequent accesses to the critical data will hit in the cache, thereby eliminating time variations. Note that, such data loading is just one possible solution. The key advantage of using informing loads over pure hardware-based defense mechanisms such as the RPcache is that the exception handler can be easily updated to defend/detect future cache-based attacks or to incorporate a better crafted defense algorithm.
5.3.2.2. Proposed Implementation

Informing loads are special load instructions that “inform” the software when the load misses in the cache. There are three ways of implementing informing loads [86]. The first is to use a cache outcome condition code and branch-and-link instructions. The second is a branch operation with a slot that is squashed if there is a hit. The third one is a low-overhead user-level cache miss trap. We choose to use the low-overhead cache-miss trap for its low hardware complexity.

```
r = random_number;
// from hardware random number generator (i.e. the one
// used by the RPeCache)
i_max = number_of_tables;
j_max = table_size / table_element_size;
for (i=0; i < i_max; i++) // Fetch each protected table
    for (j=0; j < j_max; j += cache_line_size/table_element_size)
        Prefetch(T[(i XOR r) % i_max][(j XOR r) % j_max]);
// Fetch each protected cache line in a random order
```

**Figure 33. Code of the informing load exception handler**

Using AES as an example, the proposed procedure works as following. In the cryptographic operations, those protected tables are loaded with informing load instructions. These informing load instructions work as normal loads with no extra overhead when they hit in the cache. Whenever an informing load misses in the cache, a user-level exception will be generated and the exception handler, shown in Figure 33, will be executed. As shown in Figure 33, the exception handler loads the critical tables in a random order. The reason is due to an artifact of the RPeCache, in which cache lines may be invalidated when a cache line index is randomized. As a result, if the exception handler loads the tables in a determined order, the elements that are loaded earlier have higher chances to be invalidated than those loaded later.
The random order (achieved by XORing a random number \( r \) in Figure 33) in the exception handler eliminates the determinism.

Combining informing loads with the RPcache defeats software cache-based attacks. The RPcache itself is effective against access-driven attacks and the informing load exception handling provides a defense mechanism against time-driven attacks. Due to the invalidation effect of the RPcache, in theory we still cannot guarantee that all the table access hit in the RPcache since some critical data may be invalidated during the permutation process. As a result, there may still be access latency variations among different table elements, although the randomized loading order in our exception handler already makes such variations non-deterministic. To completely overcome the problem, we can change the RPcache to let it selectively swap the cache lines instead of invalidating them during the permutation process. In other words, if a cache line to be invalidated contains some critical data, the content will be copied to the new location instead of being invalidated. Such a change increases the complexity of the RPcache but may further improve the security. With such implementation, the randomized loading order in the informing load exception handler can be removed.

5.3.3 Securing Regular Caches with Informing Loads

5.3.3.1 The Idea

To protect regular caches, we first use software random permutation to randomize the crypto process’ cache footprint against access-driven attacks. However, as discussed in Section 5.2.1, fixed permutation leaks information and frequent updates of permutation may incur unnecessary performance overheads. To overcome these problems, we propose to change the permutation only when it is necessary. As discussed in Section 5.1, cache attacks rely on
cache misses to identify whether a cache line is used by the victim process. Therefore, we choose to change the permutation whenever there is a cache miss of the critical data, which is supported by informing loads. In other words, we integrate permutation update in the exception handler of informing loads. Such informing loads assisted software permutation can also be viewed as a replacement of the permutation logic in the RPcache so that the security protection can be provided with no need for hardware changes for the RPcache.

To defeat time-driven attacks, loading of all critical data is also performed in the exception handler upon cache misses detected by informing loads, similar to the way to further secure the RPcache.

5.3.3.2. Proposed Implementation

The key of software permutation is to use one level of indirection to randomize the address mapping between the table lookup values and their memory addresses. Using the critical tables of AES as an example, we use an indirection table to produce the actual address of one protected unit, i.e. a cache line in our implementation, as shown in Figure 34.

In Figure 34, one dimension table $T$ with $N$ elements is converted into a two-dimension $K \times L$ array $T'$, where $K \times L = N$ and $L$ is the number of elements in one cache line, i.e. $L = \text{Cache line size} / \text{size of each table element}$. Compared to $T$, which occupies a continuous region of memory, the data in $T'$ are distributed in memory and are accessed through pointer indirection.
Figure 34. Converting a one-dimension table into a two-dimension array

With the critical data organized in a two-dimension array, we can easily perform address permutation. As illustrated in Figure 35, assuming that the address of table element $T[0]$ (i.e., &$T[0]$ or $T'[0]$) is 0x40, address permutation upon $T'[0]$ proceeds as follows. First, one entry among $T'[1, \ldots K-1]$ is randomly selected (assuming $T'[1]$ is selected and $T'[1] = 0x80$). Second, both the indirection pointers ($T'[0]$ and $T'[1]$) and the data pointed to ($*T'[0]$ and $*T'[1]$) are swapped. After permutation, the new address of $T[0]$ becomes 0x80. The data value of $T[0]$ (or $T'[0][0]$) is unchanged due to the data swapping between $*T'[0]$ and $*T'[1]$.

Figure 35. Address permutation by swapping both the pointers and the data

With the proposed way to perform permutation, we devise the exception handler for informing loads to defeat the cache attacks. The scheme works as follows. In the crypto algorithm (e.g., AES) implementation, two-dimension arrays are used to store the critical data (each of $T_0$-$T_4$ in AES). The table lookups use informing loads while other data accesses use regular load instructions. Once a critical data access misses in the cache, the exception handler for informing loads will be triggered. The exception handler prefetches all critical data to the
cache and meanwhile performs the permutation change upon the missing table entry, as illustrated with the pseudo code in Figure 36.

0. The cache line of $T[i]$ is missing in the cache
1. Prefetch from addresses $T'[0], T'[1], \ldots, T'[K-1]$
2. Find the corresponding $T'[p]$ that points to $T[i]$
3. Randomly select a target entry $T'[q]$ for permutation
4. Swap $*T'[p]$ and $*T'[q]$; swap $T'[p]$ and $T'[q]$

Figure 36. Pseudo code of the exception handler for informing loads, which prefetches all critical data to the cache and randomly permutes the cache lines

5.4. Defending against BTB Attacks and Instruction Cache Attacks

In this section, we propose hardware-based approaches to eliminate the sources of information leakage exploited in BTB and I-cache attacks. We propose to adopt an always BTB update policy as an effective yet efficient approach to defend against BTB attacks. We also propose to utilize the PLcache with preloading and the RPcache as an effective and efficient countermeasure against I-cache attacks.

5.4.1 An Always BTB Update Policy as a Countermeasure against BTB Attacks

Since BTB attacks exploit the fact that BTB is only updated when the conditional branch is taken, a simple yet effective solution would be to always update BTB no matter whether the branch is taken or not. This completely removes the correlation between the key-dependent branch and BTB update.

Two options are available for the proposed countermeasure. The first one would introduce a new branch instruction to the ISA. The BTB will be always updated only for these branches. It would introduce little interference to the shared BTB and therefore a very small
performance overhead. This approach is desirable for the case where key-dependent branches can be identified. The second would simply assume that it is very hard if not possible to identify all key-dependent branches. For example, an indirect correlation between control flow and the secret key is recently discovered and exploited in [74]. Therefore it is better to protect all branches in the critical process and have all of them always updated in BTB regardless of taken or not taken. Both approaches would only require a simple logic change to the BTB update policy and therefore incur almost no extra latency.

Aside from the always BTB update policy, also there could be other options, especially since BTB attacks are similar to data cache attacks in terms of exploiting cache interference. The proposed always BTB update policy can be regarded as a hardware implementation of the software access-all approach. Also BTB preloading shares the same weakness as lookup table preloading, i.e. there is no guarantee that the loaded BTB entries will not be evicted prior to the execution. For hardware approaches proposed for D-cache attacks, besides the hardware complexity, locking from the PLcache tends to occupy the BTB resource too much compared to the always update policy. For the always BTB update policy, when protected branches are not executed, the corresponding BTB entries are available to other branches. Locking, however, occupies those BTB entries all the time. Applying the RPcache to BTB would also add much hardware complexity and extra latency to the critical branch prediction unit. Overall, the always BTB update policy is a good countermeasure against BTB attacks in terms of both hardware cost and performance efficiency.
Using Hardware Countermeasures for Data Cache Attacks against I-cache Attacks

A countermeasure against BTB attacks is not good enough to defend against side channel attacks based on exploiting key-dependent control flow. The control flow divergence still exists. Attackers can exploit that by observing the instruction cache usage and still get the secret key through access-driven attacks. As I-cache attacks work in a similar way as D-cache access-driven attacks work, the same defense mechanisms for D-cache attacks might also be effective against I-cache attacks.

For preloading and locking, the first thing to do is to identify all critical instructions in divergent execution paths because of key-dependent control flow. Prior to the execution of the key-dependent control flow, preloading and locking of those critical instructions is necessary. During the process, instructions from divergent execution paths may compete for space in each cache set. Here two cases may happen. In the first case, for the cache set, all instructions can fit in. Preloading and locking can be completed successfully in this case. In the second case, not all instructions can fit in and certain critical cache lines may be left out. In both cases, however, security is still ensured as I-cache attacks cannot tell the access activities of those critical cache lines of the crypto process. The downside, however, is that there would be more cache misses and therefore performance loss if locked cache lines are not used frequently.

For the RPcache, a hardware indirection table is introduced. For every access to the instruction cache, the original cache set index needs to be translated to an actual cache set index through the hardware table. Any cache interference between the protected process and another process would trigger the randomization in order to obfuscate attackers’ observation. Compared to preloading and locking, the RPcache incurs larger hardware cost because of the indirection table. It, however, may introduce small performance overhead as cache sharing is allowed.
For informing loads, to defend against access-driven attacks, a software indirection table is required as randomization is performed through the table. For instruction caches, however, such software indirection table would be too slow to catch up with the high-throughput requirements from instruction fetch.

### Table 4. Default processor configuration

| Branch Predictor                  | 64K-entry g-share Branch Target Buffer (BTB)  
<table>
<thead>
<tr>
<th></th>
<th>15-cycle minimum branch miss-prediction penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar Core</td>
<td>7-stage pipeline: Fetch/Dispatch/Issue/RegisterRead/EXE/WriteBack/Retire, Pipeline bandwidth:4</td>
</tr>
<tr>
<td></td>
<td>Fully-symmetric Function Units: 4</td>
</tr>
<tr>
<td></td>
<td>Reorder Buffer (ROB) size: 128</td>
</tr>
<tr>
<td></td>
<td>Issue Queue (IQ) size: 64</td>
</tr>
<tr>
<td></td>
<td>Load Store Queue (LSQ) size: 64</td>
</tr>
<tr>
<td>Execution Latencies</td>
<td>Fetch Policy for SMT: round-robin</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>32KB 2-way, Block size 64B</td>
</tr>
<tr>
<td></td>
<td>10-cycle miss penalty</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32KB 2-way, Block Size 64B</td>
</tr>
<tr>
<td></td>
<td>10-cycle miss penalty</td>
</tr>
<tr>
<td></td>
<td>8 Miss Status Handling Registers (MSHRs)</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>2MB 16-way Block size: 64B</td>
</tr>
<tr>
<td></td>
<td>300-cycle miss penalty</td>
</tr>
</tbody>
</table>

## 5.5. Experiments

### 5.5.1 Methodology

Our experiments are conducted using a detailed timing simulator developed from the SimpleScalar toolset [80]. The underlying processor model is MIPS R10000 and the default configuration is listed in Table 4. We implemented both the PLcache and RPcache in the simulator. Proper architectural support for informing loads is included in the simulator. The
detailed user-level exception handling including pipeline squashing, control flow transfer to and from the user-level handler is implemented to faithfully measure the performance of our proposed approaches. For the case of BTB, when a branch is predicted taken but there is a BTB miss, the instruction fetch will simply continue fetching the next instruction as in a not-taken case.

Our AES code is extracted from the OpenSSL 0.9.7c implementation. The key size is 16 bytes. For performance evaluation, the OpenSSL speed test program, a standard microbenchmark program included in OpenSSL [90], is used as our benchmark program. In the program, AES runs in the cipher-block chaining (CBC) mode. We use the message size of 8KB in our experiments. In terms of the performance metric for AES, we use cycles per byte instead of instructions per cycle. The reason is that different hardware/software approaches have different implementations, different lookup tables or different number of instructions. Cycles per byte, in contrast, directly reflects the throughput of AES: the number of bytes encrypted per time unit.

Our RSA implementation is extracted from the OpenSSL 0.9.7c implementation. The key size is 2048 bits. Special attentions have been paid to compile the extracted RSA source code to PISA-based binary code and have it running faithfully in our simulator. To ensure the correctness, RSA signature generation and verification routines are completed successfully in our simulator. For performance evaluation, the OpenSSL speed test program, a standard microbenchmark program included in OpenSSL [90], is used as our benchmark program. In the program, RSA runs in signature mode. Similar to [72],[73], we make the window size be 1 to have a square-and-multiply version of RSA, but with no other further changes.
5.5.2 Security Analysis

5.5.2.1. Microarchitectural Effects on Cache Collision Time-driven Attacks

We first investigate the microarchitectural effects on cache-collision time-driven attacks against AES. The reason is that both instruction-level parallelism (ILP) and memory-level parallelism (MLP) affect the encryption time. With a high degree of parallelism, many cache misses can be overlapped, thereby reducing the impact from cache collisions. In this experiment, we use five processor configurations as shown in Table 5 and examine the relationship between the encryption time and the number of cache collisions. In the experiment, a clean cache state is established before the 128-bit-key standard OpenSSL AES encryption and 16-million samples are collected for each processor configuration. Since different processor configurations lead to different encryption times, we report the relative encryption times in Figure 37, in which the encryption times are normalized to the mean encryption time with the same configuration. For reference, we also include the results from a real Pentium 4 machine in Figure 37.

<table>
<thead>
<tr>
<th>Configuration 1</th>
<th>In order issue, 1way-issue, 1MSHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 2</td>
<td>In order issue, 4way-issue, 4MSHRs</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>Out-of-order execution, 4way-issue 4MSHRs, 64ROB/32IQ/32/LSQ</td>
</tr>
<tr>
<td>Default</td>
<td>Out-of-order execution, 4way-issue 8MSHRs, 128ROB/64IQ/64/LSQ</td>
</tr>
<tr>
<td>Configuration 4</td>
<td>Out-of-order execution, 8way-issue 32MSHRs, 256ROB/128IQ/128/LSQ</td>
</tr>
</tbody>
</table>

From Figure 37, it can be seen that the collision attacks are most effective against single issue, in-order processors with limited MLP. Both out-of-order (OOO) execution and high degrees of MLP reduce the effect of cache collisions. Furthermore, we perform AES final-round collision attacks on those samples. The analysis tool that we used is from [89], which performs
collision attack analysis (i.e., key search) on AES encryption samples using a number of artificial intelligence techniques and reports the number of samples required to recover the complete key. The results, which are shown in Table 6, validate our observations. For processor configuration 1, around 4k samples are enough to break the key. In comparison, it takes around 400k samples to break the key for processor configuration 4. Our experiments on these processor configurations with the RPcache also show similar results for the different processor configurations, confirming that the RPcache is still vulnerable to collision attacks. From this experiment, it can also be seen that encryption times of modern high performance processors is less correlated to the number of cache collisions (or the number of cache misses) than in-order processors. This is due to the effect of OOO execution and MLP. However the vulnerability to collision attacks remains since the number of samples required to reveal the correlation is still in feasible ranges.

![Figure 37. The relationship between the number of collisions in the final round and the normalized encryption time on various processor configurations](image-url)

-1.0%
-1.5%
-2.0%
-2.5%
-3.0%

Normalized Execution Time

0 1 2 3 4 5

Configuration 1
Configuration 2
Configuration 3
Pentium 4
Default Configuration
Configuration 4
Table 6. Required numbers of samples for key recovery on various processor configurations. (based on 25 random keys with random input plaintext)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>min</th>
<th>median</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 1</td>
<td>3k</td>
<td>4k</td>
<td>5k</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>6k</td>
<td>10k</td>
<td>13k</td>
</tr>
<tr>
<td>Configuration 3</td>
<td>7k</td>
<td>11k</td>
<td>13k</td>
</tr>
<tr>
<td>A real Pentium 4</td>
<td>12k</td>
<td>20k</td>
<td>27k</td>
</tr>
<tr>
<td>Default Configuration</td>
<td>17k</td>
<td>24k</td>
<td>29k</td>
</tr>
<tr>
<td>Configuration 4</td>
<td>328k</td>
<td>393k</td>
<td>459k</td>
</tr>
</tbody>
</table>

5.5.2.2. Security Evaluation of the Proposed Schemes

5.5.2.2.1. Protection against Data Cache Attacks

The PLcache with preloading (PLcache+PL) As discussed in Section 5.3.1, the PLcache with preloading ensures that all critical data reside in the cache throughout the crypto process' lifetime. It defeats access-driven attacks since the only information that a spy process/thread can obtain is that all the critical data are used. It defeats time-driven attacks as any access to the critical data will hit in cache, thereby no timing variations.

The RPcache with informing loads (RPcache+IL) Through random permutation, the RPcache is effective against access-driven attacks (see [104] for the theoretical proof). With informing loads, any access to the critical data, if it misses in the cache, will invoke the exception handler to load all the critical data. Next, we examine how well this approach mitigates cache-collision attacks.

We first examine the relationship between the encryption time and the number of cache collisions as shown in the Figure 38. As analyzed in Section 5.5.2.1, processor configuration 1 is most vulnerable to collision attacks. Therefore, we use this processor configuration to evaluate various cache designs. For the RPcache enhanced by informing loads (Config. 1 with RPcache+IL), we observed no evident correlation between the encryption time and the number
of cache collisions compared to the RPcache case (Config. 1 with RPcache). Then, we repeated the key recovery experiments. Compared to the RPcache, upon which 8K samples are enough for a complete key recovery, the attack fails on the RPcache+IL when presented with 16-million samples. More samples have not been tried due to the required simulation time as it takes 16 days to simulate one 16-million encryption run. Attacks on other processor configurations equipped with the RPcache+IL also failed when presented with 16-million samples.

From the experiments, we can conclude that using informing loads with the proposed exception handler can greatly enhance the security of the RPcache. However, as discussed in Section 5.3.2.2, if further cache security is desired, one can choose to selectively swap cache lines instead of invalidating them. This way, the cache collision attacks upon uniprocessors can be completely defeated since all the tables reside in the cache and all the access to those tables are cache hits. For more complex cache-collision attacks upon multi-threaded processors, further exploration is necessary and left as our future work.

**Regular caches with informing loads (Regular cache+IL)** For access-driven attacks, our solution is to use software permutation to randomize the mapping of the cache lines that contain critical data. Every time if the adversary tries to exploit a cache miss to observe the cache usage, the permutation varies. In fact, the permutation with updates on cache misses can be

![Figure 38](image-url). The effect of our informing loads approaches on the relationship between the number of collisions in the final round and the normalized encryption time.
viewed as a software implementation of the RPcache design, which is already proven to be secure from access-driven attacks from the information theory perspective [104].

For time-driven attacks against AES, again we examine the relationship between the encryption time and the number of cache collisions as in the previous sections and the results are also in Figure 38. From Figure 38, it can be seen that for processor configuration 1, there exists no evident correlation between the encryption time and the number of cache collisions in our informing loads approach (Config. 1 with Regular cache+IL) compared to the regular cache case (Config. 1 with Regular cache). Next, we repeat the cache-collision attacks to evaluate the effectiveness of our approach against time-driven attacks on the processor configuration 1. For the regular cache the tool successfully retrieves the key with less than 8K samples. For the cache protected by our approach, the tool fails on 16-million samples, demonstrating our approach’s effectiveness. Attacks on other configurations equipped our regular cache+IL also failed on 16-million samples. This is similar to what we observed from the RPcache+IL, as both use reloading of all the critical data as the countermeasure against collision attacks.

5.5.2.2.2. Protection against BTB/I-cache Attacks

As discussed in Section 5.4.1, the proposed BTB always update policy completely removes the correlation between the outcomes of key-dependent conditional branches and BTB update. Therefore our scheme defeats BTB attacks.

Similar to their protections against D-cache access-driven attacks, the PLcache with preloading and the RPcache provide complete security guarantee against I-cache attacks. The PLcache with preloading ensure that a spy process can only know that all instructions from all
diverged execution paths are executed. The RPcache ensures that a spy process may never know which exact instruction cache line is accessed through dynamic randomization.

5.5.3 Performance Evaluation

In this section, we study the performance impact of our proposed approaches against data cache attacks, i.e., preloading on top of the PLcache (PLcache+PL), informing loads combined with the RPcache (RPcache+IL) and informing loads combined with the regular cache (Regular cache+IL), upon AES, which represents the code to be protected. The baseline is the standard OpenSSL AES implementation which uses 5 precomputed lookup tables with the total size of 5KB.

We also study the performance impact of our proposed schemes upon RSA, i.e., the always BTB update policy against BTB attacks, the PLcache with preloading and the RPcache against I-cache attacks. The baseline is the OpenSSL RSA implementation with the window size of 1. For experiments on BTB, the always BTB update policy is effective for all branches in RSA. For experiments on the PLcache with preloading, the key-dependent diverged execution paths involve mainly two unique functions: $BN\_sqr$ and $BN\_mul$. These two and their associated unique sub-routines are preloaded and locked.

5.5.3.1 Performance Impact

5.5.3.1.1 The Case of AES

In this experiment, we analyze the performance impact of different protection schemes on AES using various L1 data cache configurations. We vary the L1 data cache sizes from 8KB to 32KB and the set associativity from 1-way to 4-way. The throughputs normalized to the baseline
results (i.e., regular cache) are shown in Figure 39. Among different cache designs, the PLcache and PLcache+PL have almost the same performance since all critical data are locked in both designs after the short warm-up phase.

From Figure 39, we make the following observations. First, for the 8KB direct-mapped cache, the PLcache (and PLcache+PL) incurs non-trivial performance overhead (26%). It is because locking introduces extra cache conflict misses over the protected cache lines. With larger caches and higher associativities, the number of conflict misses is reduced, resulting in smaller performance overhead (e.g. 15% of the baseline for 16KB direct mapped cache and almost 0% of the baseline for 16KB 2-way). Second, the RPcache has almost no performance difference compared to the baseline results. This is because the performance impact caused by random invalidation is small. These results also agree to that reported with the original PLcache and RPcache [104]. Third, the RPcache+IL has substantial performance overhead for 8KB direct mapped cache (63%). The reason is that because of frequent conflict misses, informing loads exception handler is invoked frequently, wasting lots of cycles. With larger caches and higher associativities, the cache is able to hold the working data set of AES and the RPcache+IL has
low performance overhead (only 8% overhead for the 8KB 2-way cache and almost 0% overhead for the 8KB 4-way cache).

For our informing loads with the regular cache (Regular cache+IL), we also compare it to the software-based hybrid protection schemes (v1, v2 and v3 in Section 5.2.1). Since v1 and v3 always cause more than 6X slowdown to the baseline in our experiments, Figure 39 only includes the results for v2. Figure 39, it can be seen that due to extra instructions (pointer indirection in our approach and S-box-based computation as well as permutation in v2), both v2 and Regular cache+IL have non-trivial performance overhead compared to the protection schemes with special cache designs. For 8KB direct mapped, 8KB 2-way, 16KB direct mapped and 32KB direct mapped caches, our scheme incurs higher performance overhead than v2. This is due to the large number of conflict cache misses over the protected cache lines, which lead to frequent invocation of the exception handler. As the cache size and set-associativity increase, such performance overhead quickly diminishes. The L1 data caches in modern commodity processors often have capacity bigger than 8KB and/or set associativity higher than one-way (as indicated in Intel® 64 and IA-32 Architectures Software Developer's Manuals). With those configurations, our approach incurs much smaller performance overhead compared to v2. For example, for the 16KB 2-way cache, our scheme incurs 40% overhead while v2 has 67%. For the 32KB 2-way cache, our scheme has 37% overhead and v2 has 67%. The main reason is that our proposed approach only responds to the potential dangerous event (i.e., cache miss over critical data), during which permutation updates and loading of all critical data are performed. In comparison, pure software approaches such as v2 have to perform these operations no matter whether there is a potential information leakage event or not. Furthermore, v2 offers relatively
lower security levels because of its treatment of the inner rounds of AES as discussed in Section 5.2.1.

5.5.3.1.2. The Case of RSA

In the first experiment, we analyze the performance impact of the always BTB update policy on RSA with various BTB cache configurations. We vary the BTB cache sizes from 256-entry to 4096-entry and the set associativites from 1-way to 4-way. The baseline configuration has a BTB update policy in which the BTB is updated only upon taken branches. For all BTB configurations, the performance differences between the new BTB update policy and the baseline one is almost negligible (0.5% or less of the baseline). One of the reasons for such small performance differences is that RSA is computation-intensive. IPCs (instructions per cycle) for RSA are from 1.84 to 2.23 in our experiments. Another reason is that the differences between the new policy and the regular one in terms of BTB cache miss rates are small (being 1.2% or less).

In the second experiment, we analyze the performance impacts of the PLcache with preloading and the RPcache for protecting the instruction cache. We vary the L1 instruction cache sizes from 8KB to 32KB and the set associativities from 1-way to 4-way. The baseline configuration has regular instruction caches. For all instruction cache configurations, the RPcache has almost the same performance as regular instruction caches since there is almost no external interference from other processes. For the PLcache with preloading, the relative performance differences are almost negligible (0.5% or less of the baseline). The reason is that for RSA instruction cache miss rates are very small. Even for the worst case, for an 8KB direct-mapped instruction cache, the cache miss rate is only 1.6%.
5.5.3.2. **Performance Impact on an SMT Processor**

In the experiment, we examine the performance impact of the protection schemes on SMT processors. We use two-way SMT processors and have AES/RSA run together with one of the ten SPEC2000 INT benchmarks (bzip2, gap, gcc, gzip, mcf, parser, perl, twolf, vortex and vpr). For each benchmark, SimPoint [100] is used to select a simulation phase of 300-million instructions. We vary the L1 data/instruction cache configurations from 8KB direct-mapped to 16KB, 32KB and 64KB 4-way set-associative. We vary the BTB cache configurations from 512-entry direct-mapped to 1024-entry, 2048-entry and 4096-entry 4-way set-associative. We report two metrics for each cache configuration, the overall instructions per cycle (IPC) for throughput and the Hmean metric (Harmonic mean of the IPC speedup/slowdown of each separate thread [81]) for fairness.

5.5.3.2.1. The Case of AES

The results for AES are shown in Figure 40. Note that although here we use instructions per cycle as the performance metric for AES implementations, these IPCs are relatively normalized to the baseline’s IPC in terms of AES throughput (cycles per byte) to ensure fair overall IPC comparison. Here, the baseline is the standard OpenSSL AES implementation and assume that it takes $N$ instructions to encrypt a message. Due to added defense mechanisms, a software approach, e.g., v2, may require a different number of instructions (e.g., $M$) to encrypt the same message. To ensure fair comparison, the IPC of v2 is computed using the baseline instruction count, $N$. Our results exclude the instructions for the user-level exception handler.
From Figure 40, we can make the following observations on PLcache+PL and RPcache+IL. First, for the 8KB direct-mapped cache, the PLcache (and PLcache+PL) incurs performance degradation in throughput (8% on average) because locking introduces extra cache misses. When measuring Hmean, however, the PLcache reports 6% improvement. The reason is due to the memory-intensive benchmark \textit{mcf}, which dominates the cache usage and significantly affects the performance of AES. With the PLcache, the AES lookup tables are locked, thereby reducing such negative impact from \textit{mcf} upon AES. With \textit{mcf} excluded, PLcache reports a 4% loss on Hmean. The RPcache improves both the throughput (7%) and Hmean (7%) because the cache line relocation alleviates the cache conflict problem associated with the direct mapped cache. Second, for 16KB, 32KB and 64KB 4-way set-associative caches, the PLcache achieves very small throughput improvement. This is because those cache configurations have enough capacity for the working sets of both AES encryption and the SPEC CINT benchmarks and locking helps to avoid cache misses on the protected tables. The RPcache has a little degradation in IPC (1% for 16KB) because of the effect of invalidations from cache line relocation. This effect tends to diminish for large caches such as 32KB (almost 0%) caches due to their higher
number of cache sets. These results also agree to that reported with the original PLcache and RPcache [104]. In terms of Hmean, the PLcache (PLcache+PL) and RPcache have very limited impact compared to the baseline caches. Third, the RPcache+IL incurs higher overhead compared to the original RPcache. The reason is due to user-level exception handling. For 8KB directed mapped and 16KB 4-way caches, the AES lookup tables are frequently replaced with the data from SPEC benchmarks. Therefore, the exception handler is invoked frequently, resulting in 49% and 45% losses in IPC and 19% and 57% losses in Hmean for those two cache configurations, respectively. The low Hmean results for the RPcache+IL are due to inherently unfair usage of informing loads. Since only the AES code uses the informing loads, the performance loss due to invocations of the exception handler is mainly upon AES rather than the SPEC workloads, thereby indeed being unfair. Such performance loss is recovered when the cache size is increased to 32KB (12% loss in IPC and 12% loss in Hmean) and 64KB (5% loss in IPC and 4% loss in Hmean) as the lookup tables will reside in the cache for much longer time.

In Figure 40, we also show the performance on the most efficient software hybrid approach (Software v2) and the regular cache with informing loads scheme (Regular cache+IL) for two-way SMT processors. The results of v1 and v3 are not shown since both are much worse than v2. For throughput, both v2 and Regular cache+IL have significant throughput degradation for all cache configurations. In the meantime, Regular cache+IL gains higher throughput over v2 except for the 8KB direct-mapped cache. This is due to the diminishing number of conflict misses as the cache can hold the working sets of both threads. Therefore it results in a small number of invocations of the exception handler, which lead to small performance overhead. For fairness, v2 reports similar results to the baseline as v2 makes no use of informing loads, thereby is not affected. Regular cache + IL reports lower fairness results. The reason is due to the unfair
usage of informing loads: informing loads are only used in AES but not in the other thread. Therefore, when an informing load exception happens, it only affects AES and does not affect the co-running thread. The fairness of Regular cache + IL improves quickly when the cache goes from 8KB up to 64KB as a result of the reducing number of invocations of the exception handler.

5.5.3.2.2. **The Case of RSA**

![Figure 41. Performance impacts of the protection scheme over various BTB cache configurations in two-way SMT processors](image)

Figure 41 shows the results for RSA under the protection against BTB attacks. Overall, the performance differences between our proposed always BTB update policy and the baseline are small (less than 1% of the baseline) for both throughput and fairness. The proposed policy actually increases the performance a little bit. This is because often the new policy tends to actively cache BTB entries for branches in RSA, while in the baseline scenario BTB is only updated when the branch is taken. In RSA, there are many iterations to execute (2048 iterations for a 2048-bit key). The new policy increases the performance of the RSA thread and the gain is often larger than the loss from the other SPEC2000 thread.
Figure 42 shows the results for RSA under the protections against I-cache attacks. Overall, because of small instruction cache miss rates, the performance differences between the protection schemes and the baseline are small for both throughput and fairness. For the PLcache with preloading, very small performance degradation for an 8KB directed-mapped instruction cache is introduced because of the locking effect. The effect tends to diminish as the cache size and associativity increase. For the RPcache, it actually increases the performance for an 8KB direct-mapped as the invalidation mitigates the interference between RSA and the other SPEC2000 thread (2% increase for the throughput and 3% increase for the fairness). When the cache size and associativity increases, the RPcache causes performance degradation as it invalidates valid cache lines. Such degradation diminishes as bigger caches can hold most of the instruction memory footprint and reduce interference between two threads.

5.5.4 Summary

Protecting computer systems is often about selecting the tradeoff between security and efficiency. A stronger level of security protection usually comes at extra cost. Here we use performance overhead, hardware complexity, flexibility, compatibility with legacy programs and
software change complexity as the factors for consideration and present a comparison among various protection schemes against software data-cache-based side channel attacks, as shown in Table 7. For security, although the PLcache and RPcache designs are effective against the two cache attacks analyzed in [104], they are still vulnerable to other cache attacks [87]. Our proposed hardware-software integrated approaches address the source of information leakage and are able to mitigate the latest cache attacks. Software hybrid approaches are also able to provide certain resistance against cache attacks but may trade security for performance, such as v2. From the perspective of performance overhead, the special hardware designs reduce the performance cost, while our informing loads with regular cache approach incurs non-trivial performance degradation because of extra pointer indirection. PLcache+PL and RPcache+IL pose a small to medium performance overhead upon the original cache designs, while pure software approaches usually incur the highest performance overhead. In terms of the ability to evolve when better software countermeasures are crafted or new attacks are developed, pure hardware designs lack the flexibility compared to software approaches, as evidenced by the vulnerabilities of the original RPcache designs to some timing-based attacks. Our proposed hardware-software integrated approaches combine some of the performance advantage of the hardware design with the flexibility of the software defense. Among them, the PLcache+PL and RPcache+IL provide performance efficiency at the cost of extra hardware and complexity and our approach for regular caches has smaller performance overhead compared to the pure software approach, for caches larger than 16 KB, while requiring only minor architectural support for informing loads. For compatibility with legacy programs, RPcache is able to provide protection without requiring any software changes. In terms of the complexity of software changes PLcache requires relatively small change in order to provide fine-grain locking control.
of cache lines and RPcache requires no software changes, while PLcache+PL and RPcache+IL need the user-level exception handler. In comparison, regular cache+IL and software v1, v2, v3 introduce software random permutation and other changes to the target programs.

Table 7. A comparison between various protection schemes

<table>
<thead>
<tr>
<th>Mitigation against access-driven attacks?</th>
<th>PLcache</th>
<th>RPcache</th>
<th>PLcache+PL</th>
<th>RPcache+IL</th>
<th>Regular cache +IL</th>
<th>Software v1, v2, v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, with SW preloading</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mitigation against time-driven attacks?</td>
<td>None-Small</td>
<td>None+ Small-Medium*</td>
<td>Small-Medium*</td>
<td>Medium-High*</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Performance overhead</td>
<td>Non-Small</td>
<td>None-trivial</td>
<td>Non-trivial</td>
<td>Non-trivial</td>
<td>Light-weight</td>
<td>None</td>
</tr>
<tr>
<td>Hardware changes</td>
<td>Trivial</td>
<td>Non-trivial</td>
<td>Non-trivial</td>
<td>Non-trivial</td>
<td>Light-weight</td>
<td>None</td>
</tr>
<tr>
<td>SW flexibility in handling misses</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Compatibility with legacy programs</td>
<td>Yes with OS change</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Software change complexity</td>
<td>Small</td>
<td>None</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

*Note: for small size and low-associativity caches, the performance overhead would be high
+Note: Performance can increase for RPcache for small cache size and direct-mapped caches

5.6. Conclusions

Software cache-based side channel attacks pose serious threats to the security of computer systems. In this chapter, we review current cache attacks including data cache attacks, instruction cache attacks and branch target buffer attacks and identify the weakness of existing hardware/software countermeasures. We then propose integrated hardware-software approaches to provide stronger security protection. To defend against data cache attacks, we use preloading to protect the PLcache from the initial loading exposure. A light-weight hardware support, informing loads, is used to detect the sign of potential danger - cache misses of critical data and then to deploy flexible software countermeasures. We propose to use informing loads combined
with simple yet effective software countermeasures to protect both the RPcache and regular caches. We also apply similar approaches such as the PLcache with preloading and the RPcache to protect instruction caches. To defend against branch target buffer attacks, we propose to adopt an always BTB update policy to remove the correlation between key-dependent conditional branches and BTB update. Experimental results show that these approaches achieve strong security protection at relatively low performance overheads.
List of References


CHAPTER 6. CONCLUSIONS

Computer security and privacy is an important issue of computer systems. As the size and complexity of modern computer hardware and software design grow, the opportunities for malicious security attacks also grow. To protect computer systems from ever-growing security threats, a good defenses mechanism needs to be effective as well as efficient. In this dissertation, we propose several novel methods to improve computer security and privacy from architectural point of view. They provide both strong protection and performance efficiency.

In our first approach, we analyze new non-control data attacks, which exploits popular vulnerabilities such as buffer overflow and format string. These new attacks differ from previous attacks in a way that they exploit non-control data instead of control data for break-in. Therefore previously proposed dynamic information flow schemes (also called tainting architectures) which focus on control data cannot detect such attacks, even though the whole taint tracking mechanism is already in place. Our new solution is to take advantage of the existing taint tracking infrastructure and to detect such new threats from the instruction side. In our approach, the instructions are classified into two different types based on tainted or non-tainted data that they handle. The instructions which are used to deal with non-tainted data only can therefore be used to guard certain non-control data and prevent some non-control data attacks. Such architectural solution incurs only small changes to existing hardware proposals while maintaining small performance overhead.

In our second approach, we investigate the issue of using counter-mode encryption to protect the privacy of a new emerging non-volatile main memory technology – phase change
memory (PCM). PCM is very promising as a next-generation memory technology because of its excellent merits. However its limited programming endurance is one weakness for its adoption in main memory. Therefore several wear leveling techniques have already been proposed to improve the lifetime of PCM-based main memory. In the meanwhile, the non-volatility of PCM jeopardizes the privacy of its content. We show that encryption for privacy protection has negative impact on some of the previously proposed wear leveling techniques for PCM because of the diffusion effect. In particular, diffusion reduces the bit-level redundancy a lot and completely eliminates the benefit of partial writes. To mitigate such negative impact on PCM lifetime, we propose a new encryption counter scheme to revive partial writes by using more fine-grain encryption counters, i.e. using one local counter for each encryption block in a cache line. To further extend PCM lifetime, we argue that error correct code (ECC) is necessary as ECC can detect and correct a certain number of bit errors to prevent the whole chip to fail early. However, a static ECC configuration would cause memory space waste since during the most of PCM lifetime there won’t be as many errors as ECC is set up to address. We propose an adaptive ECC management scheme to dynamically adjust the designated ECC protection coverage and therefore to save memory space, which can potentially improve memory performance. The adaptive management is enabled by monitoring the wear-out status of the memory. Also we leverage existing encryption counters as wear-out status (age) counters for efficient monitoring.

In our third approach, we first deconstruct two previously proposed secure cache designs (PLcache and RPcache) for security against software data-cache-based side channel attacks. We show that those secure cache designs, although effective against two particular attacks with small hardware cost and performance overhead, fail to address the root causes of software data-cache-based side channel attacks. Therefore they are still vulnerable to advanced cache attacks. We
identify that cache misses of critical data, whose addresses are dependent on secret keys, are the source of information leakage. We then propose three hardware-software integrated approaches: preloading, informing loads and informing loads with software randomization to protect PLcache, RPcache and regular caches respectively from data cache attacks. They present different tradeoffs between hardware complexity and performance overhead. Overall, compared to software approaches, our hardware-based schemes not only provide stronger security protection but also incur relatively small performance overhead.

Software cache-based side channel attacks not only exploit data caches, but also other hardware caches such as branch target buffers (BTB attacks) and instruction caches (I-cache attacks). We identify that in BTB attacks the vulnerability lies on the correlation between the outcome of key-dependent conditional branches and BTB update, i.e. the BTB is only updated when the key-dependent conditional branches is taken. We propose an always BTB update policy to defend against such attacks. Our experimental results show that the new BTB update policy incurs very small performance overhead. For I-cache attacks, we identify that cache misses of instructions, whose execution depends on secret keys, are the source of information leakage. As I-cache attacks work in a similar way as data cache attacks, we propose to apply similar defense schemes for data caches to protect instruction caches. Our experiments show that the PLcache with preloading and the RPcache work well for protecting instruction caches with very small performance overhead.