Precision Timed Machines

by

Isaac Liu

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor John Wawrzynek
Professor Alice Agogino

Spring 2012

Precision Timed Machines

Abstract

Precision Timed Machines

by

Isaac Liu
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

This is my abstract

To my wife Emily Cheung, my parents Char-Shine Liu and Shu-Jen Liu, and everyone else whom I've had the pleasure of running into for the first twenty-seven years of my life.

Contents

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**
1.1 Background . . . 1
1.2 Intro Section Header 2 . . . 1

2 Related Work **2**
2.1 Architectural Modifications . . . 2
2.2 Related Section Header 2 . . . 2

3 Hardware Architecture **3**
3.1 Architecture Design . . . 3
3.1.1 Predictable and Composable Timing . . . 3
3.1.2 Thread-Interleaved Pipeline . . . 3
3.1.3 Memory System . . . 7
3.2 Implementation . . . 7
3.2.1 PTARM Simulator . . . 7
3.2.2 PTARM VHDL Softcore . . . 7
3.2.3 Worst Case Execution Time Analysis . . . 7

4 Programming Models **8**
4.1 PRET Programming model Section Header . . . 8
4.2 Pret Programming model Section Header 2 . . . 8

5 Applications **9**
5.1 Eliminating Side-Channel-Attacks . . . 9
5.2 Real Time 1D Computational Fluid Dynamics Simulator . . . 10

6 Conclusion and Future work **12**
6.1 Summary of Results . . . 12
6.2 Future Work . . . 12

Bibliography **13**

List of Figures

1.1 Image Placeholder 1

2.1 Image Placeholder 2

3.1 Common Multi-Threaded Pipeline Architecture 4
3.2 Example Execution of a 5 Thread 5 Stage Thread-Interleaved Pipeline 4
3.3 Comparison of how data dependencies are handled between different pipelines . . 5
3.4 Illustrative example showing the replay mechanism 5
3.5 None Pipelined Floating Unit 6

4.1 Image Placeholder 8

5.1 Image Placeholder 11

List of Tables

Acknowledgments

I want to thank my wife

I want to thank my parents

I want to thank my advisor, Edward A. Lee

I want to thank the committee members

I want to thank Hiren Patel

I want to thank Jan Rieneke

I would also like to thank the ptolemy group especially Christopher and Mary, Jia for providing me the template

I would like to thank everyone else that made this possible

Chapter 1

Introduction

Outline

1.1 Background

- Discuss the problem
- show the difficulty in execution time analysis as background

Fig. 1.1 shows an image

1.2 Intro Section Header 2

Here is another header

The remaining chapters are organized as follows. Chapter 2 surveys the related research that has been done on architectures to make them more analyzable. Chapter 3 explains the architecture of PRET including the thread-interleavedpipeline and memory hierarchy, Chapter **??**, Chapter 5, Chapter 6,

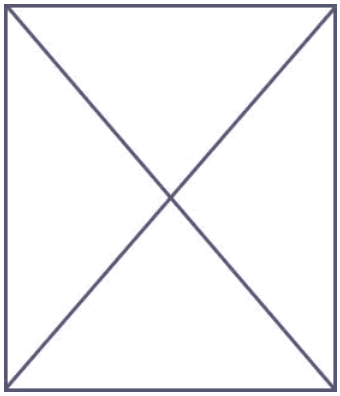

Figure 1.1: Image Placeholder

Chapter 2

Related Work

2.1 Architectural Modifications

Fig. 2.1 shows an image

2.2 Related Section Header 2

Here is another header

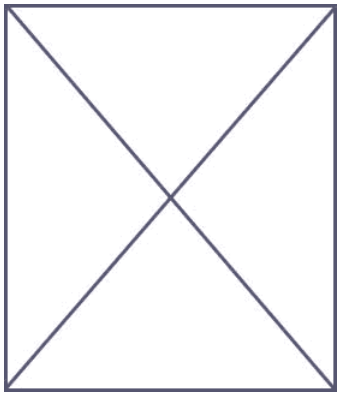

Figure 2.1: Image Placeholder

Chapter 3

Hardware Architecture

3.1 Architecture Design

(Todo: Intro and transition to this chapter)

3.1.1 Predictable and Composable Timing

The thread interleaving pipeline design allows for predictable timing analysis for all threads within the pipeline.

3.1.2 Thread-Interleaved Pipeline

The thread-interleaved pipeline was introduced to improve the response time of handling multiple I/O devices (Todo: citation). I/O operations often stall from the communication with the I/O devices. Thus, interacting with multiple I/O devices leads to wasted processor cycles that are idle waiting for the I/O device to respond. With hardware multi-threading, the pipeline throughput is improved by introducing more hardware thread contexts for the pipeline to execute. When one hardware thread is stalled, instructions from another hardware thread can be fetched into the pipeline to avoid wasting the pipeline cycles. To reduce the overhead of switching execution context, the pipeline keeps physically separate hardware thread state for each hardware thread. These hardware states include register files, program counters, specially processor registers etc. Muxes are added to select the correct hardware states, so context switches can be done by simply changing the selection bits. Figure 3.1 shows a simplified diagram of a typical multi-threaded architecture. Throughout this chapter, we will use the term "thread" to refer to hardware threads that have physically separate register files, program counters, and other thread states. This is not to be confused the typical notion of "threads", which denotes software threads that is managed by operating systems with thread states stored in memory.

Thread-interleaved pipelines uses a deterministic thread switching policy that cycles the threads through the pipeline in a round robin fashion. Every cycle a context switch occurs and a different thread is fetched into the pipeline for execution. Figure 3.2 shows an example execution sequence from a 5 stage thread-interleaved pipeline with 5 threads. The thread-interleaved pipeline shown is a single-issue, single-way architecture. Single-issue means that each cycle only one instruction is fetched into the pipeline. Single-way means that there is only one execution datapath,

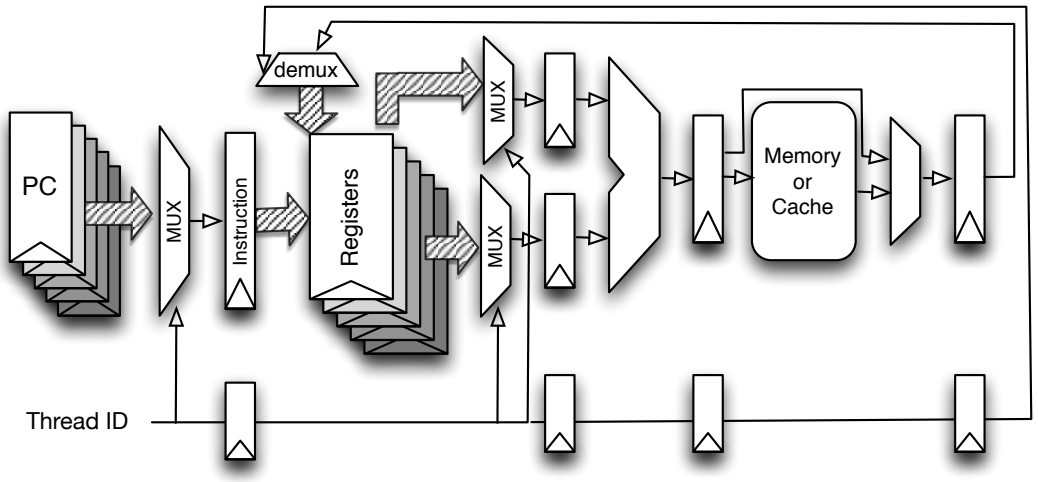

Figure 3.1: Common Multi-Threaded Pipeline Architecture

so each instruction can only hold one operation. We see that with 5 threads on a 5 stage pipeline, the pipeline stages are each executing on a different thread during the same cycle. The properties of a thread-interleaved pipeline provides the basis for our timing predictable architecture design.

With n threads occupying a n stage pipeline, we remove the data-dependencies caused by the pipeline. Data-dependencies arise when an instruction needs data from another instruction that is currently in flight in the pipeline and not committed. These include data operations that operate on registers which are written to in previous instructions. Branch instructions that conditionally branch based upon results from the previous instruction also cause data-dependencies. Pipelines often include forwarding logic and branch predictors to handle these cases so pipeline stalls can be avoided. But thread-interleaved pipelines switch thread contexts every cycle, and threads are fetched into the pipeline a predictable round robin fashion. So from each thread only one instruction is in flight at one time. As a result, data hazards caused by data dependencies from read after write instruction sequences do not occur in thread-interleaved pipelines because instructions from the same context are committed before the next instruction is dispatched. The same holds true for conditional branches, as the condition is resolved before the next instruction is fetched, thus no prediction or stall is needed. Figure 3.3 shows how data-dependencies and conditional branches are handled for both single-threaded and thread-interleaved pipelines.

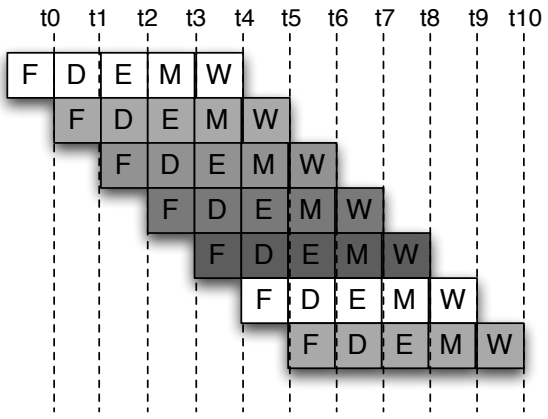

Figure 3.2: Example Execution of a 5 Thread 5 Stage Thread-Interleaved Pipeline

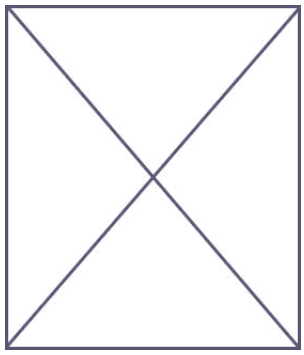

Figure 3.3: Comparison of how data dependencies are handled between different pipelines

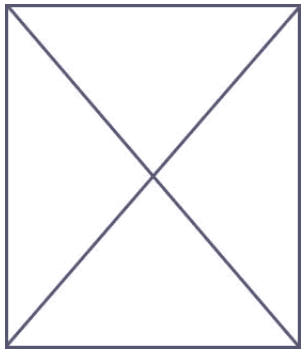

Figure 3.4: Illustrative example showing the replay mechanism

For instructions that take multiple cycles, a replay mechanism is used so the round robin thread scheduling is preserved and no interference is introduced between the threads. When a multi-cycle instruction is fetched into the pipeline from a thread, it executes as any instruction. As the instruction goes through the pipeline, no results are committed, but instead its state is saved in a hardware thread control block and does not increment the program counter for this thread. When an instruction fetch from this thread occurs, the same instruction is dispatched into the pipeline to continue its execution. If it still has not completed its execution, then the program counter for this thread is again not incremented and the same instruction is dispatched, until it is completed. Figure 3.4 illustrates this mechanism.

For instructions that take multiple cycles due to limitations of the pipeline design, this mechanism makes sense. Instructions that do 64-bit operations on a 32-bit pipeline datapath for example falls into this category. In order to abide to the round-robin thread scheduling, the thread simply saves the instruction state and continues execution when it gains access to the pipeline. However, for other multi-cycle instructions, such as memory operations, this mechanism might seem counter intuitive. These instructions require multiple cycles because data is required from other hardware components that have longer access latencies. Memory operations or floating point operations are categorized into such instructions because they are waiting for data from main memory access or

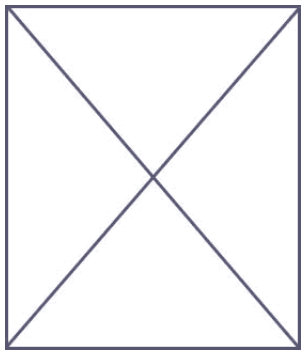

Figure 3.5: None Pipelined Floating Unit

computation results from floating point units. Often times multi-threaded architectures mark these threads inactive and the thread is not rescheduled until the data is ready. This is done to maximize the throughput of the pipeline, since threads waiting for data from other hardware components cannot make any progress until the data is returned. However, this leads to unpredictable timing behaviors in the threads. When threads are scheduled and unscheduled dynamically, the other threads in the pipeline would dynamically execute more or less frequently depending on which threads are active and inactive. This greatly complicates any timing analysis on the software running on each thread as the execution frequency of the threads would depend on the execution of other threads. Thus, our thread-interleaved pipeline does not mark threads inactive, but simply replays the instruction from the thread. The effects of latency hiding is still present, as other threads continue to progress while one thread is replaying its multi-cycle instruction.

Care must also be taken when adding datapaths that take multiple cycles, or else the interference introduced could easily disrupt the timing analysis of threads. If the added datapath isn't able to support pipelined or simultaneous operations, then it will introduce contention amongst the threads. For example, in figure 3.5 we show the effects of adding a non-pipelined floating point divider that takes 20 cycles to execute. As one thread executes a floating-point division instruction, any other thread that also executes a floating-point division must now wait until the first instruction finishes. If other threads also executes the same instruction, then queuing mechanisms must be introduced, for threads that are contending for the floating-point divider. This would greatly complicate the timing analysis, as the execution time of floating-point division instructions now depend on the execution context of other threads. Pipelining the floating-point divider would increase the throughput at the cost of area and latency. However, by pipeling the floating-point divider unit, each thread that executes a floating-point division can now access it without contention. The replay mechanism also hides the long latency of the instruction, and benefits from the improved latency. Because there is no contention, the timing analysis of floating-point operations are now trivial and predictable.

3.1.3 Memory System

3.2 Implementation

3.2.1 PTARM Simulator

3.2.2 PTARM VHDL Softcore

3.2.3 Worst Case Execution Time Analysis

Chapter 4

Programming Models

Intro text here

4.1 PRET Programming model Section Header

Fig. 4.1 shows an image

4.2 Pret Programming model Section Header 2

Here is another header

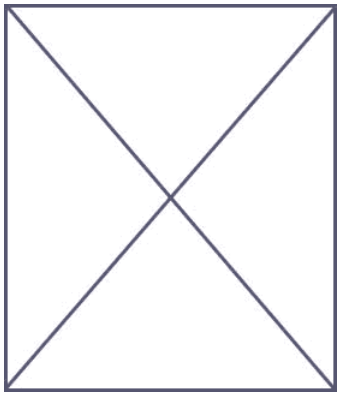

Figure 4.1: Image Placeholder

Chapter 5

Applications

5.1 Eliminating Side-Channel-Attacks

Encryption algorithms are based on strong mathematical properties to prevent attackers from deciphering the encrypted content. However, their implementations in software naturally introduce varying run times because of data-dependent control flow paths. Timing attacks [14] exploit this variability in cryptosystems and extract additional information from executions of the cipher. These can lead to deciphering the secret key. Kocher describes a timing attack as a basic signal detection problem [14]. The "signal" is the timing variation caused by the key's bits when running the cipher, while "noise" is the measurement inaccuracy and timing variations from other factors such as architecture unpredictability and multitasking. This signal to noise ratio determines the number of samples required for the attack – the greater the "noise," the more difficult the attack. It was generally conceived that this "noise" effectively masked the "signal," thereby shielding encryption systems from timing attacks. However, practical implementations of the attack have since been presented [6, 9, 24] that clearly indicate the "noise" by itself is insufficient protection. In fact, the architectural unpredictability that was initially believed to prevent timing attacks was discovered to enable even more attacks. For example, computer architects use caches, branch predictors and complex pipelines to improve the average-case performance while keeping these optimizations invisible to the programmer. These enhancements, however, result in unpredictable and uncontrollable timing behaviors, which were all shown to be vulnerabilities that led to side-channel attacks [3, 20, 2, 8].

In order to not be confused with Kocher's [14] terminology of *timing attacks* on algorithmic timing differences, we classify all above attacks that exploit the timing variability of software implementation *or* hardware architectures as *time-exploiting attacks*. In our case, a *timing attack* is only one possible *time-exploiting attack*. Other time-exploiting attacks include branch predictor, and cache attacks. Examples of other side-channel attacks are power attacks [17, 13], fault injection attacks [4, 10], and many others [24].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [4, 8, 15, 11, 1, 12, 7, 23, 22]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [8, 4, 15, 11, 1, 12, 7, 23, 22] indicates that we do not have the complete set of solutions as more and more vulnerabilities are still being discovered and exploited. Just recently, Coppens et al. [8] discovered two previously unknown time-exploiting attacks on modern x86 pro-

cessors caused by the out-of-order execution and the variable latency instructions. This suggests that while current prevention methods are effective at *defending* against their particular attacks, they do not *prevent* other attacks from occurring. This, we believe, is because they do not address the root cause of time-exploiting attacks, which is that run time variability *cannot be controlled* by the programmer.

It is important to understand that the main reason for time-exploiting attacks is *not* that the program runs in a varying amount of time, but that this variability *cannot be controlled* by the programmer. The subtle difference is that if timing variability is introduced in a controlled manner, then it is still possible to control the timing information that is leaked during execution, which can be effective against time-exploiting attacks. However, because of the programmer's *lack of control* over these timing information leaks in modern architectures, noise injection techniques are widely adopted in attempt to make the attack infeasible. These include adding random delays [14] or blinding signatures [14, 7]. Other techniques such as branch equalization [18, 24] use software techniques to rewrite algorithms such that they take equal time to execute during each conditional branch. We take a different approach, and directly address the crux of the problem, which is the *lack of control* over timing behaviors in software. We propose the use of an embedded computer architecture that is designed to allow predictable and controllable timing behaviors.

At first it may seem that a predictable architecture makes the attacker's task simpler, because it reduces the amount of "noise" emitted from the underlying architecture. However, we contend that in order for timing behaviors to be controllable, the underlying architecture *must* be predictable. This is because it is meaningless to specify any timing semantics in software if the underlying architecture is unable to honor them. And in order to guarantee the execution of the timing specifications, the architecture must be predictable. Our approach does not attempt to increase the difficulty in performing time-exploiting attacks, but to eliminate them completely.

In this paper, we present the PREcision Timed (PRET) architecture [16] in the context of embedded cryptosystems, and show that an architecture designed for predictability and controllability effectively eliminates all time-exploiting attacks. Originally proposed by Lickly et al [16], PRET provides instruction-set architecture (ISA) extensions that allow programmers to control an algorithm's temporal properties at the software level. To guarantee that the timing specifications are honored, PRET provides a predictable architecture that replaces complex pipelines and speculation units with multithread-interleaved pipelines, and replaces caches with software-managed fast access memories. This allows PRET to maintain predictability without sacrificing performance. We target embedded applications such as smartcard readers [15], key-card gates [5], set-top boxes [15], and thumbpods [21], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [19] encryption algorithms on PRET, and show its immunity against time-exploiting attacks. This work shows that a disciplined defense against time-exploiting attacks requires a combination of software and hardware techniques that ensure controllability and predictability.

5.2 Real Time 1D Computational Fluid Dynamics Simulator

Here is another header

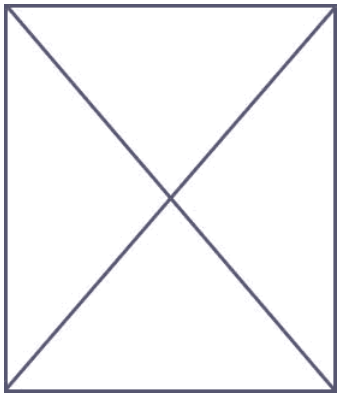

Figure 5.1: Image Placeholder

Chapter 6

Conclusion and Future work

6.1 Summary of Results

This is my summary

6.2 Future Work

Here is what you can keep doing

Bibliography

[1] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.

[2] O. Aclicmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pages 225–242. Springer-Verlag, 2007.

[3] D. J. Bernstein. Cache-timing Attacks on AES, 2004.

[4] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. *Lecture Notes in Computer Science*, 1294:513–525, 1997.

[5] S. C. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled rfid device. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[6] D. Brumley and D. Boneh. Remote timing attacks are practical. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[7] D. Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology: Proceedings of Crypto 82*, pages 199–203. Plenu Press, 1983.

[8] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors, 2009.

[9] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998)*. Springer-Verlag, 1998.

[10] M. Feng, B. B. Zhu, M. Xu, S. Li, B. B. Zhu, M. Feng, B. B. Zhu, M. Xu, and S. Li. Efficient Comb Elliptic Curve Multiplication Methods Resistant to Power Analysis, 2005.

[11] R. Karri, K. Wu, P. Mishra, and Y. Kim. Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture. In *DFT '01: Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, page 427, Washington, DC, USA, 2001. IEEE Computer Society.

[12] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In *Journal of Computer Security*, pages 97–110. Springer-Verlag, 1998.

[13] P. Kocher, J. J. E, and B. Jun. Differential Power Analysis. In *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[14] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[15] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology proceedings*, pages 9–20, 1999.

[16] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–146, New York, NY, USA, 2008. ACM.

[17] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *In USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.

[18] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[19] National Institute of Standards and Technology. "Digital Signature Standard". Federal Information Processing Standards Publication 186, 1994.

[20] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, page 05, 2005.

[21] P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, S. Yang, A. Hodjat, B. Lai, and I. Verbauwhede. Testing ThumbPod: Softcore bugs are hard to find. In *Eighth IEEE International High-Level Design Validation and Test Workshop, 2003*, pages 77–82, 2003.

[22] Z. Wang and R. B. Lee. Covert and Side Channels Due to Processor Architecture. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

[23] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494 – 505, San Diego, CA, June 2007 2007.

[24] Yongbin. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.