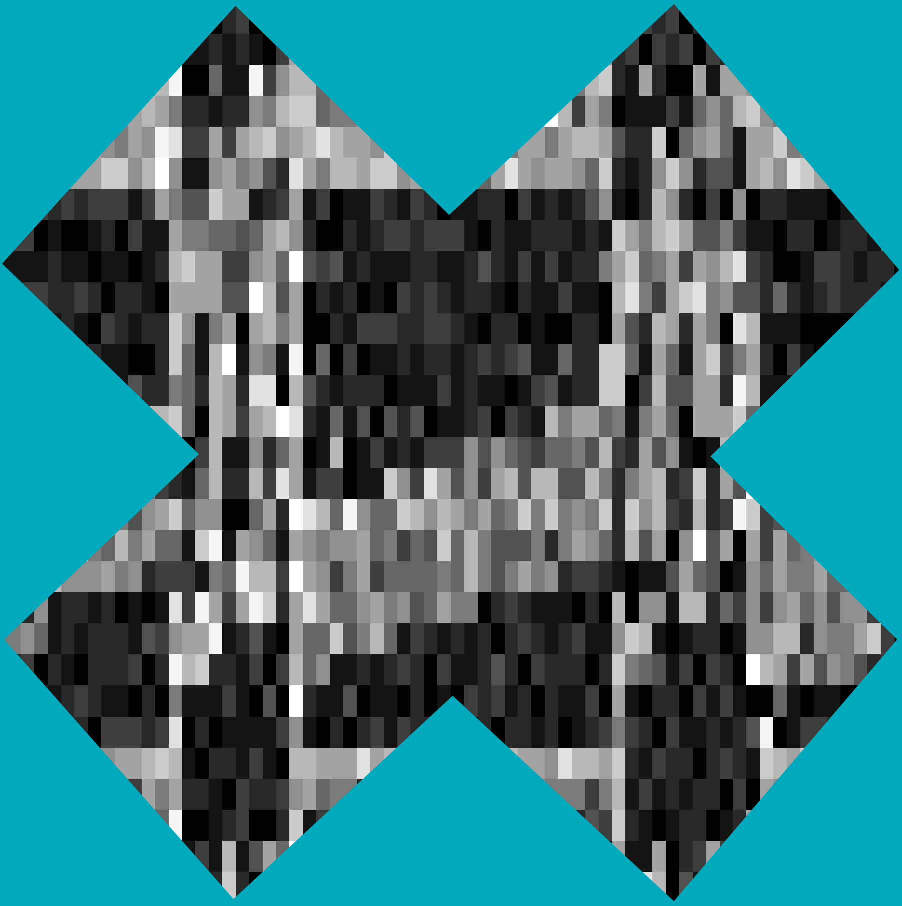
Department of Information and Computer Science

Covert Timing Channels, Caching, and Cryptography

Billy Bob Brumley



Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 136/2011

Covert Timing Channels, Caching, and Cryptography

Billy Bob Brumley

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium AS1 at the Aalto University School of Science (Espoo, Finland) on the 16th of December 2011 at 12 noon.

Aalto University
School of Science
Department of Information and Computer Science

Aalto University publication series
DOCTORAL DISSERTATIONS 136/2011

© Billy Bob Brumley

ISBN 978-952-60-4416-3 (printed)
ISBN 978-952-60-4417-0 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Billy Bob Brumley

Name of the doctoral dissertation
Covert Timing Channels, Caching, and Cryptography

Publisher School of Science

Unit Department of Information and Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 136/2011

Field of research Theoretical Computer Science

Manuscript submitted 29 July 2011 **Manuscript revised** 14 November 2011

Date of the defence 16 December 2011 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

Side-channel analysis is a cryptanalytic technique that targets not the formal description of a cryptographic primitive but the implementation of it. Examples of side-channels include power consumption or timing measurements. This is a young but very active field within applied cryptography. Modern processors are equipped with numerous mechanisms to improve the average performance of a program, including but not limited to caches. These mechanisms can often be used as side-channels to attack software implementations of cryptosystems. This area within side-channel analysis is called microarchitecture attacks, and those dealing with caching mechanisms cache-timing attacks. This dissertation presents a number of contributions to the field of side-channel analysis. The introductory portion consists of a review of common cache architectures, a literature survey of covert channels focusing mostly on covert timing channels, and a literature survey of cache-timing attacks, including selective related results that are more generally categorized as side-channel attacks such as traditional timing attacks. This dissertation includes eight publications relating to this field. They contain contributions in areas such as side-channel analysis, data cache-timing attacks, instruction cache-timing attacks, traditional timing attacks, and fault attacks. Fundamental themes also include attack mitigations and efficient yet secure software implementation of cryptosystems. Concrete results include, but are not limited to, four practical side-channel attacks against OpenSSL, each implemented and leading to full key recovery.

Keywords cryptography, covert channels, side-channel analysis, timing attacks, cache-timing attacks

ISBN (printed) 978-952-60-4416-3 **ISBN (pdf)** 978-952-60-4417-0

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 238 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Preface

The following entities generously support the work in this dissertation.

- Helsinki Doctoral Programme in Computer Science - Advanced Computing and Intelligent Systems (Hecse).
- Academy of Finland (project #122736).
- The European Commission through the ICT program under contracts ICT-2007-216499 CACE and ICT-2007-216676 ECRYPT II.
- The Nokia Foundation.

I am indebted to the following individuals for their gracious support. Please accept my sincerest thanks.

- Prof. Kaisa Nyberg, my supervisor and mentor.
- My fellow colleagues in the cryptography group at Aalto University School of Science.
- My co-authors with whom I've had the pleasure to collaborate.

Espoo, November 14, 2011,

Billy Bob Brumley

Contents

List of Publications

This dissertation consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security*, Tokyo, Japan, December 6-10, 2009, LNCS vol. 5912, pages 667-684, Springer, 2009.

II Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop*, Santa Barbara, CA, USA, August 17-20, 2010, LNCS vol. 6225, pages 110-124, Springer, 2010.

III Billy Bob Brumley, Risto M. Hakala, Kaisa Nyberg, and Sampo Sovio. Consecutive s-box lookups: a timing attack on SNOW 3G. In *Information and Communications Security - 12th International Conference, ICICS 2010*, Barcelona, Spain, December 15-17, 2010, LNCS vol. 6476, pages 171-185, Springer, 2010.

IV Billy Bob Brumley and Nicola Tuveri. Cache-timing attacks and shared contexts. In *2nd International Workshop on Constructive Side-Channel Analysis and Secure Design, COSADE 2011*, Darmstadt, Germany, 24-25 February 2011, pages 233-242, Technische Universität Darmstadt / CASED, 2011.

V Billy Bob Brumley and Dan Page. Bit-sliced binary normal basis multiplication. In *20th IEEE Symposium on Computer Arithmetic, ARITH 2011*, Tübingen, Germany, 25-27 July 2011, pages 205-212, IEEE Computer Society, 2011.

VI Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security*, Leuven, Belgium, September 12-14, 2011, LNCS vol. 6879, pages 355-371, Springer, 2011.

VII Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. Accepted for publication in *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012*, San Francisco, CA, USA, February 27-March 2, 2012, LNCS, 18 pages, Springer, 2012.

VIII Billy Bob Brumley. Secure and fast implementations of two involution ciphers. Accepted for publication in *15th Nordic Conference on Secure IT Systems, NordSec 2010*, Helsinki, Finland, 27-30 October 2010, LNCS vol. 7127, 14 pages, Springer, 2011.

Author's Contribution

Publication I: "Cache-timing template attacks"

The current author is responsible for proposing this research topic, implementing the side-channel and lattice portions of the attack, and the related writing.

Publication II: "New results on instruction cache attacks"

The current author is responsible for the attack portion of the work as well as the related writing.

Publication III: "Consecutive s-box lookups: a timing attack on SNOW 3G"

The current author is responsible for formalizing and implementing the side-channel, proposing improvements to the state recovery algorithm, implementing the bit-sliced version of the cipher, and the related writing.

Publication IV: "Cache-timing attacks and shared contexts"

The current author is responsible for proposing this research topic, interpreting experiment results, and the related writing.

Publication V: "Bit-sliced binary normal basis multiplication"

The current author is responsible for proposing this research topic, implementing the multiplication circuits, and the related writing.

Publication VI: "Remote timing attacks are still practical"

The current author is responsible for proposing this research topic, devising the attack, implementing the lattice portion of the attack, and the related writing.

Publication VII: "Practical realisation and elimination of an ECC-related software bug attack"

The current author is responsible for proposing this research topic, implementing the attack, and the related writing.

Publication VIII: "Secure and fast implementations of two involution ciphers"

The current author is solely responsible for this work.

List of Acronyms

3GPP	3rd Generation Partnership Project
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard Instruction Set
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
ARM	Advanced RISC Machines
BPA	Branch Prediction Analysis
CBC	Cipher Block Chaining mode
CERT	Computer Emergency Response Team
CFS	Completely Fair Scheduler
CLMUL	Carryless Multiplication
CPU	Central Processing Unit
CTR	Counter mode
CVE	Common Vulnerabilities and Exposures
dcache	data cache
DEC	Digital Equipment Corporation
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman key exchange
ECDHE	Elliptic Curve Diffie-Hellman key exchange, Ephemeral type
ECDSA	Elliptic Curve Digital Signature Algorithm
FSM	Finite State Machine
GCM	Galois Counter Mode
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HTT	Hyper-Threading Technology

IBM	International Business Machines
icache	instruction cache
ISO	International Organization for Standardization
KB	Kilobyte, 1024 bytes
KVM	Kernelized Virtual Machine System 370
LFSR	Linear Feedback Shift Register
LFU	Least Frequently Used
LNCS	Lecture Notes in Computer Science
LRU	Least Recently Used
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Massachusetts Institute of Technology
MRU	Most Recently Used
NAF	Non-Adjacent Form
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OR	bitwise OR, logical disjunction
OS	Operating System
PC	Personal Computer
PGP	Pretty Good Privacy
RAM	Random-Access Memory
RSA	Rivest, Shamir and Adleman public key cryptosystem
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SPN	Substitution Permutation Network
SSE	Streaming SIMD Extensions
SSE2	Streaming SIMD Extensions 2
SSH	Secure Shell
SSL	Secure Sockets Layer
SSSE3	Supplemental Streaming SIMD Extensions 3
TCP	Transmission Control Protocol
TCSEC	Trusted Computer System Evaluation Criteria
TLS	Transport Layer Security
VAX	Virtual Address extension
VM	Virtual Machine
VMM	Virtual Machine Monitor
VQ	Vector Quantization
XOR	bitwise exclusive OR, exclusive disjunction

1. Introduction

Much like signal processing, cryptology is a peculiar science due to its symbiotic relationship with application. Theory alone without an express use case can be difficult to justify in this field. One good example of this gap between theoretical and applied cryptography is the problem of secure pseudorandom number generation, perhaps for use in a stream cipher. On one hand, the venerable Blum-Blum-Shub construction has an extremely strong security proof and theorists might assert the problem solved. On the other hand, it is rarely used in practice due to its implementation aspects: it is not particularly fast and requires large area. The stream ciphers used in practice are instead built using easily implementable components such as feedback shift registers and/or native microprocessor instructions such as integer additions, but in contrast little can usually be proved about the security of said constructions.

Proposals for cryptographic primitives such as a block cipher are expected to be accompanied by extensive security proofs demonstrating resistance to known cryptanalytic techniques such as differential and linear cryptanalysis. Such proofs make certain assumptions about the abilities of the attacker and are defined by the security model. If a primitive is at all useful, at some point it must make its way from the formal written specification to a concrete implementation, perhaps in software. This task is inevitably the burden of an engineer.

When this leap from paper to practice occurs, a natural concern is how well the physical implementation of the primitive preserves the assumptions of the security model. Or perhaps, from another perspective, how well the theoretical security model reflects the realities imposed by practical applied cryptography.

Over the past two decades, the irrefutable stance of both academia and industry is clear: implementation aspects can easily invalidate the assumptions of the security model and lead to serious vulnerabilities. This breed of cryptanalytic attack is known as side-channel analysis and additionally makes use of a platform, architecture, and/or implementation dependent signal procured during the execution of a cryptographic primitive. Examples of side-channels include power consumption measurements, electromagnetic radiation measurements, acoustic emanations, computation faults, and various timing measurements. These channels are not theoretical and are born out of practice.

This dissertation exclusively concerns side-channel attacks, and within this area focuses on timing attacks. One specific type of timing attack exploits the caching mechanism commonly featured in modern microprocessors. These cache-timing attacks exploit the fact that the latency of fetching data from main memory is essentially governed by the availability of said data in the cache and, by definition and design, is not a constant duration. Cache-timing attacks are the main topic of this dissertation.

The bulk of this dissertation consists of eight publications that represent novel contributions and advance this field of study. A summary of said publications follows.

Publication I. One of the major challenges when implementing a cache-timing attack is how to accurately and efficiently process the signal, i.e., the side-channel made up of timing data. When attacking a block cipher that uses table lookups the strategy is to directly interpret the timing data to infer part of an index used into a lookup table, depending on how this lookup table maps into the cache. In contrast, the implementation of a public key cryptosystem often uses dynamic memory where a different strategy is needed: these types of cache-timing attacks attacks are essentially more about memory access patterns. With that in mind, Publication I presents a framework for efficiently processing large quantities of this type of cache-timing data. The framework uses Vector Quantization (VQ) and Hidden Markov Models (HMMs) to accomplish this. Roughly speaking, VQ reduces the dimension of the data and the HMM accounts for the control flow of the algorithm that produced the signal. This framework is used as part of a cache-timing attack on OpenSSL's implementation of ECDSA. After processing the cache-timing data with the framework, the derived key material is used to mount a lattice attack to recover the private key. The attack succeeds with only a few thousand traces and less than one hour of offline computation on a single desktop machine. The current author is responsible for proposing this research topic, implementing the side-channel and lattice portions of the attack, and the related writing.

Publication II. In contrast to data cache-timing attacks that seek to exploit key-dependent memory references, instruction cache attacks exploit the variable execution of code segments that might be caused, for example, by a logic branch. Building on previous work on instruction cache-timing attacks, Publication II presents improved analysis techniques for these attacks and also considers mitigation strategies. The signal of an instruction cache side-channel is often quite similar to that of a data cache. The work applies the framework in Publication I to timing data from the instruction cache. This is used to attack OpenSSL's implementation of DSA. Similar to the data cache attack in Publication I, the first stage uses the framework to process the signal and last stage using lattice methods to recover the private key. The work furthermore proposes, implements, and evaluates a number of countermeasures to these attacks at both software and hardware levels. The current author is responsible for the attack portion of the work as well as the related writing.

Publication III. Based on the stream cipher SNOW 2.0, SNOW 3G is a software-oriented stream cipher used in 3GPP mobile networks. Roughly speaking, the cipher consists of some linear state implemented as a word-based LFSR and some nonlinear state that emulates a block cipher round function: the linear process masks the output of the nonlinear process to produce keystream words. Both processes are often implemented using table lookups that pose a cache-timing attack threat. Building on previous work that attacks the linear process of SNOW 2.0, Publication III presents a cache-timing attack on SNOW 3G. The attack intuition is that the lookups within the nonlinear process in fact leak much more information than that of the linear process which can be leveraged to dramatically trim the search space of a state recovery algorithm. At a high level, the attack is an exhaustive search of the state space using a backtracking algorithm where the side-channel data constrains the search space. Finally, the work proposes and implements an efficient bit-slicing countermeasure that applies to batch keystream generation. The current author is responsible for formalizing and implementing the side-channel, proposing improvements to the state recovery algorithm, implementing the bit-sliced version of the cipher, and the related writing.

Publication IV. With respect to performance-critical software, dynamic allocation of memory is an expensive operation. A logical strategy is to allocate the data only once and save it for reuse later. A shared context is a software mechanism that implements said strategy. OpenSSL uses a shared context to initialize all of its element representations, such as integers or finite field elements. The mechanism works roughly like a stack, keeping a double linked list of elements and dynamically allocating memory for new elements as required, appending to the list. As suggested in Publication I, this behavior can lead to a cache-timing attack vulnerability since temporary variables get reused in a deterministic way. One countermeasure proposed, but not implemented, therein suggests that the context should randomize its allocation. To this end, Publication IV explores the ability of a shared context to mitigate cache-timing attacks. The work performs a detailed analysis of OpenSSL's shared context implementation and, based on said analysis, implements a simple memory alignment countermeasure. Surprisingly, the results suggest that this is ineffective and that the allocation policy of a shared context has little to no influence on the resulting signal. The work raises some interesting questions concerning the true origin of the side-channel. The current author is responsible for proposing this research topic, interpreting experiment results, and the related writing.

Publication V. Most mainstream processors feature at least some form of an integer multiplication instruction that can be used to implement finite fields with large prime characteristic. On the other hand, implementation of (large) binary fields requires a carryless multiplication instruction that only a minority of processors feature. The textbook way to implement said fields involves some online precomputation into a lookup table, then any number of methods resembling schoolbook multiplication but with shifts, XORs, and lookups into said table. This is not comparatively efficient and can also lead to a cache-timing attack vulnerability. Two standard representations used for finite field elements are a polynomial basis and a normal basis. The latter is normally not particularly competitive with respect to performance. Bit-slicing is a software technique where essentially a w-bit processor is used as w 1-bit processors to run w logic computations in parallel: this allows a quasi-hardware design approach to software components. Furthermore, bit-slicing has inherent resistance to cache-timing attacks since state-dependent memory accesses are replaced by their computational equivalent in terms of bit logic. Motivated by cache-timing attacks and surveying existing hardware normal basis multiplication techniques, Publication V applies the bit-slicing approach to realize secure parallel multiplications in software. The results encompass a multitude of field sizes and compare the timings to those of an existing library for efficient polynomial basis multiplication on a number of different platforms. The results suggest that, for batch operations, the performance gap between polynomial and normal basis multiplication is smaller than suggested in the literature. The current author is responsible for proposing this research topic, implementing the multiplication circuits, and the related writing.

Publication VI. In contrast to cache-timing attacks that often require some malicious code executing locally, general timing attacks measure the overall execution time of a high level operation. Hence both are side-channel attacks yet the latter is a much weaker attack. A key component to an elliptic curve cryptography implementation is the scalar multiplication routine that computes multiples of a point on the curve with itself. Montgomery's ladder is one method to perform said routine and has the inherent potential to resist many types of side-channel attacks due to its extremely regular nature: it always performs the same sequence of finite field operations regardless of the certain value of a key bit. Exploiting a timing attack vulnerability in OpenSSL's ladder implementation, Publication VI devises and implements an attack that leads to private key recovery. The attack is able to recover the private key of a TLS server that authenticates using ECDSA signatures by using only the timings of exchanged handshake messages, the messages themselves, and the signatures on the messages. The final stage of the attack uses lattice methods to compute the private key. The attack only requires a few minutes and is demonstrated to succeed in both local and remote scenarios. The current author is responsible for proposing this research topic, devising the attack, implementing the lattice portion of the attack, and the related writing.

Publication VII. Fault attacks can be considered as a type of side-channel attack in which, at some stage during operation, a device makes a computation error. This is traditionally a hardware-related topic. Modeling a known OpenSSL software bug as a fault attack, Publication VII gives a detailed analysis of said bug, discusses implications, and outlines a number of attacks exploiting it. Briefly, the modular reduction routines in OpenSSL for some finite fields associated with standardized elliptic curves can fail in very rare instances. Exploiting this bug, Publication VII devises and implements an attack against various ECDH modes in TLS that recovers the server's static private key by querying the server with cleverly chosen inputs. The work postulates that formal verification techniques, while challenging, could have prevented this bug. The current author is responsible for proposing this research topic, implementing the attack, and the related writing.

Publication VIII. Anubis and Khazad are two block ciphers that resemble AES in many respects. But different from AES, they have an involution property: decryption differs from encryption only by the key schedule, i.e., the code the cipher executes in operation is the same both ways. Motivated by cache-timing attacks and building on two recent results on AES software implementations, Publication VIII gives both serial and bit-sliced implementations of these ciphers. The intuition for the serial Anubis implementation is that the nonlinear layer has a very elegant implementation using a byte shuffler, e.g., that available through Intel's SSSE3 instruction set. This version is competitive with the reference implementation that uses table lookups, yet in contrast is resistant to cache-timing attacks. On the bit-slicing side, interestingly the results show that, in software on the considered platform, the per-round operation of Anubis is slightly faster than that of AES. The current author is solely responsible for this work.

Themes and scope. With the previous description of the publications making up this dissertation, a number of unequivocal themes emerge. A non-exhaustive list of these themes follows, along with a short discussion of each theme, defining the scope of this dissertation.

1. Naturally, the most prevalent theme of this dissertation is that of side-channel analysis. There are many ways to approach this topic, but regarding microarchitecture attacks perhaps the most logical and classical in some respects is to begin by constructing a covert channel from the microarchitecture mechanism and determine how it can be used for information transfer. Then turning said channel into a side-channel by one of the party's inadvertent use of the channel. Publication I and Publication II include cache-timing attacks on asymmetric key cryptosystems, Publication III a cache-timing attack on a symmetric key cryptosystem, Publication VI a timing attack on an asymmetric key cryptosystem, and Publication VII a fault attack on an asymmetric key cryptosystem. While their channels are realized in very different ways, these works all fall under the theme of side-channel analysis.

2. After successfully devising, and even implementing, a side-channel attack, it is tempting to proudly disseminate the results, declare victory and move on to the next conquest. This approach is not constructive. Successfully implementing an attack gives the experimenter a unique perspective on the result, including the intuition for why and how it works and, in particular, the best way to prevent it. Indeed, countermeasures are an integral, even obligatory part of the research process in this field. While all of the publications making up this dissertation treat countermeasures at least as an aside, a number of them deal exclusively with this crucial theme. In particular, Publication IV focuses on a single cache-timing attack countermeasure for asymmetric key cryptosystems, Publication V even more general timing-attack countermeasures for asymmetric key cryptosystems, and Publication VIII cache-timing attack countermeasures for symmetric key cryptosystems.

3. In light of side-channel attacks, a practical concern is whether it is possi-

ble to realize implementations that resist these attacks, yet at the same time are still computationally efficient. Since side-channel attacks themselves are implementation-specific and depend, for example, on the architecture where the compiled code executes, it is natural to consider architecture-specific features that can aid in side-channel mitigation. For example, parallel computation via Single Instruction Multiple Data (SIMD) featured on many commodity microprocessors is often used to improve efficiency but can also, in some instances, be used to thwart timing attacks. This is sometimes even an unintentional consequence of optimization. Efficient yet secure software implementations of cryptosystems is a strong theme of this dissertation. In particular, Publication V concerns asymmetric key cryptosystems and Publication VIII symmetric key cryptosystems. This theme relates to the previous one, but approaches the issue from the opposite direction and with different priorities.

In order to provide a concise yet thorough coverage on the subject matter and avoid lengthy tangents, this dissertation attempts to restrict the scope as reasonably as possible. With that in mind, regretfully there are a number of fascinating topics that, although related to this dissertation in some respects, do not fall under this scope. A short yet inevitably non-exhaustive list of these topics follows.

1. Pioneered by Acıiçmez et al. [AScKK06, AcKKS07], another type of microarchitecture attack is that which exploits the behavior of the branch predictor. To avoid excessively stalling the instruction pipeline, the job of a microprocessor's branch predictor is to intelligently guess the outcome of a logic branch based on previous outcomes, fetch the resulting instructions, and speculatively execute them. Branch predictor attacks execute malicious code that essentially spies on the branch target buffer to determine if the victim code takes a logic branch or not by measuring the execution time of its own branch statements. If the victim executes key dependent logic branches, this yields a side-channel that can be used for cryptanalytic purposes. This dissertation omits this topic.

2. The framework in Publication I leverages two well-established signal processing techniques: VQ and HMMs. VQ maps vectors from a given domain to a finite set of vectors called a codebook. This is usually implemented by mapping said vectors to the closest codebook vector by Euclidean distance. HMMs

are formal models of discrete-time stochastic processes. Given such a process that, with certain probabilities, emits one of many observable events when changing states, one use of an HMM is calculating the most likely sequence of states that explains the observations. This dissertation does not cover the theory behind these techniques. The textbook by Russell and Norvig is a standard reference for HMMs [RN10].

3. Lattices are mathematical objects that have many uses in cryptography from cryptographic primitives to attacking schemes with partially known secret data. They are generally useful for finding small solutions to underdetermined systems of equations. Lattice methods are an effective endgame for many side-channel attacks: combining public information with (private) partial key material derived in the analysis phase, i.e., procured from the signal, to recover the complete private key. The work of Howgrave-Graham and Smart is an excellent example [HGS01]. Although three publications that are part of this dissertation apply lattices accordingly, the scope does not encompass the theory behind lattice methods.

4. This dissertation gives an extensive survey of existing results on cache-timing attacks and also advances the field by presenting some novel contributions to the topic. In the end, caching is the culprit here and one natural response is to consider alternative cache designs and architectures that can provide a higher degree of security. Such designs do indeed exist (e.g., Page's partitioned cache [Pag05]), but are not discussed in this dissertation.

5. Traditional models for provable security of ciphers fail to capture side-channel attacks. In response, some recent models (e.g., leakage-resilient cryptography), attempt to account for side-channels and even parameterize the security as a function of the side-channel capacity. Such models are not a topic of this dissertation. In fact, one can argue that such models are predisposed to failure and can do more harm than good: see the work of Koblitz and Menezes for an insightful discussion [KM11].

Outline. The structure of this dissertation is as follows. Chapter 2 contains brief background on caching mechanisms. Chapter 3 contains background on covert channels, including a selective chronological literature survey. Transitioning from the previous chapter, Chapter 4 contains a selective chronological literature survey of timing attacks that focuses on cache-timing attacks. Said survey is interleaved with discussions summarizing the main contribution of the publications appearing in this dissertation, placing them in context to show how they build and improve upon previous work. Chapter 5 is similar in structure, but is less extensive and concerns efficient and secure software methods for cryptography engineering. Chapter 6 draws conclusions, summarizes the impact of this dissertation, and examines outlook in this field. Much of the subject matter of this dissertation deals with concrete realization of side-channels, and to this end Appendix A contains some helpful tools of the trade. This culminates with the listed publications, in order.

2. Microprocessor Caches

cache: a secure place of storage.
Webster's New Collegiate
Dictionary

There are a number of thorough references for cache design that treat the subject very formally. This dissertation is not one of them. In particular, the textbooks by Hennessy and Patterson [PH90, 8.3], Patterson and Hennesey [PH07, 7.2], and Page [Pag09, 8.3] are excellent references on the topic.

This chapter contains a brief review of practical cache architectures. These concepts are useful in understanding the nature of the cache as a covert channel and side-channel, that being the focus of subsequent chapters.

Microprocessor operation. Figure 2.1 illustrates typical microprocessor operation. The instruction fetcher is responsible for fetching instructions to be executed from main memory. It then passes these to the instruction decoder that interprets these instructions. The result then gets executed, generally by, for example, passing it to the arithmetic logic unit (ALU) that might operate on a number of register values and store the result in a register or main memory, or move a value between a register and main memory.

Cache architectures. The memory interface component depicted in Fig. 2.1 commonly contains at least two critical components: a **data cache** and an **instruction cache**. The discussion focuses on the former. Moving values between registers is an extremely low latency operation, while movement between registers and main memory suffers higher latency. The number of registers is unfortunately limited. To offset the cost of such movement, modern microprocessors employ a data cache; this has a much higher capacity than the working set of registers, higher latency than register access, but much lower latency than memory access. Data from main memory is stored first in the cache; when the CPU needs data from main memory, it looks for it initially in the cache. If it finds it, this is a **cache hit** and it looks no further; otherwise, a **cache miss**

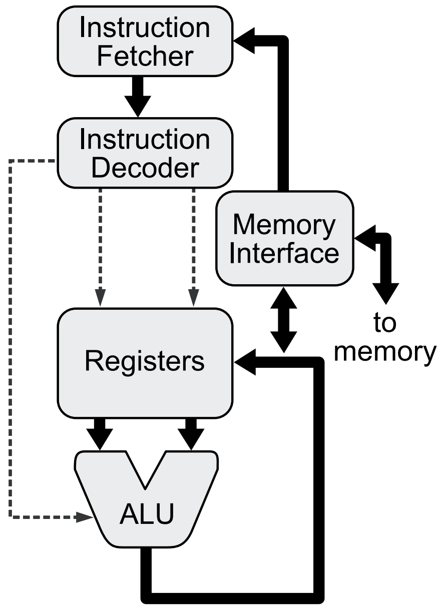


Figure 2.1. Simplified microprocessor operation. Source: `http://commons.wikimedia.org/wiki/File:CPU_block_diagram.svg`

and it reads the data from main memory. The blocks of memory in the cache are called **cache lines**. There are three common ways to implement such a cache.

Direct mapped cache. The simplest form allows a block of memory to be stored in one and only one location in a **direct mapped cache**, illustrated in Fig. 2.2. The bits in a memory address are split into three parts. The **tag** is a unique identifier for a block of memory. The **line** identifies which cache line a block of memory can be stored in. The **offset** identifies the byte offset within the block. Along with the data, the cache lines also have a tag associated with them, representing what block of memory currently resides within a cache line. The cache logic works as follows when processing a request for data at a given memory address. The line portion of the address identifies the specific cache line for the memory block; the cache controller compares the corresponding tag of the cache line to the tag of the memory address. If they match, a cache hit occurs and it uses the offset portion of the address to load the data at an offset within the given cache line. Otherwise, a cache miss occurs and it loads the data from the next level in the memory hierarchy. Advantages of a direct mapped cache include simplicity and low latency. The minimal amount of logic means they are easy to implement compactly. Disadvantages can include poor cache utilization and **cache thrashing**. Consider an extreme example where only two memory blocks are accessed. By chance, these memory blocks have

```
                                         address tag line offset
  +--+----+                              ------- --- ---- ------
0 |  |    |    +-------------------- D831    D8  3    1
  +--+----+    |                         35A6    35  A    6
1 |  |    |    |                         937F    93  7    F
  +--+----+    |                         0384    03  8    4
2 |  |    |    |                         2B93    2B  9    3
  +--+----+    |                         FA63    FA  6    3
3 |MM|YYYY| <--+                         4B91    4B  9    1
  +--+----+       MM == D8 ? H : M       0F9C    0F  9    C
4 |  |    |                              CD44    CD  4    4
  +--+----+                              D49F    D4  9    F
5 |  |    |                              14BA    14  B    A
  +--+----+                              E4F1    E4  F    1
6 |  |    |                              B319    B3  1    9
  +--+----+                              14F3    14  F    3
7 |  |    |                              3E09    3E  0    9
  +--+----+                              13DA    13  D    A
8 |  |    |                              0C6E    0C  6    E
  +--+----+                              6BBC    6B  B    C
9 |  |    |                              44A2    44  A    2
  +--+----+                              26D4    26  D    4
A |  |    |                              04A3    04  A    3
  +--+----+                              5F59    5F  5    9
B |  |    |                              9FB3    9F  B    3
  +--+----+                              65DE    65  D    E
C |  |    |                              D65B    D6  5    B
  +--+----+                              6D6F    6D  6    F
D |  |    |                              8FAC    8F  A    C
  +--+----+                              90DC    90  D    C
E |  |    |                              C8E0    C8  E    0
  +--+----+                              7A4B    7A  4    B
F |  |    |                              76D3    76  D    3
  +--+----+                              05A9    05  A    9
```

Figure 2.2. A 256B direct mapped cache with 16 lines of 16B each. The line portion of a memory address points to a single line in the cache. Only said line can match the tag portion of the memory address.

the same line portion of the address: they both map to the same cache line and compete for the same cache space. The remaining cache lines go unused. When memory accesses occur at these addresses, the cache controller **evicts** the current data: it replaces the data in the cache line with the new data loaded as a result of a cache miss. These evictions as a result of repeatedly accessing different memory locations is called cache thrashing: the data ends up being continually swapped out and the cache becomes a burden instead of an asset.

Fully associative cache. The disadvantages of a direct mapped cache are unfortunately fundamental; each memory block can only reside in one cache line. To remedy this, one might consider an extreme solution and instead allow each memory block to reside in *any* cache line; Fig. 2.3 illustrates a **fully associative cache**. The cache logic works as follows. The cache controller compares the corresponding tag of *each* cache line to the tag of the memory address. If there is a match, a cache hit occurs; otherwise, a cache miss. Advantages of a fully associative cache include better cache utilization and minimal cache thrashing. The main disadvantage is implementation complexity. Instead of

```
                                      address tag offset
  +--+----+                           ------- --- ------
0 |NN|YYYY| <--+--------------------- D831    D83 1
  +--+----+    | NN == D83 ? H :      35A6    35A 6
1 |ZZ| .. | <--+                      937F    937 F
  +--+----+    | ZZ == D83 ? H :      0384    038 4
2 |LL| .. | <--+                      2B93    2B9 3
  +--+----+    | LL == D83 ? H :      FA63    FA6 3
3 |MM| .. | <--+                      4B91    4B9 1
  +--+----+    | MM == D83 ? H :      0F9C    0F9 C
4 |TT| .. | <--+                      CD44    CD4 4
  +--+----+    | TT == D83 ? H :      D49F    D49 F
5 |YY| .. | <--+                      14BA    14B A
  +--+----+    | YY == D83 ? H :      E4F1    E4F 1
6 |HH| .. | <--+                      B319    B31 9
  +--+----+    | HH == D83 ? H :      14F3    14F 3
7 |KK| .. | <--+                      3E09    3E0 9
  +--+----+    | KK == D83 ? H :      13DA    13D A
8 |WW| .. | <--+                      0C6E    0C6 E
  +--+----+    | WW == D83 ? H :      6BBC    6BB C
9 |II| .. | <--+                      44A2    44A 2
  +--+----+    | II == D83 ? H :      26D4    26D 4
A |GG| .. | <--+                      04A3    04A 3
  +--+----+    | GG == D83 ? H :      5F59    5F5 9
B |RR| .. | <--+                      9FB3    9FB 3
  +--+----+    | RR == D83 ? H :      65DE    65D E
C |UU| .. | <--+                      D65B    D65 B
  +--+----+    | UU == D83 ? H :      6D6F    6D6 F
D |SS| .. | <--+                      8FAC    8FA C
  +--+----+    | SS == D83 ? H :      90DC    90D C
E |XX| .. | <--+                      C8E0    C8E 0
  +--+----+    | XX == D83 ? H :      7A4B    7A4 B
F |JJ| .. | <--+                      76D3    76D 3
  +--+----+      JJ == D83 ? H : M    05A9    05A 9
```

Figure 2.3. A 256B fully associative cache with 16 lines of 16B each. Any single line in the cache can match the tag portion of the memory address.

needing to check only one cache line, the cache controller must check all cache lines; this fundamental difference is noticeable in the input to the compare logic in Fig. 2.3. Furthermore, the cache controller must also implement some kind of intelligent policy for evicting data from the cache. Such policies are covered later.

Set associative cache. The logical compromise between the previous two approaches is to allow each memory block to reside in one of *many* cache lines; Fig. 2.3 illustrates a **set associative cache**. The cache logic works as follows. The **set** portion of the address identifies the subset of cache lines in which the memory block can reside; the cache controller compares the corresponding tag of each cache line in the subset to the tag of the memory address. If they match, a cache hit occurs; otherwise, a cache miss. A set associative cache enjoys the benefits of both previous types. It is a trade-off between performance and complexity. Compared to a fully associative cache, the cache controller can more easily identify if a memory location resides in the cache, and the needed policy is easier to implement. This is by far the most widely implemented cache architecture for modern microprocessors.

```
                                      address tag set offset
  +--+----+                           ------- --- --- ------
0 |  |    |    +--------------------- D831    D80 3   1
  +--+----+    |                      35A6    358 2   6
1 |  |    |    |                      937F    934 3   F
  +--+----+    |                      0384    038 0   4
2 |  |    |    |                      2B93    2B8 1   3
  +--+----+    |                      FA63    FA4 2   3
3 |MM|YYYY| <--+                      4B91    4B8 1   1
  +--+----+    | MM == D80 ? H :      0F9C    0F8 1   C
4 |  |    |    |                      CD44    CD4 0   4
  +--+----+    |                      D49F    D48 1   F
5 |  |    |    |                      14BA    148 3   A
  +--+----+    |                      E4F1    E4C 3   1
6 |  |    |    |                      B319    B30 1   9
  +--+----+    |                      14F3    14C 3   3
7 |KK| .. | <--+                      3E09    3E0 0   9
  +--+----+    | KK == D80 ? H :      13DA    13C 1   A
8 |  |    |    |                      0C6E    0C4 2   E
  +--+----+    |                      6BBC    6B8 3   C
9 |  |    |    |                      44A2    448 2   2
  +--+----+    |                      26D4    26C 1   4
A |  |    |    |                      04A3    048 2   3
  +--+----+    |                      5F59    5F4 1   9
B |RR| .. | <--+                      9FB3    9F8 3   3
  +--+----+    | RR == D80 ? H :      65DE    65C 1   E
C |  |    |    |                      D65B    D64 1   B
  +--+----+    |                      6D6F    6D4 2   F
D |  |    |    |                      8FAC    8F8 2   C
  +--+----+    |                      90DC    90C 1   C
E |  |    |    |                      C8E0    C8C 2   0
  +--+----+    |                      7A4B    7A4 0   B
F |JJ| .. | <--+                      76D3    76C 1   3
  +--+----+      JJ == D80 ? H : M    05A9    058 2   9
```

Figure 2.4. A 256B set associative cache with 16 lines of 16B each and 4 ways. The set portion of a memory address points to four distinct lines in the cache. Only one of these four lines can match the tag portion of the memory address.

Cache replacement policies. When data at a memory location can reside in more than one place in the cache, a policy must exist for evicting existing data in the event of a cache miss. What data should be evicted from possible cache locations to make room for the new data? Said logic defines the **cache replacement policy**. The most common cache replacement policy is **Least Recently Used (LRU)**. The cache controller can implement this by maintaining age fields for the cache lines; the cache line with the oldest field, or least recently used, gets evicted. A microcontroller used for a common desktop workstation generally benefits from such a policy. If data is used, it is likely to be used again in the near future. For large data sets or random access patterns, LRU poses a problem; an item from the set is not likely to be needed in the near future once accessed. The exact opposite idea of a **Most Recently Used (MRU)** policy would be more appropriate, where the cache controller instead evicts the line with the youngest field. A microcontroller used for such a special purpose would benefit from this. Instead of an age, a valuable metric might be the number of times a memory block is accessed; a **Least Frequently Used (LFU)** policy discards data that is used least often. Last but not least, a **random** policy evicts a random line from the cache to make room for incoming data. A big advantage of such a policy is ease of implementation: it involves very little logic and does not hinge on any assumptions about the memory access habits of a typical program.

3. Covert Channels

The topic of covert channels usually falls under the umbrella of system security. Academic interest in this specific area dates back to at least the 1970s. Roughly speaking, covert channels provide a method to transmit information in an unconventional way in a system where such communication is not explicitly allowed. These channels can be either **intentional** and used between consenting parties to communicate unregulated by the system, or **unintentional** and used by a malicious party to monitor the activities of a legitimate party. In this context, a party can be a user, program, or process. This chapter contains a selective literature review of results pertaining to covert channels, focusing mostly on covert timing channels, beginning from the initial works and proceeding chronologically. These concepts build the foundation for side-channel attacks (i.e., attacks that exploit unintentional covert channels) and, in particular, cache-timing attacks.

The confinement problem. In his seminal work, Lampson defines the **confinement problem** [Lam73].

> This note explores the problem of confining a program during its execution so that it cannot transmit information to any other program except its caller.

He gives a number of examples of how a program might leak data, among them the following that he attributes to A. G. Fraser at Bell Laboratories.

> By varying its ratio of computing to input/output or its paging rate, the service can transmit information which a concurrently running process can receive by observing the performance of the system. The communication channel thus established is a noisy one, but the techniques of information theory can be used to devise an encoding which will allow the information to get through reliably no matter how small the effects the service on system performance are, provided they are not zero. The data

rate of this channel may be very low, of course.

This is an important example for a number of reasons, but mostly because it captures the essence of covert and side-channels succinctly: the observation that the ratio or rate at which a program performs a given task can provide information. Classifying the channels for numerous examples of program leakage, he goes on to give an informal definition of a **covert channel**.

> Covert channels [are] those not intended for information transfer at all.

Analyzing the confinement problem. Taking a more policy-based approach and applying existing principles of computer security, Lipner further examines the confinement problem [Lip75]. The discussion on closing covert timing channels suggest that the system assign a virtual time or virtual clock to each process that depends solely on its own activity and not any outside process. Continuing from the example of Lampson [Lam73], Lipner gives an example of paging, i.e., the touching of each page must take a constant amount of time. The work observes that the problem of limiting the observed time of a process to the virtual one is indeed difficult: the program can choose to perceive the passage of time however it wishes, independent of the virtual clock provided by the system. The goals and implementation of a dynamic time-based scheduler are counteractive to those of eliminating the ability of a process to correlate between virtual and real time. The compromise results in (at best) reduced timing granularity: introducing more noise into the covert channel and reducing the throughput.

Covert storage and timing channels. In 1972, IBM released an operating system for their mainframes called VM/370. While retrofitting a security architecture to this operating system called KVM/370, Schaefer et al. explore the confinement problem [SGLS77]. They give their own informal definition of a covert channel.

> Covert channels are [data] paths not meant for communication but that can be used to transmit data indirectly.

They further distinguish between two types of covert channels: **storage channels** and **timing channels**. This dissertation concerns mainly the latter.

> Storage channels consist of variables that are set by a system process on behalf of the sender, e.g., interlocks, thresholds, or an ordering. In timing channels, the time variable is controlled: resource allocations are made to a receiver at intervals of time controlled by the sender. In both cases, the state of the variable ("on" or "off", "time-interval is 2 seconds") is made to represent information, e.g., digits or characters.

They go on to give numerous examples of practical covert channels they encountered including CPU scheduling, I/O scheduling, and timing of I/O operations. One of the most interesting examples they give is disk arm movement that exploits the disk scheduling algorithm. This is now one of the classical examples of a storage channel. An elevator disk scheduling algorithm behaves similarly to a normal elevator, moving the arm in one direction until no further requests are pending in that direction, then starts moving in the opposite direction. They explain how a covert channel exists essentially because the order in which cylinder requests are filled depends on the current direction of the arm.

> Let R own a minidisk at cylinders 51 through 59 of some real disk, to which S has read access. R issues a request for cylinder 55, loops until notified of its completion, then relinquishes the CPU. S then issues a request for either cylinder 53 (to send a 0) or 57 (for a 1) and relinquishes the CPU. R then issues requests for both cylinder 58 and 52. If the request for cylinder 58 completes first, a 1 is received because S left the arm at cylinder 57 and the algorithm continues the upward motion; a 0 is similarly received if the request for cylinder 52 completes first.

They remain pessimistic about the feasibility of eliminating all said channels and settle on mitigations that minimize the bandwidth and increase the noise of the covert channels.

Identifying covert channels. In contrast to previous work that identifies covert storage and timing channels in an ad hoc fashion, Kemmerer presents a more rigorous methodology [Kem83]. He gives his own definition of a covert channel, noting that it differs from that of Lampson [Lam73] due to the distinction between storage and timing channels.

> Covert channels use entities not normally viewed as data objects to transfer information from one subject to another. These nondata objects, such as file locks, device busy flags, and the passing of time, are needed to register the state of the system.

The methodology first identifies all shared resources of the system. This includes not only the resource itself, but possibly numerous attributes of the resource. For example, a file can be a shared resource but attributes like a lock flag or file size are distinct attributes. It then identifies various primitives available on the system, for example read file or write file. One can then examine every possible pair and determine what the potential throughput of the channel is, if one exists. This allows incremental evaluation as new resources and primitives are added to the system. Kemmerer lists quite concrete requirements for the existence of storage and timing channels, the latter of which follows.

> The minimum criteria necessary in order for a timing channel to exist are as follows:
>
> 1. The sending and receiving processes must have access to the same attribute of a shared resource.
>
> 2. The sending and receiving processes must have access to a time reference such as a real-time clock.
>
> 3. The sender must be capable of modulating the receiver's response time for detecting a change in the shared attribute.
>
> 4. There must be some mechanism for initiating the processes and for sequencing the events.

The work reinforces the idea that it is infeasible to eliminate all covert channels and efforts should concentrate on identifying such channels and minimizing their throughput.

The Orange Book. The Rainbow Series is a collection of security-related guidelines issued jointly by the United States Department of Defense (DoD) and the National Computer Security Center (NCSC), part of the National Security Agency (NSA). Appearing in 1983, the first publication in the series is the DoD "Trusted Computer System Evaluation Criteria" (TCSEC) fondly referred to as the Orange Book. Its purpose is to more rigorously define security aspects of computer systems and allow for a concrete evaluation with respect to the defined criteria. It contains a chapter dedicated to covert channels and offers the following definition [dod85, Sec. 8].

> A covert channel is any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy.

It also differentiates between covert storage and timing channels, offering the rather insightful definition of the latter.

> Covert timing channels include all vehicles that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information.

Although now dated, the Orange Book heavily influences its modern successor, the ISO Common Criteria. The inclusion of covert channels in the Orange Book is significant because it formally recognizes covert channels as a security threat in a documented standard. The guidelines suggest identifying covert channels and their associated throughput.

A practical covert timing channel. Multiplexed Information and Computing Service (Multics) is an operating system that began development in 1964 and originally was a joint effort between MIT, General Electric, and Bell Laboratories. In a concrete realization (on Multics) of the covert timing channel using paging described by Lampson [Lam73], Van Vleck gives an account of what is now one of the most classical examples of practical covert timing channels [Vle90].

> My friend, Bob Mullen, astounded me a few days later by showing me two processes in the terminal room. You could type into one and the other would type out the same phrase a few seconds later. The processes were communicating at teletype speed by causing page faults and observing how long they took. The sender process read or didn't read certain pages of a public library file. The receiver process read the pages and determined whether they were already in core by seeing if the read took a long or short real time.

Considering mitigation of general covert timing channels, similar to the permission attribute that controls the reading and writing of files, Van Vleck suggests a certification attribute that controls the execution of a program. Furthermore, similar to review process of operating system kernels, software should be examined manually for covert channels to classify their certification level.

Operating system design and covert channels. Ideally, security should be considered a first class citizen when designing an operating system, and not an afterthought through retrofitting. Digital Equipment Corporation (DEC) adopted this principle beginning in 1981 when designing a VMM security kernel for their VAX architecture, described by Karger et al. [KZB$^+$90]. Somewhat different from a modern operating system, the task of the VAX Virtual Machine Monitor (VMM) is to manage virtual machines at different security levels. One of its design goals is to identify covert channels from the onset and apply appropriate countermeasures.

Disk scheduling algorithms and covert channels. In the early 1990s, a team of researchers at DEC dedicated to the analysis of covert channels within the VAX security kernel released a flurry of results to the academic community. Inspired by the disk arm covert channel described by Schaefer et al. [SGLS77], Karger and Wray analyze said channel with respect to VAX [KW91]. Building on the existing result of the elevator disk scheduling algorithm covert channel, the work examines a number of different disk scheduling algorithms for potential covert channels. They show that the straightforward countermeasure of repositioning the arm after every request is, in general, too costly. One reason the work is significant is because it demonstrates that covert channels are not solely problems to consider at the operating system level. The specific related hardware they work with implements an efficient disk scheduling algorithm on the controller itself that provides a covert channel, yet the operating system has no authority over the controller's policy. Ironically, in deploying covert channel countermeasures the operating system must spend disproportional time in nullifying the optimizations implemented at the lower hardware level.

Mitigating covert channels with fuzzy time. The drawback of Lipner's idea to use virtual time to eliminate covert timing channels is that each process still has some concept of real time, independent of whatever virtual time the operating system presents to it [Lip75]. Furthermore, exposing only a virtual time to programs inherently imposes formidable restrictions on them: the functionality of those relying on being able to accurately measure the passage of time comes into question. Focusing on the VAX security kernel, Hu proposes fuzzy time to address these concerns and mitigate covert timing channels introduced by either the operating system or attached hardware [Hu91]. To address operating system clocks, fuzzy time randomizes the length of the operating system tick interval and reduces the granularity of the system-wide time register. To address I/O clocks, fuzzy time randomizes the response time of requests, i.e., the time at which notification of the completion of an event occurs. Hu states that the implementation of fuzzy time within the VAX security kernel proved to be an extremely effective mechanism to mitigate covert timing channels, overwhelmingly inhibiting their usability measured by increase of noise and reduced throughput.

Data caching and covert channels. The line between a covert storage channel and a covert timing channel is not always clear, as demonstrated by Wray [Wra91]. Using the disk arm storage channel example of Schaefer et al. [SGLS77] previously discussed, Wray explains that if, in the final step, S issues only a single request and measures the time required for the request to complete, S can infer the exact same bit value by relatively comparing the obtained timings. Although this is the same physical channel, with respect to classification it changes from a storage to a timing channel. As such, Wray questions the common distinction between storage and timing channels since such a classification can be misleading, for example when auditing covert channels using a classification-based methodology. For the purposes of this dissertation, one of the most interesting examples Wray gives of a covert channel is as follows.

> Consider a uniprocessor with a direct-mapped cache, running two processes containing covert channel exploitation programs. One, at low secrecy, reads sufficient memory locations to fill the cache with low secrecy data, and then relinquishes the CPU. The other process, running at a high secrecy level, read certain memory locations, causing some cache slots to be re-filled with high secrecy data, and then relinquishes the CPU. Finally, the first process re-reads the data it read earlier, but measuring the time of each read attempt against a reference clock. Those memory locations which correspond to cache locations that were filled with high secrecy data will take significantly longer to read than the locations that still contain low secrecy data, as the displaced low-secrecy data must be re-fetched from main memory. This exploitation allows the timing of individual instructions to be modulated with precision.

This lovely example demonstrates how microprocessors equipped with caching mechanisms run the risk of introducing a covert timing channel through the use of a cache as a shared resource. Furthermore, it is another good example of a covert channel introduced by the underlying hardware instead of the operating system. An extended version of the work appears later in a journal [Wra92].

Data caching and further covert channels. As Wray's example of the cache channel illustrates [Wra91], hardware timing channels potentially pose a much greater risk to security than software channels. This is in part due to their comparatively higher throughput. Hu reiterates this point and uses the cache channel as a case study, providing the following example of its exploitation [Hu92].

> The cache channel can be exploited as follows. Assume that there are only 2 user processes in the system: a Trojan horse process at a high access class and a Spy process at a lower access class. The Spy process initially loads the cache with known values by making a series of memory references. It then releases the CPU. Since the Trojan horse process is the only other process, the scheduler runs the Trojan horse process next. To send a "1", the Trojan horse process would overwrite all the contents of the cache. To send a "0", the Trojan horse process would immediately give up the CPU so as to minimize its effect on cache contents. When the Spy process regains the CPU, it reads the same memory locations it previously accessed and measures the read latency. The read latency is just the amount of time it takes to read the memory location. If the references take a relatively short time to complete, then the Spy process knows that the references were satisfied from the cache. Thus, the original cache contents were undisturbed, and the Trojan horse transmitted a "0". Otherwise, the Trojan horse transmitted a "1".

Flushing the cache during context switching is a straightforward mechanism to mitigate the cache channel. This usually has a detrimental effect on performance and hence can only be applied in rare cases. One logical trade-off is to flush the cache only when necessary, i.e., when lower secrecy data can potentially evict higher secrecy data from the cache. This poses an interesting research challenge: scheduling processes in a manner which minimizes the invocation of costly timing channel mitigations, such as cache flushing in the case of the cache channel. Hu proposes such a novel scheduler called a lattice scheduler.

Architectural approaches to covert channel mitigation. Multilevel security seeks to manage data with different sensitivity levels and users with different clearances within a single system. There are different ways to realize this, but any shared resource between security levels is a potential covert channel and/or violation of the security policy. For example, a data cache in a multilevel processor can yield a covert channel. A disk scheduling algorithm in a multilevel disk drive can yield a covert channel. A job scheduling algorithm in a multilevel operating system can yield a covert channel. Lamenting the current state of covert channel mitigation and research trends aimed at multilevel processors, Proctor and Neumann argue that the problem of covert channels in multilevel security systems is insurmountable and suggest solutions at the architecture level [PN92]. For example, instead of attempting to identify and mitigate all possible covert channels in a multilevel processor (this is arguably not even feasible, let alone practical), assign a single-level processor to each level. They argue that the trend of decreasing hardware costs makes this approach viable. Indeed, decades later a modern common desktop system is equipped with multiple CPUs and/or multiple computing cores, and even possibly one or more graphics processing units (GPUs) with hundreds of computing cores. In contrast to covert channel security in multilevel processors, they suggest research should focus on multilevel disk drives and networks, where covert channel security can be more adequately addressed and, in the end, provide more concrete guarantees.

The Light Pink Book. Achieving the higher TCSEC security ratings detailed in the Orange Book requires an analysis of covert channels and, in the case of high bandwidth, assurance that the channel has been closed or sufficiently throttled using documented countermeasures. Another entry in the Rainbow Series, the Light Pink Book “A Guide to Understanding Covert Channel Analysis of Trusted Systems” aims to help developers meet these criteria, in part by identifying and classifying covert channels [ncs93]. The document contains a number of interesting, practical examples of both storage and timing channels, including those introduced by CPU scheduling, shared hardware resources, resource exhaustion, I/O scheduling, I/O operation completion, and memory management. The guidelines recommend various tools for identifying storage channels, but emphasize that the tools do not address timing channels.

4. Cache-Timing Attacks

cache: a ~~secure~~ place of storage.

B. B. Brumley

A **side-channel** or **side-band** is an unintentional covert channel: one party is knowingly communicating via the channel yet the other is unaware. This chapter makes the transition from covert channels as a security topic to side-channels as a cryptology topic, using them for cryptanalytic purposes. The topic of side-channels is a history-rich one, dating back at least to World War II. Koblitz and Menezes give a concise review of its origins [KM11, Sec. 4.1]. Academic interest in this specific area dates back to at least the 1990s.

In contrast to traditional cryptanalytic techniques such as differential and linear cryptanalysis that target the formal mathematical description of a cryptographic primitive, **physical attacks** or **implementation attacks** target a concrete implementation of said primitive. A non-exhaustive list of physical attack techniques includes power analysis, timing analysis, electromagnetic radiation analysis, acoustic analysis, fault injection, power glitching, physical tampering, bus probing, cold boots [CPGR05], and bug attacks [BCS08]: consult Popp's work for a survey [Pop09]. Standaert offers the following classification of physical attacks [Ver10, p. 27].

1. Invasive vs. non-invasive: Invasive attacks require depackaging the chip to get direct access to its inside components; a typical example of this is the connection of a wire on a data bus to see the data transfers. A non-invasive attack only exploits externally available information (the emission of which is, however, often unintentional) such as running time, power consumption.

2. Active vs. passive: Active attacks try to tamper with the devices' proper functioning, for example, fault-induction attacks will try to induce errors in the computation. As opposed, passive attacks will simply observe the devices behavior during

their processing, without disturbing it.

As the above definitions connotate, hardware devices (e.g., smart cards and cryptographic tokens) are ordinarily the main target of physical attacks. **Side-channel attacks** are a class of physical attacks that often fall into the non-invasive, passive category: they utilize a side-channel exposed through, for example, power consumption or timing measurements.

The above physical attacks classification is more rigorous for hardware-based side-channel attacks, while most of the side-channels considered in this dissertation are software-based. On the software side, a subset of the attacks will require execution of a malicious program or **spy process** that runs in unprivileged user space [GGP07, Sec. 2]. Precisely how to place said attacks in the above context of physical attacks is debatable. The following examples support this statement.

1. The victim is a system. The spy process must execute within this system. The attack is therefore invasive.

2. The victim is a legitimate process within a system. The spy process executes in unprivileged user space, independent of the legitimate process. The attack is therefore non-invasive.

3. The spy process influences the execution of the legitimate process indirectly by manipulating the system state. The attack is therefore active.

4. It is normal, not disturbing, for a multitasking operating system to handle multiple processes (including the spy process) pseudo-concurrently. The attack is therefore passive.

Irrespective of how these attacks requiring execution of a spy process are classified as physical attacks, the threat model is clear. The typical attack scenario is a spy process running concurrently on the same physical CPU as the victim process (usually the execution of a cryptographic primitive). This dictates that the attacker must gain authorization to execute the spy process locally. This would be trivial, for example, if the attacker has valid login credentials to a victim Secure Shell (SSH) server. This would be impossible, for example, if the attacker cannot install and execute the spy process on the target system.

A **timing attack** is a side-channel attack that recovers key material by exploiting cryptosystem implementations that do not run in constant time, i.e., their execution time measured by the attacker is somehow state-dependent and hence key-dependent. The previous chapter discusses covert channels, with a focus on covert timing channels. One such covert channel was that made available through the processor's cache memory: **cache-timing attacks** exploit this as a side-channel. The focus of this survey is on cache-timing attacks, but also includes selective results on more general side-channel attacks that are partially related and build the foundation for cache-timing attacks. This chapter proceeds chronologically through said related literature. The survey is interleaved with discussions on the publications included in this dissertation, outlining how said publications relate to the existing work.

Academic literature contains a plethora of results concerning cache-timing attacks. Said results can be organized and presented in any number of ways: for example, chronologically, by cryptosystem, by side-channel, or by attack model. Indeed, it is feasible to produce a taxonomy of the results based on any of these criteria. The approach in this dissertation to proceed chronologically is neither rash nor arbitrary. The major stimulus for results in this area is new side-channels and new methods to exploit and/or realize them. This is largely cryptosystem-independent and naturally occurs in a chronological fashion.

Timing attacks. Giving a number of remarkably simple timing attacks, the seminal work of Kocher is the first to consider side-channel attacks on cryptosystem implementations [Koc96]. Consider a right-to-left square-and-multiply algorithm for exponentiation. If the exponent bit is a 1, the algorithm performs the assignments $B := B \cdot A$ then $A := A^2$. Otherwise, a 0-bit and the algorithm performs only the assignment $A := A^2$. The attacker chooses operand A hence its value in each iteration is known. To mount a timing attack, the attacker is tasked with finding input A that distinguishes former cases from the latter. This could be done by choosing A such that the former case incurs measurably increased execution time over the entire exponentiation yet the latter case does not. Varying the number of computer words in A could be one method to induce this behavior. Starting with the least significant bit, the attacker repeats this process to recover the key iteratively. In this manner, the attacker traces its way through the states of the exponentiation algorithm using the timings as evidence. Kocher gives further examples of software mechanisms that lead to timing vulnerabilities as well as attack experiment results. Kocher focuses on public key cryptosystems with a static key such as RSA and static Diffie-Hellman. Effectively a harbinger of cache-timing attacks, Kocher closes with the following statement.

> Timing attacks can potentially be used against other cryptosystems, including symmetric functions . . . RAM cache hits can produce timing characteristics in implementations of Blowfish, SEAL, DES, and other ciphers if tables in memory are not used identically in every encryption.

Remark. In a very general way, Kocher's timing attack traces its way through the states of the server-side execution of a known algorithm with states that should critically remain secret. Viewing the state space as a tree, it decides what paths are taken in this space using the timings as evidence. Therefore, it requires some timing characteristic that:

1. Occurs with high enough probability that it is possible to find inputs that induce said timing characteristic.

2. Occurs with low enough probability that said timing characteristic is unlikely to occur for random inputs.

It seems reasonable to extend the attack to other side-channels. In fact, the algorithm in Publication VII can be viewed as a fault attack (or bug attack [BCS08], i.e., passive fault attack) analogue of Kocher's timing attack. To be specific, it requires a computational fault that:

1. Occurs with high enough probability that it is possible to find inputs that induce said fault.

2. Occurs with low enough probability that said fault is unlikely to occur for random inputs.

A nice, practical example of such a fault that Publication VII exploits is the implementation of fast modular reduction routines for NIST standard elliptic curves P-256 and P-384 in OpenSSL versions up to and including 0.9.8g. Due to an implementation error, the modular reduction computes incorrectly in quite rare instances: assuming random inputs to a modular multiplication or squaring routine, the faulty reduction occurs with probability roughly 2^{-32} and for that reason went undetected by test vectors. The attack in Publication VII uses this fault to recover private keys of TLS servers supporting ECDH and ECDHE ciphers; see Fig. 4.1 for an illustration of the TLS handshake. To be able to exploit the fault, the attacker must be able to observe whether its chosen inputs induce the fault or not. This heavily depends on the cryptosystem or protocol under attack. To attack ECDH and ECDHE ciphers in TLS, the attacker client does not explicitly see the result of the server-side computation of the shared secret. Hence to determine if the fault occurred or not, the attacker instead attempts to complete the handshake successfully. In the key confirmation messages occurring at the end of the handshake, the server will terminate the session if it does not receive the expected messages from the client encrypted with a key derived from the shared key. Thus if the handshake completes successfully, this tells the attacker that the fault did not occur. On the other hand, if the handshake fails, this tells the attacker that the fault did indeed occur. However, this does complicate matters for the attacker, since finding inputs that induce the fault, i.e., a point on the curve, cannot be done freely by simply choosing the coordinates of the point. To be able to complete the handshake successfully, the attacker must know the discrete logarithm of its submitted Diffie-Hellman key. The attack implementation in Publication VII resorts to finding said inputs at random by generating random scalars and demonstrates that this is computationally feasible. The attack in Publication VII is able to fully recover private keys associated with ECDH-ECDSA and ECDHE-ECDSA ciphers from `s_server`, OpenSSL's generic TLS server, as well as applications linked against OpenSSL ($\leq 0.9.8g$) such as the widely-deployed TLS wrapper application `stunnel`.

Side-channel analysis: a cryptanalytic technique. Kocher is careful to point out that measuring execution time is not the only possible side-channel and other potentially harmful channels can exist. Kelsey et al. extend Kocher's work by considering side-channel attacks on product ciphers [KSWH98]. They explain that mathematical attacks on block ciphers such as linear and differential cryptanalysis attempt to exploit some non-random behavior in the cipher. Thus the goal of side-channel cryptanalysis is similar yet makes use of additional data available through the side-channel. They give examples of three different side-channels along with attacks on block ciphers.

1. The first is a timing attack against IDEA, a block cipher by Lai and Massey [LM90]. Most symmetric key primitives are straight-line in the sense that they contain no logic branches. The IDEA block cipher is an exception. It makes use of many constant-time operations on 16-bit words such as bitwise XOR and integer addition modulo 2^{16}. However, it also contains multiplications modulo $2^{16}+1$ where the all-zero word represents 2^{16}. In software, this might be implemented by comparing the operands to zero and handling

```
Client                                           Server
------                                           ------

ClientHello          -------->
                                            ServerHello
                                           Certificate*
                                     ServerKeyExchange*
                                    CertificateRequest*+
                     <--------          ServerHelloDone
Certificate*+
ClientKeyExchange
CertificateVerify*+
[ChangeCipherSpec]
Finished             -------->
                                     [ChangeCipherSpec]
                     <--------                 Finished

Application Data     <------->         Application Data


     * message is not sent under some conditions
     + message is not sent unless client authentication
       is desired
```

Figure 4.1. Message flow in a full TLS handshake.

those cases separately without the use of a multiplication instruction. Such a straightforward approach most likely involves a conditional branch and causes the cipher to not execute in constant-time: if either operand is zero, a multiplication instruction is not performed and the total execution time is lower. The authors outline two attacks that make use of this side-channel. One is a ciphertext-only attack that recovers words of the key iteratively working backwards from the ciphertext and targeting the operands of different modular multiplications using the timing data as evidence. The authors state that this side-channel is inspired by the IDEA implementation in PGP 2.3 running on a 486SX-33.

2. The second attack is against RC5, a block cipher by Rivest [Riv94]. A processor flag is extra internal state that stores additional output from the result of a computation. One such flag on x86 is the carry flag that stores any overflow from the integer addition of two registers. RC5 makes use of two integer additions modulo 2^{32} per-round. The authors define a side-channel consisting of the value of the carry flag after each modular addition. This model is particularly detrimental to RC5 because round keys are one of the integer addition operands. The authors outline two simple attacks: one using ciphertext-only and the other an adaptive chosen-plaintext attack. While the authors explain

the attacks in sufficient detail, it remains unclear how to realize this side-channel. One novel feature in RC5 is its use of data-dependent rotations: register rotation distances that are not constant or fixed but take their value from a register. While this is simple to realize on x86, some architectures are not equipped with a variable-distance rotation instruction. In either case, the authors mention that the manner in which the developer implements the variable-distance rotation can introduce a timing side-channel. The same argument carries over to hardware implementations where power consumption may leak the rotation distance. It seems feasible to apply similar analysis techniques to other ciphers that employ integer additions.

3. The third attack is against the DES block cipher. The authors first consider the case where the Hamming weight of the state leaks after execution of the penultimate round. A plethora of attacks then exist: the simplest is perhaps guessing the last 48-bit round key and checking the result against a small number of ciphertexts, i.e., peeling off the last round and checking the Hamming weight of the resulting state. The authors extend this idea and describe a more statistical attack that encompasses a wide range of side-channel models involving Hamming weight leakage.

The authors sympathize with Kocher's concern on the data cache acting as a side-channel [KSWH98, Sec. 7].

> The purpose of this paper was to demonstrate the power of side-channel cryptanalysis against product ciphers. Our attacks are by no means exhaustive; the algorithms discussed have other possible side channels and other attacks are possible given other side channel information. And other product ciphers are vulnerable to similar attacks. We believe attacks based on cache hit ratio in large S-box ciphers like Blowfish, CAST, and Khufu are possible.

An extended version of the work appears later in a journal [KSWH00].

A theoretical cache-timing attack on DES. The previously mentioned works warn that a data cache can potentially be used as a side-channel to attack implementations of cryptographic primitives. Developing this notion, Page presents a theoretical cache-timing attack against the DES block cipher [Pag02a, Pag02b]. This is the first public work to formalize a side-channel model for a data cache. The cache architecture under consideration is a 1kB direct-mapped cache with 4B cache lines, modeled around ARM and MIPS embedded processors at the time. The attack assumes the cipher execution begins with an empty cache and that for each memory access in the cipher description, the attacker is told whether the access generates a cache hit or miss. The DES description contains eight S-boxes and the attack is carried out using relations derived using the first two rounds of the cipher's execution. The attack intuition is as follows. Consider the execution of DES in the first two rounds. By definition, starting with an empty cache all accesses in the first round are cache misses. In the second round, a cache hit (miss) implies that the input for the corresponding S-box lies (does not lie) on the same cache line as that of the first round. Using a single S-box as an example, Page carefully expresses the inputs to the S-boxes for the first two rounds as a function of round key bits and plaintext input, the latter of which is under the attacker's control. The attacker carries out an efficient search for key material by varying the plaintext input using carefully chosen plaintexts and observing the resulting cache behavior. The attack requires 2^{10} chosen plaintexts and a computational effort of 2^{32} to recover the 56-bit key. Page also considers countermeasures to these types of attacks. Pre-loading the tables can be as costly as disabling the cache entirely. Randomizing the cache state before cipher execution provides only probabilistic protection. The work shows that non-deterministic cache placement, i.e., dynamically altering the cache mapping policy, is too costly. The only countermeasure presented optimistically is non-deterministic access ordering, i.e., executing the memory accesses in random order. This is not always possible and depends on the degree of parallelism present in the cipher description, i.e., if the memory accesses are inherently serial this countermeasure is not effective. Page's work marks the first theoretical cache-timing attack on a cryptosystem.

A practical cache-timing attack on DES. Page's seminal work models the side-channel as a series of cache hits and misses, each corresponding to a distinct memory access in the description of the cipher. Given the application, he argues that such a model might be realized by power consumption measurements, e.g., of a smart card. The attack is theoretical in the respect that no implementation is presented in the work. Using a different (arguably weaker) side-channel model, Tsunoo et al. devise and implement a cache-timing attack on DES [TSS$^+$03]. For example, consider the following simple round structure: $L_1 = S(L_0 \oplus K_0)$ and $R_1 = S(R_0 \oplus K_1)$ where L_0 and R_0 are plaintext inputs and K_0 and K_1 are round or subkeys. The attacker collects many plaintext-ciphertext pairs along with the time required to produce the ciphertext: that is, the side-channel model provides the total execution time of the cipher. Consider the following two scenarios.

1. The equality $L_0 \oplus K_0 = R_0 \oplus K_1$ results in a cache hit and one infers $L_0 \oplus R_0 = K_0 \oplus K_1$. Tuples with a lower encryption time experienced more cache hits and the relation and differential are more likely to hold. Tsunoo et al. call this the non-elimination method because the resulting exhaustive search proceeds through key material from where the more probable differential holds to the least probable, i.e., the attacker uses timing evidence to explore more likely key material first [TSS+03, Sec. 2.4].

2. The inequality $L_0 \oplus K_0 \neq R_0 \oplus K_1$ results in a cache miss and one infers $L_0 \oplus R_0 \neq K_0 \oplus K_1$. Similarly, Tsunoo et al. call this the elimination method because the resulting exhaustive search proceeds through key material from where the least probable differential holds to the most probable, i.e., the attacker uses timing evidence to explore less likely key material last.

To attack DES, the authors concentrate on the elimination method, explaining that the number of S-box evaluations per-round compared to the size of the S-box means that collisions that ultimately lead to cache hits are less likely. The above example overlooks the fact that a cache line normally holds a number of contiguous values and the attacker can only infer a relation on a portion of the round keys, not the entire round key. The authors design an attack targeting such a relation between the first and last encryption rounds. They implement the attack on a Pentium III "Katmai" that contains a 16kB 4-way set-associative L1 data cache with 32B lines. They use a C implementation of DES for 32-bit platforms that unrolls the linear layer (bit permutations) following the non-linear layer (S-boxes): in practice, this means that the table for each of the eight S-boxes is $64 \cdot 4$ bytes in size, spanning eight cache lines. So while a theoretical attack cannot determine the least significant lg(8) = 3 bits, interestingly the authors are able to reduce this to two bits in practice as a result of unaligned tables, i.e., the tables for each S-box will not always land on a 32B address boundary. As a result, they are able to derive relations on the four most significant bits of the S-box inputs [TSS+03, Sec. 3.2]. With this setup, the authors are able to recover the full 56-bit DES key using 2^{23} known plaintexts and 2^{24} computation effort. This is the first published cache-timing attack on DES with an implementation. The authors cite their previous work, a similar attack against the block cipher MISTY, yet the publication is not readily available [TTMM02]. An extended version of the work appears later in a journal

[TTS+06].

Classifying and mitigating cache-timing attacks. The previous two attacks by Page [Pag02b] and Tsunoo et al. [TSS+03] are similar in that they exploit the behavior of the data cache on the CPU but differ in the side-channel model. Page subsequently categorizes these two results and examines general countermeasures to them [Pag03]. He terms the former **trace-driven** attacks where, for each memory access in the description of the cipher, the attacker is told whether said access induced a cache hit or cache miss [Pag03, Sec. 2.1]: this series of cache hits and misses is called a **trace**. He terms the latter **time-driven** attacks that rely instead on the total execution time of the cipher and operate under the assumption that a cache miss (hit) implies longer (shorter) execution times [Pag03, Sec. 2.2]. He discusses a number of countermeasures considering both trace-driven and time-driven attacks [Pag03, Sec. 3]. The following summarizes three such countermeasures to time-driven attacks that Page proposes, implements, and evaluates. He measures their effectiveness by implementing the attack by Tsunoo et al., confirming the validity of their results [TSS+03]. The author states a success rate of approximately 97% with 2^{17} chosen plaintexts [Pag03, Sec. 4.2].

1. Time skewing performs a random number of dummy operations at the beginning of the encryption [Pag03, Sec. 3.6]. It is up to the implementor to ensure that the compiler does not remove these dummy operations during optimization. This is a reasonable defense because the random time the algorithm spends doing dummy operations masks the timing variation due to caching effects. When performing dummy operations for up to 1000 cycles, the original attack no longer succeeds [Pag03, Tbl. 2].

2. Miss skewing performs a random number of memory accesses into a dummy table; Page calls such a “fake” S-box an F-box [Pag03, Sec. 3.7]. This induces an unpredictable number of cache misses, removing the correlation between the plaintext and the number of cache misses and hence execution time. Performing up to 32 random accesses into the F-box per encryption reduces the success rate of the original attack to 2% [Pag03, Tbl. 2].

3. Instead of starting the encryption with an empty cache, cache warming brings all, or a portion of, the S-box entries into the cache prior to encryption [Pag03, Sec. 3.4]. In this instance, full cache warming is quite detrimental to performance and the author experiments with random cache warming that pre-

loads up to half of the S-box entries into the cache. This can be implemented by simply performing a random number of random accesses to the S-box. This reduces the success rate of the original attack to 5% [Pag03, Tbl. 2].

A remote timing attack on RSA. Many RSA implementations including that in OpenSSL use Montgomery reduction to replace expensive modular reductions with cheaper modular reductions modulo a power of two. Said Montgomery reduction step follows each multi-precision multiplication and squaring step. If the output of the Montgomery reduction step is greater than one of the modulus factors, an extra multi-precision subtraction is required. Walter and Thompson show how a side-channel consisting of the presence or absence of these extra reduction steps can be used to reveal exponent bits [WT01]. Schindler optimizes and slightly generalizes the attack [Sch02]. Furthermore, OpenSSL varies its choice of multi-precision multiplication methods based on the number of words in the operands. If the number of words are the same, it applies the Karatsuba divide-and-conquer method. If the number of words differ, it applies schoolbook multiplication. D. Brumley and D. Boneh devise and implement a timing attack on OpenSSL that exploits the behavior of both of these mechanisms: the extra reduction step and the varying multiplication routine [BB03]. The attack is essentially an iterative binary search on the bits of one of the modulus factors, starting from the most significant bit, using the time required for the server to decrypt the attacker input as evidence of the correctness of bit guesses [BB03, Sec. 3]. The authors give extensive experiment results that demonstrate the effectiveness of their implemented attack in a number of scenarios, including local, remote, and virtual machine environments [BB03, Sec. 5]. The attack requires roughly one million queries to a TLS server and succeeds in roughly two hours. Before this work, the general view of remote timing attacks was that they are impractical due to the noise introduced by network and other latency. This work is one of the most significant in its area because it changed that perception. An extended version of the work appears later in a journal [BB05].

Remark. There are at least two major contributions in the discussed work by D. Brumley and D. Boneh:

1. They analyze the feasibility of running remote timing attacks in a variety of network scenarios with varying degrees of network latency. This contribution is somewhat independent of the timing attack itself, i.e., it is seemingly reasonable to replace the underlying timing attack vulnerability with any such

vulnerability where the server-side secret inputs are not changing.

2. They devise and implement a remote timing attack using two such vulnerabilities in OpenSSL's implementation of RSA. One of the more practical scenarios they consider is using the timings of the messages exchanged during the TLS handshake to recover the private key.

Regarding the first point, this is not necessarily an incredibly interesting cryptology topic: it can indeed be an obstacle when mounting the attack, but from the theoretical standpoint the security of the private key should be independent of network latency. Regarding the second point, a natural concern is how such timing attacks affect other ciphers that can be used for server authentication in the TLS handshake. Considering DSA and ECDSA, these are quite different from RSA since generating DSA and ECDSA signatures use nonces, i.e., the secret input is always distinct for the most time consuming operation during signature generation. This is in stark contrast with RSA with a fixed private exponent. Building on the work of D. Brumley and D. Boneh, Publication VI presents a remote timing attack on OpenSSL's implementation of ECDSA. The implementation of scalar multiplication of points on elliptic curves over binary fields uses an algorithm that has a fixed cost per bit, i.e., performs the same sequence of curve and finite field operations regardless of the value that any particular bit takes. Contrast this with a left-to-right scalar multiplication method that only performs point additions on non-zero key bits. As a result, in this case the implementation has a fixed time and computation cost for a single iteration of the scalar multiplication loop. However, the computation starts from the most significant bit: the implementation drills down using bit tests to find the top bit of the scalar and begin the computations from there. As a result, there is a direct correlation between the time required to compute a scalar multiplication and the number of bits (i.e., base two logarithm) in the scalar. After identifying this vulnerability, Publication VI uses it to steal the private key of a TLS server that authenticates using ECDSA signatures. The attack proceeds in two phases. In the first phase, the attacker repeatedly opens TLS sessions and measures the time required for the server to respond with a message authenticating messages in the beginning of the TLS handshake with an ECDSA signature. This is repeated for a moderate number of sessions (2^{13} or 2^{14}). Due to the identified timing attack vulnerability, those signatures with a lower time measurement correspond to signatures with nonces that have a lower number of bits. This provides the attacker information on the top part of the nonce. In the second phase of the attack, the attacker uses this information to mount a lattice attack to recover the private key. The attacker is essentially left with an underdetermined system of equations for which a small solution must be found, and such solutions should be quite rare. This is precisely where lattice algorithms are useful. Implementing the attack, the attack results in Publication VI show that the attacker is able to succeed in both local and remote attack scenarios with as few as 2^{13} queries to the server. In response to the attack in Publication VI, CERT issued[1] vulnerability note VU#536044 and the OpenSSL team integrated the countermeasure patch provided in Publication VI into their development code for future stable releases.

A cache-timing attack on AES. AES is arguably the most relevant modern block cipher. Software implementations of AES targeting high-speed applications are largely table-based, i.e., replacing low-level finite field operations involved in the non-linear and linear layers with a series of table lookups that compute the linear layer output from the non-linear layer input. Bernstein devises time-driven cache-timing attack on such implementations [Ber04]. The intuition is that, due to caching effects, the time required to encrypt one AES block is correlated with (part of) the key, which can be exploited by examining the distribution of said timings as a function of attacker-chosen plaintext. This is used to recover (part of) the inputs to the table lookups in the first AES round, which are a function of the key and plaintext, the latter being known to the attacker. What follows is a simplified example of the attack. Consider a known function $f : X \times Y \to Z$ implemented with a table lookup by $(x, y) \mapsto T[x \oplus y]$. In a profiling stage with an environment identical to that of the victim, the attacker measures the (noisy) time to compute $f(a, 0)$ for many random $a \in X$. Consider the average time as a function of a and assume this value is maximal for a distinct $b \in X$. The victim computes $f(a', y')$ with $a' \in X$ chosen by the attacker and $y' \in Y$ private and fixed, discards the output, and returns the (noisy) time required to compute f. The attacker submits many random $a' \in X$ and records said time. Consider the average time as a function of a' and assume this is maximal for a distinct $b' \in X$. The attacker deduces that $b \oplus 0 = b' \oplus y'$ holds and solves for the only unknown y'. This scenario immediately applies to AES because x is a plaintext byte and y a key byte. In this manner, the attacker derives a single key byte: the same logic applies to the other key bytes with additional profiling. The author's implementation runs with a custom server process that plays the role of the victim and returns the number of Pentium III cycles required to encrypt a single block from the client with AES in OpenSSL

[1]http://www.kb.cert.org/vuls/id/536044

(0.9.7*a*). The 32-bit word-based AES implementation uses four distinct tables with 256 words per table and performs a round using sixteen table lookups. The profiling stage may not produce a distinct maximum but instead a number of candidates with which the attacker carries out an efficient search for the key. The required 2^{22} queries to the server dominates the attack effort. In reality, the attacker is not given such an accurate cycle count: the author claims that this can be compensated for by simply taking more measurements [Ber04, Sec. 4]. This work is significant because of the ubiquity of AES and its software implementations: the target is not a straw man implementation but one within OpenSSL, a widely-deployed library. Furthermore, the author encourages researchers to reproduce the results by providing the source code.

A trace-driven cache-timing attack on AES. Page describes his cache-timing attack on DES under the assumption that the cache is completely empty when the encryption begins and that the attacker procures the trace by measuring the power consumption of the device, e.g., a smart card [Pag02a]. The simplest software implementation of AES uses a 256B lookup table to implement the `SubBytes` step; this is an 8-bit S-box S. Somewhat similar to Page's attack on DES, Bertoni, Zaccaria, Breveglieri, Monchiero, and Palermo give a trace-driven cache-timing attack on such AES implementations [BZB+05]. The attack assumes the attacker can execute malicious code in user space that manipulates the state of the cache by referencing its own memory locations [BZB+05, Sec. 4.1]. An example of the simplest version of the attack follows, assuming a direct-mapped cache with single byte lines. Assume $S[0]$ maps to line 0 without loss of generality. The attacker submits an all-zero block for encryption. With overwhelming probability all bytes of S are now present in the cache. The attacker references an address mapping to line 0, replacing the line contents with the value at said address. The attacker resubmits the all-zero block for encryption and monitors the power consumption of the device. Encrypting the all-zero block, the indices for the sixteen lookups in the first round are in fact all the bytes of the key. If the power trace reveals a cache miss for any of these sixteen lookups, the attacker deduces the corresponding key byte is 0. The attacker iterates this process for all byte values. This requires two queries per byte value, or a maximum of 512 queries total. In reality, a line contains a number of bytes and the attacker cannot deduce the exact value of the lower bits of the indices; they carry out experiments simulating a 4kB direct-mapped cache with 8B lines, leaking the top five bits of each key byte, or 80 out of 128 bits. The authors give convincing results of running the attack in a simulated environment [BZB+05, Sec. 5]. As a countermeasure they propose preloading the S-box into memory, i.e., performing exactly the amount and type of memory references to bring all entries of the S-box into the cache [BZB+05, Sec. 6]. As noted by Page, as a general countermeasure performance-wise this is not always feasible and depends on the size of the tables and the cache geometry [Pag02a, Sec. 5.1].

Access-driven cache-timing attacks. With a single processor core, an operating system can only physically execute one process at a time. Modern operating systems implement multithreading through context switching, i.e., letting a process run for a fixed time quantum then swapping it out for another process. With two physical cores, the operating system can assign two processes to run in parallel. For better resource allocation, another option is that a single physical core present two logical processors to the operating system and instead run two processes in pseudo-parallel. While this requires some duplication of architectural state such as registers and flags as well as arithmetic logic units (ALUs), it allows some components to be shared: for example, data and instruction caches. This type of hardware parallelism is called simultaneous multithreading (SMT). Intel's Pentium 4 is the first modern processor to feature SMT. Intel's implementation is called Hyper-Threading Technology (HTT) [MBH+02]. This also implies that the two processes running in pseudo-parallel also compete for the shared data cache. As such, Percival identifies HTT as a covert channel and security risk [Per05]. He provides a concrete realization of the cache-based covert channel first described by Wray [Wra91]. Percival engineers carefully crafted assembly that can be used to spy on the data cache of the Pentium 4 [Per05, Sec. 3]. The intuition is to bring the cache to some predetermined state, then examine changes to this state induced by another process; a brief description follows. The Pentium 4 has an 8kB 4-way set-associative L1 data cache with 64B lines. It divides its 128 lines into 32 sets, each with four lines. Alice fills all the lines in a single set by reading from four addresses that all share the same set portion of the address, yet do not share the tag portion. Alice repeats this for all cache sets. Bob can then send 32 bits of data by, for each cache set, choosing to read (a 1-bit) or not read (a 0-bit) an address that maps to said set. A read changes the cache state by evicting one of Alice's values from the set. Not reading does not change the state. Alice can receive the 32 bits from Bob by, for each cache set, measuring the time required to re-read the data at her addresses. If the latency is large, this implies a cache miss induced by Bob's read, transmitting a 1-bit. If the latency is small, this implies a cache hit and Bob did not read from the set, transmitting a 0-bit. Percival uses this covert timing channel as a side-channel to spy on an OpenSSL 0.9.7*c* process performing an RSA signature operation. This performs two modular exponentiations using the sliding window technique, where a series of bits in the exponent are used as an index into a lookup table. Percival shows there is significant correlation between the timing traces obtained by the spy and said indices, leaking critical private key material [Per05, Sec. 5]. He designs a factoring algorithm that takes this partial key material into account. Briefly, the algorithm is a breadth first search on the bits of two forms of the private exponent, where at each level nodes get trimmed based on the partial key material available and number-theoretic constraints [Per05, Sec. 6]. It is worth noting that it is not difficult to derive the later-published algorithm by Heninger and Shacham [HS08, HS09] from Percival's version, although there are oversights in the latter. For example, the description assumes that a single linear equation with a single unknown has at most a single solution, but this does not always hold in a ring. As a result, the partial solution space can grow larger than the claimed two-fold. Nevertheless, this efficiently recovers the private key using as little as a single cache-timing trace. Percival's work marks the first practical cache-timing attack on a public key cryptosystem. Moreover, the side-channel itself is not one introduced by the operating system via scheduling and context switching, but the underlying hardware itself: in the end, it is Intel's HTT running two processes in pseudo-parallel that makes this particular attack feasible. In the literature, this breed of attack is now termed an **access-driven** attack. These attacks are similar to trace-driven attacks, yet differ in the attack model and the side-channel implementation. Here the attacker is allowed to manipulate the cache state as the victim executes. This is a significantly more powerful attacker that, in a theoretical model, for each memory access in the cipher description, is told not only whether it induces a cache hit or miss but also what cache set it maps to, and furthermore is able to evict all victim data between successive accesses.

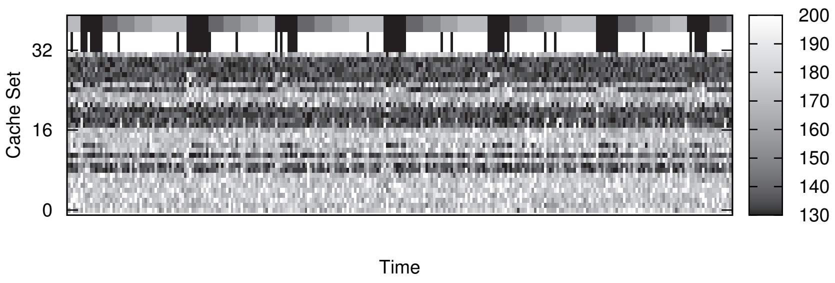
Remark. One of the major contributions of Percival's work is explicitly providing a strategy and technical details for realizing a data cache spy process. This allows researchers to verify the result. One significant issue is how to process the resulting signal to recover internal algorithm state. This is precisely the problem that Publication I addresses. At a very concrete level, the spy process yields successive vectors of cache-timing data where each vector component is a timing measurement for a distinct cache set. The framework that Publication I proposes processes this vector timing data in two stages.

1. Vector Quantization (VQ) is a signal processing technique that approximates

vectors from a large input domain with a smaller set of vectors called a codebook. It matches vectors to their closest (Euclidean distance-wise) representative in this codebook. It can be helpful to view VQ in the context of lossy data compression. Instead of transmitting the original vector, the index of the most similar codebook representative is transmitted. On the receiving end, this index is converted back to the codebook vector: the original vector is most likely not obtained, but the closest available approximation of it. The framework in Publication I uses VQ for classification: it matches an incoming timing vector to the closest existing timing vector in the codebook. The codebook itself is produced in a profiling stage where the attacker spies on the victim algorithm with its own known secret inputs, for example by executing the same algorithm in its own user space. To summarize, the first stage of the framework in Publication I matches new incoming timing data to existing timing data, the latter of which represents the algorithm in a known state, and thus this known state is a good guess at the state for the incoming data. This stage essentially reduces the dimension of the side-channel to facilitate further processing. This "templating" portion of the framework relates to the template attacks concept by Chari et al. concerning power analysis [CRR02].

2. A Hidden Markov Model (HMM) is a statistical model of a discrete-time stochastic process. The internal state of the process remains hidden, yet as the process changes states it emits one of a number of observations with given probabilities. HMMs are an effective signal processing technique to eliminate noise in the signal, i.e., recover the (most likely) original states that the process passed through by only observing the state emissions. It may come as no surprise that HMMs have been applied in side-channel analysis before: see the works of Karlof and Wagner [KW03] and Green et al. [GNS05]. The framework in Publication I uses the output from the VQ stage as observation input to the HMM stage. The goal of the HMM stage is to output the most likely sequence of algorithm states that produced the observations. Similar to the VQ stage, the HMM training takes place during a profiling stage.

As an example of applying this framework, Publication I mounts an access-driven cache-timing attack on OpenSSL's implementation of ECDSA. Said implementation uses a textbook left-to-right double-and-add algorithm for scalar multiplication and a wide modified Non-Adjacent Form (NAF) for scalar representation where, in each point addition step, a lookup into a memory-resident table of precomputed points takes place. The attack uses the framework to re-



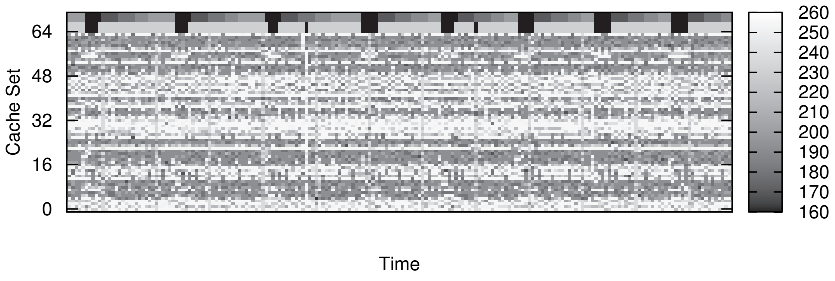


Figure 4.2. Using the framework in Publication I, cache-timing data from a spy process running concurrently with an OpenSSL 0.9.8*k* ECDSA signature operation; 160-bit curve with a width-4 NAF. Top: Pentium 4 timing data, seven point additions denoted with black metadata. Bottom: Atom timing data, eight point additions denoted with black metadata. Repeated point doublings occur between the additions. The top eight rows are metadata; the bottom half the VQ label and top half the HMM state. All other cells are the raw timing data, viewed as column vectors from left to right with time. Any timing data cell with a higher value, i.e., lighter shade, suggests a cache miss. Any cell with a lower value, i.e., darker shade, suggests a cache hit.

cover a portion of the nonce used for ECDSA signature generation. This small portion of the nonce is then used as input to a lattice attack to recover the long-term ECDSA private key from many such signatures. This requires only a few thousand signatures to be collected, along with the corresponding side-channel data, and runs in a matter of hours. The attack in Publication I gives implementation results for both the Intel Pentium 4 and Atom processors. Figure 4.2 illustrates the effectiveness of the framework. As mentioned in Chapter 1, another microarchitecture attack breed is that which exploits the behavior of the branch predictor. Further applying the framework to process the resulting side-channel of the branch predictor is an important topic for future work.

Remark. Percival's work also led to changes in the OpenSSL source code, implementing countermeasures to the attack for RSA modular exponentiations using a sliding window. This specific countermeasure aligns values in the pre-computed lookup table so that lookups into said table reflect the same pattern in the trace, the goal being to hide the lookup table index, i.e., exponent key bits. While this can be effective, said countermeasure only applies to one cryptosystem: namely, RSA. Treating the vulnerability in a more general fashion, Publication IV examines the use of a shared context within OpenSSL as a cache-timing attack countermeasure. A robust library such as OpenSSL uses dynamic memory allocation to reserve space for objects such as multiprecision integers, finite ring elements, and finite field elements. Since one object type can be used for multiple cryptosystems, e.g., RSA, DSA, and ECDSA all use multiprecision integers, only one implementation of these objects exists and is very flexible with respect to size. That is, their size is not necessarily known in advance, e.g., the integers in RSA are much larger than those in ECDSA. This implies that dynamic memory allocation must be used at runtime to set aside the needed amount of space for the object. However, all of the mentioned cryptosystems need some temporary space for intermediate values. For example, to compute $x \cdot y \mod p$, it first computes $z = x \cdot y$ where z requires twice the space to be stored temporarily, then computes $z \mod p$ reducing the magnitude. Dynamically allocating memory is a costly operation, so to optimize functions where temporary space is needed repeatedly, it makes sense to store the dynamically allocated memory and reuse it across functions: for example, within a modular exponentiation function. This is precisely what OpenSSL does (as well as other libraries offering multiprecision integer arithmetic) and terms the mechanism a shared context. One general cache-timing attack countermeasure suggested, but not implemented, in Publication I is to modify the allocation policy of the shared context by randomizing its behavior. Taking a more straightforward approach, Publication IV modifies the allocation policy of the shared context to align all dynamically-allocated memory at the same address boundary, said boundary determined at library compile time depending on the architecture. The intuition is that if all dynamically-allocated data maps to the same cache set, this significantly impedes access-driven cache attacks since little to no information can be inferred about operands. After implementing the countermeasure as a patch to the OpenSSL source code and evaluating its effectiveness, Publication IV arrives at a counter-intuitive result. The traces obtained for patched and unpatched versions are not significantly different, suggesting that the allocation policy of the shared context has little impact on the resulting side-channel, at least on the Intel Pentium 4 architecture under consideration. This poses the open question of what software trait is triggering some microarchitecture feature that contributes most to this signal.

Access-driven attacks on AES. Percival's attack is a local attack in that it requires the attacker to execute the spy program locally on the same processor core as the victim program. Independent and concurrent to Percival's work, Osvik, Shamir, and Tromer develop a similar local cache-timing attack on AES [OST05, OST06]. They consider both synchronous and asynchronous attacks. In this context, synchronous means that the attacker can bring the cache to some known state, trigger the victim to encrypt some known plaintext block, then examine the cache after encryption of the block. That is, the attacker can only examine the cache before and after the encryption, not during. The asynchronous model allows the latter: it can be realized as a result of Intel's HTT. A short description of the synchronous attack follows. Denote key bytes k_i and plaintext bytes p_i. The first nine out of ten AES rounds use four lookups into each table T_j for $0 \leq j < 4$. That is, sixteen lookups per round, one for each state byte. The attacker fills the cache with his own data, submits sixteen bytes to be encrypted, then tries to re-read his data cache set-wise. A cache miss implies that the victim accessed that set. However, the attacker cannot infer much from this: there are a total of 36 lookups into a single T_j over the encryption of one block and the attacker cannot deduce which lookup(s) led to the data eviction. On the other hand, a cache hit implies that no lookup mapped to said set. In the first round, the lookup indices are $p_i \oplus k_i$ and with p_i known such an observation eliminates a large number of candidates for k_i. More precisely, reasonably assuming 64B cache lines, each table contains 256 4-byte entries and each line sixteen entries, hence this eliminates sixteen candidates for each of four key bytes. The authors state the probability of this event occurring is approximately 0.104 and iterating this strategy recovers 64 bits of the key using only a few dozen queries. The authors also show how to extend the attack using relations between the first and second round of AES: this recovers the full 128 bits of the key after 8220 encryption queries and approximately 2^{29} steps in analysis. To carry out such cache-timing attacks in practice, one needs to realize the side-channel. To this end, the authors propose two methods to obtain cache-timing data; these methods are important because, to some extent, they have become the de facto method of procuring said data for researchers in the field.

1. In the first strategy, the attacker submits a plaintext block for encryption. After the encryption completes, the attacker completely fills a single cache set. The attacker then submits the same plaintext and measures the encryption time. If the average time is higher, this implies the victim suffers a cache miss from accessing data mapping to said cache set. The authors term this method Evict+Time [OST05, Sec. 3.4].

2. In the second strategy, the attacker first brings the cache to a known state. Normally this involves either filling the entire cache or relevant cache sets. The attacker then submits a plaintext block. After the encryption completes, the attacker, cache set-wise, measures the time required to re-read the data in the cache sets. High latency implies cache hits and that the victim accessed data mapping to the cache set. There are two fundamental differences from the former strategy. In the former, the attacker obtains only a single timing measurement for a single cache set. Here the attacker obtains a single measurement for a number of cache sets. Also, here the attacker measures the execution latency of its own instructions; in the former, the latency of the victim's instructions. The authors term this method Prime+Probe [OST05, Sec. 3.5].

The authors carry out their attacks on AES through OpenSSL 0.9.8 function calls as well as the `dm-crypt` disk encryption subsystem for Linux. An extended version of the work appears later in a journal [TOS10].

An improved trace-driven attack on AES. As previously mentioned, the trace-driven attack on AES by Bertoni et al. assumes that the encryption begins with the cache in a state chosen by the attacker [BZB+05]. This is a much more powerful attacker than one in Page's model where, in contrast, encryption begins with an empty cache [Pag02a]. Lauradoux adopts the latter model to develop a trace-driven cache-timing attack on AES implementations that use the large T tables [Lau05]. A brief description of the attack follows; assume 32-bit words and 64B cache lines. The four indices into T_0 during the first round are $p_i \oplus k_i$ for $0 \le i < 4$. The first access induces a cache miss since the cache is empty. The goal of the attacker is to derive relations of the form $p_0 \oplus k_0 = p_j \oplus k_j$ for all $1 \le j < 4$ that hold on the four most significant bits. The attacker accomplishes this by cycling through all values in the four most significant bits of p_0 yet fixing the four remaining bits and all other bytes p_i. The attacker submits these sixteen plaintexts for encryption and collects the corresponding traces. One trace must reflect a hit for $p_1 \oplus k_1$. This yields four relations on eight key bits: four bits each of k_0 and k_1. Their are two options for the third access. One possibility is that one trace must reflect a hit for $p_2 \oplus k_2$, meaning the change in attacker-chosen input induced the cache hit. This yields four relations on eight key bits: four bits each of k_0 and k_2. The other possibility is that all traces reflect a hit, meaning the second access induced the cache hit. This also yields four relations on eight key bits: four bits each of k_1 and k_2. Similar logic applies to the fourth access. In total, this provides twelve relations on sixteen key bits.

Applying the same logic in parallel to the other T tables yields 48 relations on 64 key bits. With 16 chosen plaintexts and corresponding traces, this reduces a brute force attack on the key to 2^{80} steps, deferring a more in-depth cryptanalysis to future work [Lau05, Sec. 3.1]. The author also analyzes the attack against implementations that use only a single T table but with rotations: with 240 chosen plaintexts and corresponding traces, this reduces a brute force attack on the key to 2^{68} steps [Lau05, Sec. 3.2]. One non-trivial countermeasure considers randomizing the addresses by using instead a number of permuted tables that remain local and secret, decreasing the amount of information an attacker can infer from the traces [Lau05, Sec. 4].

Cache-timing attack mitigations for AES. The previously discussed cache-timing attacks by Bernstein [Ber04] and Osvik et al. [OST05] demonstrate that cache-timing attacks pose a practical threat to software implementations of AES. Brickell, Graunke, Neve, and Seifert recognize this threat and propose, implement, and evaluate a number of carefully devised countermeasures [BGNS06]. A brief summary of three methods follows.

1. As the size of a table decreases, it spans fewer lines in the cache and reduces the amount of information an attacker can infer from a cache hit or miss. In an extreme case, if a table fits into exactly one cache line, then (in theory) an attacker cannot infer any information whatsoever. The authors propose to use the original, smaller 256-byte AES S-box, and furthermore prefetch all of the lines in each round; for the common case of 64-byte cache lines, this implies four extra memory references per round. This is generally effective against time-driven and access-driven attacks with limited granularity. Unfortunately this approach dictates that the linear layer be explicitly computed. SIMD through Intel's SSE2 aids in keeping performance competitive.

2. The simplest cache-timing attacks against AES use relations derived from either the first round or the last round: the cipher's diffusion properties make it difficult to derive exploitable relations involving variables from multiple rounds. The authors propose to apply the previous countermeasure during only these critical rounds, yet for all rounds between use the single large compressed T table and unaligned loads. They suggest this compromise is effective when attackers can only obtain cache-timing measurements between the contiguous execution of a number of AES rounds, i.e., access-driven attacks with only limited granularity.

3. Lastly, the authors consider access-driven attacks with high granularity, i.e., capable of observing the cache a number of times per AES round. The authors again suggest the smaller 256-byte S-box and prefetching all of the lines in each round, but combining this with permuting the S-box with a dynamic permutation changed frequently. Intel's SSE2 aids this time in executing said permutation.

When measuring the performance impact of these countermeasures, the authors state that the execution time is between 1.35 and 2.85 of the stock AES code, depending on what countermeasures are employed [BGNS06, Sec. 5.1]. Furthermore, they state that any of the proposed countermeasures are effective against Bernstein's attack [Ber04]: the average execution times follow a normal distribution [BGNS06, Sec. 5.2].

An access-driven attack on the last AES round. The first nine rounds of AES are all identical in the sense that, as previously discussed, in software each round contains sixteen table lookups, four into each T_j for $0 \le j < 4$. The last round omits the `MixColumns` step and, as such, is exceptional. One way to implement this is using a table T_4 that holds the `SubBytes` output as 4-byte words. The last round is then implemented as sixteen lookups into T_4 combined with some masking or shifting. Neve and Seifert design a cache-timing attack against such implementations [NS06]. To implement a trace-driven cache-timing attack on AES, one challenging practical issue is synchronizing with the cipher: for example, determining where one round ends and the next begins in a cache-timing trace. However, with the above implementation the last round only makes memory accesses from T_4 while all other rounds do not access T_4 at all: these accesses in the final round should be easy to distinguish from the other accesses. The authors devise two attacks building on the two search strategies of Tsunoo et al. [TSS$^+$03]. Both methods seek to recover the last round key and then derive the original key by easily inverting the key schedule.

1. The non-elimination method first sorts all ciphertexts according to the value of a given byte [NS06, Sec. 7.1]. All bytes that share the same value all made the same memory access in the last round: each ciphertext byte is a function of a single state byte and a round key byte in the last round. Hence all such bytes that have the same value must reflect a cache miss for the corresponding cache set in the trace, i.e., there will be one cache set amongst sixteen reflecting all cache misses while access to other sets is fairly random; the au-

thors calculate the average number of sets accessed in the last round as 10.3 [NS06, Sec. 5], i.e., with $t = 1$ ciphertexts there are 10.3 candidates for the correct cache line. Proceeding with $t = 2$ eliminates many of these candidates, i.e., any lines that went unaccessed for this new ciphertext. With overwhelming probability, all lines will be accessed by $t = 7$ [NS06, Tbl. 1] and the correct cache line can then be identified. This identifies the upper four bits of all state bytes in the last round. Recover the lower four bits of each byte as follows. Find bytes a and b, the top four bits fixed by the known state from the previous step, such that the difference $T_4[a] \oplus T_4[b]$ agrees with the corresponding difference in their ciphertexts: the relation derived by summing the two ciphertexts. Most likely a and b will be unique, otherwise this process can be repeated. This recovers the entire state at the beginning of the last round. Express each byte of the last round key as a function of a single byte of said state and a single byte of ciphertext. Finally, solve for all bytes of the last round key. This method requires roughly 186 queries [NS06, Sec. 7.1].

2. The elimination method is more efficient and focuses instead on unreferenced cache sets; the average number of these per trace is 5.7 [NS06, Sec. 7.2]. This method keeps a set of candidate bytes for each round key byte. An unreferenced set implies the corresponding upper four bits of state are not possible for any state byte. Use the corresponding ciphertext to compute the resulting impossible key bytes. This eliminates up to sixteen key byte candidates from each key byte, or 256 candidates total. The attack proceeds iteratively through the traces in this fashion, trimming the candidate sets. Naturally as more traces are processed less trims are made as collisions start occurring, i.e., eliminating bytes that have already been eliminated, but the authors show that roughly 20 queries suffices to recover the key using this method [NS06, Sec. 7.2].

The authors consider three different levels of granularity in cache-timing traces [NS06, Sec. 6]: low resolution that allows only one measurement for the encryption of multiple blocks, one line resolution that allows only one measurement for the encryption of a single block, and high resolution that allows multiple measurements throughout the encryption of a single block. The authors focus mostly on one line resolution that is fairly easy to realize, for example with the Prime+Probe strategy of Osvik et al. [OST05, Sec. 3.5]. The authors describe implementing the attack by exploiting the operating system scheduler [NS06, Sec. 3]. The spy slows down the victim process by using up nearly all of its allotted time then yielding the remainder, allowing the victim process to execute for only a very short period: this improves the resolution of the spy's measurements. This is an important result because it shows that SMT such as Intel's HTT is not a prerequisite for carrying out practical access-driven cache-timing attacks.

Improved trace-driven attacks on AES. In the previously discussed trace-driven attack on AES by Lauradoux, he mentions the possibility of using more of the cache trace to reduce the 2^{80} search space but defers the analysis to future work [Lau05, Sec. 3.1]. The cache-timing attack on AES by Neve and Seifert is access-driven and not trace-driven, i.e., the side-channel provides information on specific sets that are accessed [NS06]. Acıiçmez and Koç build on both of these results by, in the former case, extending the attack to relations across the first two AES rounds and, in the latter case, developing a trace-driven attack on the last AES round [AcKK06a, AcKK06b]. A summary of these two topics follows.

1. The authors begin by reformulating the known attack in terms of hypothesis key bytes, then performing a search trimmed by the relations derived from the cache trace. Although not stated as such, the alluded algorithm is essentially a backtracking search algorithm where the search space is all possible 16-tuples of key bytes. For example, consider the four accesses in to T_0 in the first round and a trace of $MHMM$. The first miss provides no information due to the empty cache. The second hit implies the previously discussed relation $p_0 \oplus p_1 = k_0 \oplus k_1$ on the four top bits. Tuples that do not satisfy this relation get trimmed, and due to the backtracking as soon as possible, i.e., the second level of the search. The second miss implies $p_0 \oplus p_2 \neq k_0 \oplus k_2$ on the four top bits: this trims the search space at the third level. This in and of itself is not novel. The novelty is the manner in which the authors further trim the search space using trace data from the second round. This is done by expressing the bytes of the second round key as a function of bytes in the first round key, i.e., using how the AES key schedule evolves from the first round to the second. The authors also point out that such a strategy cannot extend beyond the first two round since AES achieves complete diffusion at that point: after the first two rounds every state bit depends on every key bit [AcKK06a, Sec. 3.4]. When simulating the attack their experiment results state that this reduces the number of steps in analysis to roughly 2^{48} with fifteen traces or roughly 2^{33} using 40 traces [AcKK06a, Sec. 5].

2. The trace-driven attack on the last AES round follows a similar strategy as that of Neve and Seifert in that it starts with a set of candidate bytes for each key byte and trims these sets based on the side-channel data [NS06]. Yet how these sets are trimmed differs substantially because of the differences in the two side-channel models. The intuition is as follows. Take a trace in the last round that starts with MH. The first miss provides no information due to the empty cache: no entry from T_4 is accessed until the last round. Call the inputs to the first and second lookups a and b, respectively. The cache hit implies that the top four bits of a and b are equal. The ciphertext bytes are $c_m = S[a] \oplus k_i$ and $c_n = S[b] \oplus k_j$ for some m, n, i, j where k_i and k_j are bytes of the last round key. Or rearranged, $a = S^{-1}[c_m \oplus k_i]$ and $b = S^{-1}[c_n \oplus k_j]$. Find k_i and k_j such that the top four bits of $S^{-1}[c_m \oplus k_i]$ and $S^{-1}[c_n \oplus k_j]$ are equal; the search space is 16 bits. This recovers two bytes of the last round key. To implement this, the authors use a search strategy similar to that on the first two rounds: it can be thought of as an exhaustive search on key bytes yet trimming the search using equalities and inequalities derived from the trace. When simulating the attack their experiment results state that this attack requires roughly 2^{35} steps using ten traces [AcKK06a, Sec. 5].

An improved time-driven attack on AES. In the time-driven attack on DES by Tsunoo et al., the elimination method focuses on encryptions that experienced high latency, i.e., those more likely to have incurred cache misses and concentrates on an implied inequality between parts of the first and last round states [TSS+03]. Bonneau and Mionov derive a related attack on AES that instead focuses on encryptions that experienced low latency, i.e., those more likely to have incurred cache hits [BM06]. The inferred equations are similar to, but obtained independently from, those previously discussed by Neve and Seifert [NS06] and Acıiçmez and Koç [AcKK06a]. The authors first outline an attack exploiting cache hits in the first AES round [BM06, Sec. 5]. What follows is a short summary of their final round attack [BM06, Sec. 6]. At the beginning of the last round, consider two state bytes x_i and x_j such that $x_i = x_j$: the state bytes collide. Then $T_4[x_i] = T_4[x_j]$ holds and this induces a cache hit. This furthermore implies $c_m \oplus c_n = k_m \oplus k_n$ for some fixed m and n: that is, a cache hit should result in a low latency timing, and that particular key differential will hold (or from the attacker perspective, is more likely to hold given only the low latency as evidence). Based on this observation, the attacker triggers the device to perform an encryption and measures the execution time. The attacker then builds a table $t[i, j, \Delta]$ that holds the average execution time for the differential Δ: that is, with ciphertext c for each combination of ciphertext bytes c_i and c_j increments a counter and accumulates the time at $t[i,j,c_i \oplus c_j]$. After iterating this process sufficiently, one particular differential for each i and j combination will have a lower average execution time: the attacker will be able to identify a distinct Δ' such that $t[i,j,\Delta']$ reflects a minimal average execution time. The attacker then uses these differentials to build a system of linear equations and solve for the last round key. This method can be extended by considering cache line collisions instead of full state byte collisions: i.e, collisions on only the upper portions of the state bytes [BM06, Sec. 7]. This is done by expanding the table t to include 2-tuples of key bytes instead of only differentials. That is, with ciphertext c for each combination of ciphertext bytes c_i and c_j the attacker increments a counter and accumulates the time at $t[i,j,k_i,k_j]$ whenever the top bits of $S^{-1}[c_i \oplus k_i]$ are equal to the top bits of $S^{-1}[c_j \oplus k_j]$, i.e., induced a cache hit in the last round. This requires more memory and time in the analysis phase, but less measurements. The authors implement the attack and, for example, carrying it out against AES software running on a Pentium III requires as few as 2^{13} ciphertexts and measurements. It is worth mentioning that a single process is used to collect the timings and perform the encryption, i.e., a single program that measures the latency of a function call. In closing, the authors state that the simple countermeasure of eliminating the T_4 table dedicated to the last round, replacing it with lookups into the other existing T tables and different masking techniques, has no measurable effect on the cipher performance and deserves consideration [BM06, Sec. 9].

Cache-timing attacks and constraint satisfaction. The trace-driven AES final round attack by Acıiçmez and Koç assumes that encryption begins with an empty cache [AcKK06a]. Or more accurately, the authors adopt Page's model for trace-driven attacks where the assumption holds [Pag03]. Bonneau builds on this result by developing a related trace-driven attack on the last AES round that, in contrast, does not necessarily assume an empty cache [Bon06]. Somewhat similar to the elimination method in the attack by Tsunoo et al. [TSS+03], the attack uses implied inequalities derived from cache misses in the trace. A brief summary follows. Assume two ciphertext bytes $c_0 = T_4[x_m] \oplus k_0$ and $c_1 = T_4[x_n] \oplus k_1$ for some fixed m and n and a cache trace starting with MM. The first cache miss conveys no relevant information; consider the second cache miss. This implies that x_m and x_n are not on the same cache line, thus do not have the same values in the upper bits. In contrast, nothing can be immediately inferred from a cache hit since this work assumes the previous cache state is unknown. The attacker guesses byte k_0 and computes $x_m = S^{-1}[c_0 \oplus k_0]$. Depending on the number of values that a cache line fits, this eliminates a large number of candidates for x_n: all those that share the same upper bits with x_m. This eliminates a corresponding number of candidate k_1 values using the equation for c_1, trimming the search space significantly. The author extracts all such constraints from the side-channel and cleverly views the problem as a one of constraint satisfaction [Bon06, Sec. 5]. As such, he is able to apply existing constraint propagation methods to carry out the search efficiently. This is an elegant solution to the problem because, as opposed to much of the previous work, it leans on existing techniques instead of constructing a dedicated, one-off algorithm. In that respect, the method can be adapted and applied more generally to handle searches that should take into account constraints imposed by side-channel data. The author simulates the attack for different cache line sizes [Bon06, Sec. 6]. The results confirm those of Acıiçmez and Koç assuming an empty cache requiring ten traces [AcKK06a, Sec. 5], but interestingly when relaxing this assumption the average number of measurements needed is only nineteen [Bon06, Tbl. 1].

An extended time-driven attack on AES. One novelty in the previously discussed trace-driven attack on AES by Acıiçmez and Koç is how the authors extend a first round attack to an attack over the first two rounds by considering the trace data in the second round in combination with the operation of the AES key schedule [AcKK06a]. The work is, however, theoretical as the attack only simulates the side-channel. In contrast, consider the time-driven attacks on AES by Bonneau and Mironov that remotely exploit the same features as a trace-driven attack, i.e., whether table lookups in single round collide and map to the same cache line or not [BM06]. The authors implement their last round attack and demonstrate that it is feasible, at least when timing noise is at a minimum. On the other hand, the first round attack the authors propose leaves an impractical search space for the remaining key bits. Leaning on both of these results, Acıiçmez, Schindler, and Koç implement a time-driven attack on the first two AES rounds [AScKK07]. In a chosen plaintext attack scenario, the authors are able to improve the attack complexity by fixing a number of plaintext bytes, allowing individual key bytes to be attacked instead of 4-tuples of key bytes [AScKK07, Sec. 3.4]. They implement their attack in two different scenarios [AScKK07, Sec. 4]. They term the first an innerprocess attack: the scenario is analogous to that of Bonneau and Mironov where a single process is measuring the latency of an encryption directly through a function call [BM06]. This minimizes noise. They term the second an interprocess attack: it uses a custom TCP client that takes timing measurements and server that encrypts data. In their experiments, the client and server run on the same physical machine so there is no transmission delay, only that introduced from the network stack. This is a much weaker attacker than the former and closer to a realistic remote attack. In the latter case, the authors state that 106 million queries to the server are enough to make the attack succeed and recover the entire 128-bit AES key: the attack is carried out against OpenSSL 0.9.7*e* and Linux running on an Intel Xeon with HTT [AScKK07, Sec. 4].

Process scheduling and cache-timing attacks. For access-driven attacks, implementing such a spy process that exploits the OS scheduler can indeed be a tedious task: as such Neve and Seifert omit the explicit details of its construction [NS06, Sec. 3]. Tsafrir, Etsion, and Feitelson present a "cheat" attack that can inadvertently be used to run local trace-driven cache-timing attacks [TEF07]. They place the attack in a much more general context: a malicious user program that is able to control the execution of another user program in such a way that the latter consumes an arbitrary percentage of real CPU time, yet this is undetectable through typical administrative auditing (such as `ps`, `top`, etc.). An OS scheduler divides CPU time into evenly-spaced intervals called ticks. At the end of a tick, the scheduler "bills" the running process, i.e., increments a counter, determines whether said process has exceeded its alloted time or quantum, and either continues running the process or performs a context switch and runs another waiting process. The intuition of the "cheat" attack is to have the malicious program start running immediately following a tick, yet stop execution before the next tick and the scheduler allocates the remaining time by resuming execution of a waiting process. Upon the next tick, said resumed process is billed for the tick; the malicious program is no longer running. To implement this, the authors use the CPU cycle counter `rdtsc` to determine the approximate number of real CPU cycles in the duration of a tick. Hence the malicious program can execute for any number of CPU cycles less than said duration, continuously checking `rdtsc` during execution to ensure that it does not exceed the tick duration. The malicious program synchronizes with scheduler ticks by performing a dummy sleep request, causing it to wake exactly upon the next tick. One can imagine a number of security threats this poses. For example: a user is able to bypass administrator policies limiting CPU time; avoid monetary liability for CPU usage; perform a denial-of-service attack by consuming system resources. The authors outline a number of other malicious uses [TEF07, Sec. 2]. While not explicitly stated as a use case, the "cheat" attack can be used to carry out access-driven cache-timing attacks: the malicious program is able to monitor the memory references of the program that is resumed immediately following it. In this manner, the malicious program slows down the execution of the victim program since it only receives a very small window of the tick to execute during. The authors include the complete source code for their malicious program [TEF07, Sec. 3.1] as well as extensive experiment results [TEF07, Sec. 3.2].

Instruction cache-timing attacks. All of the previously discussed cache-timing attacks exploit the data cache, i.e., the notion that the latency of retrieving values from memory depends on the availability of said data in the data cache. Modern processors also feature an instruction cache (icache) that provides similar functionality for program code. That is, to decrease the latency of fetching instructions from main memory to be executed, once fetched said instructions are cached to decrease the latency of subsequent requests. Acıiçmez demonstrates that this microarchitecture feature can indeed be used as a side-channel and poses a security risk [Acı07a, Acı07b]: a brief summary follows. Consider a spy process that wishes to determine whether a victim process executes or does not execute a certain code segment. The spy executes a series of dummy instructions that map precisely to the same region of the icache as the target code segment of the victim process. In this manner, the spy fills the icache with its own code. Now assume the victim process is allowed to briefly execute. Similarly to data cache attacks, this could be realized through context switching and exploiting the operating system scheduler or SMT such as Intel's HTT. Now the spy process is again allowed to execute. It measures the latency of re-executing its dummy instructions. High (low) latency execution implies that the victim process executed (did not execute) a code segment that maps to the target region of the icache. In a proof-of-concept realization, Acıiçmez describes a spy process that targets the multi-precision multiplication function in OpenSSL 0.9.8*d* used during an RSA encryption via modular exponentiation [Acı07a, Sec. 5]. Hence the spy process executes dummy instructions that map to the same regions as said function and measures the total execution time: said regions can be determined by disassembling the executable. Using this side-channel to determine the sequence of modular squarings and multiplications, the author states that this allows recovery of 200 bits out of each 512-bit exponent [Acı07a, Sec. 6]. Although this is not carried over to a complete attack that factors the modulus and recovers the private key, it does indeed demonstrate a serious threat. This work is important because it demonstrates the dangers of state-dependent code execution and calls for future work in this area, including the reevaluation of security-critical software and methods to mitigate this new breed of microarchitecture attack.

Remark. This new side-channel attack breed of icache attacks raises questions about the most effective way to construct a spy process, i.e., realize the side-channel. One way to accomplish this is to build an icache analogue of Percival's previously discussed data cache spy process [Per05]. This is one contribution of Publication II. The generic icache spy process it proposes lays out a segment of program code the same size as the icache. It steps through this code cache-set wise with unconditional jumps: each code block completely fills an entire cache line and, in each iteration, it jumps through numerous blocks of code that map to a distinct cache set, enough to completely pollute said cache set. It measures the time required to do so, then proceeds to the next cache set, then repeats the entire process indefinitely. Abstractly, like Percival's data cache spy process jumps around in memory, the icache spy process proposed in Publication II jumps around in code. With the side-channel realized, another concern is how to most effectively process the resulting side-channel data. The work in Publication II proposes applying the framework in Publication I, originally proposed for data cache-timing measurements, to icache timing measurements. This turns out to be quite effective, demonstrated in Publication II by mounting an icache-timing attack on OpenSSL's implementation of DSA. Carried out on an Intel Atom processor, the attack in Publication II uses the framework in Publication I to identify the sequence of modular multiplications and squarings in the sliding window modular exponentiation algorithm used in OpenSSL's implementation of DSA. Figure 4.3 illustrates the effectiveness of this approach. This yields a significant amount of key material for the secret nonce, which Publication II leverages in a lattice attack to recover the long term DSA key. This requires only a few thousand signatures along with the corresponding side-channel data, and a few hours of offline computation. The work in Publication II also considers general attack mitigations at many different levels, such as the operating system and compiler level.

Further cache-timing attack mitigations for AES. From the work of Brickell et al., as discussed one interesting countermeasure to AES access-driven cache attacks they propose is the use of a random secret permutation to compute the nonlinear layer [BGNS06]. Blömer and Krummel analyze said countermeasure and propose additional strategies [BK07a, BK07b]. They show that if the secret permutation is not changed often enough, the countermeasure is ineffective. Specifically, using a random permutation to implement the last AES round and assuming 64B cache lines, they calculate that roughly 2300 measurements is usually enough to recover the complete round key for the last AES round [BK07a, Sec. 7.1]. Seeking to reach the theoretical minimum amount of pos-

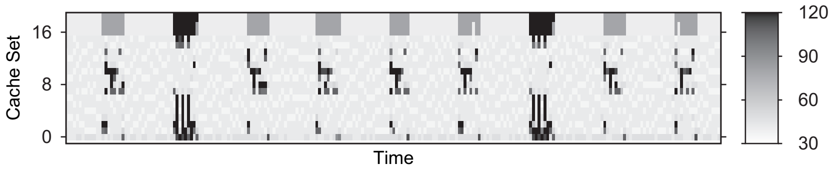


Figure 4.3. Live I-cache timing data produced by the spy process in Publication II running in parallel with an OpenSSL DSA sign operation; roughly 250 timing vectors (in CPU cycles), and time moves left-to-right. The bottom 16 rows are the timing vector components on 16 out of 64 possible cache sets. The top four are meta-data, of which the bottom two are the VQ classification and the top two the HMM state guess given the VQ output. Seven squarings are depicted in dark gray and two multiplications in black. Any timing data cell with a higher value, i.e., darker shade, suggests a cache miss. Any cell with a lower value, i.e., lighter shade, suggests a cache hit.

sible key leakage with the permutation approach, they continue by defining a subset of random permutations called distinguished permutations [BK07a, Sec. 7.2]. They give a method to construct said permutations that are susceptible to a cache-timing attack that reveals the minimal 64 out of 128 of the last round key bits. In contrast, the implementation of distinguished permutations does not require any updating of the permutation. Another interesting countermeasure they propose splits the small 256B AES S-box S into smaller S-boxes S_i where each S_i fits entirely within a single cache line [BK07a, Sec. 6]. A brief description follows. Assume 64B cache lines. Construct four tables S_i containing 64B such that $S_i[x]$ yields two bits of the result. Four lookups, one into each table, yield a total of eight bits that are then concatenated together to arrive at the desired result. The authors report that implementing this approach in the last round only increases the running time of the cipher by a factor of 1.6 on a Pentium M. In practice, this requires a fair amount of shifting, masking, and bitwise operations. Although not stated in the work, one can instead accomplish the same goal more logically with a 4-to-1 "software multiplexer" using the pseudocode shown in Fig. 4.4.

An instruction cache attack on RSA. As previously discussed, Acıiçmez demonstrates the feasibility of extracting the sequence of modular squarings and modular multiplications within the modular exponentiation routine executed during an RSA decryption using the icache as a side-channel [Acı07a]. With the sliding window method and OpenSSL default settings, this reveals a significant amount of the (secret) exponent inputs into each of two modular exponentiations, yet the attack stops there after identifying the vulnerability and does not extend this to a key recovery attack. Acıiçmez and Schindler combine their results to describe an icache attack against a well-protected RSA implementation that can lead to full key recovery [AS08]. The extra reduction step that

```
xl = x & 0x3F           // mask off top two bits
v0 = S[       xl]       // pull out four possible values
v1 = S[0x40 | xl]
v2 = S[0x80 | xl]
v3 = S[0xC0 | xl]
i0 = sar(x << 1,7)      // mask for LO control wire
i1 = sar(x     ,7)      // mask for HI control wire
t0 = ~i0                // mux it all together
t1 = ~i1
t2 = v0 & t1 & t0       // 0 0
t3 = v1 & t1 & i0       // 0 1
t4 = v2 & i1 & t0       // 1 0
t5 = v3 & i1 & i0       // 1 1
t6 = t2 | t3 | t4 | t5  // result: S[x]
```

Figure 4.4. Pseudocode for an alternate implementation of an 8 to 8-bit lookup using a 4-to-1 multiplexer to resist cache-timing attacks.

is sometimes required, as previously discussed in the attack of Brumley and Boneh [BB03], is realized in a straightforward software implementation with a logical branch that calls a multi-precision subtraction function. Acıiçmez and Schindler describe an icache spy process that targets the multi-precision subtraction function [AS08, Sec. 3.1]. The spy executes dummy instructions that map precisely to the same icache sets as said subtraction function. By measuring the time required to re-execute these dummy instructions, the attacker is able to infer whether the multi-precision subtraction function executes or not during the Montgomery reduction step. The described attack can be successful even in the presence of RSA blinding and modular exponentiation functions with fixed windows, two strong side-channel countermeasures. This work is interesting because it takes an existing theoretical attack and suggests realizing it using the icache side-channel, yet still rather artificial since the spy process is not implemented as an independent process but integrated into OpenSSL's modular exponentiation routine. In this respect, the attack is more proof-of-concept.

An access-driven attack on a stream cipher. The eSTREAM project is a public competition for stream cipher designs than ran from 2004 to 2008 [RB08]. One of the phase 3 candidates for software designs is Wu's HC-256 that takes a 256-bit key and 256-bit initialization vector [Wu04]. For this brief discussion, the most relevant feature of HC-256 is that the internal state consists of two large tables P and Q each with 1024 entries of 32-bit words. The contents of these tables remains secret. The cipher performs lookups into these tables to produce keystream words. Zenner devises a theoretical cache-timing attack on HC-256 [Zen08]. The attacker is granted access to two oracles: one that provides keystream word i and the other that provides the unordered list of cache line accesses made during the creation of keystream word i [Zen08, Sec. 3.2]. In HC-256, many of the per-round lookups into P and Q have public indices based on the round number and hence do not reveal any secret state. However, five lookups per-round are state dependent. Assuming 64B cache lines, P and Q span 64 lines each in the cache and an attacker potentially learns the top six bits of every lookup index. Four of these lookups are into distinct regions of either P or Q where the low eight of the ten index bits are secret state-dependent: this potentially leaks the top four bits of said state. The fifth lookup, however, slightly complicates matters in Zenner's model where the order of lookups is not given. It potentially leaks six bits of internal state, but into the same region as one of the four previous lookups. Zenner first gives an analytical method to determine the order of said lookups and hence directly map the implied state bits to their corresponding lookup in the cipher description [Zen08, Sec. 4]. He continues by describing a method to significantly trim the resulting search space, and furthermore a backtracking algorithm to recover the full state. The attack is theoretical in that it assumes the error-free side-channel is given and the attack complexity is 2^{67} thus no implementation exists, and also assumes known keystream. Two interesting aspects of Zenner's work are as follows. In contrast to attacks on AES where the actions of the round function implemented by the T tables are public, the tables P and Q in HC-256 are private, but in the end can be recovered using the side-channel data. Lastly, the attack models that Zenner defines abstract away the concrete implementation of the side-channel, thus allowing cache-timing analysis of stream ciphers to focus solely on the cryptanalytic aspects [Zen08, Sec. 6].

Further access-driven attacks on stream ciphers. One design pattern for software-oriented stream ciphers is having a word-based Linear Feedback Shift Register (LFSR), for example over $\mathbb{F}_{2^{32}}$, with state consisting of a number of state words. As the cipher clocks, this LFSR updates its state in a linear fashion. Stream ciphers combine this with a small number of state words that evolve in a nonlinear fashion, for example using S-boxes, shifts, rotations, XORs, and additions modulo 2^{32}. This is combined in some way to produce keystream words. A number of stream ciphers fall under this description, including SNOW, SNOW 2.0, SOSEMANUK, and SOBER-128. Leander, Zenner, and Hawkes give a framework for cache-timing attacks against such stream ciphers [LZH09]. Building on Zenner's previous work, the authors assume the same side-channel model: namely, the attacker is given access to a keystream oracle and a side-channel oracle. The basic idea of the attack is in fact quite simple and focuses on the linear process. Consider an LFSR consisting of i bits with a feedback function that leaks j bits per clock. The process is $\mathbb{F}_2$-linear so after i/j clocks the attacker simply solves the resulting system of linear equations to recover the complete LFSR state. The attacker uses basic guess-and-determine techniques to recover any remaining state that evolves nonlinearly: this is done in a straightforward way by guessing register values, clocking the cipher and comparing the resulting keystream words with the known keystream words. For a concrete example, consider the stream cipher SNOW 2.0 by Ekdahl and Johansson [EJ02]. The LFSR consists of 16 words in $\mathbb{F}_{2^{32}}$ and the cipher has an FSM consisting of two 32-bit registers that evolve nonlinearly. To implement the LFSR feedback function efficiently, two distinct tables with 256 entries of 4-byte words are used. Assuming 64B cache lines, lookups potentially leak the top four bits of each index, or eight bits per clock. With 512 bits of LFSR state, after $512/8 = 64$ clocks the attacker can recover the LFSR state. The attacker recovers the two 32-bit registers in the FSM by guessing the value of one register, thus fixing the value of the other register in either the previous or next clock, then clocking the cipher repeatedly and comparing the output to a few known keystream words: this takes at most 2^{32} steps. This work is interesting because under a single framework the authors are able to devise theoretical cache-timing attacks on an impressive number of stream ciphers.

Remark. The attack scenario that Leander et al. consider assumes a side-channel that leaks part of the linear state of the cipher. This is reasonable since it allows applying their attacks to many different ciphers. However, many stream ciphers additionally employ some non-linear state that, similar to a block cipher, is most efficient to implement with a table lookup. An example of such a cipher is SNOW 2.0 and its variant SNOW 3G for mobile networks. Specifically, SNOW 3G maintains three 32-bit registers that pass values between each other through AES-like S-boxes. Building upon the work of Leander et al., Publication III presents an access-driven cache-timing attack on SNOW 3G that uses the leakage from within this non-linear state and, optionally, combines it with that of the linear state. This reduces the required number of observed clock cycles, as well as the attack complexity (to roughly 2^{16}). The algorithm in Publication III is essentially a backtracking algorithm that runs a depth-first search on the complete cipher state space, then trims this space as soon as the search reaches an invalid state as evidenced by the information available through the side-channel. Apart from this theoretical attack, Publication III also includes a concrete implementation of the attack, mounted against the SNOW 3G reference implementation, running on an AMD Athlon 64 3200+ Venice with a 64KB

2-way associative L1 data cache and 64-byte cache lines, i.e., 512 cache sets. Experiment results show that the average number of attack iterations to succeed in this environment is only two and, due to the extremely low complexity of the theoretical attack, the implemented attack requires only a few seconds to run. As a countermeasure, Publication III presents a bit-sliced implementation of the SNOW 3G cipher that runs 128 parallel instances and, in situations where batch keystream generation is relevant, suggests that such a constant-time implementation can be realized with no performance penalty when compared to high-speed serial versions employing table lookups. It is worth noting that the circuit-level design of the second S-box that Publication III presents is the most compact publicly-available design, requiring 498 logic gates. What makes the attack in Publication III particularly efficient is the presence of consecutive S-boxes where values move between registers by only passing through an S-box without any additional inputs: this provides the attacker with information on both the input and output of the S-box. This observation yields insight into cryptographic primitive design that is more resistant to side-channel attacks.

Process scheduling and access-driven attacks on AES. On of the interesting features of the previously discussed access-driven cache-timing attack on AES by Neve and Seifert is their description of a spy process that exploits the operating system scheduler policy to obtain per-round granular measurements AES [NS06]. The cheat attack by Tsafrir et al. also exploits the operating system scheduler policy and the work goes into great detail on implementing a malicious process to do so [TEF07]. Building on both of these results, Bangerter, Gullasch, and Krenn present an access-driven cache-timing attack on AES that is able to obtain per-lookup granular measurements [BGK10, BGK11, GBK11]. The spy process does this by exploiting the policy of the Completely Fair Scheduler (CFS) present in Linux from kernel version 2.6.23. The spy does this by creating a large number of identical spy threads [BGK10, Sec. 4]. When activated, a single spy thread takes the cache measurements then sets a timer to wake up the next spy thread soon after the current thread's alloted time expires. This allows the victim to execute in an extremely small quantum, at which time the next spy thread is activated and this process repeats. The victim process makes very slow progress: the scheduler knows this and, according to its policy, compensates by lowering the priority of the particular spy thread that executed immediately preceding the victim. However, with many spy threads, only one of which makes progress preceding the victim, the remaining spy threads have even higher priority than the victim. This is the key observation that makes this particular spy process so effective. The authors use this to mount an access-driven cache-timing attack on AES [BGK10, Sec. 3]. Surprisingly, the attack target is the AES implementation in OpenSSL 0.9.8*n* that uses compressed tables, a frequently suggested countermeasure to such attacks. With the stated granularity and assuming 64B cache lines, this leaks five bits of each byte used in the table lookup per round: that is, 80 out of 128 state bits. Using equations derived between two contiguous rounds this reveals a large amount of key material. The authors are able to synchronize with the cipher rounds by considering all sixteen possible offsets and finding the particular offset where all the inferred key material agrees. Finally, the authors recover the original key using this inferred round key material through a clever heuristic search that is error-tolerant and considers the evolution of the AES key schedule. This essentially searches for the full key that best explains all of the inferred round key material, starting with the most probable of said material. This work is significant because it demonstrates that an attacker can indeed obtain extremely high resolution measurements with respect to the victim's memory accesses: restricting to per-round or per-iteration countermeasures to access-driven cache-timing attacks does not suffice.

5. Cryptography Engineering

As the previous chapter emphasizes, side-channel attacks are a serious threat to cryptosystem implementations. The majority of the discussed countermeasures attempt to strike a balance between performance and security. Indeed, realizing constant time implementations usually comes at a hefty cost. To carry out parallel computations in software, one often needs to straight-line code, removing logic branches and any control flow that is state-dependent. While the goal of such parallel computation is to improve efficiency, oddly enough a common side effect is an implementation that runs in constant time. This can greatly improve the side-channel security of software implementations. This chapter discusses a number of results along those lines: in these approaches, performance is top priority yet the applied methods have some inherent features that improve resistance to side-channel attacks.

Fast and secure serial AES in software. Many of the previously discussed cache-timing attack countermeasures for AES come at either a significant cost, or do not provide sufficient guarantees in all attack scenarios. To achieve constant time software implementations that are still competitive with fast table-based implementations, one might consider architecture-specific features: for example, Single Instruction Multiple Data (SIMD) capabilities available in commodity microprocessors. Such instructions operate on vectors of values instead of a single value. Hamburg leverages vector permute instructions to obtain fast and secure software implementations of AES [Ham09]. The nonlinearity of the AES S-box comes from mapping finite field elements to their multiplicative inverse. An intriguing hardware design technique used to realize a compact AES S-box is to first use a finite field isomorphism $\mathbb{F}_{2^8} \to \mathbb{F}_{2^4}^2$, i.e., representing the larger composite field as a pair of subfield elements. It is then efficient to derive a formula for inversion in $\mathbb{F}_{2^4}^2$ where the only inversion explicitly computed is done in $\mathbb{F}_{2^4}$, i.e., computing a single inverse in the subfield to facilitate computing an inversion in the composite field. Leveraging this hardware technique, Hamburg first derives a novel representation for this finite field to facilitate its computation with vector permute instructions [Ham09, Sec. 2]. The author considers vector permute instructions (i.e., variable byte shuffles) on a number of different platforms, but for concreteness focuses on the AltiVec and Intel's SSSE3. Their features differ slightly: AltiVec's `vperm` allows a 5-to-8 bit 16-way parallel table lookup while SSSE3's `pshufb` restricts to a 4-to-8 bit 16-way parallel lookup. Indeed, these are powerful instructions allowing a constant-time dynamic byte shuffle. To be clear, (the lower bits of) the component values in one register define shuffle indices for values in another register (or many): i.e., the former are the indices into the lookup table and the latter the lookup table itself. Figure 5.1 illustrates the operation of Intel's SSSE3 `pshufb` instruction that dynamically shuffles a 16-byte vector: vector a shuffles the values in b resulting in r. Hamburg adapts classical logarithm-based multiplication for AltiVec [Ham09, Sec. 2]. The author derives a number of formulae for finite field inversion that heavily leverage the dynamic byte shuffler where formula choice will depend on the capabilities of the underlying shuffle instruction [Ham09, Sec. 3]. Outside of this inner nonlinear layer, the linear layers are much simpler to account for since the maps naturally decompose into a sum of multiple maps. The author also considers byte-sliced approaches for parallel modes. Benchmarking the implementations, Hamburg achieves a speed of 10.8 cycles per byte in (serial) CBC mode and 5.4 cycles per byte in (parallel) CTR mode on a Motorola PowerPC G4 7447a, and 10.3 cycles per byte in CBC mode on an Intel Core i7 920 [Ham09, Sec. 7]. These figures represent a dramatic improvement over existing techniques. One of the significant contributions is that the implementation approach applies not only to parallel modes, but serial modes and thus is extremely flexible and applicable.

Fast and secure parallel AES in software. In the previously discussed work, Hamburg takes a byte-sliced approach for implementing parallel modes of AES where, explicitly, a 16-byte vector holds sixteen values for the same byte index in sixteen different instances of the same cipher under the same key. AES features a high degree of parallelism where sixteen S-box substitutions occur in parallel in the nonlinear layer. An alternative approach is to instead gather the individual bits of each byte for each instance in a single register. For example, running eight parallel instances where 128-bit registers are available, byte i of the register r_j holds bit j of state byte i for all eight instances of the cipher, each instance at a fixed offset within said byte. In work occurring concurrently with Hamburg's, this is the bit-sliced approach that Käsper and Schwabe propose

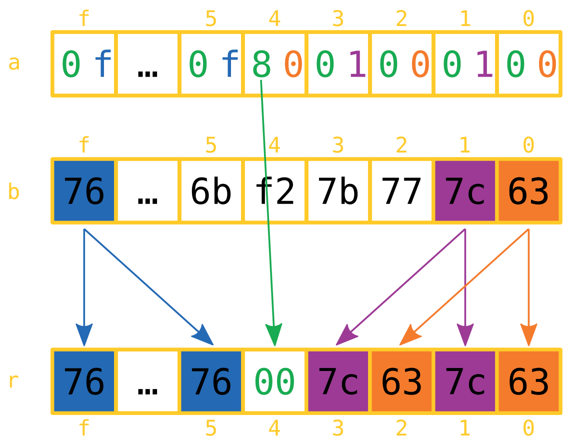


Figure 5.1. Intel's SSSE3 dynamic byte shuffle instruction pshufb. When used as a lookup table or S-box, b defines the lookup table and a the inputs. Put another way, the instruction implements a 16-to-1 multiplexer in 16-way parallel.

[KS09]. This incredibly novel representation for the AES state allows (fairly) straight adoption of compact hardware designs for the AES S-box, where in software the logic gates are realized in parallel with bitwise operations. Considering the nonlinear layer, the exact choice of which bit within the register corresponds to which state byte and/or cipher instance is irrelevant, but of critical importance for the linear layer [KS09, Sec. 4.1]. Said representation implies a conversion to bit-sliced form, but only at the beginning and end of the encryption process. Käsper and Schwabe focus their efforts on the Intel platform and their implementation achieves speeds up to 6.9 cycles per byte on an Intel Core i7 920 [KS09, Sec. 6]. The implementation is constant-time and resists cache-timing attacks, yet remarkably at the same time yields the fastest public software implementation for AES without dedicated AES instructions. Fast table-based implementations of AES are throttled by the sixteen required lookups per round, and with ten AES round this inherently limits their speed to ten cycles per byte: the techniques devised by the authors circumvent this limit [KS09, Sec. 1]. The authors additionally consider AES-GCM mode for authenticated encryption, recently standardize by NIST [KS09, Sec. 5]. Recognizing the significant contribution, this work received a best paper award at its conference.

Remark. An involution block cipher is a cipher where decryption differs from encryption only in the key schedule. This is a useful feature to reduce the size of implementations, since the same machinery can essentially be used for both purposes. Anubis, operating on 128-bit blocks, and Khazad, operating on 64-bit blocks, are two involution ciphers by Barreto and Rijmen [BR01a, BR01b].

These ciphers share many similarities with AES and, in light of the two previously discussed results on efficient and secure AES software implementations, it is reasonable to apply and build on these techniques to realize fast and secure software implementations of both Anubis and Khazad. This is precisely the analysis that Publication VIII carries out. Considering Hamburg's serial AES implementations, the first contribution of Publication VIII is efficient implementation of the nonlinear layer of Anubis and Khazad using a vector byte shuffler. The nonlinear layers of these involution ciphers is identical, only differing in the number of parallel applications of the S-box. In contrast to the algebraically-derived AES S-box, this particular 8 to 8-bit S-box is built using a Substitution Permutation Network (SPN) where at each layer two distinct 4 to 4-bit S-boxes are applied in parallel, followed by a simple linear layer that is only a permutation of bits. The observation in Publication VIII is that this has an elegant implementation using a vector byte shuffler: facilitating a 4 to 8-bit lookup, Intel's SSSE3 `pshufb` allows unrolling the linear layer following the 4-bit S-box, then combining the result of the two parallel 4-bit S-boxes with a bitwise OR. Considering the parallel bit-sliced AES implementations by Käsper and Schwabe, Publication VIII applies an analogous approach to Anubis and Khazad. The logic expressions for the nonlinear layer leads to a bit-sliced implementation that requires fewer instructions (147) compared to that of AES (167 [KS09, Tbl. 2]), but the bit-sliced implementation of the linear layer is slightly heavier, intrinsically due to the fact that the entries in the matrix for the linear map are larger than that of AES. Benchmarking the resulting implementations on an Intel Core 2 Duo E8400, Publication VIII reports speeds of 21.7 cycles per byte for serial modes such as CBC or up to 9.2 cycles per byte for Anubis with parallel CTR (compared to 20.7 cycles per byte for the table-based reference implementation), and 10.3 cycles per byte for Khazad with CTR (compared to 19.8 cycles per byte for the table-based reference implementation). To put these figures in context, Käsper and Schwabe's benchmarked code on this platform performs at 8.0 cycles per byte for AES with CTR. Hence the resulting throughput of Anubis is lower, but Anubis performs twelve rounds compared to AES with ten, suggesting that the per-round performance of bit-sliced Anubis software is more efficient than that of AES. In conclusion, Publication VIII provides evidence that the nonlinear layer design decisions for Anubis and Khazad, although initially intended for hardware optimization, can have a significant positive impact on their software implementation as well.

Fast and secure parallel ECC in software. When implementing an Elliptic Curve Cryptosystem (ECC), one weights numerous issues that drastically affect implementation aspects. For example, to name a few: scalar multiplication method, scalar representation, coordinate system, finite field, curve coefficients. Such choices impact the speed and, ultimately, side-channel security of said implementation. Using the recently proposed Edwards form of elliptic curves, Bernstein presents a fast and secure implementation of batch scalar multiplication [Ber09]. From the implementation perspective, Edwards curves have a number of advantages, a prominent one being a complete addition law. Traditional elliptic curves in Weierstrass form must handle the identity element explicitly as a special case. Edwards curves do not suffer from this drawback. This is a useful feature to resist side-channel attacks. Bernstein's bit-sliced implementation takes a circuit-level design approach, using a width-w register for w parallel 1-bit logic operations. The number of finite field multiplications is a solid metric for the speed of an ECC implementation. Bernstein sets the underlying finite field to $\mathbb{F}_{2^{251}} \cong \mathbb{F}_2[t]/(t^{251} + t^7 + t^4 + t^2 + 1)$. Multiplication in characteristic two finite fields in software is normally more costly compared to large characteristic, although Intel acknowledges the ubiquity of such fields and, on recent processors, features a carryless multiplication instruction to facilitate multiplying polynomials in $\mathbb{F}_2[t]$. To get an efficient ECC implementation, Bernstein gives a thorough analysis of polynomial multiplication in $\mathbb{F}_2[t]$ with a focus on logic minimization [Ber09, Sec. 2]. The author devises a multiplication circuit with 33096 logic gates for multiplying two 251-bit polynomials in $\mathbb{F}_2[t]$; the final reduction step does not require significant logic. Simultaneously, the devised method also attempts to limit the code size and number of loads and stores. While these are not necessarily goals in logic minimization, when realizing a bit-sliced implementation excessive code size can adversely affect performance, and the number of registers is quite restrictive (sixteen `xmm` registers on AMD64) so in practice values must be pushed on the stack, requiring extra CPU cycles. For curve operations, Bernstein uses differential addition and doubling along with Montgomery's ladder for scalar multiplication, realizing the key bit-dependent logic branches with a conditional bitwise swap [Ber09, Sec. 3]. In the end, the implementation requires 44 million Intel Core2 cycles to compute 128 parallel scalar multiplications in constant time. This work yields an extremely fast and secure implementation where batch operations are applicable, yet suffers from the drawback that said curve parameters are not yet explicitly part of any standard.

Remark. In the discussion on polynomial multiplication in $\mathbb{F}_2[t]$, Bernstein calls for a collaborative effort to reduce logic for multiplication of characteristic two polynomials of varying degrees [Ber09, Sec. 2]. Regarding existing cryptography standards, the application is finite field multiplication with respect to a polynomial basis: this is the most common form implemented in the wild. However, the standards often include a second representation for such finite fields called a normal basis. Squaring elements of a characteristic two finite field is a linear operation. With a polynomial basis, this involves simply relabeling the components then reducing the resulting polynomial. With a normal basis, however, squaring is simply a rotation: this is an attractive feature when performing many field squarings. Implementing a finite field with respect to a normal basis is far less common due to implementation aspects. In parallel to Bernstein's work, Publication V explores bit-sliced implementations of batch binary field multiplication with respect to a normal basis. Such methods are constant time and address arguments that normal basis multiplication is inefficient in software simply due to the tedious bitwise manipulations it requires: bit-slicing allows a pseudo-hardware design approach to components where access to individual bits is trivial. The analysis in Publication V begins with a broad survey of existing hardware methods for normal basis multiplication, and continues by implementing said methods over a wide range of field sizes and providing timings of said implementations on a number of different architectures. The results in Publication V show that, when batching applies, significant performance gains are possible with bit-sliced implementations when compared to traditional serial methods for normal basis multiplication. Furthermore, in some cases said implementations can be competitive when compared to efficient serial methods for polynomial basis multiplication. Finally, quoting Hankerson et al. [HMV04, p. 72]:

> Experience has shown that optimal normal bases do not have any significant advantages over polynomial bases for hardware implementation. Moreover, field multiplication in software for normal basis representations is very slow in comparison to multiplication with a polynomial basis.

One important and necessary result in Publication V is that any inefficiency of normal basis multiplication is, in contrast to popular belief, not due to its unsuitability for software implementation, but instead its excessive logic gate requirement compared to polynomial basis methods. The issue is one of algorithm design, not of platform choice.

6. Conclusion

This chapter marks the end of the introductory portion of this dissertation, providing sufficient background and showing how the contributions of the publications that make up this dissertation relate to the existing literature. The remainder of this dissertation consists of the eight aforementioned publications: the main topic is side-channel analysis. The most common side-channel of study here is the cache-timing channel that exploits the behavior of various caching mechanisms present on commodity microprocessors. Some of the publications instead concern more traditional remote timing attacks and fault attacks. Complementary to this, another topic therein is efficient and effective countermeasures to these types of attacks. This requires a deep understanding of the underlying side-channel, often gained as a result of implementing an attack. The leveraged tools are often architecture-specific, side-channel attacks themselves being implementation-specific.

The remainder of this chapter consists of a brief, conclusive analysis of the main results achieved in this dissertation, followed by an outlook on future developments in this field.

Publication I develops a powerful framework for processing access-driven cache-timing data that can be generally useful for any high-dimension side-channel data. The framework creates vector cache-timing data templates that reflect the victim algorithm's cache access behavior. It then uses VQ to match incoming cache-timing data to these existing templates, effectively categorizing the vectors and reducing their dimension. This VQ output is then used as observation input to an HMM that accurately models the control flow of the victim algorithm. The HMM then provides the most likely sequence of algorithm states, leading to partial key material. The work applies the framework in combination with lattice methods to mount a data cache-timing attack on OpenSSL's implementation of ECDSA, leading to complete private key recovery in less than an hour of computation on a desktop PC. Building on this work, Publication II further applies the framework to instruction cache-timing data. This is then used to mount an access-driven instruction cache-timing attack on OpenSSL's implementation of DSA. Similarly, the attack implementation makes use of lattice methods and leads to complete private key recovery in less than an hour of computation on a desktop PC. The success of these two attacks is due in large part to the effectiveness of the developed framework: as such, it is likely that it will be utilized to carry out further attacks, highlighting that long term mitigation strategies are needed to address these vulnerabilities.

The main application of the above framework is to timing data procured by a spy process, required to run locally and pseudo-concurrently with the victim process on the same physical CPU. In contrast, this is not a requirement for traditional remote timing attacks that merely measure the response time of the victim across a network. Remote timing attacks pose a much greater threat in this sense. Publication VI identifies a remote timing attack vulnerability in OpenSSL's implementation of scalar multiplication for elliptic curves over binary fields. Said scalar multiplication method (Montgomery's powering ladder) is widely-regarded as a highly side-channel resistant algorithm due to its regularity: the same sequence of finite field operations are performed in each iteration regardless of the value of any given key bit. In a twist of irony, it is precisely this feature that Publication VI exploits: there is a direct correlation between the time required for the server to produce a signature (i.e., authenticate in the TLS protocol) and the number of bits in the key. Devising and implementing an attack employing lattice methods, this work shows that said vulnerability can be exploited to remotely recover the private key of a TLS server in a matter of minutes. This caused CERT to issue[1] vulnerability note VU#536044 and the OpenSSL development team to integrate the proposed source code patch as a countermeasure, enabled by default as of stable release 1.0.0e[2]. Hence this work directly impacts the way that hundreds of millions of users around the world utilize cryptography. Furthermore, it highlights the fact that side-channel attacks target implementations: an algorithm featuring desirable physical security properties provides no guarantees with respect to the implementation of said algorithm.

In conclusion, the contributions of this dissertation advance the field of applied cryptography by deepening the understanding of side-channel attacks, the main emphasis falling on cache-timing attacks. In turn, this leads to a better understanding and recognition of microarchitecture attacks as a serious

[1]`http://www.kb.cert.org/vuls/id/536044`
[2]`http://www.openssl.org/news/changelog.html`

threat to security-critical software.

The landscape of microprocessors is rapidly changing. The growth of GPUs for general purpose computing and multi-core and SIMD-capable embedded processors is astounding. Chip manufacturers are constantly challenged to push the envelope in terms of computing performance, often at the expense of security. If there is only one lesson to be taken away from this dissertation, it is this: security must be regarded as a first-class concept in microprocessor design, and not a retrospective afterthought. It seems that industry often sends a mixed message with respect to side-channel consideration. For example, the Hyper-Threading feature in Intel's Pentium 4 has repeatedly been exploited to demonstrate data cache-timing attacks. The first such public case is documented soon after its 2003 release. As a result, one might assume that chip designers would hesitate to feature such SMT capabilities. Indeed, no subsequent mainstream microprocessor featured SMT. That is, until 2008 when Intel reintroduced HTT in its Atom line, and continued this trend in 2009 with the Core i7 and later Core i5. It seems Intel intends to propagate this trend. On the other hand, around the same time in 2010 Intel introduced the AES-NI instruction set, providing an on-chip implementation of AES, and the CLMUL instruction set providing carryless multiplication for binary polynomial multiplication. Both sets are strong cache-timing attack countermeasures. The message seems to be that chip manufacturers are concerned with side-channel attacks, but are more concerned with performance: they are willing to deploy countermeasures, but not if they adversely affect chip performance. This quandary is compounded by the fact that the concrete security of an executing cryptosystem ultimately depends on many different layers. There is the hardware level where microarchitecture features such as SMT, instruction caching, data caching, branch prediction, hardware prefetching, and speculative execution can adversely affect security. There is then the operating system level where features such as job scheduling, virtual memory, and virtual machine managers can adversely affect security. Finally, there is then the application software level where features such as variable control flow, memory-resident table lookups, and software bugs can adversely affect security. What level is held accountable for the security of a running program? While the ideal answer is *all of them*, unfortunately the grim reality is that the current answer is usually *none of them*. With practical examples of side-channel attacks spanning at least two decades, what catalyst is needed to induce change in this answer? Quoting Bernstein [Ber04, Sec. 6]:

> This area offers many obvious opportunities for cryptanalytic research . . . I do not

> intend to pursue any of these research directions. As far as I'm concerned, it is already blazingly obvious that the AES time variability is a huge security threat. We need constant-time cryptographic software! Once we've all switched to constant-time software, we won't care about the exact difficulty of attacking variable-time software.

In one respect, Bernstein's 2004 prediction has panned out: Intel's AES-NI instruction set released six years later can be wielded as a blanket response to any suggested cache-timing attack on AES. However, AES is only one cryptosystem and this approach is not sustainable. Bernstein suggests the solution is a radical change in software development habits. What catalyst is needed to induce this change? While this question is rhetorical in nature, this dissertation is at least part of that catalyst: the change can only come through repeatedly demonstrating the dangers of bartering security for performance. This pushes the current state closer to the alluded utopia that is ultimately unreachable.

Bibliography

[Abe06] Masayuki Abe, editor. *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*. Springer, 2006.

[Acı07a] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting i-cache. Cryptology ePrint Archive, Report 2007/164, 2007. `http://eprint.iacr.org/`.

[Acı07b] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting i-cache. In Peng Ning and Vijay Atluri, editors, *CSAW*, pages 11–18. ACM, 2007.

[AcKK06a] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES. Cryptology ePrint Archive, Report 2006/138, 2006. `http://eprint.iacr.org/`.

[AcKK06b] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.

[AcKKS07] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Abe [Abe06], pages 225–242.

[AS08] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In Tal Malkin, editor, *CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 256–273. Springer, 2008.

[AScKK06] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. Cryptology ePrint Archive, Report 2006/288, 2006. `http://eprint.iacr.org/`.

[AScKK07] Onur Acıiçmez, Werner Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In Abe [Abe06], pages 271–286.

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

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BCS08] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 221–240. Springer, 2008.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. `http://cr.yp.to/papers.html#cachetiming`.

[Ber09] Daniel J. Bernstein. Batch binary Edwards. In Halevi [Hal09], pages 317–336.

[BGK10] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache games - bringing access based cache attacks on AES to practice. Cryptology ePrint Archive, Report 2010/594, 2010. `http://eprint.iacr.org/`.

[BGK11] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache games - bringing access based cache attacks on AES to practice. Constructive Side-Channel Analysis and Secure Design – COSADE 2011, 2011.

[BGNS06] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. `http://eprint.iacr.org/`.

[BK07a] Johannes Blömer and Volker Krummel. Analysis of countermeasures against access driven cache attacks on AES. Cryptology ePrint Archive, Report 2007/282, 2007. `http://eprint.iacr.org/`.

[BK07b] Johannes Blömer and Volker Krummel. Analysis of countermeasures against access driven cache attacks on AES. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2007.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[Bon06] Joseph Bonneau. Robust final-round cache-trace attacks against AES. Cryptology ePrint Archive, Report 2006/374, 2006. `http://eprint.iacr.org/`.

[BR01a] Paulo S. L. M. Barreto and Vincent Rijmen. The Anubis block cipher. `http://www.larc.usp.br/~pbarreto/anubis-tweak.zip`, 2001.

[BR01b] Paulo S. L. M. Barreto and Vincent Rijmen. The Khazad legacy-level block cipher. `http://www.larc.usp.br/~pbarreto/khazad-tweak.zip`, 2001.

[BZB$^+$05] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *ITCC (1)*, pages 586–591. IEEE Computer Society, 2005.

[CG09] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.

[CPGR05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime. In *Proc. 14th USENIX Security Symposium*, pages 331–346, August 2005.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[dod85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[EJ02] Patrik Ekdahl and Thomas Johansson. A new version of the stream cipher SNOW. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2002.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505. IEEE Computer Society, 2011.

[GGP07] Philipp Grabher, Johann Großschädl, and Dan Page. Cryptographic side-channels from low-power cache memory. In Steven D. Galbraith, editor, *IMA Int. Conf.*, volume 4887 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2007.

[GNS05] P. J. Green, Richard Noad, and Nigel P. Smart. Further hidden Markov model cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2005.

[Hal09] Shai Halevi, editor. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*. Springer, 2009.

[Ham09] Mike Hamburg. Accelerating AES with vector permute instructions. In Clavier and Gaj [CG09], pages 18–32.

[HGS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[HS08] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. Cryptology ePrint Archive, Report 2008/510, 2008. `http://eprint.iacr.org/`.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Halevi [Hal09], pages 1–17.

[Hu91] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, pages 8–20, 1991.

[Hu92] Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*, pages 52–61, 1992.

[Kem83] Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*, 1(3):256–277, 1983.

[KM11] Neal Koblitz and Alfred Menezes. Another look at security definitions. Cryptology ePrint Archive, Report 2011/343, 2011. `http://eprint.iacr.org/`.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Clavier and Gaj [CG09], pages 1–17.

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *ESORICS*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 1998.

[KSWH00] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2/3), 2000.

[KW91] Paul A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *IEEE Symposium on Security and Privacy*, pages 52–63, 1991.

[KW03] Chris Karlof and David Wagner. Hidden Markov model cryptoanalysis. In Walter et al. [WcKKP03], pages 17–34.

[KZB$^+$90] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *IEEE Symposium on Security and Privacy*, pages 2–19, 1990.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[Lau05] Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In Christopher Wolf, Stefan Lucks, and Po-Wah Yau, editors, *WEWoRC*, volume 74 of *LNI*, pages 76–85. GI, 2005.

[Lip75] Steven B. Lipner. A comment on the confinement problem. In *SOSP*, pages 192–196, 1975.

[LM90] Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.

[LZH09] Gregor Leander, Erik Zenner, and Philip Hawkes. Cache timing analysis of LFSR-based stream ciphers. In Matthew G. Parker, editor, *IMA Int. Conf.*, volume 5921 of *Lecture Notes in Computer Science*, pages 433–445. Springer, 2009.

[MBH+02] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[ncs93] National Computer Security Center. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, November 1993. NCSC-TG-030.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[OST05] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. `http://eprint.iacr.org/`.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[Pag02a] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[Pag02b] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. `http://eprint.iacr.org/`.

[Pag03] D. Page. Defending against cache based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, April 2003.

[Pag05] D. Page. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280, 2005. `http://eprint.iacr.org/`.

[Pag09] D. Page. *A Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer, 2009.

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

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[PH07] David A. Patterson and John L. Hennessy. *Computer organization and design - the hardware / software interface (3. ed.)*. Morgan Kaufmann, 2007.

[PN92] Norman E. Proctor and Peter G. Neumann. Architectural implications of covert channels. In *Proceedings of the 15th National Computer Security Conference*, pages 28–43, 1992.

[Pop09] Thomas Popp. An introduction to implementation attacks and countermeasures. In *7th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2009), July 13-15, 2009, Cambridge, Massachusetts, USA*, pages 108–115. IEEE Computer Society, 2009.

[RB08] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.

[Riv94] Ronald L. Rivest. The RC5 encryption algorithm. In Bart Preneel, editor, *FSE*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 1994.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[Sch02] Werner Schindler. A combined timing and power attack. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 263–279. Springer, 2002.

[SGLS77] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the 1977 annual conference*, ACM '77, pages 404–410, New York, NY, USA, 1977. ACM.

[TEF07] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *USENIX Security Symposium*, pages 239–256, Boston, Massachusetts, August 2007.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[TSS$^+$03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Walter et al. [WcKKP03], pages 62–76.

[TTMM02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proc. Int. Symp. on Inform. Theory and Its Applications (ISITA)*, Beijing, China, 2002.

[TTS$^+$06] Yukiyasu Tsunoo, Etsuko Tsujihara, Maki Shigeri, Hiroyasu Kubo, and Kazuhiko Minematsu. Improving cache attacks by considering cipher structure. *Int. J. Inf. Sec.*, 5(3):166–176, 2006.

[Ver10] Ingrid M. R. Verbauwhede, editor. *Secure Integrated Circuits and Systems*. Integrated Circuits and Systems. Springer, 2010.

[Vle90] Tom Van Vleck. Timing channels. `http://www.multicians.org/timing-chn.html`, May 1990.

[WcKKP03] Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.

[Wra91] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[Wra92] J. C. Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4):219–232, 1992.

[WT01] Colin D. Walter and Susan Thompson. Distinguishing exponent digits by observing modular subtractions. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2001.

[Wu04] Hongjun Wu. A new stream cipher HC-256. In Bimal K. Roy and Willi Meier, editors, *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 226–244. Springer, 2004.

[Zen08] Erik Zenner. A cache timing analysis of HC-256. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2008.

A. Recipes

This appendix contains sample spy processes, sample timing data, and a small example of constant-time table lookups using vector byte shuffles. Figure 1.1 shows the assembly logic for a data cache spy process written for the Pentium 4. Figure 1.2 shows a graphical representation of a portion of side-channel data collected by said process. Figure 1.3 shows the assembly logic for an instruction cache spy process written for the Atom. Figure 1.4 contains the encoded archive for both of these spy processes: unpack, compile, and execute to procure timing data. Figure 1.5 shows how to perform constant-time table lookups efficiently in parallel using SSSE3 compiler intrinsics.

```
mov $8192,%edi
LOOPA:
sub $4,%edi
mov $1,(%ecx,%edi)
jnz LOOPA
xor %edi,%edi
rdtsc
mov %eax,%esi
LOOPB:
; cache set 00
imul 0x0000(%ecx),%ecx
imul 0x0800(%ecx),%ecx
imul 0x1000(%ecx),%ecx
imul 0x1800(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x00(%ebx,%edi)
add %eax,%esi
; cache set 01
imul 0x0040(%ecx),%ecx
imul 0x0840(%ecx),%ecx
imul 0x1040(%ecx),%ecx
imul 0x1840(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x01(%ebx,%edi)
add %eax,%esi
; cache set 02
imul 0x0080(%ecx),%ecx
imul 0x0880(%ecx),%ecx
imul 0x1080(%ecx),%ecx
imul 0x1880(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x02(%ebx,%edi)
add %eax,%esi
; cache set 03
imul 0x00c0(%ecx),%ecx
imul 0x08c0(%ecx),%ecx
imul 0x10c0(%ecx),%ecx
imul 0x18c0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x03(%ebx,%edi)
add %eax,%esi
; cache set 04
imul 0x0100(%ecx),%ecx
imul 0x0900(%ecx),%ecx
imul 0x1100(%ecx),%ecx
imul 0x1900(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x04(%ebx,%edi)
add %eax,%esi
; cache set 05
imul 0x0140(%ecx),%ecx
imul 0x0940(%ecx),%ecx
imul 0x1140(%ecx),%ecx
imul 0x1940(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x05(%ebx,%edi)
add %eax,%esi
; cache set 06
imul 0x0180(%ecx),%ecx
imul 0x0980(%ecx),%ecx
imul 0x1180(%ecx),%ecx
imul 0x1980(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x06(%ebx,%edi)
add %eax,%esi
; cache set 07
imul 0x01c0(%ecx),%ecx
imul 0x09c0(%ecx),%ecx
imul 0x11c0(%ecx),%ecx
imul 0x19c0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x07(%ebx,%edi)
add %eax,%esi
; cache set 08
imul 0x0200(%ecx),%ecx
imul 0x0a00(%ecx),%ecx
imul 0x1200(%ecx),%ecx
imul 0x1a00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x08(%ebx,%edi)
add %eax,%esi
; cache set 09
imul 0x0240(%ecx),%ecx
imul 0x0a40(%ecx),%ecx
imul 0x1240(%ecx),%ecx
imul 0x1a40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x09(%ebx,%edi)
add %eax,%esi
; cache set 10
imul 0x0280(%ecx),%ecx
imul 0x0a80(%ecx),%ecx
imul 0x1280(%ecx),%ecx
imul 0x1a80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0a(%ebx,%edi)
add %eax,%esi
; cache set 11
imul 0x02c0(%ecx),%ecx
imul 0x0ac0(%ecx),%ecx
imul 0x12c0(%ecx),%ecx
imul 0x1ac0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0b(%ebx,%edi)
add %eax,%esi
; cache set 12
imul 0x0300(%ecx),%ecx
imul 0x0b00(%ecx),%ecx
imul 0x1300(%ecx),%ecx
imul 0x1b00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0c(%ebx,%edi)
add %eax,%esi
; cache set 13
imul 0x0340(%ecx),%ecx
imul 0x0b40(%ecx),%ecx
imul 0x1340(%ecx),%ecx
imul 0x1b40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0d(%ebx,%edi)
add %eax,%esi
; cache set 14
imul 0x0380(%ecx),%ecx
imul 0x0b80(%ecx),%ecx
imul 0x1380(%ecx),%ecx
imul 0x1b80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0e(%ebx,%edi)
add %eax,%esi
; cache set 15
imul 0x03c0(%ecx),%ecx
imul 0x0bc0(%ecx),%ecx
imul 0x13c0(%ecx),%ecx
imul 0x1bc0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x0f(%ebx,%edi)
add %eax,%esi
; cache set 16
imul 0x0400(%ecx),%ecx
imul 0x0c00(%ecx),%ecx
imul 0x1400(%ecx),%ecx
imul 0x1c00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x10(%ebx,%edi)
add %eax,%esi
; cache set 17
imul 0x0440(%ecx),%ecx
imul 0x0c40(%ecx),%ecx
imul 0x1440(%ecx),%ecx
imul 0x1c40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x11(%ebx,%edi)
add %eax,%esi
; cache set 18
imul 0x0480(%ecx),%ecx
imul 0x0c80(%ecx),%ecx
imul 0x1480(%ecx),%ecx
imul 0x1c80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x12(%ebx,%edi)
add %eax,%esi
; cache set 19
imul 0x04c0(%ecx),%ecx
imul 0x0cc0(%ecx),%ecx
imul 0x14c0(%ecx),%ecx
imul 0x1cc0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x13(%ebx,%edi)
add %eax,%esi
; cache set 20
imul 0x0500(%ecx),%ecx
imul 0x0d00(%ecx),%ecx
imul 0x1500(%ecx),%ecx
imul 0x1d00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x14(%ebx,%edi)
add %eax,%esi
; cache set 21
imul 0x0540(%ecx),%ecx
imul 0x0d40(%ecx),%ecx
imul 0x1540(%ecx),%ecx
imul 0x1d40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x15(%ebx,%edi)
add %eax,%esi
; cache set 22
imul 0x0580(%ecx),%ecx
imul 0x0d80(%ecx),%ecx
imul 0x1580(%ecx),%ecx
imul 0x1d80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x16(%ebx,%edi)
add %eax,%esi
; cache set 23
imul 0x05c0(%ecx),%ecx
imul 0x0dc0(%ecx),%ecx
imul 0x15c0(%ecx),%ecx
imul 0x1dc0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x17(%ebx,%edi)
add %eax,%esi
; cache set 24
imul 0x0600(%ecx),%ecx
imul 0x0e00(%ecx),%ecx
imul 0x1600(%ecx),%ecx
imul 0x1e00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x18(%ebx,%edi)
add %eax,%esi
; cache set 25
imul 0x0640(%ecx),%ecx
imul 0x0e40(%ecx),%ecx
imul 0x1640(%ecx),%ecx
imul 0x1e40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x19(%ebx,%edi)
add %eax,%esi
; cache set 26
imul 0x0680(%ecx),%ecx
imul 0x0e80(%ecx),%ecx
imul 0x1680(%ecx),%ecx
imul 0x1e80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1a(%ebx,%edi)
add %eax,%esi
; cache set 27
imul 0x06c0(%ecx),%ecx
imul 0x0ec0(%ecx),%ecx
imul 0x16c0(%ecx),%ecx
imul 0x1ec0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1b(%ebx,%edi)
add %eax,%esi
; cache set 28
imul 0x0700(%ecx),%ecx
imul 0x0f00(%ecx),%ecx
imul 0x1700(%ecx),%ecx
imul 0x1f00(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1c(%ebx,%edi)
add %eax,%esi
; cache set 29
imul 0x0740(%ecx),%ecx
imul 0x0f40(%ecx),%ecx
imul 0x1740(%ecx),%ecx
imul 0x1f40(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1d(%ebx,%edi)
add %eax,%esi
; cache set 30
imul 0x0780(%ecx),%ecx
imul 0x0f80(%ecx),%ecx
imul 0x1780(%ecx),%ecx
imul 0x1f80(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1e(%ebx,%edi)
add %eax,%esi
; cache set 31
imul 0x07c0(%ecx),%ecx
imul 0x0fc0(%ecx),%ecx
imul 0x17c0(%ecx),%ecx
imul 0x1fc0(%ecx),%ecx
rdtsc
sub %esi,%eax
movnti %eax,0x1f(%ebx,%edi)
add %eax,%esi
add $32,%edi
cmp <buffer len>,%edi
jge END
jmp LOOPB
END:
```

Figure 1.1. Data cache spy process for the Intel Pentium 4.

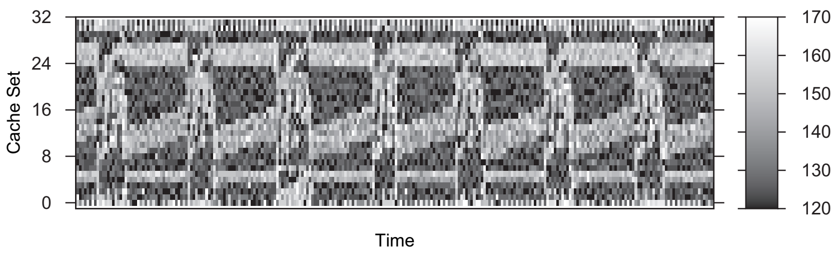


Figure 1.2. Live dcache timing data produced by the spy process in Fig. 1.1 running in parallel with an OpenSSL RSA encryption operation, i.e., a left-to-right sliding window modular exponentiation algorithm. It depicts seven modular multiplications (high latency in sets 20-22), interleaved by a number of repeated modular squarings (high latency in sets 24-27).

```
xor %edi,%edi
rdtsc
mov %eax,%esi
jmp L0
.align 4096
L0:
jmp L64
.rept 59
nop
.endr
L1:
jmp L65
.rept 59
nop
.endr
L2:
jmp L66
.rept 59
nop
.endr
L3:
jmp L67
.rept 59
nop
.endr
L4:
jmp L68
.rept 59
nop
.endr
L5:
jmp L69
.rept 59
nop
.endr
L6:
jmp L70
.rept 59
nop
.endr
L7:
jmp L71
.rept 59
nop
.endr
L8:
jmp L72
.rept 59
nop
.endr
L9:
jmp L73
.rept 59
nop
.endr
L10:
jmp L74
.rept 59
nop
.endr
L11:
jmp L75
.rept 59
nop
.endr
L12:
jmp L76
.rept 59
nop
.endr
L13:
jmp L77
.rept 59
nop
.endr
L14:
jmp L78
.rept 59
nop
.endr
L15:
jmp L79
.rept 59
nop
.endr
L16:
jmp L80
.rept 59
nop
.endr
L17:
jmp L81
.rept 59
nop
.endr
L18:
jmp L82
.rept 59
nop
.endr
L19:
jmp L83
.rept 59
nop
.endr
L20:
jmp L84
.rept 59
nop
.endr
L21:
jmp L85
.rept 59
nop
.endr
L22:
jmp L86
.rept 59
nop
.endr
L23:
jmp L87
.rept 59
nop
.endr
L24:
jmp L88
.rept 59
nop
.endr
L25:
jmp L89
.rept 59
nop
.endr
L26:
jmp L90
.rept 59
nop
.endr
L27:
jmp L91
.rept 59
nop
.endr
L28:
jmp L92
.rept 59
nop
.endr
L29:
jmp L93
.rept 59
nop
.endr
L30:
jmp L94
.rept 59
nop
.endr
L31:
jmp L95
.rept 59
nop
.endr
L32:
jmp L96
.rept 59
nop
.endr
L33:
jmp L97
.rept 59
nop
.endr
L34:
jmp L98
.rept 59
nop
.endr
L35:
jmp L99
.rept 59
nop
.endr
L36:
jmp L100
.rept 59
nop
.endr
L37:
jmp L101
.rept 59
nop
.endr
L38:
jmp L102
.rept 59
nop
.endr
L39:
jmp L103
.rept 59
nop
.endr
L40:
jmp L104
.rept 59
nop
.endr
L41:
jmp L105
.rept 59
nop
.endr
L42:
jmp L106
.rept 59
nop
.endr
L43:
jmp L107
.rept 59
nop
.endr
L44:
jmp L108
.rept 59
nop
.endr
L45:
jmp L109
.rept 59
nop
.endr
L46:
jmp L110
.rept 59
nop
.endr
L47:
jmp L111
.rept 59
nop
.endr
L48:
jmp L112
.rept 59
nop
.endr
L49:
jmp L113
.rept 59
nop
.endr
L50:
jmp L114
.rept 59
nop
.endr
L51:
jmp L115
.rept 59
nop
.endr
L52:
jmp L116
.rept 59
nop
.endr
L53:
jmp L117
.rept 59
nop
.endr
L54:
jmp L118
.rept 59
nop
.endr
L55:
jmp L119
.rept 59
nop
.endr
L56:
jmp L120
.rept 59
nop
.endr
L57:
jmp L121
.rept 59
nop
.endr
L58:
jmp L122
.rept 59
nop
.endr
L59:
jmp L123
.rept 59
nop
.endr
L60:
jmp L124
.rept 59
nop
.endr
L61:
jmp L125
.rept 59
nop
.endr
L62:
jmp L126
.rept 59
nop
.endr
L63:
jmp L127
.rept 59
nop
.endr
L64:
jmp L128
.rept 59
nop
.endr
L65:
jmp L129
.rept 59
nop
.endr
L66:
jmp L130
.rept 59
nop
.endr
L67:
jmp L131
.rept 59
nop
.endr
L68:
jmp L132
.rept 59
nop
.endr
L69:
jmp L133
.rept 59
nop
.endr
L70:
jmp L134
.rept 59
nop
.endr
L71:
jmp L135
.rept 59
nop
.endr
L72:
jmp L136
.rept 59
nop
.endr
L73:
jmp L137
.rept 59
nop
.endr
L74:
jmp L138
.rept 59
nop
.endr
L75:
jmp L139
.rept 59
nop
.endr
L76:
jmp L140
.rept 59
nop
.endr
L77:
jmp L141
.rept 59
nop
.endr
L78:
jmp L142
.rept 59
nop
.endr
L79:
jmp L143
.rept 59
nop
.endr
L80:
jmp L144
.rept 59
nop
.endr
L81:
jmp L145
.rept 59
nop
.endr
L82:
jmp L146
.rept 59
nop
.endr
L83:
jmp L147
.rept 59
nop
.endr
L84:
jmp L148
.rept 59
nop
.endr
L85:
jmp L149
.rept 59
nop
.endr
L86:
jmp L150
.rept 59
nop
.endr
L87:
jmp L151
.rept 59
nop
.endr
L88:
jmp L152
.rept 59
nop
.endr
L89:
jmp L153
.rept 59
nop
.endr
L90:
jmp L154
.rept 59
nop
.endr
L91:
jmp L155
.rept 59
nop
.endr
L92:
jmp L156
.rept 59
nop
.endr
L93:
jmp L157
.rept 59
nop
.endr
L94:
jmp L158
.rept 59
nop
.endr
L95:
jmp L159
.rept 59
nop
.endr
L96:
jmp L160
.rept 59
nop
.endr
L97:
jmp L161
.rept 59
nop
.endr
L98:
jmp L162
.rept 59
nop
.endr
L99:
jmp L163
.rept 59
nop
.endr
L100:
jmp L164
.rept 59
nop
.endr
L101:
jmp L165
.rept 59
nop
.endr
L102:
jmp L166
.rept 59
nop
.endr
L103:
jmp L167
.rept 59
nop
.endr
L104:
jmp L168
.rept 59
nop
.endr
L105:
jmp L169
.rept 59
nop
.endr
L106:
jmp L170
.rept 59
nop
.endr
L107:
jmp L171
.rept 59
nop
.endr
L108:
jmp L172
.rept 59
nop
.endr
L109:
jmp L173
.rept 59
nop
.endr
L110:
jmp L174
.rept 59
nop
.endr
L111:
jmp L175
.rept 59
nop
.endr
L112:
jmp L176
.rept 59
nop
.endr
L113:
jmp L177
.rept 59
nop
.endr
L114:
jmp L178
.rept 59
nop
.endr
L115:
jmp L179
.rept 59
nop
.endr
L116:
jmp L180
.rept 59
nop
.endr
L117:
jmp L181
.rept 59
nop
.endr
L118:
jmp L182
.rept 59
nop
.endr
L119:
jmp L183
.rept 59
nop
.endr
L120:
jmp L184
.rept 59
nop
.endr
L121:
jmp L185
.rept 59
nop
.endr
L122:
jmp L186
.rept 59
nop
.endr
L123:
jmp L187
.rept 59
nop
.endr
L124:
jmp L188
.rept 59
nop
.endr
L125:
jmp L189
.rept 59
nop
.endr
L126:
jmp L190
.rept 59
nop
.endr
L127:
jmp L191
.rept 59
nop
.endr
L128:
jmp L192
.rept 59
nop
.endr
L129:
jmp L193
.rept 59
nop
.endr
L130:
jmp L194
.rept 59
nop
.endr
L131:
jmp L195
.rept 59
nop
.endr
L132:
jmp L196
.rept 59
nop
.endr
L133:
jmp L197
.rept 59
nop
.endr
L134:
jmp L198
.rept 59
nop
.endr
L135:
jmp L199
.rept 59
nop
.endr
L136:
jmp L200
.rept 59
nop
.endr
L137:
jmp L201
.rept 59
nop
.endr
L138:
jmp L202
.rept 59
nop
.endr
L139:
jmp L203
.rept 59
nop
.endr
L140:
jmp L204
.rept 59
nop
.endr
L141:
jmp L205
.rept 59
nop
.endr
L142:
jmp L206
.rept 59
nop
.endr
L143:
jmp L207
.rept 59
nop
.endr
L144:
jmp L208
.rept 59
nop
.endr
L145:
jmp L209
.rept 59
nop
.endr
L146:
jmp L210
.rept 59
nop
.endr
L147:
jmp L211
.rept 59
nop
.endr
L148:
jmp L212
.rept 59
nop
.endr
L149:
jmp L213
.rept 59
nop
.endr
L150:
jmp L214
.rept 59
nop
.endr
L151:
jmp L215
.rept 59
nop
.endr
L152:
jmp L216
.rept 59
nop
.endr
L153:
jmp L217
.rept 59
nop
.endr
L154:
jmp L218
.rept 59
nop
.endr
L155:
jmp L219
.rept 59
nop
.endr
L156:
jmp L220
.rept 59
nop
.endr
L157:
jmp L221
.rept 59
nop
.endr
L158:
jmp L222
.rept 59
nop
.endr
L159:
jmp L223
.rept 59
nop
.endr
L160:
jmp L224
.rept 59
nop
.endr
L161:
jmp L225
.rept 59
nop
.endr
L162:
jmp L226
.rept 59
nop
.endr
L163:
jmp L227
.rept 59
nop
.endr
L164:
jmp L228
.rept 59
nop
.endr
L165:
jmp L229
.rept 59
nop
.endr
L166:
jmp L230
.rept 59
nop
.endr
L167:
jmp L231
.rept 59
nop
.endr
L168:
jmp L232
.rept 59
nop
.endr
L169:
jmp L233
.rept 59
nop
.endr
L170:
jmp L234
.rept 59
nop
.endr
L171:
jmp L235
.rept 59
nop
.endr
L172:
jmp L236
.rept 59
nop
.endr
L173:
jmp L237
.rept 59
nop
.endr
L174:
jmp L238
.rept 59
nop
.endr
L175:
jmp L239
.rept 59
nop
.endr
L176:
jmp L240
.rept 59
nop
.endr
L177:
jmp L241
.rept 59
nop
.endr
L178:
jmp L242
.rept 59
nop
.endr
L179:
jmp L243
.rept 59
nop
.endr
L180:
jmp L244
.rept 59
nop
.endr
L181:
jmp L245
.rept 59
nop
.endr
L182:
jmp L246
.rept 59
nop
.endr
L183:
jmp L247
.rept 59
nop
.endr
L184:
jmp L248
.rept 59
nop
.endr
L185:
jmp L249
.rept 59
nop
.endr
L186:
jmp L250
.rept 59
nop
.endr
L187:
jmp L251
.rept 59
nop
.endr
L188:
jmp L252
.rept 59
nop
.endr
L189:
jmp L253
.rept 59
nop
.endr
L190:
jmp L254
.rept 59
nop
.endr
L191:
jmp L255
.rept 59
nop
.endr
L192:
jmp L256
.rept 59
nop
.endr
L193:
jmp L257
.rept 59
nop
.endr
L194:
jmp L258
.rept 59
nop
.endr
L195:
jmp L259
.rept 59
nop
.endr
L196:
jmp L260
.rept 59
nop
.endr
L197:
jmp L261
.rept 59
nop
.endr
L198:
jmp L262
.rept 59
nop
.endr
L199:
jmp L263
.rept 59
nop
.endr
L200:
jmp L264
.rept 59
nop
.endr
L201:
jmp L265
.rept 59
nop
.endr
L202:
jmp L266
.rept 59
nop
.endr
L203:
jmp L267
.rept 59
nop
.endr
L204:
jmp L268
.rept 59
nop
.endr
L205:
jmp L269
.rept 59
nop
.endr
L206:
jmp L270
.rept 59
nop
.endr
L207:
jmp L271
.rept 59
nop
.endr
L208:
jmp L272
.rept 59
nop
.endr
L209:
jmp L273
.rept 59
nop
.endr
L210:
jmp L274
.rept 59
nop
.endr
L211:
jmp L275
.rept 59
nop
.endr
L212:
jmp L276
.rept 59
nop
.endr
L213:
jmp L277
.rept 59
nop
.endr
L214:
jmp L278
.rept 59
nop
.endr
L215:
jmp L279
.rept 59
nop
.endr
L216:
jmp L280
.rept 59
nop
.endr
L217:
jmp L281
.rept 59
nop
.endr
L218:
jmp L282
.rept 59
nop
.endr
L219:
jmp L283
.rept 59
nop
.endr
L220:
jmp L284
.rept 59
nop
.endr
L221:
jmp L285
.rept 59
nop
.endr
L222:
jmp L286
.rept 59
nop
.endr
L223:
jmp L287
.rept 59
nop
.endr
L224:
jmp L288
.rept 59
nop
.endr
L225:
jmp L289
.rept 59
nop
.endr
L226:
jmp L290
.rept 59
nop
.endr
L227:
jmp L291
.rept 59
nop
.endr
L228:
jmp L292
.rept 59
nop
.endr
L229:
jmp L293
.rept 59
nop
.endr
L230:
jmp L294
.rept 59
nop
.endr
L231:
jmp L295
.rept 59
nop
.endr
L232:
jmp L296
.rept 59
nop
.endr
L233:
jmp L297
.rept 59
nop
.endr
L234:
jmp L298
.rept 59
nop
.endr
L235:
jmp L299
.rept 59
nop
.endr
L236:
jmp L300
.rept 59
nop
.endr
L237:
jmp L301
.rept 59
nop
.endr
L238:
jmp L302
.rept 59
nop
.endr
L239:
jmp L303
.rept 59
nop
.endr
L240:
jmp L304
.rept 59
nop
.endr
L241:
jmp L305
.rept 59
nop
.endr
L242:
jmp L306
.rept 59
nop
.endr
L243:
jmp L307
.rept 59
nop
.endr
L244:
jmp L308
.rept 59
nop
.endr
L245:
jmp L309
.rept 59
nop
.endr
L246:
jmp L310
.rept 59
nop
.endr
L247:
jmp L311
.rept 59
nop
.endr
L248:
jmp L312
.rept 59
nop
.endr
L249:
jmp L313
.rept 59
nop
.endr
L250:
jmp L314
.rept 59
nop
.endr
L251:
jmp L315
.rept 59
nop
.endr
L252:
jmp L316
.rept 59
nop
.endr
L253:
jmp L317
.rept 59
nop
.endr
L254:
jmp L318
.rept 59
nop
.endr
L255:
jmp L319
.rept 59
nop
.endr
L256:
jmp L320
.rept 59
nop
.endr
L257:
jmp L321
.rept 59
nop
.endr
L258:
jmp L322
.rept 59
nop
.endr
L259:
jmp L323
.rept 59
nop
.endr
L260:
jmp L324
.rept 59
nop
.endr
L261:
jmp L325
.rept 59
nop
.endr
L262:
jmp L326
.rept 59
nop
.endr
L263:
jmp L327
.rept 59
nop
.endr
L264:
jmp L328
.rept 59
nop
.endr
L265:
jmp L329
.rept 59
nop
.endr
L266:
jmp L330
.rept 59
nop
.endr
L267:
jmp L331
.rept 59
nop
.endr
L268:
jmp L332
.rept 59
nop
.endr
L269:
jmp L333
.rept 59
nop
.endr
L270:
jmp L334
.rept 59
nop
.endr
L271:
jmp L335
.rept 59
nop
.endr
L272:
jmp L336
.rept 59
nop
.endr
L273:
jmp L337
.rept 59
nop
.endr
L274:
jmp L338
.rept 59
nop
.endr
L275:
jmp L339
.rept 59
nop
.endr
L276:
jmp L340
.rept 59
nop
.endr
L277:
jmp L341
.rept 59
nop
.endr
L278:
jmp L342
.rept 59
nop
.endr
L279:
jmp L343
.rept 59
nop
.endr
L280:
jmp L344
.rept 59
nop
.endr
L281:
jmp L345
.rept 59
nop
.endr
L282:
jmp L346
.rept 59
nop
.endr
L283:
jmp L347
.rept 59
nop
.endr
L284:
jmp L348
.rept 59
nop
.endr
L285:
jmp L349
.rept 59
nop
.endr
L286:
jmp L350
.rept 59
nop
.endr
L287:
jmp L351
.rept 59
nop
.endr
L288:
jmp L352
.rept 59
nop
.endr
L289:
jmp L353
.rept 59
nop
.endr
L290:
jmp L354
.rept 59
nop
.endr
L291:
jmp L355
.rept 59
nop
.endr
L292:
jmp L356
.rept 59
nop
.endr
L293:
jmp L357
.rept 59
nop
.endr
L294:
jmp L358
.rept 59
nop
.endr
L295:
jmp L359
.rept 59
nop
.endr
L296:
jmp L360
.rept 59
nop
.endr
L297:
jmp L361
.rept 59
nop
.endr
L298:
jmp L362
.rept 59
nop
.endr
L299:
jmp L363
.rept 59
nop
.endr
L300:
jmp L364
.rept 59
nop
.endr
L301:
jmp L365
.rept 59
nop
.endr
L302:
jmp L366
.rept 59
nop
.endr
L303:
jmp L367
.rept 59
nop
.endr
L304:
jmp L368
.rept 59
nop
.endr
L305:
jmp L369
.rept 59
nop
.endr
L306:
jmp L370
.rept 59
nop
.endr
L307:
jmp L371
.rept 59
nop
.endr
L308:
jmp L372
.rept 59
nop
.endr
L309:
jmp L373
.rept 59
nop
.endr
L310:
jmp L374
.rept 59
nop
.endr
L311:
jmp L375
.rept 59
nop
.endr
L312:
jmp L376
.rept 59
nop
.endr
L313:
jmp L377
.rept 59
nop
.endr
L314:
jmp L378
.rept 59
nop
.endr
L315:
jmp L379
.rept 59
nop
.endr
L316:
jmp L380
.rept 59
nop
.endr
L317:
jmp L381
.rept 59
nop
.endr
L318:
jmp L382
.rept 59
nop
.endr
L319:
jmp L383
.rept 59
nop
.endr
L320:
jmp L384
.rept 59
nop
.endr
L321:
jmp L385
.rept 59
nop
.endr
L322:
jmp L386
.rept 59
nop
.endr
L323:
jmp L387
.rept 59
nop
.endr
L324:
jmp L388
.rept 59
nop
.endr
L325:
jmp L389
.rept 59
nop
.endr
L326:
jmp L390
.rept 59
nop
.endr
L327:
jmp L391
.rept 59
nop
.endr
L328:
jmp L392
.rept 59
nop
.endr
L329:
jmp L393
.rept 59
nop
.endr
L330:
jmp L394
.rept 59
nop
.endr
L331:
jmp L395
.rept 59
nop
.endr
L332:
jmp L396
.rept 59
nop
.endr
L333:
jmp L397
.rept 59
nop
.endr
L334:
jmp L398
.rept 59
nop
.endr
L335:
jmp L399
.rept 59
nop
.endr
L336:
jmp L400
.rept 59
nop
.endr
L337:
jmp L401
.rept 59
nop
.endr
L338:
jmp L402
.rept 59
nop
.endr
L339:
jmp L403
.rept 59
nop
.endr
L340:
jmp L404
.rept 59
nop
.endr
L341:
jmp L405
.rept 59
nop
.endr
L342:
jmp L406
.rept 59
nop
.endr
L343:
jmp L407
.rept 59
nop
.endr
L344:
jmp L408
.rept 59
nop
.endr
L345:
jmp L409
.rept 59
nop
.endr
L346:
jmp L410
.rept 59
nop
.endr
L347:
jmp L411
.rept 59
nop
.endr
L348:
jmp L412
.rept 59
nop
.endr
L349:
jmp L413
.rept 59
nop
.endr
L350:
jmp L414
.rept 59
nop
.endr
L351:
jmp L415
.rept 59
nop
.endr
L352:
jmp L416
.rept 59
nop
.endr
L353:
jmp L417
.rept 59
nop
.endr
L354:
jmp L418
.rept 59
nop
.endr
L355:
jmp L419
.rept 59
nop
.endr
L356:
jmp L420
.rept 59
nop
.endr
L357:
jmp L421
.rept 59
nop
.endr
L358:
jmp L422
.rept 59
nop
.endr
L359:
jmp L423
.rept 59
nop
.endr
L360:
jmp L424
.rept 59
nop
.endr
L361:
jmp L425
.rept 59
nop
.endr
L362:
jmp L426
.rept 59
nop
.endr
L363:
jmp L427
.rept 59
nop
.endr
L364:
jmp L428
.rept 59
nop
.endr
L365:
jmp L429
.rept 59
nop
.endr
L366:
jmp L430
.rept 59
nop
.endr
L367:
jmp L431
.rept 59
nop
.endr
L368:
jmp L432
.rept 59
nop
.endr
L369:
jmp L433
.rept 59
nop
.endr
L370:
jmp L434
.rept 59
nop
.endr
L371:
jmp L435
.rept 59
nop
.endr
L372:
jmp L436
.rept 59
nop
.endr
L373:
jmp L437
.rept 59
nop
.endr
L374:
jmp L438
.rept 59
nop
.endr
L375:
jmp L439
.rept 59
nop
.endr
L376:
jmp L440
.rept 59
nop
.endr
L377:
jmp L441
.rept 59
nop
.endr
L378:
jmp L442
.rept 59
nop
.endr
L379:
jmp L443
.rept 59
nop
.endr
L380:
jmp L444
.rept 59
nop
.endr
L381:
jmp L445
.rept 59
nop
.endr
L382:
jmp L446
.rept 59
nop
.endr
L383:
jmp L447
.rept 59
nop
.endr
L384:
jmp L448
.rept 59
nop
.endr
L385:
jmp L449
.rept 59
nop
.endr
L386:
jmp L450
.rept 59
nop
.endr
L387:
jmp L451
.rept 59
nop
.endr
L388:
jmp L452
.rept 59
nop
.endr
L389:
jmp L453
.rept 59
nop
.endr
L390:
jmp L454
.rept 59
nop
.endr
L391:
jmp L455
.rept 59
nop
.endr
L392:
jmp L456
.rept 59
nop
.endr
L393:
jmp L457
.rept 59
nop
.endr
L394:
jmp L458
.rept 59
nop
.endr
L395:
jmp L459
.rept 59
nop
.endr
L396:
jmp L460
.rept 59
nop
.endr
L397:
jmp L461
.rept 59
nop
.endr
L398:
jmp L462
.rept 59
nop
.endr
L399:
jmp L463
.rept 59
nop
.endr
L400:
jmp L464
.rept 59
nop
.endr
L401:
jmp L465
.rept 59
nop
.endr
L402:
jmp L466
.rept 59
nop
.endr
L403:
jmp L467
.rept 59
nop
.endr
L404:
jmp L468
.rept 59
nop
.endr
L405:
jmp L469
.rept 59
nop
.endr
L406:
jmp L470
.rept 59
nop
.endr
L407:
jmp L471
.rept 59
nop
.endr
L408:
jmp L472
.rept 59
nop
.endr
L409:
jmp L473
.rept 59
nop
.endr
L410:
jmp L474
.rept 59
nop
.endr
L411:
jmp L475
.rept 59
nop
.endr
L412:
jmp L476
.rept 59
nop
.endr
L413:
jmp L477
.rept 59
nop
.endr
L414:
jmp L478
.rept 59
nop
.endr
L415:
jmp L479
.rept 59
nop
.endr
L416:
jmp L480
.rept 59
nop
.endr
L417:
jmp L481
.rept 59
nop
.endr
L418:
jmp L482
.rept 59
nop
.endr
L419:
jmp L483
.rept 59
nop
.endr
L420:
jmp L484
.rept 59
nop
.endr
L421:
jmp L485
.rept 59
nop
.endr
L422:
jmp L486
.rept 59
nop
.endr
L423:
jmp L487
.rept 59
nop
.endr
L424:
jmp L488
.rept 59
nop
.endr
L425:
jmp L489
.rept 59
nop
.endr
L426:
jmp L490
.rept 59
nop
.endr
L427:
jmp L491
.rept 59
nop
.endr
L428:
jmp L492
.rept 59
nop
.endr
L429:
jmp L493
.rept 59
nop
.endr
L430:
jmp L494
.rept 59
nop
.endr
L431:
jmp L495
.rept 59
nop
.endr
L432:
jmp L496
.rept 59
nop
.endr
L433:
jmp L497
.rept 59
nop
.endr
L434:
jmp L498
.rept 59
nop
.endr
L435:
jmp L499
.rept 59
nop
.endr
L436:
jmp L500
.rept 59
nop
.endr
L437:
jmp L501
.rept 59
nop
.endr
L438:
jmp L502
.rept 59
nop
.endr
L439:
jmp L503
.rept 59
nop
.endr
L440:
jmp L504
.rept 59
nop
.endr
L441:
jmp L505
.rept 59
nop
.endr
L442:
jmp L506
.rept 59
nop
.endr
L443:
jmp L507
.rept 59
nop
.endr
L444:
jmp L508
.rept 59
nop
.endr
L445:
jmp L509
.rept 59
nop
.endr
L446:
jmp L510
.rept 59
nop
.endr
L447:
jmp L511
.rept 59
nop
.endr
L448:
rdtsc
sub %esi,%eax
movb %al,0x00(%ebx,%edi)
add %eax,%esi
jmp L1
.rept 50
nop
.endr
L449:
rdtsc
sub %esi,%eax
movb %al,0x01(%ebx,%edi)
add %eax,%esi
jmp L2
.rept 49
nop
.endr
L450:
rdtsc
sub %esi,%eax
movb %al,0x02(%ebx,%edi)
add %eax,%esi
jmp L3
.rept 49
nop
.endr
L451:
rdtsc
sub %esi,%eax
movb %al,0x03(%ebx,%edi)
add %eax,%esi
jmp L4
.rept 49
nop
.endr
L452:
rdtsc
sub %esi,%eax
movb %al,0x04(%ebx,%edi)
add %eax,%esi
jmp L5
.rept 49
nop
.endr
L453:
rdtsc
sub %esi,%eax
movb %al,0x05(%ebx,%edi)
add %eax,%esi
jmp L6
.rept 49
nop
.endr
L454:
rdtsc
sub %esi,%eax
movb %al,0x06(%ebx,%edi)
add %eax,%esi
jmp L7
.rept 49
nop
.endr
L455:
rdtsc
sub %esi,%eax
movb %al,0x07(%ebx,%edi)
add %eax,%esi
jmp L8
.rept 49
nop
.endr
L456:
rdtsc
sub %esi,%eax
movb %al,0x08(%ebx,%edi)
add %eax,%esi
jmp L9
.rept 49
nop
.endr
L457:
rdtsc
sub %esi,%eax
movb %al,0x09(%ebx,%edi)
add %eax,%esi
jmp L10
.rept 49
nop
.endr
L458:
rdtsc
sub %esi,%eax
movb %al,0x0a(%ebx,%edi)
add %eax,%esi
jmp L11
.rept 49
nop
.endr
L459:
rdtsc
sub %esi,%eax
movb %al,0x0b(%ebx,%edi)
add %eax,%esi
jmp L12
.rept 49
nop
.endr
L460:
rdtsc
sub %esi,%eax
movb %al,0x0c(%ebx,%edi)
add %eax,%esi
jmp L13
.rept 49
nop
.endr
L461:
rdtsc
sub %esi,%eax
movb %al,0x0d(%ebx,%edi)
add %eax,%esi
jmp L14
.rept 49
nop
.endr
L462:
rdtsc
sub %esi,%eax
movb %al,0x0e(%ebx,%edi)
add %eax,%esi
jmp L15
.rept 49
nop
.endr
L463:
rdtsc
sub %esi,%eax
movb %al,0x0f(%ebx,%edi)
add %eax,%esi
jmp L16
.rept 49
nop
.endr
L464:
rdtsc
sub %esi,%eax
movb %al,0x10(%ebx,%edi)
add %eax,%esi
jmp L17
.rept 49
nop
.endr
L465:
rdtsc
sub %esi,%eax
movb %al,0x11(%ebx,%edi)
add %eax,%esi
jmp L18
.rept 49
nop
.endr
L466:
rdtsc
sub %esi,%eax
movb %al,0x12(%ebx,%edi)
add %eax,%esi
jmp L19
.rept 49
nop
.endr
L467:
rdtsc
sub %esi,%eax
movb %al,0x13(%ebx,%edi)
add %eax,%esi
jmp L20
.rept 49
nop
.endr
L468:
rdtsc
sub %esi,%eax
movb %al,0x14(%ebx,%edi)
add %eax,%esi
jmp L21
.rept 49
nop
.endr
L469:
rdtsc
sub %esi,%eax
movb %al,0x15(%ebx,%edi)
add %eax,%esi
jmp L22
.rept 49
nop
.endr
L470:
rdtsc
sub %esi,%eax
movb %al,0x16(%ebx,%edi)
add %eax,%esi
jmp L23
.rept 49
nop
.endr
L471:
rdtsc
sub %esi,%eax
movb %al,0x17(%ebx,%edi)
add %eax,%esi
jmp L24
.rept 49
nop
.endr
L472:
rdtsc
sub %esi,%eax
movb %al,0x18(%ebx,%edi)
add %eax,%esi
jmp L25
.rept 49
nop
.endr
L473:
rdtsc
sub %esi,%eax
movb %al,0x19(%ebx,%edi)
add %eax,%esi
jmp L26
.rept 49
nop
.endr
L474:
rdtsc
sub %esi,%eax
movb %al,0x1a(%ebx,%edi)
add %eax,%esi
jmp L27
.rept 49
nop
.endr
L475:
rdtsc
sub %esi,%eax
movb %al,0x1b(%ebx,%edi)
add %eax,%esi
jmp L28
.rept 49
nop
.endr
L476:
rdtsc
sub %esi,%eax
movb %al,0x1c(%ebx,%edi)
add %eax,%esi
jmp L29
.rept 49
nop
.endr
L477:
rdtsc
sub %esi,%eax
movb %al,0x1d(%ebx,%edi)
add %eax,%esi
jmp L30
.rept 49
nop
.endr
L478:
rdtsc
sub %esi,%eax
movb %al,0x1e(%ebx,%edi)
add %eax,%esi
jmp L31
.rept 49
nop
.endr
L479:
rdtsc
sub %esi,%eax
movb %al,0x1f(%ebx,%edi)
add %eax,%esi
jmp L32
.rept 49
nop
.endr
L480:
rdtsc
sub %esi,%eax
movb %al,0x20(%ebx,%edi)
add %eax,%esi
jmp L33
.rept 49
nop
.endr
L481:
rdtsc
sub %esi,%eax
movb %al,0x21(%ebx,%edi)
add %eax,%esi
jmp L34
.rept 49
nop
.endr
L482:
rdtsc
sub %esi,%eax
movb %al,0x22(%ebx,%edi)
add %eax,%esi
jmp L35
.rept 49
nop
.endr
L483:
rdtsc
sub %esi,%eax
movb %al,0x23(%ebx,%edi)
add %eax,%esi
jmp L36
.rept 49
nop
.endr
L484:
rdtsc
sub %esi,%eax
movb %al,0x24(%ebx,%edi)
add %eax,%esi
jmp L37
.rept 49
nop
.endr
L485:
rdtsc
sub %esi,%eax
movb %al,0x25(%ebx,%edi)
add %eax,%esi
jmp L38
.rept 49
nop
.endr
L486:
rdtsc
sub %esi,%eax
movb %al,0x26(%ebx,%edi)
add %eax,%esi
jmp L39
.rept 49
nop
.endr
L487:
rdtsc
sub %esi,%eax
movb %al,0x27(%ebx,%edi)
add %eax,%esi
jmp L40
.rept 49
nop
.endr
L488:
rdtsc
sub %esi,%eax
movb %al,0x28(%ebx,%edi)
add %eax,%esi
jmp L41
.rept 49
nop
.endr
L489:
rdtsc
sub %esi,%eax
movb %al,0x29(%ebx,%edi)
add %eax,%esi
jmp L42
.rept 49
nop
.endr
L490:
rdtsc
sub %esi,%eax
movb %al,0x2a(%ebx,%edi)
add %eax,%esi
jmp L43
.rept 49
nop
.endr
L491:
rdtsc
sub %esi,%eax
movb %al,0x2b(%ebx,%edi)
add %eax,%esi
jmp L44
.rept 49
nop
.endr
L492:
rdtsc
sub %esi,%eax
movb %al,0x2c(%ebx,%edi)
add %eax,%esi
jmp L45
.rept 49
nop
.endr
L493:
rdtsc
sub %esi,%eax
movb %al,0x2d(%ebx,%edi)
add %eax,%esi
jmp L46
.rept 49
nop
.endr
L494:
rdtsc
sub %esi,%eax
movb %al,0x2e(%ebx,%edi)
add %eax,%esi
jmp L47
.rept 49
nop
.endr
L495:
rdtsc
sub %esi,%eax
movb %al,0x2f(%ebx,%edi)
add %eax,%esi
jmp L48
.rept 49
nop
.endr
L496:
rdtsc
sub %esi,%eax
movb %al,0x30(%ebx,%edi)
add %eax,%esi
jmp L49
.rept 49
nop
.endr
L497:
rdtsc
sub %esi,%eax
movb %al,0x31(%ebx,%edi)
add %eax,%esi
jmp L50
.rept 49
nop
.endr
L498:
rdtsc
sub %esi,%eax
movb %al,0x32(%ebx,%edi)
add %eax,%esi
jmp L51
.rept 49
nop
.endr
L499:
rdtsc
sub %esi,%eax
movb %al,0x33(%ebx,%edi)
add %eax,%esi
jmp L52
.rept 49
nop
.endr
L500:
rdtsc
sub %esi,%eax
movb %al,0x34(%ebx,%edi)
add %eax,%esi
jmp L53
.rept 49
nop
.endr
L501:
rdtsc
sub %esi,%eax
movb %al,0x35(%ebx,%edi)
add %eax,%esi
jmp L54
.rept 49
nop
.endr
L502:
rdtsc
sub %esi,%eax
movb %al,0x36(%ebx,%edi)
add %eax,%esi
jmp L55
.rept 49
nop
.endr
L503:
rdtsc
sub %esi,%eax
movb %al,0x37(%ebx,%edi)
add %eax,%esi
jmp L56
.rept 49
nop
.endr
L504:
rdtsc
sub %esi,%eax
movb %al,0x38(%ebx,%edi)
add %eax,%esi
jmp L57
.rept 49
nop
.endr
L505:
rdtsc
sub %esi,%eax
movb %al,0x39(%ebx,%edi)
add %eax,%esi
jmp L58
.rept 49
nop
.endr
L506:
rdtsc
sub %esi,%eax
movb %al,0x3a(%ebx,%edi)
add %eax,%esi
jmp L59
.rept 49
nop
.endr
L507:
rdtsc
sub %esi,%eax
movb %al,0x3b(%ebx,%edi)
add %eax,%esi
jmp L60
.rept 49
nop
.endr
L508:
rdtsc
sub %esi,%eax
movb %al,0x3c(%ebx,%edi)
add %eax,%esi
jmp L61
.rept 49
nop
.endr
L509:
rdtsc
sub %esi,%eax
movb %al,0x3d(%ebx,%edi)
add %eax,%esi
jmp L62
.rept 49
nop
.endr
L510:
rdtsc
sub %esi,%eax
movb %al,0x3e(%ebx,%edi)
add %eax,%esi
jmp L63
.rept 49
nop
.endr
L511:
rdtsc
sub %esi,%eax
movb %al,0x3f(%ebx,%edi)
add %eax,%esi
add $64,%edi
cmp <buffer len>,%edi
jge END
jmp L0
END:
```

Figure 1.3. Instruction cache spy process for the Intel Atom.

```
begin-base64 644 spy.zip
UEsDBAoAAAAAAK+Uhz0AAAAAAAAAAAAAAAAAEABwAc3B5L1VUCQADymL+TD5k
/kx1eAsAAQToAwAABOgDAABQSwMECgAAAAAAVpWHPQAAAAAAAAAAAAAAAAA8A
HABzcHkvc3B5X2ljYWNoZS9VVAkAAwNk/kwEZP5MdXgLAAEE6AMAAAToAwAA
UEsDBAoAAAAAAHyUhz0AAAAAAAAAAAAAAAAAPABwAc3B5L3NweV9kY2FjaGUv
VVQJAANrYv5MRWT+THV4CwABBOgDAAAE6AMAAFBLAwQUAAAACADqlIc9vB5+
O7QOAADPrgAAGwAcAHNweS9zcHlfaWNhY2hlL3NweV9pY2FjaGUuY1VUCQAD
N2P+TFVj/kx1eAsAAQToAwAABOgDAADF3VtvG9cZheFr8VcMmBiQDNXmdxge
7CZA06aBAdspmqQ3iSHIEmUzlSmBpJK0Rf57KR9gz8oeaPDebCCAKIrc0hcs
aFl89uY8vN+cnTV/umq21/85WZ2dnr1efnLzwf5Lr0bb5a453bzaNvbg5Wo9
erlZnv678XZ0vtpeP1w1n1+fjTY369F2Nbr/cPTZan12eXO+bP683Z2vrh68
/rJz12a1fnV73+iz8+XFar1snv/w7F9f//X7b//5XWM+mUw+/cL3T549ef7N
d80073982McHfPuPvzTjdjH+9J6vmnG249Fotd41b05X68PbG/sf/uy4OXt9
umnu72//8uOLo+Z/o9HBu3suVpfL9emb5ePRweri8PaxzZeNHTUf7v/i7VPs
xf7ry8vtcv/Mg4Pr/Ry7i8PxD9vTV8tHzb1t8+OHh7/4aT0+bt4+Z/LiaP+k
g81yd7NZN5P97d/33/VmvV29Wi/P3/1AL3/8O0j+W4wOTk5Ot29OTpqTk1+u
Lk93+1X3nzSHo/FvV5vm3r3l+er43Yef1j/txqPx5ny3PXt/+83VL7df0/3t
7UO2Hx7y85vr5unk/ScPTi/3377JyWL6/p6nk0efPnCaHx65WV7vmvH7/9Hv
71xfXX/48nJ9vvmwhHWXaMES3l1iCpaI7hIzsER2l5iDJdruEguwxLSzxGwC
lph1lzCwxLy7hIMlFt0lgkSrG88Zimc3nzOST+sGdEYCat2EzkhCrRvRGYmo
dTM6Ixm1bkjnJKTWTemcpNS6MZ2TmFo3p3OSU+/mdE5y6t2cztHvOW5O5ySn
3s3pnOTUuzmdk5x6N6dzklPv5nRBcurdnC5ITr2bOwXJqXdzuiA5jW5OFySn
Oc3pguQ0ujldoMLv5nRBchrdnC5ITqOb0wXJaRzahMS1JjJIiSpMZdFSFRj
IYuQrOZEFiFhTZNFSFrTZRES1wxZBPOLNWUREthsZRGS2JTEGklsSmKNJDYl
sUYSm5JYI4lltJbFGEttKYo0ktpXEOknsVBLrJLFTSayTxE4lsU4SO5XEOknsVBIbJLFTSWyQxM4ksUESO5PEBknsTBIbJLEzSWyQxM4k
sUESO5PEBknsTBKb6NUrSWySxM4ksUkSO5PEJknsXBKbJLFzSWySxM4lsUkS
O5fEJknsXBKbJLFzSWySxM4lsS1J7FwS26JXXCWxLUnsXBLbksQuJLEtSexC
EtuSxC4ksS1J7EIS25LELiSxLUnsQhLbksQuJLFTktiFJHZKEruQxE6REkhi
p8wJJLIMsiaSWWRZNpHQIs6yiaQWiZZNJLYItWwiuUWuZRMJLqItm0hykW7Z
RKKLgMsmkl1mXIJcBpVLssucS6DLmHQJdRmzLsEuY9ol3GXMuwS8jImXkJcx
8xL0Mqzewl6G3MsEvgzJlwl9GbIvE/wypF8m/GXIt0yBLI8l8iaSWWRZNpHQIs6yiaQWiZZNJLYItWwiuUWuZRMJLqItm0hykW7Z
```

Figure 1.4. Source code for spy processes in uuencode format.

```
void subbytes(__m128i *x) {
  __m128i lo, hi, t0, t1;
  lo = _mm_and_si128(mask,*x);
  hi = _mm_and_si128(mask,_mm_srli_epi64(*x,4));
  t0 = t1 = _mm_xor_si128(t0,t0);
  for(int i=0; i<16; i++, t1 = _mm_add_epi8(t1,one))
    t0 = _mm_or_si128(t0,_mm_and_si128(_mm_cmpeq_epi8(hi,t1),_mm_shuffle_epi8(S[i],lo)));
  *x = t0;
}
```

Figure 1.5. C code with SSSE3 intrinsics for a 16-way parallel 8 to 8-bit lookup to resist cache-timing attacks.

In contrast to traditional cryptanalysis that targets a mathematical abstraction of a cryptographic primitive, side-channel analysis is a cryptanalytic technique that targets the implementation of said primitive. Timing attacks exploit a side-channel consisting of elapsed time measurements: for example, the duration required for a primitive to produce its output given an arbitrary input. Cache-timing attacks exploit the fact that the varying latency of data load instructions is essentially governed by the availability of said data in the microprocessor's data cache. This dissertation contains a number of contributions related to side-channel attacks, timing attacks, and cache-timing attacks: from novel high-dimension side-channel signal processing techniques to four devised and implemented attacks against OpenSSL, arguably the most ubiquitous cryptographic software library in use today.

Aalto-DD 136/2011

9 789526 044163

ISBN 978-952-60-4416-3
ISBN 978-952-60-4417-0 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Information and Computer Science
www.aalto.fi