UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

Tools for the Evaluation and Choice of Countermeasures against Side-Channel Attacks

PHD THESIS

Goran Doychev

DEPARTAMENTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERIA DE SOFTWARE

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

UNIVERSIDAD POLITÉCNICA DE MADRID

Tools for the Evaluation and Choice of Countermeasures against Side-Channel Attacks

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF:
Doctor of Philosophy in Computer Science

Author: **Goran Doychev**
M.Sc. in Computer Science, Saarland University

Advisor:

Prof. Dr. Boris Köpf	IMDEA Software Institute

Thesis Committee:

Prof. Dr. Somesh Jha (Chair)	University of Wisconsin
Prof. Dr. Manuel Carro Liñares (Secretary)	Universidad Politécnica de Madrid
Prof. Dr. Juan Caballero	IMDEA Software Institute
Dr. François Dupressoir	IMDEA Software Institute
Prof. Dr. Jens Grossklags	The Pennsylvania State University

2016

Abstract

Side-channel attacks have been successful in breaking cryptographic protections of systems, by using secret-dependent variations of non-functional properties such as timing or traffic volume. Countermeasures against side-channel attacks usually attempt to eliminate or reduce these variations, which may lead to performance penalties such as increases in the running time of programs, or in the traffic volume they induce. This thesis investigates the trade-off between the security of side-channel countermeasures, and their cost in terms of performance penalties. For this, we seek rigorous answers to two research questions:

Q1: How to *choose* a balance between the security guarantees and the performance penalties of side-channel countermeasures?

Q2: How to *measure* the security of side-channel countermeasures on *practical systems*?

This thesis develops tools that enable the security quantification and the choice of practical countermeasures against side-channel attacks. These tools include the necessary formal models, as well as algorithms and software tools to allow the automatic evaluation of practical systems.

In addressing Q1, we develop the first systematic approach for choosing side-channel countermeasures. We do this in a game-theoretic model, where a defender chooses a protection against an adversary who performs an attack. We apply this approach for reasoning about countermeasures against timing attacks, i.e., attacks where an adversary can exploit secret-dependent execution time of programs. We identify cases where leaky countermeasures are preferable to leak-free, constant-time implementations, as they offer better performance without sacrificing security.

In addressing Q2, we develop the first tools for the automatic formal quantification of the security of side-channel countermeasures in practical systems. We do this for two types of attacks: cache attacks, where an adversary exploits secret-dependent timing differences due to the use of the CPU cache, and web-traffic attacks, where an adversary exploits secret-dependent differences in the volume of encrypted traffic.

To capture cache attacks, we develop the tool CacheAudit, which performs static analysis of x86 binaries, and quantifies their security with respect to cache adversaries. Using CacheAudit, we analyze implementations of AES from the PolarSSL library, as well as of the finalists of the eSTREAM stream cipher competition, and we reason about the effects of architectural features such as cache size and replacement policy to side-channel leakage. Furthermore, we devise novel techniques that provide support for bit-level and symbolic reasoning about pointers in the presence of dynamic memory allocation, which we apply for reasoning about the effectiveness of several widely deployed side-channel countermeasures from the libgcrypt and OpenSSL libraries.

To capture web-traffic attacks, we develop scalable algorithms that enable the formal quantification of web-traffic leakage, as well as the generating of provable protections. We apply these algorithms on practical instances of web applications.

Resumen

Los ataques de canal lateral han sido utilizados con éxito para romper sistemas protegidos criptográficamente. Dichos ataques explotan variaciones en propiedades no funcionales que dependen de la clave secreta, como por ejemplo variaciones en el volumen de tráfico web o en el tiempo de ejecución de un programa. Como protección ante estos ataques de canal lateral, normalmente se intenta eliminar o reducir dichas variaciones, lo que puede empeorar la eficiencia, por ejemplo aumentando el tiempo de ejecución de los programas o el volumen de tráfico que producen. En esta tesis se investiga cómo encontrar un balance entre seguridad contra estos ataques y coste en términos de eficiencia. Para ello, intentamos dar una respuesta rigurosa a dos preguntas de clave:

P1: ¿Cómo *elegir* protecciones contra un canal lateral? Es decir, ¿cuál es un buen balance entre seguridad y eficiencia?

P2: ¿Cómo *medir* la seguridad de dichas protecciones contra canales laterales en *sistemas reales*?

En esta tesis se desarrollan herramientas que permiten cuantificar la seguridad y elegir protecciones prácticas contra ataques de canal lateral. Estas herramientas se basan tanto en modelos formales como en algoritmos y software que permiten el análisis automático de sistemas reales.

Para contestar a P1, hemos desarrollado un método para elegir protecciones contra canales laterales de forma sistemática. Para ello utilizamos un modelo de teoría de juegos, en el que un defensor elige una protección contra un adversario que intenta llevar a cabo un ataque. Hemos aplicado este modelo para prevenir ataques de tiempo, es decir, ataques en los que un adversario puede deducir información sobre la clave secreta midiendo el tiempo de ejecución de programas, ya que existe una dependencia entre ambos. Hemos encontrado casos en los que permitir ataques de tiempo es preferible a implementaciones en tiempo constante (que son completamente seguras ante estos ataques), ya que se consigue mejor eficiencia sin sacrificar seguridad.

En lo referente a P2, hemos desarrollado las primeras herramientas para cuantificar automática y formalmente la seguridad de protecciones contra ataques de canal lateral. Distinguimos entre dos tipos de ataque: ataques de caché, en los que un adversario explota las diferencias de tiempo provocadas por el uso de la caché de CPU; y ataques sobre el volumen de tráfico web, en los que un adversario explota las diferencias de volumen de tráfico encriptado.

Para analizar ataques de caché, hemos desarrollado la herramienta CacheAudit, que a través de un análisis estático de binarios x86 cuantifica la seguridad de éstos contra ataques de este tipo. Utilizando CacheAudit, hemos analizado implementaciones de AES de la librería PolarSSL, así como los esquemas finalistas de la competición de cifrados en flujo eSTREAM. Además, hemos analizado los efectos de diferentes características dependientes de la arquitectura, como el tamaño de la chaché o las políticas de reemplazo. Incluso, hemos ideado nuevas técnicas que proporcionan soporte para razonamiento simbólico (a nivel de bit) de punteros en el caso de asignación

dinámica de memoria. Aplicando estas técnicas, hemos analizado la efectividad de protecciones ampliamente extendidas y utilizadas de las librerías libgcrypt y OpenSSL.

Para analizar ataques sobre el volumen de tráfico web, hemos desarrollado algoritmos eficientes que permiten cuantificar de manera formal el posible filtramiento de información debido al volumen de tráfico, así como proporcionar protecciones confiables. Hemos aplicado estos algoritmos en ejemplos prácticos de aplicaciones web.

Acknowledgements

I am mostly grateful to my advisor Boris Köpf, for a number of reasons. First, for seeing a potential in me early on, and encouraging me to pursue a doctorate. Second, for being patient with me, for providing guidance in my endeavors, and for giving priceless advice. Third, for teaching me his views on work ethic, on quality research, on career growth, and for being more of a friend than a boss. It has been a great honour working with you! I would also like to thank my co-authors Laurent Mauborgne, Jan Reineke, Dominik Feld, Michael Backes, as well as my collaborators Ignacio Echeverría and Guillermo Guridi, for sharing their invaluable expertise, and making useful contributions to the projects we have been developing. I am grateful to François Dupressoir, who provided me with detailed feedback on early thesis drafts, as well as to Miguel Ambrona, who helped with the Spanish translation of the thesis abstract and gave useful feedback on my thesis draft. I want to thank the anonymous reviewers of our papers for pointing out weaknesses in our approaches, and for pushing us into making stronger contributions. The IMDEA Software Institute has provided a great environment for work, from the researchers who have been doing inspirational research, to the staff who have made sure that all aspects of our stay there have been taken care of.

On a personal note, I am deeply indebted to my girlfriend Elissaveta for putting up with me following my passion in a different country, for being there for me in the darkest hours, for not stopping giving all her love and support. This thesis is for you! I would also like to thank my parents Nelly and Valentin, as well as my sister Cveta, for supporting me throughout this journey. I am grateful to my grandmother Nadezhda, who was encouraging my education ever since I was six, who taught me to value work well done and the goodness in people, who would have been so proud of me, as always. I want to thank my friends Alexi, Nadya, Violeta, and Milan, for staying close to me despite the distance, and Gantcho, for being such a great support to me during my frequent stays in Berlin. Almost all my time away from research was spent travelling together with Gergana and Sergey; thanks for always being such awesome travel companions, and for not condemning my choice for research over personal life. I want to apologize to all those who I have ignored while being busy with the development of my thesis: you have always stayed close to my heart, no matter how infrequently I contacted you! I was happy to make a lot of good friends while being in Madrid: Antonio, Srdjan, Platon, Miguel, Germán, Julián, Juan Manuel, Natalia, Artem, José Miguel, et al.; thanks for providing me with the best possible environment for balancing work and life, for countless discussions, lunches, beers, jokes.

Madrid, May 28, 2016

Contents

1 Introduction

Side-channel attacks are a class of cyber-attacks that exploit information revealed by non-functional properties of implementations. Usually, side-channel attacks target cryptographically-secured or privacy-sensitive systems. Examples for non-functional properties used by side-channel attacks, and the attacks' goals, are:

- programs' execution times [1], devices' power consumption [2], or devices' electromagnetic radiation [3], used for recovering cryptographic keys;
- the volume of encrypted network traffic, used for revealing web browsing patterns [4], as well as the content of VoIP conversations [5];
- noise produced by keyboards, used for revealing the sequence of pressed keys [6], as well as noise produced by printers, used for revealing the content of printed text [7].

This thesis performs a rigorous study of countermeasures against three types of attacks, which we present in the following.

Timing Attacks Classical timing attacks are attacks that exploit differences in the total execution time of programs. The first timing attacks against cryptosystems were demonstrated in 1996 by Kocher [1]. Brumley and Boneh [8] demonstrate remote timing attacks, which extract private keys of an OpenSSL-based remote web server. In general, targets for timing attacks have been implementations with input-dependent execution times, such as cryptosystems in which the execution time depends on the secret key. An example of input-dependent execution time is exhibited by the square-and-multiply algorithm for modular exponentiation (see Figure 1.1), used in popular public-key cryptosystems such as RSA and ElGamal. The algorithm goes through the bits of the exponent and performs a (slow) multiplication only for bits equal to 1. Measuring the total exponentiation time can reveal the number of times a multiplication was performed, thus revealing the number of set bits of the exponent (i.e., the exponent's Hamming weight).

```
1  // Input: b, e, m
2  // Output: b^e  mod m
3  r := 1
4  for i := |e| − 1 downto 0 do
5      r := sqr(r)
6      r := mod(r, m)
7      if e_i = 1 then
8          r := mul(b, r)
9          r := mod(r, m)
10     return r
```

Figure 1.1: Pseudocode for square-and-multiply modular exponentiation

Cache-Timing Attacks Cache-timing attacks[1] are attacks that exploit differences in the time that memory accesses take, due to the presence or absence of data in the CPU cache. These differences can be detected e.g. by an adversary who shares the CPU cache with the victim process, and concurrently probes the cache. The first cache attacks against cryptosystems were presented in 2005 [9, 10, 11]. Cache attacks for key recovery have been demonstrated to a remote server [12], as well as between co-located virtual machines [13, 14]. Oren et al. demonstrate in-browser cache attacks [15], which infer mouse and network activity of users who have loaded malicious JavaScript code. In general, targets for cache attacks have been implementations that perform input-dependent cache accesses. Vulnerable input-dependent cache accesses may be due to input-dependent lookups in precomputed lookup tables, e.g. in implementations of AES [9, 11], or due to input-dependent control-flow, e.g. in implementations of the square-and-multiply algorithm [13, 14]. An example of input-dependent cache accesses is given in Figure 1.2, which shows the compiled code corresponding to the conditional if-branch in square-and-multiply (see Figure 1.1, lines 7–9). If the input-dependent if-branch is taken, a high number of memory accesses is performed; an adversary able to detect these cache accesses, can determine the value of the exponent.

Web-Traffic Attacks Web-traffic attacks exploit differences in the volume of encrypted network traffic. In these attacks, the adversary inspects the sizes and numbers of network packets, which allows the recovery of information about the user's web-browsing behavior. Early web-traffic attacks were demonstrated by Cheng and Avnur in 1998 [16]. Web-traffic attacks have used passive analysis of encrypted traffic for website fingerprinting [17, 18], i.e., revealing which websites were visited by users; for reconstruction of the execution path in web applications, which may reveal health or financial information about users [4]; for deanonymization of Tor hidden services [19].

[1] Throughout this thesis, we use the wide-spread namings of *cache attacks* for cache-timing attacks, and *timing attacks* for classical timing attacks exploiting the total execution time.

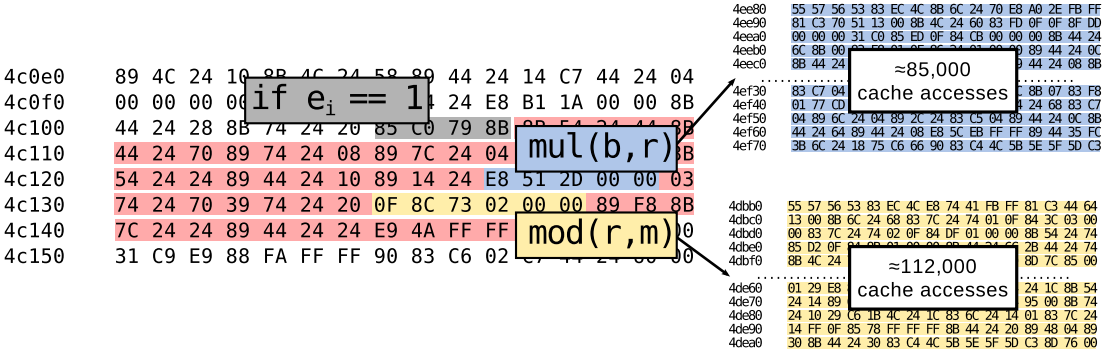


Figure 1.2: Code from the square-and-multiply modular exponentiation implementation from libgcrypt 1.5.2 with a 3072-bit base, compiled with gcc. The highlighted code is executed in case the if-branch is taken. This code contains calls to the functions `mul` and `mod`, and results in almost $2 \cdot 10^5$ cache accesses.

Furthermore, active web-traffic attacks have been demonstrated that allow recovering values from secure cookies [20, 21]. Web-traffic attacks are possible because despite encryption, the sizes of objects transmitted over the network may reveal their content. For example, input-fields with an auto-complete functionality are usually implemented by sending a query after each key-stroke, and downloading lists of suggestions for that query. These lists have different sizes, depending on the number of characters they contain. Figure 1.3 shows the byte-sizes of such lists corresponding to the first typed character in a query; observing sizes corresponding to a sequence of typed characters can reveal the content of the user's queries.

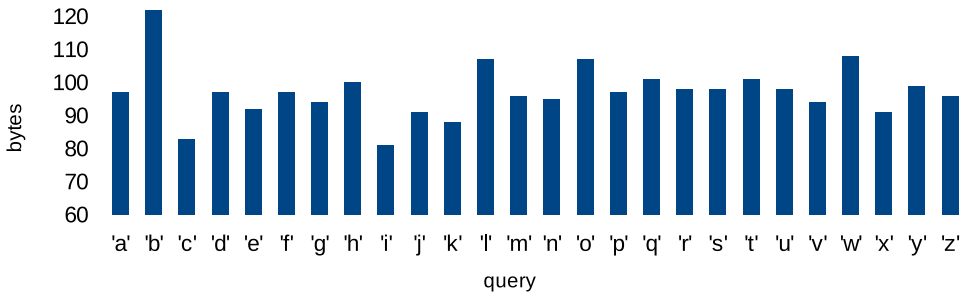


Figure 1.3: The sizes of HTTP frames downloaded after typing one letter into an auto-complete input field, using the Google Autocomplete API. The size depends on the number of characters in the downloaded list of suggestions.

1.1 Trade-offs in Side-Channel Protection

Diverse mechanisms for protecting against side-channel attacks have been proposed, ranging from changes in the underlying software, through changes in the operating

systems (e.g., [22]), deployment of specific network proxies (e.g., [23, 24]), to changes in hardware (e.g., [25, 26]).

A practical example for hardware-based protections against cache attacks is the AES instruction set (AES-NI), which was announced by Intel in 2009 [26, 27]. AES-NI allows implementing cache-attack resistant implementations that run faster than purely software implementations of AES. Despite the attractive features of such hardware-based protections, they cannot be considered a universal solution for the immediate future because of their slow roll-out (AES-NI was still missing in some Intel CPUs released in 2015 [28]), as well as the prevalence of legacy hardware.

In this thesis, we focus on software-based protections against side-channel attacks. Software-based protections are the most immediate path between the design of protections and their deployment: software patches to existing software packages offer the possibility of fixing security problems in a matter of days after their discovery. A major challenge in the implementation and deployment of such protections is that there is usually a conflict between two goals that the protections aim to achieve:

(1) the protection should eliminate the *security* problem, i.e., make the side-channel attack impossible;

(2) the protection should not harm the *performance* of the system.

To illustrate this conflict, we consider two extreme cases. First, goal (2) can be trivially met if no dedicated side-channel protection is applied; this will leave the system exposed to potential attacks. Second, goal (1) can be met by eliminating the root cause of the side-channel. Straightforward approaches to turn a leaky implementation into a leak-free implementation are: disable the cache to eliminate cache attacks; make a program return in the worst-case execution time to eliminate timing attacks; transmit constant traffic to eliminate web-traffic attacks. Such straightforward approaches may induce heavy performance penalties.

An active research area in cryptography is the development of fast side-channel-free implementations of cryptographic systems, e.g., see [29, 30, 31, 32, 33]. These works aim at implementing algorithms that are immune to timing-based attacks by ensuring that there are no secret-dependent timing variations, e.g. by avoiding secret-dependent control flow or table lookups. However, even very efficient side-channel-free implementations may offer worse performance than vulnerable implementations, which may hamper their adoption. For example, the constant-time implementation of AES-GCM authenticated encryption by Käsper and Schwabe is twice slower than non-constant-time, lookup-table-based implementations [29].

Often, instead of eliminating possible side-channel leaks, they are being *mitigated*. This is done by applying a protection that defeats some known side-channel attacks, or makes attacks harder for adversaries, without offering a proof that the protection defeats *all* side-channel attacks of a certain class. The result is a system that is believed to be less vulnerable to attacks, in which performance penalties are limited. For example, Kocher [1] proposed the use of *blinding* for protecting against timing attacks

against modular exponentiation: input blinding (randomizing the base), or exponent blinding (randomizing the exponent). While input blinding has seen a wide adoption in practice [34, 35, 36], it has not been shown that it eliminates all timing attacks; as for exponent blinding, there are indications that it does not eliminate timing attacks, but makes them harder for adversaries [37].

1.2 Research Questions

In the presence of performance penalties introduced by side-channel protections, when deploying a protection, both the security and the performance play a role in deciding in favor of a protection. This leads to our first research question:

Q1: How to *choose* a balance between the security guarantees and the performance penalties of side-channel countermeasures?

While numerous side-channel countermeasures have been proposed, systematic approaches for making a decision in favor of one such countermeasure have not been, to the best of our knowledge, the subject of theoretical or practical research.

In order for a systematic choice between two (or more) protections to be made, a way of measuring and comparing the security and the performance of systems is needed. In this thesis, we measure performance in terms of average execution time (in the case of timing attacks), and average network traffic consumption (in the case of web-traffic attacks). The need for measuring security leads to our second research question:

Q2: How to *quantify* the security of side-channel protections on *practical systems*?

We favor methods for a formal quantification of the security of systems. These methods come with a proof, which is valid for clearly defined models. For quantifying the security of side-channel protections, we rely on models and methods for quantitative information flow (QIF) analysis [38]. QIF analysis usually uses information-theoretic metrics to quantify the information that flows from a secret input to a public output. In the context of this thesis, the secret input is either a cryptographic key or a user's browsing pattern, and the public output is the side-channel observations, e.g. a program's execution time, a cache state, a sequence of memory blocks loaded into cache, or a sequence of byte-sizes of encrypted network packets. QIF analysis has been used for quantifying side-channel leaks [39], and is particularly appealing because it has shown potential for automation [40,41,42]. Prior to the work presented in this thesis, it had been an open question whether QIF analysis of side-channel leaks was possible on practical security-critical systems, i.e., whether it can be scalable enough to capture practically deployed software such as implementations of popular cryptographic algorithms.

This thesis develops tools that enable rigorous answers to the two research questions. These tools include the necessary models needed for formal reasoning about side-channel protections, as well as algorithms and software tools needed for the automatic evaluation of practical systems.

1.3 Contributions

The contributions of the thesis are summarized in the following.

Modeling Side-Channel Adversaries We develop novel models that enable the reasoning about several types of practically relevant side-channel adversaries. First, our models capture timing adversaries who can perform both online steps (i.e. collect side-channel measurements), and offline steps (i.e. perform a computational attack). Second, our models capture cache adversaries of several types, among which access-based adversaries who can observe the final cache state, and trace-based adversaries who can observe sequences of accessed memory addresses or memory blocks. Third, our models capture web-traffic adversaries who can observe traffic corresponding to sequences of visited webpages. We apply these models for addressing the two research questions introduced in Section 1.2.

Systematic Choice of Protections Addressing Q1, we develop the first systematic approach for choosing side-channel countermeasures. For this, we cast the problem in a game-theoretic model, where a defender chooses a protection against an adversary who performs an attack. Both the defender and the adversary are assumed to be rational, i.e., they seek to optimize their personal utilities. We apply this approach for reasoning about timing attack countermeasures, and we identify cases where leaky countermeasures are preferable to leak-free, constant-time implementations, as they offer better performance without sacrificing security. This result is the first formal argument that favors a leaky side-channel protection over a constant-time implementation.

Quantification of Security in Practical Systems Addressing Q2, we develop the first tools for the automatic formal quantification of the security of side-channel protections in practical systems. To enable the analysis of protections against cache attacks, we develop the tool CacheAudit, which performs static analysis of x86 binaries, and quantifies their security with respect to cache adversaries. CacheAudit enables us to perform several case studies, where we analyze implementations of symmetric cryptosystems from libraries such as PolarSSL and NaCl, and reason about the effectiveness of widely-deployed countermeasures against side-channel attacks on asymmetric cryptosystems from the libgcrypt and OpenSSL libraries. To enable the analysis of protections against web-traffic attacks, we develop scalable algorithms that enable the analysis and protection of web applications, and we demonstrate these algorithms on practical instances of web applications.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we present background on the information-theoretic notions we use for quantifying side-channel leakage, and review related work. The exposition of the thesis contributions is presented in Chapters 3 to 7, which are divided into three parts, according to the type of side-channel attacks we address: in Part I we address timing attack protection, in Part II we address cache attack protection, and in Part III we address web-traffic attack protection. The thesis concludes in Chapter 8.

In Part I, which consists of Chapter 3, we demonstrate a systematic approach for choosing protections against timing attacks. The choice is a solution in a two-staged game between a defender and an adversary. In this game, first the defender chooses a protection, and then the adversary performs an attack combining online steps (collecting side-channel measurements), and offline (computational) steps. The offline steps of the adversary are captured in the generic group model, and the combination between computational and side-channel steps is done relying on unpredictability entropy, a computational entropy notion (see Chapter 2). We demonstrate a practical evaluation of our approach, on an implementation of the ElGamal encryption scheme from libgcrypt.

Part II is organized in three chapters. In Chapter 4 we provide the necessary background and formalization on caches, static analysis, and cache side-channel quantification. In Chapter 5 we present CacheAudit, a tool that enables the formal quantification of cache leakage from x86 executables, by performing static analysis based on abstract interpretation. CacheAudit supports cache characteristics such as the LRU, FIFO, PLRU replacement policies, and allows the analysis of executables with respect to adversaries with different observational capabilities: observing the final cache state, a sequence of cache hits or misses, or the total execution time. Using CacheAudit, we analyze the implementation of AES from PolarSSL (now mbed TLS), as well as the finalists of the eSTREAM stream cipher competition. In Chapter 6 we present a tool built as an extension to CacheAudit, which enables the static analysis of leaks from dynamic heap memory, and captures adversaries who can make fine-grained observations, e.g. observing sequences of accessed memory addresses or memory blocks. We use this tool to analyze side-channel countermeasures from implementations of modular exponentiation in different versions of libgcrypt and OpenSSL.

In Part III, which consists of Chapter 7, we present algorithms for dealing with web-traffic attacks. We present an algorithm for the efficient quantification of the information leaked to an adversary observing traffic patterns corresponding to sequences of accessed web pages; furthermore, we present an algorithm for generating protections against web-traffic attacks. The generated protections span a specter of trade-offs between the security guarantees and the performance overhead. We use these algorithms to evaluate a regional-language Wikipedia, and an auto-complete input field.

1.5 Thesis Publications

The content of this thesis is based on several publications. These publications are the result of collaborative effort, and I have contributed significant parts in them.

- Chapter 3 is based on publication [43], which is joint work with Boris Köpf, and was published in the proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF) in 2015.
- Chapter 4 and Chapter 5 are based on publication [44], which is joint work with Dominik Feld, Boris Köpf, Laurent Mauborgne and Jan Reineke, and was published in the proceedings of the 22nd USENIX Security Symposium in 2013, and on its significantly extended version [45], which was published in the ACM Transactions on Information and System Security (TISSEC) in 2015.
- Chapter 6 is based on publication [46], which is joint work with Boris Köpf, and at the time of writing of this thesis is under submission.
- Chapter 7 is based on publication [47], which is joint work with Michael Backes and Boris Köpf, and was published in the 20th Network and Distributed Systems Security Symposium (NDSS) in 2013. This chapter is a continuation of the research the author published in his Master thesis [48], which presents models for formal reasoning about web leaks.

2 Background and Related Work

In this chapter, we start by presenting background on the information-theoretic notions used in this thesis for quantifying the security of side-channel protections. Furthermore, we present related work.

2.1 Information-Theoretic Notions

For quantifying the security of side-channel protections, throughout this thesis we use three different information-theoretic entropy notions: Shannon entropy, min-entropy, and unpredictability entropy. The first two have seen a wide use in quantitative information flow (QIF) literature [38, 39, 49]; the last one is an entropy notion that captures computationally bounded adversaries [50]. In this section, we present the three entropy notions.

2.1.1 Entropy Definitions

Information-theoretic entropy is used for quantifying the amount of randomness of a probability distribution. Consider the random variables X and Y. The *entropy of* X quantifies the randomness of the probability distribution $P[X]$, and the *conditional entropy of* X *given* Y quantifies the randomness of the conditional probability distribution $P[X|Y]$. In the context of quantitative information flow, we model a secret value (e.g., a cryptographic key) as the random variable X, and an observable value (e.g., a side-channel observation) as the random variable Y. We use the entropy of X as a quantitative measure of the a-priori uncertainty about the value of the secret, and we use the conditional entropy of X given Y as a quantitative measure of the uncertainty about the secret value, when the observable value is known.

The first information-theoretic notion we consider is *Shannon entropy*, which was proposed by Claude Shannon in 1948, as part of a general theory of communication [51].

Definition 1 (Shannon entropy)**.** *The* Shannon entropy *of X is defined as*

$$H(X) = -\sum_x P[X = x] \log_2 P[X = x] \,.$$

The conditional entropy $H(X|Y)$ *of X given Y is defined as*

$$H(X|Y) = \sum_y P[Y = y] H(X|Y = y) \,.$$

Shannon entropy is interesting as a measure of confidentiality because one can use it to give a lower bound for the expected number of steps that are required for determining the value of X by brute-force search. Observe that the optimal strategy for this is to try all values of X in order of their decreasing probabilities. For this, let X be indexed accordingly: $P[X = x_1] \geq P[X = x_2] \geq \ldots$. Then the expected number of guesses (also called *guessing entropy* [52]) required to determine X is defined as $G(X) = \sum_{1 \leq i \leq |X|} i\, P[X = x_i]$, and the conditional version $G(X|Y)$ is defined in the natural way [53]. The following result [52, 53] bounds $G(X|Y)$ in terms of Shannon entropy.

Proposition 1. $G(X|Y) \geq \frac{1}{4} 2^{H(X|Y)}$

Note however that for heavily skewed X, the *expected* number of guesses required to determine X can be large, even if an attacker has a considerable chance of correctly guessing the value of X in one attempt. To see this, consider a random variable X distributed by $P[X = x_1] = \frac{1}{2}$ and $P[X = x_i] = \frac{1}{2n}$, for $i \in \{2, \ldots, n\}$: the expected number of guesses to determine X grows linearly with n, but the probability of correct guessing in one shot remains $\frac{1}{2}$.

The *min-entropy* H_∞ accounts for such scenarios by capturing the probability of correctly guessing the secret in *one* attempt, i.e., it delivers worst-case rather than average-case guarantees.

Definition 2 (Min-entropy)**.** *The* min-entropy *of X is defined as*

$$H_\infty(X) = -\log_2 \max_{x \in \mathcal{X}} P[X = x].$$

The conditional min-entropy *of X given Y, is defined as*

$$H_\infty(X|Y) = -\log \sum_{y \in \mathcal{Y}} P[Y = y] \max_{x \in \mathcal{X}} P[X = x|Y = y].$$

As discussed above, the entropy definitions 1 and 2 can be used for reasoning about the ability of adversaries to guess the value of random variables. Those definitions capture adversaries with unbounded computational capabilities; however, cryptography considers probability distributions that *appear to be* random to computationally bounded adversaries, e.g. pseudo-random distributions [54].

Computational entropy notions [50] allow reasoning about the apparent randomness of probability distributions. We use one such entropy notion – *unpredictability entropy*,

which generalizes min-entropy to computational settings. More specifically, it captures the probability of a resource-bounded algorithm to estimate the value of a random variable. Unpredictability entropy has been defined for different computational models, such as circuits of bounded size [50] and polynomial time algorithms [55].

In this thesis, we consider computational models in which computational steps are modeled as queries to an oracle [56, 57]. For example, brute-force guesses can be modeled as queries to an oracle which outputs `true` iff the guess is correct. We define unpredictability entropy w.r.t. bounds on the number of accesses to an oracle $O(X)$ that receives as input the value of a random variable X.

Definition 3 (Unpredictability entropy)**.** *X has* unpredictability entropy at least t bits *for m calls to O, written $H^m_O(X) \geq t$, if for all algorithms $\mathsf{A}^m_{O(X)}$ that can make at most m queries to $O(X)$,*

$$P[\mathsf{A}^m_{O(X)} = X] \leq 2^{-t} .$$

Note that for $m = 0$, unpredictability entropy and min-entropy coincide, i.e., $H^0_O(X) = H_\infty(X)$. The reason for this is that the best algorithm for predicting X is one that has the most likely value of X hardcoded.

Next we define a conditional version of unpredictability entropy. For this we consider algorithms $\mathsf{A}^m_{O(X)}(Y)$ that can observe the output of a random variable Y that is jointly distributed with the oracle input X.

Definition 4 (Conditional unpredictability entropy)**.** *X has* unpredictability entropy at least t conditioned on Y *with respect to oracle O, written $H^m_O(X|Y) \geq t$, if for all algorithms that can take one sample from Y and can make at most m queries to $O(X)$,*

$$P[\mathsf{A}^m_{O(X)}(Y) = X] \leq 2^{-t} .$$

2.1.2 Chain Rules

Key properties for reasoning about entropy are its *chain rules*. Shannon entropy satisfies a strong chain rule, which establishes the connection between the conditional and the joint entropy.

Lemma 1 (Strong chain rule for Shannon entropy)**.**

$$H(X|Y) = H(X, Y) - H(Y) .$$

While this chain rule is not satisfied by the remaining entropy notions we consider, weaker chain rules exist, which find a broad use. Those chain rules are sufficient to show that when conditioned on Y, the entropy of X decreases gracefully. Below, we give such chain rules for Shannon, min-, and unpredictability entropy, and we enclose the proof of the latter. Note that the chain rule is not necessarily satisfied for other entropy notions, such as HILL entropy [58].

Lemma 2 (Chain rule for Shannon entropy)**.**

$$H(X|Y) \geq H(X) - |\,ran(Y)|\ .$$

Lemma 3 (Chain rule for min-entropy)**.**

$$H_\infty(X|Y) \geq H_\infty(X) - |\,ran(Y)|\ .$$

Lemma 4 (Chain rule for unpredictability entropy [58])**.** *If* $\log_2 |\,ran(Y)| = \ell$*, then*

$$H^m_O(X) \geq t \Rightarrow H^m_O(X|Y) \geq t - \ell\ .$$

Proof. Assume that $H^m_O(X|Y) < t - \ell$. By the definition of unpredictability entropy it follows that there exists an algorithm A^m_O such that $P[\mathsf{A}^m_{O(X)}(Y) = X] > 2^{-t+\ell}$. We construct the algorithm B^m_O, which takes no input, chooses $y' \in ran(Y)$ u.a.r. (which we represent by the random variable Y'), and returns $\mathsf{A}^m_{O(X)}(y')$. Then we obtain

$$\begin{aligned}
P[\mathsf{B}^m_{O(X)} = X] &= P[\mathsf{A}^m_{O(X)}(Y') = X] && (2.1)\\
&= P[\mathsf{A}^m_{O(X)}(Y) = X \wedge Y = Y'] && (2.2)\\
&= P[\mathsf{A}^m_{O(X)}(Y) = X] \cdot P[Y = Y'] && (2.3)\\
&= P[\mathsf{A}^m_{O(X)}(Y) = X] \cdot 2^{-\ell} && (2.4)\\
&> 2^{-t+\ell} \cdot 2^{-\ell} = 2^{-t}\ , && (2.5)
\end{aligned}$$

where (2.3) follows because both events are independent, i.e., the fact that Y collides with a uniformly chosen element does not affect the probability that the algorithm guesses X correctly, and vice versa. □

2.2 Related Work

Quantitative Information Flow Analysis

Early work on quantitative information flow (QIF) analysis by Denning [59] uses information-theoretic definitions to quantify information transferred between variables in different states of program execution. More recent works provide definitions to quantify the transformations from system inputs to system outputs [38], which have been used for statically bounding secret information that leaks in a program [60]. Such definitions have been used for quantifying leaks in anonymity protocols [61]. The works cited above quantify flows in terms of Shannon entropy; further works on QIF analysis use min-entropy [49, 62], and Alvim et al. [63] propose a generalization of min-entropy which allows flexible definitions of the adversary's gain [63].

QIF analysis for reasoning about side-channels was introduced by Köpf and Basin [39]. Earlier work exists on the quantification of the related but distinct problem of covert

channels [64, 65], i.e., where an adversary aims to transmit information from within a system.

This thesis directly builds upon the results of two related papers. First, Köpf and Dürmuth [53] is a starting point of the development in Chapter 3. The authors prove bounds on the information leaked by a cryptosystem protected from timing attacks using input blinding and bucketing as a countermeasure, where tuning the bucketing countermeasure leads to a trade-off between the security and performance of the protection. We connect those leakage bounds with cryptographic models of computation [56] to capture adversaries who combine side-channel information with computational steps. Further, the authors propose algorithms for optimal choice of the bucketing when given a constraint on the overhead, or on the desired security. We use those algorithms as part of our more general approach for choosing a protection: in addition to tuning the bucketing countermeasure, our defender chooses the size of the key, and is opposed to a rational adversary who chooses an optimal attack.

Second, the development of CacheAudit (see Chapter 5) is inspired by a feasibility study by Köpf, Mauborgne, and Ochoa [66]. They quantify cache side-channels by connecting a commercial, closed-source tool for the static analysis of worst-case execution times [67] to an algorithm for counting concretizations of abstract cache states. The application of the tool to side-channel analysis is limited to access-based adversaries and requires heavy code instrumentation. In contrast, CacheAudit was built with cache attacks in mind, and while the results are not directly comparable because they are obtained for different platforms (x86 vs. ARM), the results of our experiments suggest that the bounds provided by CacheAudit are significantly tighter. CacheAudit also provides tailored abstract domains for various kinds of cache side-channel adversaries, and its modularity allows the support of various architecture features such as the PLRU replacement policy, and it is open for further extensions. In contrast to [66], analysis of binaries is possible without any code instrumentation.

Our work is related to other works on automation of QIF analysis [60], and the automation by reduction to counting problems appears in [40, 42, 68, 69]; the connection to abstract interpretation appears in [41]. Approaches that can deal with adversarially chosen input [39, 70] are currently limited to systems with small state-spaces due to the lack of efficient abstractions. With progress on such abstractions, bounds such as the ones from Chapter 3 could be obtained automatically from code and platform models using CacheAudit or similar analyses (see Chapter 5).

McCamant and Ernst [71] develop a tool that uses dynamic taint analysis to bound information leakage along an execution path. They demonstrate their approach on practical programs, among which an SSH client performing an authentication. In contrast to this work, we perform static analysis that considers all execution paths.

Mardziel et al. [72] consider information-flow in dynamic systems, where defender and adversary can interact. Their focus is on secrets that are dynamically changing, whereas our approach specifically considers the aggregation of information about long-term secrets, such as secret keys.

Zhang et al. [73] study quantitative approaches for mitigating timing leaks by

adaptively delaying outgoing messages. They consider a covert channel adversary, i.e. one that aims to transmit information from within a system. This adversary is stronger than, and the corresponding notion of success (i.e. successful transmission) is different from, the ones we consider in this thesis.

Game Theory and Security Trade-Offs

A substantial body of research uses game theory for reasoning about trade-offs between security (or privacy) and conflicting goals. Stackelberg games are particularly prevalent in the literature, and the fact that the adversary reacts to the defender's commitment reflects Kerckhoff's principle.

For example, [74] uses Stackelberg games for computing the optimal placement of checkpoints and canine patrol routes for achieving (physical) security, e.g., at airports. In that setting, [75] considers a defender who aims to minimize cost while maintaining a fixed level of protection, which is similar in spirit to our definitions of security. In [76], the authors use Stackelberg games to reason about the configuration of audit mechanisms, where the trade-off is between the cost of detecting or preventing incidents. Furthermore, Stackelberg games have been used for choosing an optimal balance between privacy and service quality [77, 78], e.g. in location privacy mechanisms.

Khouzani et al. [79] explore game-theoretic models for capturing trade-offs between security and usability when picking secrets. As examples, they consider picking unique passwords and picking the size of a short cryptographic key.

Work in rational cryptography (see [80] for a recent overview) also considers adversaries as players aiming to maximize their utilities. One advantage of this adversary model is that one can circumvent impossibility results that hold for stronger, worst-case adversaries. The game definition we propose in Chapter 3 is atypical in the sense that it captures worst-case adversaries who spend all resources and relies on the utility function to describe their optimal usage.

Leakage-Resilient Cryptography

Leakage-resilient cryptographic constructions have been proposed, where information leakage is limited within each iteration of the construction [81, 82]. Such constructions include pseudo-random generators and pseudo-random functions, and usually rely on re-keying strategies [83]. Most works in that direction consider physical side-channel attacks such as analysis of power and electromagnetic radiation. For reasoning about cache-based attacks, Barthe et al. [84] propose an extension to CacheAudit (see Chapter 5), which allows reasoning about leakage-resilient cryptosystems, for concurrent access-based cache adversaries.

Related to our work in Chapter 3, Belaïd et al. [85] also reason about trade-offs between security and performance in side-channel attacks, where they focus on the decision between the masking countermeasure and a leakage-resilient primitive. As in our work, they investigate which implementation offers the best performance for a fixed

level of security. They consider power analysis attacks and use the best known attacks as a security benchmark, whereas we consider timing attacks and use the best known security guarantees as a benchmark. Our approach relies on game theory for framing and solving the decision problem, which offers the advantage of a clean interface between the security guarantees and the algorithmic challenges.

Constant-Time Cryptography

Constant-time cryptographic implementations aim at securing against timing-based attacks by ensuring that there are no secret-dependent timing variations, e.g. by avoiding secret-dependent control flow or table lookups [29, 30, 31, 32, 33]. Examples for this can be found in the NaCl cryptographic library [31]. Formal approaches have been proposed for reasoning about constant-time implementations [86, 87, 88]. For example, Almeida et al. develop a static analyzer that can automatically confirm that programs follow that regime [86], and Barthe et al. [87] establish that adhering to that policy provides security against very strong adversary models. Using CacheAudit, we analyze such implementations, e.g. NaCl's Salsa20 (see Chapter 5, Section 5.6.2), as well as constant-time coding patterns for access to lookup tables (see Chapter 6, Section 6.6.4), and we establish their security with respect to several models of cache-based adversaries.

Part I

Timing Attack Protection

The first part of the thesis presents a systematic approach for choosing a protection against timing attacks, on the example of cryptosystems based on discrete logarithms. The solution relies on a game-theoretic equilibrium in a game between a defender who strives to reduce the protection costs by choosing the key size and a side-channel protection, and a resource-bounded adversary who strives to maximize the probability of key recovery by choosing an attack strategy. At the heart of the equilibrium computation are novel bounds for the probability of key recovery, which are expressed as a function of the applied protection and the attack strategy.

The approach is applied in a practical case study, where we identify optimal protections for libgcrypt's ElGamal implementation. We determine situations in which the optimal choice is to use a defensive, constant-time implementation and a small key, and situations in which the optimal choice is a more aggressively tuned (but leaky) implementation with a longer key.

The work presented in this part is the first systematic approach for obtaining an answer to research question 1 (Q1) in Section 1.2. Our main contributions are exposed in Chapter 3.

3 Rational Protection Against Timing Attacks

3.1 Introduction

Side-channel attacks break the security of systems by exploiting signals that are unwittingly emitted by the implementation. Examples of such signals are the consumption of power [2], memory [89], and execution time [1]. Execution time is a particularly daunting signal because it can be measured and exploited from a long distance [90], which opens the door for a potentially large number of attackers.

In theory, one can get rid of timing side-channels by avoiding the use of secret-dependent control flow and of performance-enhancing features of the hardware architecture, such as caches and branch prediction units. However, this defensive approach comes at the price of a performance penalty. In practice, one is hence faced with the problem of striking a balance between performance and security against timing attacks.

In this chapter we present a game-theoretic approach for solving this problem. The key novelty of our approach is a simple and practical model of the side-channel adversary as a player that can distribute the available resources between timing measurements and offline search for the secret. Our approach is focused in that we consider only cryptosystems based on discrete logarithms and input blinding as a countermeasure, yet it is comprehensive in that it goes all the way from formal modeling to identifying the optimal protection for a library implementation of ElGamal. A highlight of our results is that we are the first to formally justify the use of a fast but (slightly) leaky implementation over a defensive constant-time implementation, for some parameter ranges.

On a technical level, we identify the optimal countermeasure configuration as an equilibrium of a two-stage (Stackelberg) game between two rational players: an adversary and a defender. The adversary strives to maximize the probability of key recovery, by distributing bounded resources between timing measurements and computational search for the key. The defender strives to minimize the cost of protection, while maintaining a certain security level given in terms of an upper bound on the probability of adversary success. The defender's means to achieve this are the choice of the key

length and the configuration of the countermeasure.

At the heart of the equilibrium computation are novel bounds for the probability of key recovery in the presence of side-channel information. We derive these bounds in the generic group model and under the assumption that the cryptosystem is based on discrete logarithms and protected against timing attacks by an idealized form of input blinding.

Our starting point is an existing upper bound for the amount of information contained in n timing measurements, when the execution time is discretized [53]. The technical challenge we face is to turn this bound into a guarantee against an adversary that can mount a combined timing/algebraic attack. We identify unpredictability entropy [50] as a suitable tool for this task. In particular, unpredictability entropy satisfies a chain rule [58], which limits the extent to which bounded leakage can decrease the hardness of a computational problem. We then cast Shoup's lower bound for computing discrete logs in generic groups [56] in terms of the unpredictability entropy w.r.t. an adversary who can perform m group operations. Finally, we connect the leakage bound with the bound for the discrete log to obtain the desired bound (in terms of n and m) for combined side-channel/algebraic adversaries.

We put our approach to work in a case study where we seek to optimally configure countermeasures against timing attacks in libgcrypt's ElGamal implementation. Experimentally, we identify optimal choices of key lengths and countermeasure configurations that guarantee the same degree of security as a constant-time implementation using a reference key length. We do this for realistic server configurations, and target commonly used key lengths. In the course of our experiments we observe that the time of deployment of a key, or the number allowed connections per second can influence which configuration is optimal: the defensive, constant-time implementation with a short key, or the more aggressively tuned and leaky implementation with a longer key.

In summary, our contributions are conceptual and practical. Conceptually, we combine game theory with novel, quantitative security guarantees to tackle the problem of systematically choosing the optimal balance between security and performance. Practically, we perform a case-study on a realistic ElGamal implementation, where we illustrate how our techniques can be used to identify cost-effective countermeasure configurations.

Organization In Section 3.2, we present the countermeasure configuration game, based on parametric security guarantees. We instantiate these guarantees in Section 3.3, before we discuss their application in Sections 3.4 and 3.5. We present related work in Section 7.5, and conclude in Section 7.6.

3.2 Choice of Optimal Protection

In this section we cast the optimal configuration of a countermeasure as a game between the adversary and the defender. We conclude the section with a discussion of the effect of using safe approximations of the security of a system (instead of exact values) for the solution of the game.

3.2.1 Motivating Example

Input blinding is a widely deployed countermeasure against timing attacks on cryptosystems based on modular exponentiation. A formal security analysis of input blinding [53] shows that the amount of information about the key that is leaked by a blinded cryptosystem is bounded from above by

$$(b-1)\log_2(n+1) \tag{3.1}$$

bits, where b is the number of possible execution times, and n is the number of side-channel measurements made.

For real systems, b can be as large as the difference between the worst-case execution time and the best-case execution time (e.g., in clock ticks), in which case (3.1) does not imply meaningful guarantees. However, b can be reduced by applying *bucketing*, which is the discretization of system's execution times into intervals (*buckets*) where, for each execution time, one waits until the enclosing bucket's upper bound before returning the result of the computation.

Choosing a smaller number of buckets b leads to better security guarantees, but it also leads to a decrease in performance. Likewise, picking a larger key size leads to better security and a decrease in performance. The techniques presented in this chapter enable identifying the sweet spot in the resulting parameter space.

3.2.2 Countermeasure Configuration as a Game

We formalize the configuration of a countermeasure as a two-stage game between a defender ($\mathcal{D}$) and an adversary ($\mathcal{A}$). Similar games are known in the literature as *Stackelberg* games [91]. In the first stage, $\mathcal{D}$ chooses an element d from a finite set D of *defender actions*, which can be parameters of a protection mechanism. In the second stage, having seen d, $\mathcal{A}$ responds by choosing an element of a finite set A of *adversary actions*. We model this choice as a function $r\colon D \to A$, which captures that the adversary knows and responds to d.

If a situation is preferred by a player, we say that it has a higher *utility*. Utilities are captured in terms of real-valued functions of both player's actions:

$$u_{\mathcal{D}}, u_{\mathcal{A}}\colon D \times A \to \mathbb{R}\,.$$

Putting together all ingredients, a two-stage game is the tuple $G = [(\mathcal{D}, \mathcal{A}), (D, A), (u_{\mathcal{D}}, u_{\mathcal{A}})]$.

3.2.3 Utilities of the Players

Now we define the utilities that make up the countermeasure configuration game. For this, we rely on (upper bounds on) the probability of a breach (i.e. a successful attack), which we model as functions

$$p\colon D \times A \to [0,1]\ .$$

Throughout this section we leave p parametric; we show how to instantiate it in Section 3.3.

Defender's utility A natural notion for the defender's utility is to consider the loss l in case of a successful breach together with the cost of the defense d, i.e.,

$$-\ p(d, r(d)) \cdot l - cost_{\mathcal{D}}(d)\ . \tag{3.2}$$

In practice, this definition suffers from two problems: First, in advance it may be difficult to put a number l on the loss in case of breach. Second, the definition may give misleading results when working with *bounds* on p, as we discuss in Section 3.2.5.

To address those problems, we impose a hard upper bound $p_{\max}$ on the probability of breach, which is common practice, for example, in safety requirements [92]. We then define the defender's utility as the cost of defense under this constraint.

Definition 5. *The* defender's utility $u_{\mathcal{D}}$ *is defined as*

$$u_{\mathcal{D}}(d, r(d)) = -cost_{\mathcal{D}}(d)\ ,$$

where we require $p(d, r(d)) \leq p_{\max}$.

Example 1. *For the countermeasure described in Section 3.2.1, the defender's action is to choose a key length k and a number of buckets b, such that the adversary's chance of key recovery remains below* $p_{\max}$*:*

$$D = \{(k,b) \mid p((k,b), r(k,b)) \leq p_{\max}\}\ ,$$

where $cost_{\mathcal{D}}(k,b)$ *is the average execution time of the corresponding implementation. Here we instantiate* $p(\cdot)$ *using the bounds in Section 3.3. We rely on key length recommendations [93] for protecting against traditional (non-side-channel) adversaries as a basis for instantiating* $p_{\max}$*. Specifically, the defender has to pick the key length k and number of buckets b in such a way that the security level* with *side-channel matches that of a reference key length* k_{ref} without *side-channel.*

Adversary's utility A natural notion for the adversary's utility is the adversary's expected benefits in terms of the gain g in case of a successful breach and the cost of attack, i.e.,

$$p(d, r(d)) \cdot g - cost_{\mathcal{A}}(r(d)) \ .$$

In practice, however, it is more common to capture adversaries in terms of assumptions on their resources and capabilities [93]. We follow this approach and impose a hard upper bound Δ on the attacker's resources. We then define the adversary's utility as the probability of breach under this constraint.

Definition 6. *The* adversary's utility $u_{\mathcal{A}}$ *is defined as*

$$u_{\mathcal{A}}(d, r(d)) = p(d, r(d)) \ ,$$

where we require $cost_{\mathcal{A}}(r(d)) \leq \Delta$.

Example 2. *In this chapter we consider as resource* Δ *the time available to the adversary for breaking the key (e.g., the key deployment time), which the adversary can spend between sequentially making* m *side-channel measurements (each of which we assume take time* τ_{on}*) and* n *search steps (each of which we assume take time* τ_{off}*). The adversary's actions are hence*

$$A = \{(m, n) \mid m\tau_{off} + n\tau_{on} \leq \Delta\} \ .$$

Note that this definition of A also caters for parallel attacks by considering more general notions of cost and resources, which we forgo for the sake of concreteness.

3.2.4 Solving the Game

We use the standard solution concept for multi-stage games for characterizing optimal countermeasure configurations. Namely, we use subgame perfect equilibrium (see e.g. [91]), which is a combination of strategies (d^*, r^*) of the two players such that none of the players can obtain a higher payoff by deviating from the strategy, if the other player sticks to their strategy.

Definition 7. *A* subgame perfect (Nash) equilibrium (SPE) *of a game* $G = [(\mathcal{D}, \mathcal{A}), (D, A), (u_{\mathcal{D}}, u_{\mathcal{A}})]$ *is a defender's action* d^* *and an adversary's response function* r^**, such that*

(i) for all $d \in D$: $u_{\mathcal{D}}(d, r^*(d)) \leq u_{\mathcal{D}}(d^*, r^*(d^*))$*, and*

(ii) for all r: $u_{\mathcal{A}}(d^*, r(d^*)) \leq u_{\mathcal{A}}(d^*, r^*(d^*))$.

The SPEs of a game can be obtained by *backward induction*, which consists of solving two optimization problems: First, we find the adversary's response function that gives an optimal response to each of the defender's actions. Then, taking into account the optimal response function, we find the optimal defense. The existence of an equilibrium is guaranteed by the following theorem.

Theorem 1 ([94,95,96]). *A finite game of perfect information has a pure-strategy SPE. This SPE can be computed in time $O(|D| \cdot |A|)$, under the assumption that $u_{\mathcal{D}}$ and $u_{\mathcal{A}}$ can be evaluated in time $O(1)$.*

Here, *finite* refers to the number of players' actions and number of stages in the game, *perfect information* refers to the fact that the second player knows which action the first player has done, and *pure-strategy* refers to the players' strategies being deterministic.

3.2.5 Soundness of Solutions Based on Probability Bounds

The definition of the countermeasure configuration game, and hence its solution, rely on the probability p of a breach. When solving the game for practical systems, we need to resort to approximations of p. Specifically, in this chapter we work with upper bounds $\hat{p}$ on p, which we introduce in Section 3.3.

However, using upper bounds may lead to unsound decisions. For example, over-approximating p by $\hat{p} = 1$ in the utility function defined in (3.2) may suggest not to deploy any countermeasures. That is, the solution of the game based on $\hat{p}$ will leave the system vulnerable, while the solution based on p may command the deployment of a countermeasure. We now show that the countermeasure configuration game yields sound solutions when used with bounds on the probability of breach, in the sense that one errs on the secure side when solving the game using $\hat{p}$ instead of p.

For this, consider countermeasure configuration games G and $\hat{G}$ with $u_{\mathcal{A}}(\cdot) = p(\cdot)$, $\hat{u}_{\mathcal{A}}(\cdot) = \hat{p}(\cdot)$, and $u_{\mathcal{D}} = \hat{u}_{\mathcal{D}}$, such that $\hat{G}$ uses an over-approximation of the probability of breach in the sense that $\hat{p} \geq p$ pointwise. The following proposition shows that if a defender chooses an optimal defense strategy with respect to $\hat{G}$, and uses it to play G, the probability of breach will still be upper bounded by $p_{\max}$.

Proposition 2. *Let d^*, r^* be a SPE of $\hat{G}$. Then, for all r,*

$$p(d^*, r(d^*)) \leq p_{\max}$$

Proof. For all adversary strategies r, we have

$$p(d^*, r(d^*)) \overset{(1)}{\leq} \hat{p}(d^*, r(d^*)) \overset{(2)}{\leq} \hat{p}(d^*, r^*(d^*)) \,,$$

where (1) follows from $p \leq \hat{p}$; and (2) holds because $r^* = \arg\max_r \hat{p}(d, r(d))$, i.e., for each r, d, $\hat{p}(d, r(d)) \leq \hat{p}(d, r^*(d))$. As d^* is the defender's optimal action in $\hat{G}$, $\hat{p}(d^*, r(d^*)) \leq p_{\max}$, from which the assertion follows. □

Proposition 2 shows that the countermeasure configuration game leads to sound solutions when used with bounds on the probability of breach, whereas we have seen that the utility defined by (3.2) does not. We leave a general characterization of these two kinds of utilities to future work.

3.3 Bounds on the Probability of Key Recovery

3.3.1 Our Approach

In this section, we derive bounds for the probability of recovering the secret key of a public-key cryptosystem in a combined timing/algebraic attack. Our bounds are practical enough to justify the choice of a leaky, non-constant time implementation over a defensive constant-time implementation for standard ElGamal in some settings, see Section 3.5. We derive our bounds in the generic group model and under the assumption that the cryptosystem is protected against timing attacks by an idealized form of input blinding.

The starting point for our proof is the bound discussed in Section 3.2.1 of the amount of information contained in n execution time measurements of a blinded implementation of modular exponentiation. The technical challenge we face is to turn this bound into a guarantee against an adversary that can mount a combined timing/algebraic attack: Existing notions of leakage (e.g. [49, 63]) do not easily combine with the adversary models used in cryptography, and cryptographic notions of security (e.g. [97]) do not cater for this weak yet realistic leakage model.

We identify unpredictability entropy [50] as a suitable notion of entropy for extending leakage bounds to notions of hardness w.r.t. computational adversaries. Unpredictability entropy is versatile enough to accommodate for different classes of adversaries and it satisfies a chain rule, which provides an interface to the leakage bounds. We give a definition of unpredictability entropy and the proof of its chain rule in Section 2.1.

There are few facts known about the unpredictability entropy of computational problems [55], nor are there generally accepted hardness assumptions.[1] To compensate for this lack, we cast Shoup's lower bound for computing discrete logs in generic groups in terms of the unpredictability entropy w.r.t. an adversary who can perform m group operations. This step requires a slight adaptation of unpredictability entropy and the chain rule to the generic group model. We then connect the leakage bounds with the entropy bounds for the discrete log to obtain the desired bound (in terms of n and m) for combined side-channel/algebraic adversaries.

3.3.2 Generic Algorithms for Computing Discrete Logarithms

A generic group algorithm is an algorithm that solves problems over groups by only performing group operations and equality tests. That is, it does not make any use of the specific representation of group elements. Generic algorithms are an attractive model of computation for security because one can use them to establish lower bounds for computational problems [56]; their disadvantage is that they may underrate the power of real adversaries.

[1] It is known that RSA private keys have low unpredictability entropy because they can be efficiently recovered if a small fraction of the key bits are known [98].

We next introduce the notion of generic algorithms for computing discrete logarithms in cyclic groups of order k, which are isomorphic to the additive group $\mathbb{Z}_k$. We use the definition by Maurer [57].

Definition 8 (Generic algorithm in groups)**.** *A* generic algorithm $\mathsf{A}^m_{\mathcal{G}(X)}$ *over* $\mathbb{Z}_k$ *is an algorithm that can make m computation steps, each consisting of one query to a group oracle* $\mathcal{G}$*. The oracle has an internal state* $v_0, v_1, v_2, \ldots, v_m$*, where* $v_0 = x$ *is initialized with a secret sampled from a random variable X with* $ran(X) = \mathbb{Z}_k$*. The t-th query to the oracle consists of one of the following two operations:*

1. *constant insertion: algorithm inputs* $c \in \mathbb{Z}_k$*,*
   oracle sets $v_t := c$
2. *variable addition: algorithm inputs* $i, j < t$*,*
   oracle sets $v_t := v_i + v_j \pmod{k}$

The oracle then outputs the results of testing v_t *for equality with all other elements in the internal state, i.e.* $(v_t = v_i)_{i \in \{0,\ldots,t-1\}}$.

In this model, each v_i can be represented as a polynomial $v_i = ax + b$, where a and b are known to the adversary. This is because each v_i is derived from x and the inserted constants by repeated addition. If $v_i = v_j$ for $v_i = ax + b$ and $v_j = a'x + b'$ with $(a, b) \neq (a', b')$, we say that there is a *collision* between v_i and v_j. For the special case of a group of prime order q, a collision can be used for recovering x, because the equation $(a - a')x + (b - b') = 0$ has a unique solution in $\mathbb{Z}_q$.

In m queries, an adversary can establish at most $\binom{m}{2}$ equations of the form $v_i = v_j$, each of which has a unique solution. The probability of hitting one of them when sampling uniformly at random from $\mathbb{Z}_k$ is upper-bounded by $\binom{m}{2}/q$. Note that collisions are the only way for recovering x in this model, i.e. the bound holds for *any* generic algorithm for extracting x from the oracle.

The bound extends to bounds for cyclic groups of arbitrary order k via the Chinese remainder theorem.

Theorem 2 ([56, 57])**.** *Let* $\mathsf{A}^m_{\mathcal{G}}$ *be a generic algorithm over* $\mathbb{Z}_k$*, q the largest prime factor of k, and X uniformly distributed. Then*

$$P[\mathsf{A}^m_{\mathcal{G}(X)} = X] \leq \frac{m^2}{q} .$$

Next, we consider the definition of unpredictability entropy w.r.t. bounds on the number of accesses to an oracle $O(X)$ that receives as input the value of a random variable X, see Section 2.1.

Example 3. *Let p be prime, g a generator of* $\mathbb{Z}^*_p$*, and q the largest prime factor of* $p - 1$*. The problem of computing the discrete logarithm x of* g^x *has unpredictability entropy of least* $\log_2 \frac{q}{m^2}$ *for m calls to* $\mathcal{G}$*. This is because the multiplicative group* $\mathbb{Z}^*_p$ *is cyclic, and hence isomorphic to the additive group* $\mathbb{Z}_{p-1}$*. Then Theorem 2 applies.*

3.3.3 Blinded Side-Channels

We briefly revisit the notion of leakage that captures timing side-channels of cryptographic algorithms with input blinding and bucketing as a countermeasure, see Section 3.2.1. The assumption is that the execution time depends only on the key (which remains fixed over multiple executions) and the blinded message (which is chosen randomly and independently in each execution). The model abstracts from potential timing leaks through the blinding/unblinding operations and system state such as caches.

A *blinded channel* for X is a family of random variables $\{O_x\}_{x \in ran(X)}$, one for every x, with shared range $B = ran(O_x)$ of bounded size $b = |B|$. For a fixed secret x, making n timing measurements corresponds to taking n independent samples from O_x. As the samples are independent, their relative ordering does not contain information about x; the information about x contained in these samples can hence be represented by the *type* Y_n of the sequence, i.e. the vector of relative frequencies [53]. The number of such vectors (and hence the information about X contained in n timing measurements of a blinded channel) is bounded as follows.

Theorem 3. $|ran(Y_n)| \leq (n+1)^{b-1}$

Theorem 3 yields an upper bound for the amount of information contained in n timing measurements. It can be improved slightly using a more careful counting argument [99], and more significantly by using the fact that timing observations are not uniformly distributed but rather follow a multinomial distribution with b possible outcomes. We opt for the simple bound of Theorem 3 because it is tight enough for our purposes and has the advantage of a polynomial expression.

3.3.4 Bounds for Combined Adversaries

We now leverage the results presented in this section to derive bounds on the probability of key recovery by a combined side-channel/algebraic attack. In particular, we use the chain rule (Lemma 4) to combine the lower bounds for the unpredictability entropy (Theorem 2 and Example 3) with the upper bounds on the leakage (Theorem 3) and obtain the following result.

Theorem 4 (Generic algorithms with side-channels)**.** *Let $\mathsf{A}^m_{\mathcal{G}}$ be an algorithm that can make m calls to a group oracle and n measurements of a blinded channel. Then*

$$P[\mathsf{A}^m_{\mathcal{G}(X)}(Y_n) = X] \leq \frac{m^2(n+1)^{b-1}}{q},$$

where parameters b and q denote the range of the blinded channel and the largest prime divisor of the group order, respectively.

Our modeling captures the case in which the adversary first performs n timing measurements and then performs m calls to the group oracle. However, the bounds we

derive also hold for adversaries that can interleave timing measurements and oracle queries. The reason for this is that the adversary cannot influence the timing measurements, which is why nothing is gained by postponing a timing measurement until after an oracle query.

3.4 Computing the Equilibrium

In this section we show how to solve the countermeasure configuration game for a discrete logarithm based cryptosystem that is protected with blinding and bucketing. As described in Section 3.2.4, a pure-strategy equilibrium of a generic two-stage game can be computed by backward induction, that is, sequentially solving two optimization problems. We describe both optimization problems below.

For this, we rely on notation introduced in Examples 1 and 2 and the bounds derived in Section 3.3. For their connection, consider a cyclic group $\mathbb{Z}_p^*$, where p is prime and q is the largest prime factor of $p-1$, as in Example 3. We use the bit lengths $(|p|, |q|)$ to instantiate the abstract notion of key length k in Section 3.2. In particular, we use $(|p|, |q|)$ to describe desired properties of the modulus, but without referring to specific p and q.

3.4.1 Adversary's Optimization Problem

The adversary's optimization problem is to find a response function r^* that maps defender actions to an adversary action that maximizes the utility $u_{\mathcal{A}} = p(\cdot)$. That is, for a defender's choice of key length $k = (|p|, |q|)$ and bucketing b, we need to compute the numbers m^* and n^* of offline and online steps, respectively, that maximize $p((k, b), (m, n))$ subject to the resource constraint $m\tau_{off} + n\tau_{on} \leq \Delta$.

For the solution, first observe that the adversary's utility (the probability of breach) is maximized when all resources Δ are invested in the attack; thus we treat the resource constraint as an equality, and use it to express m in the bound from Theorem 4:

$$p((k, b), (m, n)) \leq \frac{1}{\tau_{off}^2} \frac{(\Delta - n\tau_{on})^2 (n+1)^{b-1}}{2^{|q|}} \tag{3.3}$$

The case $b = 1$ corresponds to a constant-time implementation, which is why the utility is maximized when time is spent on offline guessing rather than on online queries, i.e., $n^* = 0$ and $m^* = \Delta/\tau_{off}$.

The case $b \geq 2$ corresponds to an implementation with timing leaks. We symbolically compute the maximum utility by solving for n in $\frac{\partial}{\partial n}(\cdot) = 0$ applied to (3.3). This yields the maximum utility at $n^* = (\Delta(b-1) - 2\tau_{on})/((b+1) \cdot \tau_{on})$ and the adversary response

$$r^*(k, b) = ((\Delta - n^*\tau_{on})/\tau_{off}, n^*) \,.$$

3.4.2 Defender's Optimization Problem

The defender's optimization problem is to identify the defense d^* that maximizes the defender's utility $-cost_{\mathcal{D}}(d^*)$, where the adversary response r^* is given. Technically, solving this problem requires computing a bucketing b^* and a key length k^* that minimize the average execution time $cost_{\mathcal{D}}$ subject to the constraint $p((k^*, b^*), r^*(k^*, b^*)) \leq p_{\max}$.

Here we face the challenge that there are no analytical models describing $cost_{\mathcal{D}}$ as a function of key length and number of buckets. Instead, we instantiate each value of $cost_{\mathcal{D}}$ by empirical analysis of the corresponding implementation. We propose a simple heuristic to guide the search and avoid the evaluation of $cost_{\mathcal{D}}$ on too many parameters.

Restricting the search space We instantiate $p_{\max}$ as the probability of breach within time Δ of a key of reference length k_{ref} by an adversary that has no access to timing information. The rationale behind using such adversaries as a baseline is that we can rely on standards such as key length recommendations [93] for justified parameter values.

We then use (3.3) and the adversary's response n^* (as a function of (k, b)) to restrict the parameter space to the (k, b) that satisfy:

$$\frac{(\Delta - n^*\tau_{on})^2(n^* + 1)^{b-1}}{\tau_{off}^2\, 2^{|q|}} \leq \frac{\Delta^2}{\tau_{off}^2 2^{|q|_{ref}}} = p_{\max} \tag{3.4}$$

Empirically finding the optimum The empirical evaluation of $cost_{\mathcal{D}}(k, b)$ of a real implementation is expensive. To avoid evaluation on the entire parameter space defined by (3.4) we use a simple heuristic based on a weak assumption about the implementation, namely, that the average execution time grows with the length of the key.

With this, we explore the search space as follows. For $p_{\max}$ fixed and $b = 1, 2, \ldots$ we define k_b to be the smallest key that achieves security $p_{\max}$, i.e.

$$k_b = \min\{k \mid p((k, b), r^*(k, b)) \leq p_{\max}\}$$

As the execution time grows with the key length we do not need to consider keys beyond k_b because they will have less utility to the defender. For $b = 1, 2, \ldots$ we empirically determine the distribution of execution times of an implementation with keys of length k_b, compute the optimal bucket boundaries into b buckets using dynamic programming [53], and define $cost_{\mathcal{D}}(k_b, b)$ as the corresponding average execution time.

In theory, the number of buckets that need to be considered is upper bounded by the number of possible execution times. In practice (see Section 3.5), the shape of $cost_{\mathcal{D}}(k_b, b)$ becomes apparent after inspection of small values, which allows identifying (k^*, b^*).

3.5 Case Study

In this section, we report on a case study where we identify the optimal configuration of countermeasures against timing attacks on ElGamal decryption, following the approach developed in Sections 3.2, 3.3, and 3.4.

3.5.1 Experimental Setup

We analyze the ElGamal implementation of libgcrypt 1.6.1, compiled with GCC 4.8.2 on Ubuntu 14.04. We measure execution time in terms of the number of executed CPU instructions instead of real time, thereby abstracting from the influence of the microarchitecture. We determine the distribution of execution times corresponding to particular key lengths and countermeasure configurations by sampling the time for decrypting 10^5 random ciphertexts with different keys. As a benchmarking tool, we rely on the PAPI library [100].

3.5.2 ElGamal Implementation in Libgcrypt

In libgcrypt, ElGamal is implemented over the multiplicative group $\mathbb{Z}_p^*$. For deriving bounds on the probability of breach using Theorem 4, we need to know the size of the largest prime factor q of $p-1$. In libgcrypt, lower bounds for q can be directly read off the source code: the key-generation algorithm ensures that q has a bit-length of at least `qbits`, which is chosen according to a pre-defined table (see Figure 3.1). The secret exponent x is then taken of size `qbits` plus a safety margin, which results in faster decryption[2].

ElGamal decryption in libgcrypt is performed in three steps: modular exponentiation (we chose the square-and-multiply option), inversion, and multiplication. Blinding is not readily available in libgcrypt 1.6.1; we implement it relying on pre-computed randomness (see, e.g., [1]), which adds the overhead of two multiplications to the decryption time.

$\lvert p\rvert$	1024	1536	2048	2560	3072
`qbits`	165	198	255	249	269

Figure 3.1: Excerpt from "Wiener's table", used in libgcrypt for determining the minimal bit-length `qbits` of p's factors.

[2]In libgcrypt's source code, this is explained vividly: "I don't see a reason to have a x of about the same size as the p. It should be sufficient to have one about the size of q or the later used k plus a large safety margin. Decryption will be much faster with such an x."

3.5.3 Constant-Time ElGamal

We modify the libgcrypt source code to estimate the timing of a constant-time implementation of ElGamal. The modifications include always performing multiplication in the square-and-multiply exponentiation (≈ 35% overhead), forcing multi-precision integer comparison to always iterate over the entire numbers (≈ 60% overhead), as well as performing dummy operations to even out the timing of conditional branches in the routines for squaring, multiplication, and division (≈ 4% overhead). Additionally, compiler optimizations were switched off (≈ 10% overhead). In total, the overhead of the performed changes is ≈ 125 to 155%, depending on the modulus size.

To eliminate timing variations in modular inversion, we replace libgcrypt's implementation with a straightforward application of Fermat's little theorem ($a^{-1} = a^{p-2} \bmod p$), where we rely on the constant-time exponentiation. Because the slowdown using this approach is significant compared to state-of-the-art algorithms for modular inversion [101], for fairness of our analysis we use the Fermat-based inversion for both constant-time and non-constant-time timing measurement.

3.5.4 Results

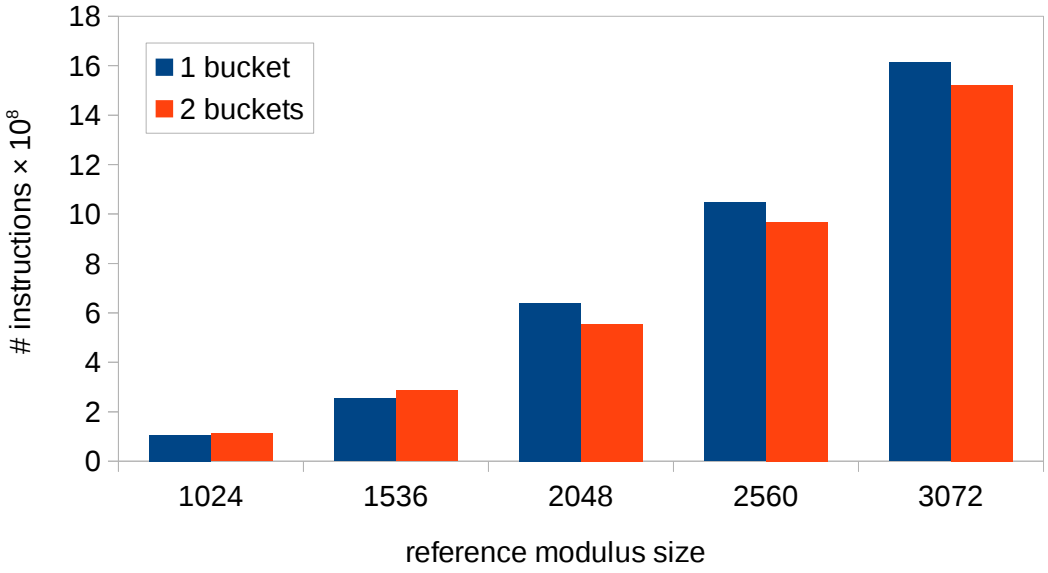
In our experiments we analyze the influence of the following parameters on the optimal countermeasure configuration: reference modulus size $|p|_{ref}$ as specified by k_{ref}; the key deployment time Δ; the key generation algorithm; and the access rate $\rho_{acc} = \tau_{on}^{-1}$. The latter parameter gives the number of accesses the adversary can make to timing observations per second, and can be influenced e.g. by limiting the rate of server requests, or by using denial-of-service preventions. In practice, the choice of a key size affects decryption time, and thus also the timing of an offline step τ_{off}. We avoid explicit instantiation of τ_{off} in dependence of the key size by over-approximating the adversary's capabilities, setting the value of τ_{off} with a key of size k to be τ_{off} with the corresponding k_{ref}, which gives a sound solution according to Proposition 2.

3.5.4.1 Varying the Modulus Size

We first consider varying the modulus size of keys generated with the libgcrypt default key generation algorithm, for fixed $\Delta = 365$ days and $\rho_{acc} = 100$ accesses per second. As depicted in Figure 3.2, for modulus sizes $|p|_{ref} \in \{1024, 1536\}$, the optimal defense is to use a constant-time implementation, i.e., $d^* = (k_{ref}, 1)$. For bigger modulus sizes, we obtain the optimum at a non-constant-time implementation with $b = 2$ buckets; the corresponding optimal modulus sizes are depicted in Figure 3.2b.

3.5.4.2 Varying the Access Rate ρ_{acc}

Increasing the access rate ρ_{acc} gives the adversary more possibilities to collect timing observations; thus, a defender deploying a non-constant time implementation needs to increase the modulus size to compensate for this information loss. In Figure 3.3 we



(a) Average cost (in number of CPU instructions).

b						
1	$\|p\|_{ref}$	1024	1536	2048	2560	3072
2	$\|p\|$	1478	2087	2683	3323	3898

(b) Modulus sizes $|p|$ (in bits) providing the same protection with 2-bucketing as modulus sizes $|p|_{ref}$ with one bucket.

Figure 3.2: Varying the modulus sizes for default libgcrypt keys. The results are given for $\Delta = 365$ days, $\rho_{acc} = 100$ accesses per second.

demonstrate that this can increase the cost of the non-constant time implementation enough for the optimum to shift from $b = 2$ to $b = 1$, i.e., the defender will prefer a constant-time implementation.

3.5.4.3 Varying the Key Deployment Time

Varying the deployment time has a similar effect as varying the access rate: an adversary has more time to collect timing observations. For example, if the deployment time is decreased, the defender's preference may shift from a constant to a non-constant-time implementation, as depicted in Figure 3.4.

3.5.4.4 Using a Safe Prime Modulus

An alternative approach for key generation makes sure that the group modulus p is a *safe prime*, i.e., $p = 2q + 1$ for a prime q. For example, the pycrypto library uses Algorithm 4.86 in [102] to generate p as a safe prime. As a result, q is guaranteed to have a bit length of $|p| - 1$.

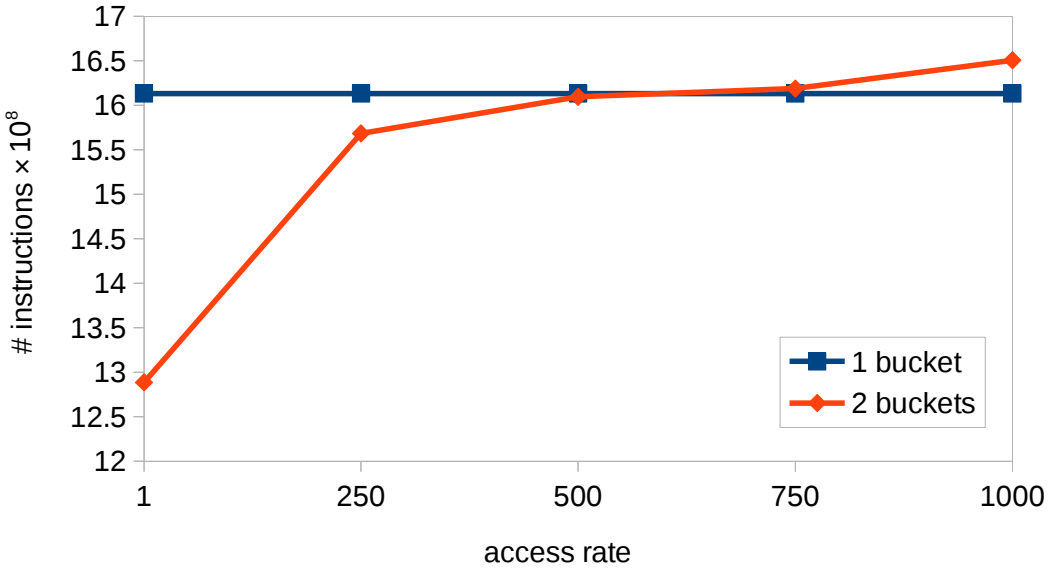


Figure 3.3: Average cost (in number of CPU instructions) for varying access rate ρ_{acc} (in accesses per second). The results are given for $|p|_{ref} = 3072$ bit, $\Delta = 365$ days.

Figure 3.5 illustrates the effect of varying the bit-size of the safe prime p, for fixed $\Delta = 365$ days and $\rho_{acc} = 100$ accesses per second. A comparison with Figure 3.2 shows that the benefits of using a non-constant time implementation are more substantial for safe primes than for default libgcrypt keys. The reason is that, for the same bit-length, safe primes provide more security in terms of Theorem 2, which is why the information loss from side-channel observations in a non-constant time implementation can be compensated by adding fewer bits to the key.

3.5.4.5 Varying the Number of Buckets

When increasing b, the defender has to increase the corresponding modulus size in order to ensure that the desired security level is met; for this to be economically feasible, the overhead from the bigger key needs to be compensated by savings from the countermeasure. In all cases we consider, increasing b above 2 did not fulfill this requirement, and thus all obtained optima were for $b = 1$ or $b = 2$. This is the case even in cases where an increase in b requires only a small increase in $|p|_{ref}$, as is the case with safe primes (see Figure 3.6).

3.5.5 Use Cases

In the following we choose the parameters to reflect two example use cases. We set the $|p|_{ref} = 2048$, which is the default GnuPG setting.

As a first use case, we consider a proxy server that decrypts incoming emails. In

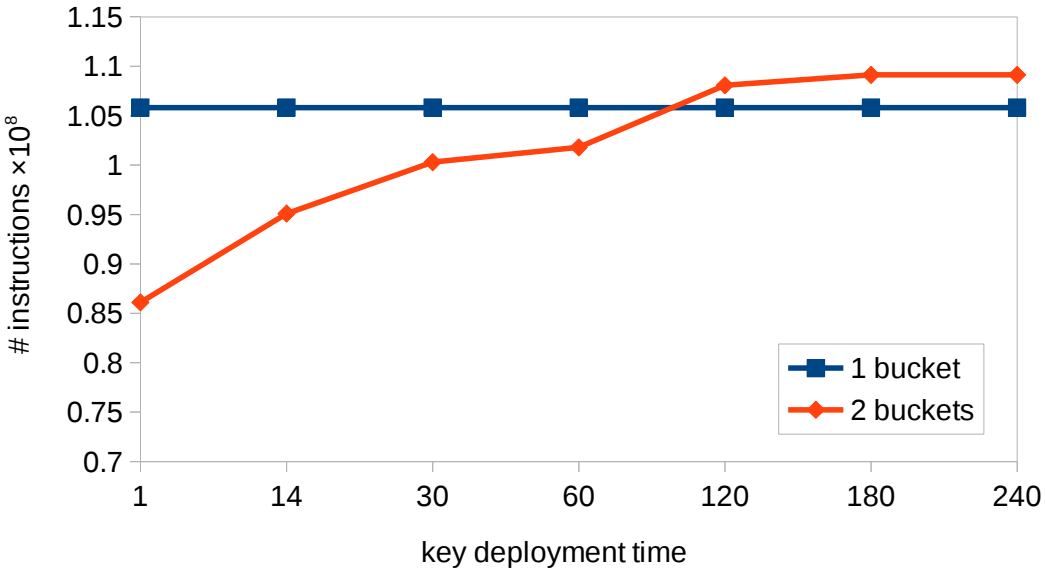


Figure 3.4: Average cost (in number of CPU instructions) for varying deployment time Δ (in days). The results are given for $|p|_{ref} = 1024$ bits, $\rho_{acc} = 100$ accesses per second.

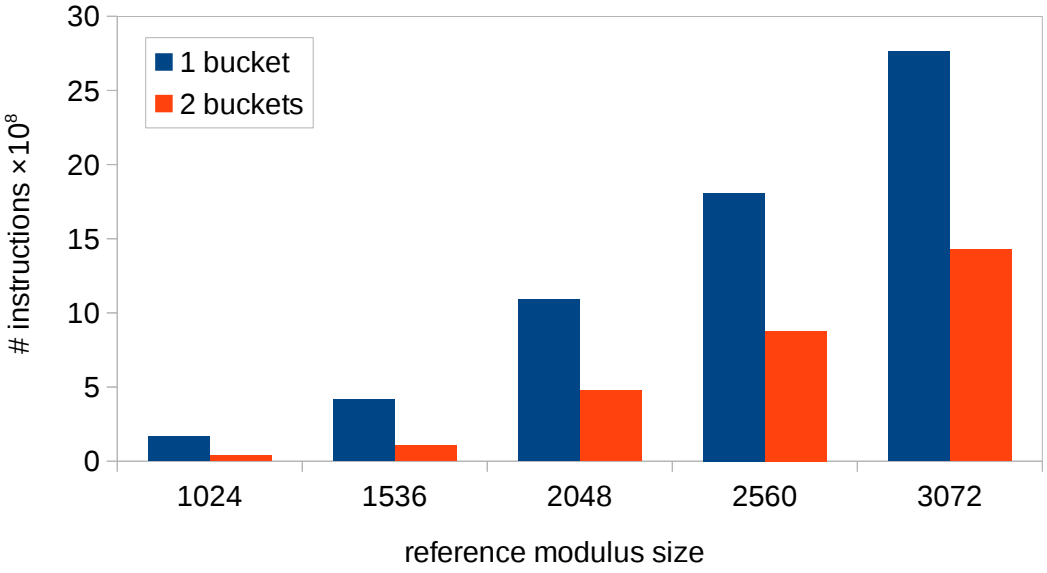
this scenario, the access rate ρ_{acc} can be expected to be small (we set it to $\rho_{acc} = 10$), while key deployment times can be expected to be higher. Figure 3.7 depicts that in this scenario, even for very long periods of time, we obtain the optimum configuration with the non-constant-time implementation ($b = 2$). Compared to the constant-time implementation, the optimal configurations give savings between 23% and 30%.

As a second use case, we consider an Internet-facing server that handles user requests. In this scenario, keys may have a shorter life-time, which we set to $\Delta = 90$ days; however, a higher access rate translates to a higher throughput, which may be a critical goal. For this scenario, Figure 3.8 shows that the non-constant-time implementation is the optimal solution even for large access rates; the expected savings compared to the constant-time implementation are between 10% and 28%.

3.6 Related Work

The substantial body of work related to the developments in this chapter is covered in Chapter 2. Below we clarify three additional points.

First, g-vulnerability [63] is a notion of entropy that accounts for general notions of adversary's gain. While it is possible to cast the generic discrete logarithm problem in terms of a specific g-function (the adversary gains 1 with a collision and 0 otherwise), we chose to rely on unpredictability entropy as a notion because it explicitly models resource-bounded computation and hence provides a natural connection to a variety of adversary models in cryptography.



(a) Average cost (in number of CPU instructions).

b						
1	$\|p\|_{ref}$	1024	1536	2048	2560	3072
2	$\|p\|$	1054	1566	2078	2590	3102

(b) Modulus sizes $|p|$ (in bits) providing the same protection with 2-bucketing as modulus sizes $|p|_{ref}$ with one bucket.

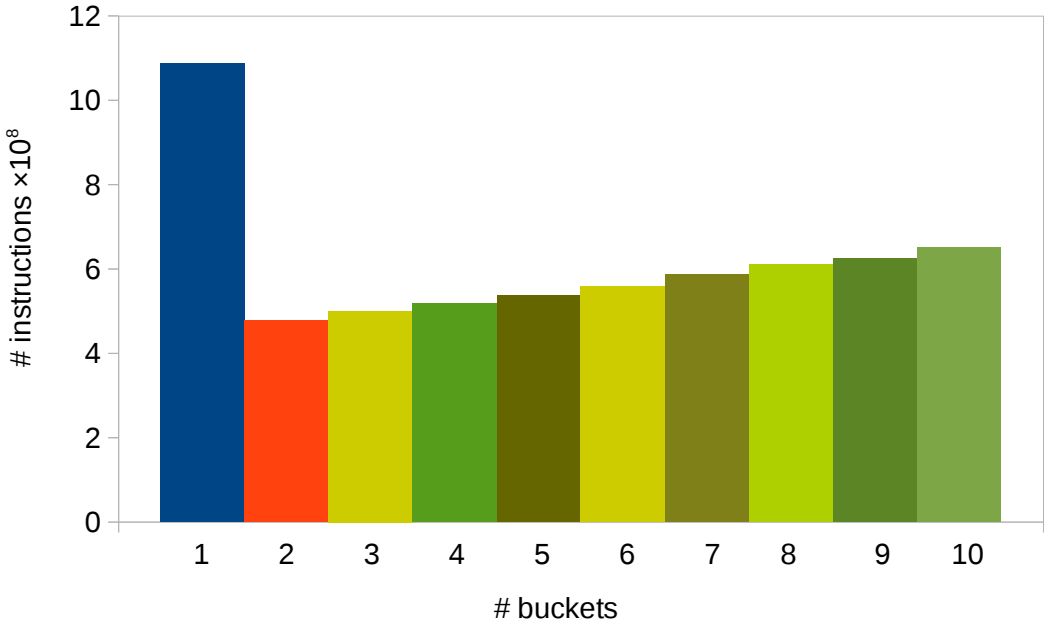
Figure 3.5: Varying the modulus sizes for safe primes. The results are given for $\Delta = 365$ days and $\rho_{acc} = 100$ accesses per second.

Second, Kiltz and Pietrzak [97] present a leakage-resilient variant of ElGamal, based on multiplicative secret sharing. They consider a more general leakage model and prove indistinguishability under chosen ciphertext attack (also in the generic group model), whereas we only prove security against key recovery attacks. The advantage of aiming for weaker guarantees is that they apply to *standard* ElGamal is that their simplicity makes them easily applicable in a game-theoretic context.

Third, the connection of timing leakage of blinded implementations to cryptographic security has been studied in [99], for asymptotic notions of security. In contrast, the bounds we develop in this chapter are concrete, which is required for using them in the context of the countermeasure configuration game.

3.7 Conclusions and Future Work

We have presented a systematic approach for determining the optimal protection against timing attacks, where we make use of a number of simple but powerful tools from game theory, information theory, and cryptography. The results we obtain are rigorous but



(a) Average cost (in number of CPU instructions).

buckets	1	2	3	4	5	6
$\|p\|$	2048	2078	2109	2139	2170	2201

(b) Modulus sizes $|p|$ (in bits) providing the same protection for varying number of buckets.

Figure 3.6: Varying the number of buckets, with safe prime modulus of reference size $|p|_{ref} = 2048$. The results are given for $\rho_{acc} = 100$ accesses per second and time of deployment $\Delta = 365$ days.

practical enough to justify the use of a fast but leaky implementation of ElGamal over a defensive constant-time implementation.

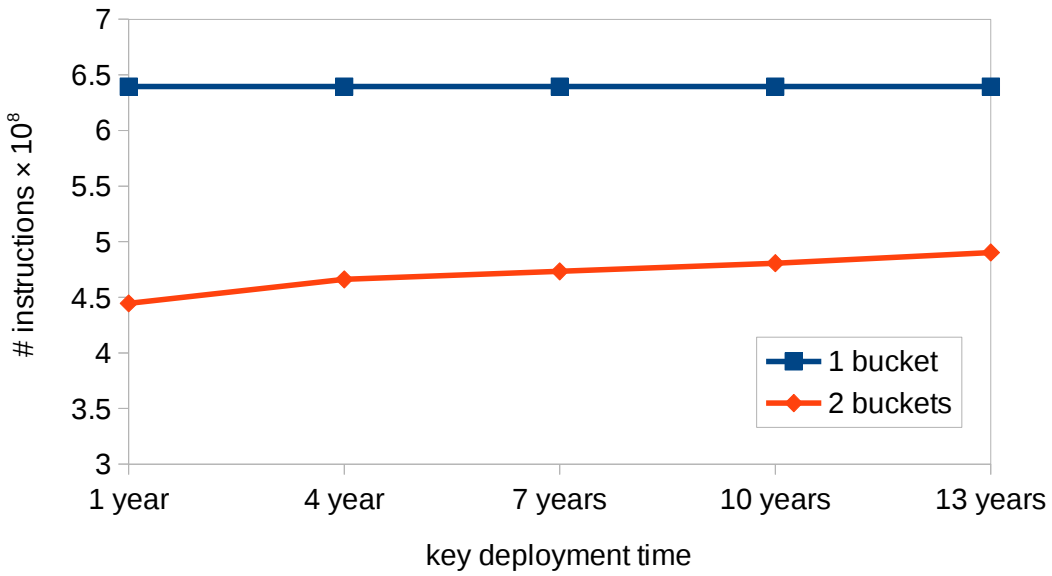


Figure 3.7: Average cost (in number of CPU instructions) for varying deployment time Δ. The results are given for $|p|_{ref} = 2048$ bit, $\rho_{acc} = 10$ accesses per second.

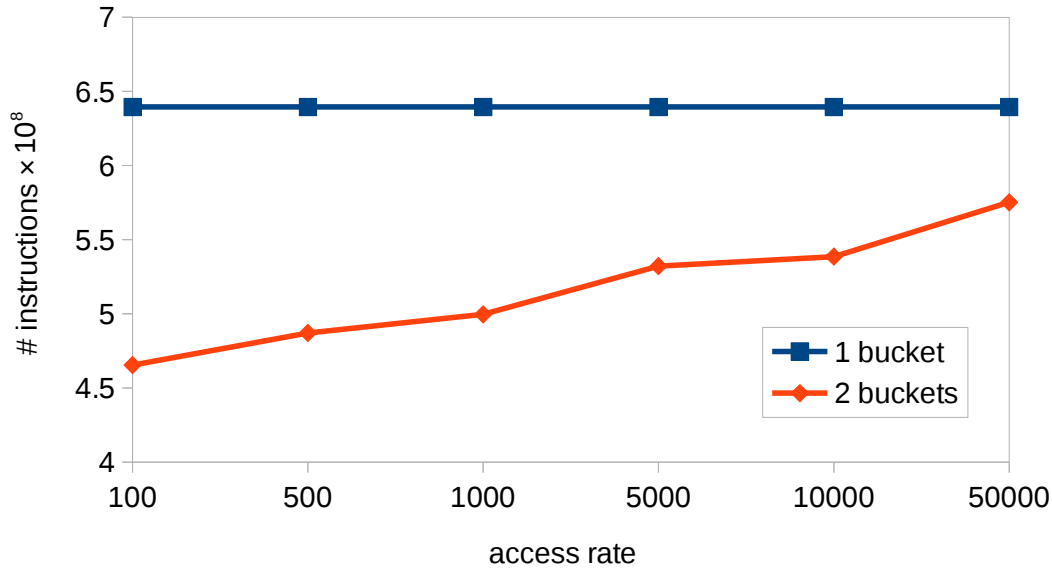


Figure 3.8: Average cost (in number of CPU instructions) for varying access rate ρ_{acc} (in accesses per second). The results are given for $|p|_{ref} = 2048$ bit, $\Delta = 90$ days.

Part II

Cache Attack Protection

The second part of this thesis presents CacheAudit, a versatile framework for the automatic, static analysis of cache side-channels. CacheAudit takes as input a program binary and a cache configuration, and derives formal, quantitative security guarantees for a comprehensive set of side-channel adversaries. Our technical contributions include novel abstractions to efficiently compute precise overapproximations of the possible side-channel observations for each of these adversaries. These approximations then yield upper bounds on the amount of information that is revealed.

The work presented in this part is the first tool for automatic quantification of cache leaks from practical implementations, and addresses research question 2 (Q2) in Section 1.2. The exposition of this part is organized in three chapters. Chapter 4 provides the necessary background on caches, static analysis, and quantifying side-channel leaks. Our main contributions are presented in Chapter 5 and Chapter 6.

In Chapter 5, we introduce CacheAudit and the novel abstract domains on which it is based. In this chapter we address adversaries who can observe the final cache states, traces of cache hits and misses, and the total execution time of programs. In case studies we apply CacheAudit to binary executables of algorithms for sorting and encryption, including the AES implementation from the PolarSSL library, and the reference implementations of the finalists of the eSTREAM stream cipher competition.

In Chapter 6, we address stronger adversaries who can observe traces of memory addresses and memory blocks, and enable reasoning about pointer arithmetic in dynamic memory. We achieve this by devising novel techniques that provide support for bit-level and arithmetic reasoning about unknown memory locations. These techniques enable us to perform the first rigorous analysis of widely deployed software countermeasures against cache attacks on modular exponentiation, based on executable code.

4 Static Analysis of Cache Side-Channels

4.1 Caches and Programs

We begin with a primer on caches, where we also define terminology. We then develop a program semantics that includes cache behavior, and we show how it can be used as a basis for quantifying the amount of information leaked by cache side-channels.

4.1.1 A Primer on Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To profit from spatial locality and to reduce management overhead, main memory is logically partitioned into a set of *memory blocks* $\mathcal{B}$. Each block is cached as a whole in a cache line of the same size. Upon a memory access, if the accessed memory block is located in the cache, a *cache hit* occurs; otherwise, a *cache miss* occurs.

Caches are usually partitioned into equally-sized *cache sets* $\mathcal{S}$, and the size k of a cache is called the *associativity* of the cache. Caches with $k > 1$ are called *k-way set-associative*, and caches with $k = 1$ are called *directly-mapped*. Each block is only stored in the lines of one set, and we denote the mapping between blocks and sets as the function $set : \mathcal{B} \to \mathcal{S}$. A cache set is identified by the *set index*; blocks within a cache set are identified by the *tag*; bytes within a block are identified by the *offset*. Usually, the tag, set index, and offset, are determined either from bits of the physical, or of the virtual memory addresses, as e.g. shown in Figure 4.1.

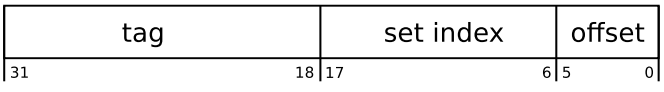


Figure 4.1: An example split of a 32-bit memory address into tag, index, and offset, for a physically indexed and tagged, 32KB, 8-way set associative cache, with 64-byte lines.

Since the cache is much smaller than main memory, a *replacement policy* must

decide which memory block to replace upon a cache miss. Usually, replacement policies treat sets independently, so that accesses to one set do not influence replacement decisions in other sets. Well-known replacement policies in this class are least-recently used (LRU), used in various Freescale processors such as the MPC603E and the TriCore17xx; pseudo-LRU (PLRU), a cost-efficient variant of LRU, used in the Freescale MPC750 family and multiple Intel microarchitectures; and first-in first-out (FIFO), also known as Round Robin, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC7450 family. A more comprehensive overview can be found in [103].

4.1.2 Programs and Computations

We introduce an abstract notion of programs and computations, which we then refine to capture cache behavior. Namely, a *program* $P = (\Sigma, I, F, \mathcal{E}, \mathcal{T})$ consists of the following components:

- Σ - a set of *states*
- $I \subseteq \Sigma$ - a set of *initial* states
- $F \subseteq \Sigma$ - a set of *final* states
- $\mathcal{E}$ - a set of *events*
- $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ - a *transition relation*

A *computation* of P is an alternating sequence of states and events $\sigma_0 e_0 \sigma_1 e_1 \dots \sigma_n$ such that $\sigma_0 \in I$ and that for all $i \in \{0, \dots, n-1\}$, $(\sigma_i, e_i, \sigma_{i+1}) \in \mathcal{T}$. The set of all computations of P is its *trace collecting semantics* $Col(P) \subseteq Traces$, where *Traces* denotes the set of all alternating sequences of states and events. When considering terminating programs, the trace collecting semantics can be formally defined as the least fixpoint of the *next* operator containing I:

$$Col(P) = I \cup next(I) \cup next^2(I) \cup \dots ,$$

where $next : Traces \to Traces$ describes the effect of one computation step:

$$next(S) = \{t.\sigma_n e_n \sigma_{n+1} \mid t.\sigma_n \in S \wedge (\sigma_n, e_n, \sigma_{n+1}) \in \mathcal{T}\}$$

In the rest of the chapter, we assume that P is fixed and abbreviate its trace collecting semantics by *Col*.

4.1.3 Cache Updates and Cache Effects

For reasoning about cache side-channels, we consider a semantics in which the cache is part of the program state. Namely, the program state consists of logical memories in $\mathcal{M}$ (representing the values of main memory locations and CPU registers, including the program counter) and a cache state in C, i.e., $\Sigma = \mathcal{M} \times C$.

The *memory update* $upd_{\mathcal{M}}$ is a function $upd_{\mathcal{M}} \colon \mathcal{M} \to \mathcal{M}$ that is determined solely by the instruction set semantics. The memory update has effects on the cache that are described by a function $eff_{\mathcal{M}} \colon \mathcal{M} \to \mathcal{E}_{\mathcal{M}}$, which we call *memory effect*. In the setting of this thesis, $eff_{\mathcal{M}}$ determines the sequence of memory accesses $e \in \mathcal{A}^*$ issued during the update, i.e., $\mathcal{E}_{\mathcal{M}} = \mathcal{A}^*$, which includes the addresses accessed when fetching instructions from the code segment, as well as the addresses containing accessed data. The memory effect determines which blocks are loaded into cache, for which we define the function $block \colon \mathcal{A}^* \to \mathcal{B}^*$ mapping addresses to the corresponding blocks.

We model the cache state as a function that assigns an *age* in $\{0, \ldots, k-1, k\}$ to every memory block, where the age determines the order in which blocks are evicted. Here, we require that no two blocks that reside in the same cache set have the same age, and we represent blocks that are *not* cached using age k. Formally:

$$C := \{c \in \mathcal{B} \to A \mid \forall a, b \in \mathcal{B} : a \neq b \Rightarrow \\ ((set(a) = set(b)) \Rightarrow (c(a) \neq c(b) \vee c(a) = c(b) = k))\}$$

Note that C includes states that cannot occur under some replacement policies: For example, under LRU and FIFO, a block of age $a \in \{1, \ldots, k-1\}$ is always preceded by a block of age $a - 1$.

The *cache update* is a function $upd_C \colon C \times \mathcal{B} \to C$, and it works as follows. Upon a cache miss, a block is loaded from main memory into a cache set, where it gets assigned age 0. The ages of the other memory blocks in this cache set are incremented by one. In particular, this means that a block of age $k-1$ is evicted from the cache. Upon a cache hit to a block of age $a \in \{0, \ldots, k-1\}$, the ages of the blocks in the same set are updated by applying a permutation $\Pi_a \colon A \to A$, which is determined by the replacement policy. We first give a formalization of the cache update in which the replacement policy is kept parametric, before we define concrete permutations describing LRU, FIFO, and PLRU replacement in Section 4.1.4:

$$upd_C(c, b) := \lambda b' \in \mathcal{B}. \begin{cases} c(b') & : set(b') \neq set(b) \\ c(b') & : set(b') = set(b) \wedge b' \neq b \wedge c(b') = k \\ 0 & : set(b') = set(b) \wedge b' = b \wedge c(b) = k \\ c(b') + 1 & : set(b') = set(b) \wedge b' \neq b \wedge c(b') < k \wedge c(b) = k \\ \Pi_{c(b)}(c(b')) & : set(b') = set(b) \wedge c(b') < k \wedge c(b) < k \end{cases}$$

Each cache update results in a cache hit or a cache miss, which we formally capture in

terms of a function *cache effect* $\mathit{eff}_C \colon \mathcal{C} \times \mathcal{B} \to \mathcal{E}$:

$$\mathit{eff}_C(c, b) := \begin{cases} \mathit{hit} & : c(b) < k \\ \mathit{miss} & : \text{otherwise} \end{cases}$$

To capture cases where no memory access occurs, or where multiple accesses occur, to the set of events is defined as $\mathcal{E} = \{\mathit{hit}, \mathit{miss}\}^*$.

With this, we can now connect the components and obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ by

$$\begin{aligned} \mathcal{T} = \{((m_1, c_1),\ e\ , (m_2, c_2)) \mid m_2 = \mathit{upd}_{\mathcal{M}}(m_1) \wedge\ & c_2 = \mathit{upd}_C^*(c_1, (\mathit{block})(\mathit{eff}_{\mathcal{M}}(m_1))) \\ & \wedge\ e = \mathit{eff}_C^*(c_1, (\mathit{block})(\mathit{eff}_{\mathcal{M}}(m_1)))\}\ , \end{aligned}$$

which formally captures the asymmetric relationship between caches, logical memories, and events. Here, upd_C^* denotes the repeated application of upd_C on a sequence of blocks, which returns the cache state after the sequence of accesses; eff_C^* denotes the repeated application of eff_C on a sequence of blocks, which returns the sequence of cache effects caused by the sequence of accesses.

4.1.4 Replacement Policies Defined by Permutations

Upon a cache hit, the different replacement policies update the ages of blocks within a cache set according to different permutations. In the following, we define these permutations for the FIFO, LRU, and PLRU replacement policies.

The FIFO replacement policy does not change the ages of the blocks upon cache hits. Its is thus readily modeled as the identity permutation.

$$\Pi_a^{\mathit{FIFO}}(a') = a'$$

The LRU replacement policy sets the age of an accessed block to 0 upon a cache hit, making sure that always the least-recently used blocks get evicted. Formally, we cast this behavior as

$$\Pi_a^{\mathit{LRU}}(a') = \begin{cases} 0 & : a' = a \\ a' + 1 & : a' < a \\ a' & : a' > a \end{cases}$$

The operation of the PLRU replacement policy, which is a cost-efficient approximation to LRU, requires a more detailed explanation. For an associativity that is a power of two (the case considered in this chapter), PLRU represents each cache set as a full binary tree storing the blocks at its leaves, and each non-leaf stores a bit that represents an arrow pointing to one of the children. Upon a cache miss, the block to be evicted is determined by following the arrows starting from the root. Upon any cache access (regardless whether it is a hit or a miss), the arrows on the way to the accessed block are

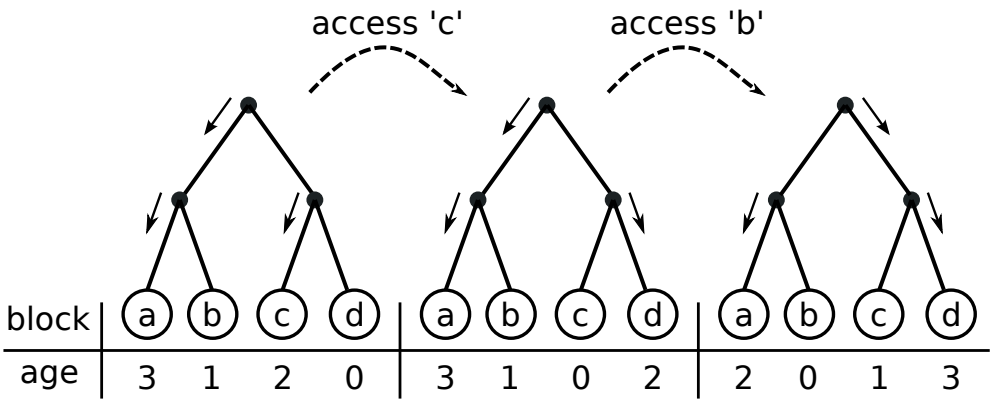


Figure 4.2: An example of two consecutive cache hits with PLRU.

flipped. Figure 4.2 shows an example of two consecutive cache hits in a 4-way cache. This construction ensures that upon consecutive cache misses, all cached blocks will be evicted in an order depending on the current settings of the arrows, which allows casting the effect of cache hits as a permutation of the ages. We formally define this PLRU permutation policy Π^{PLRU} as

$$\Pi_a^{PLRU}(a') = \begin{cases} 0 & : a' = a \\ a' & : a \text{ even} \wedge a' \text{ odd} \\ a' + 1 & : a \text{ odd} \wedge a' \text{ even} \\ 2 \cdot \Pi_{\lfloor a/2 \rfloor}^{PLRU}(\lfloor a'/2 \rfloor) & : \text{ otherwise} \end{cases}$$

The intuition behind this formalization is presented in the following. The case distinction in the definition of Π^{PLRU} stems from the observation that in the PLRU binary tree, all blocks stored in the subtree to which the arrow at the root points (the *odd subtree*) have an odd age, and the remaining blocks (in the *even subtree*) have an even age. In the second and the third case in the definition of Π^{PLRU}, we update the age of blocks that are in a different subtree than the accessed block. If the accessed block is from the even subtree (the second case), then the arrow at the root is not flipped, and all the blocks in the odd subtree retain their ages; if the accessed block is from the odd subtree (the third case), the arrow at the root is flipped, which increases the age of all blocks in the even subtree by one. For the blocks in the same subtree as the accessed blocks (the fourth case), the relative order of the ages of those blocks is the same as the order of ages of those blocks if only the subtree is considered as a (twice-smaller) cache set; the new ages are twice the ages in the smaller cache set as only every second evicted block in the actual cache is going to be from this subtree.

4.2 Side-Channels

We now define side-channels corresponding to access-based, trace-based, and timing-based side-channel adversaries. For the access-based adversaries, we restrict the presentation to *synchronous* adversary models, i.e. those that can control and observe the cache state before and after, but not during, the execution of the victim program. A description of CacheAudit's support for concurrent, *asynchronous* access-based adversaries as in [104] can be found in [84].

For a deterministic, terminating program P, the transition relation is a function, and the program can be modeled as a mapping $P\colon I \to Col$. We model an adversary's view on the computations of P as a function $view\colon Col \to O$ that maps computations to a finite set of observations O. The composition

$$C = (view \circ P)\colon I \to O$$

defines a function from initial states to observations, which we call a *channel* of P. Whenever *view* is determined by the cache and event components of traces, we call C a *side-channel* of P.

Quantification of Side-Channels

We characterize the security of a channel $C\colon I \to O$ as the difficulty of guessing the secret input from the channel output.

Formally, we model the choice of a secret input by a random variable X with $ran(X) \subseteq I$ and the corresponding observation by a random variable $C(X)$ (or just C) with $ran(C) \subseteq O$. Here $ran(\cdot)$ denotes the range of the respective random variable. We model the adversary as another random variable $\hat{X}$. The goal of the adversary is to estimate the value of X, i.e. it is successful if $\hat{X} = X$.

Consider first the special case where the adversary does not have access to the side-channel information, but knows the distribution of X. A straightforward upper bound for the probability of correctly guessing the value of X in one shot is given by the probability of the most likely value, where equality can be achieved:

$$P(\hat{X} = X) \leq \max_{\sigma \in I} P(X = \sigma)\ . \tag{4.1}$$

Consider now the case where the adversary can observe C, and where moreover this is the only information he has about X. We formalize this as the requirement that $X \to C \to \hat{X}$ form a Markov chain, which means that X and $\hat{X}$ do not share information beyond what is contained in C or, more technically, is equivalent to requiring that X and $\hat{X}$ are statistically independent when conditioned on C. The following theorem expresses a security guarantee as an upper bound on the adversary's success probability in terms of the size of the range of C.

Theorem 5. *Let $X \to C \to \hat{X}$ be a Markov chain. Then*

$$P(X = \hat{X}) \leq \max_{\sigma \in I} P(X = \sigma) \cdot |ran(C)|$$

Proof.

$$
\begin{aligned}
P(X=\hat{X}) &= \sum_{\sigma,o} P(X=\hat{X}=\sigma \mid C=o)P(C=o) \\
&\overset{(I)}{=} \sum_{o} P(C=o) \sum_{\sigma} P(X=\sigma|C=o)P(\hat{X}=\sigma|C=o) \\
&\leq \sum_{o} P(C=o) \max_{\sigma} P(X=\sigma|C=o) \qquad (4.2) \\
&\overset{(II)}{=} \sum_{o} \max_{\sigma} P(X=\sigma)P(C=o|X=\sigma) \\
&\overset{(III)}{\leq} \max_{\sigma} P(X=\sigma) \sum_{o} \max_{\sigma} P(C=o|X=\sigma) \qquad (4.3) \\
&\leq \max_{\sigma} P(X=\sigma)\,|ran(C)|
\end{aligned}
$$

where (I) is due to the conditional independence of X and $\hat{X}$ (i.e. the Markov property). Equality (II) follows directly from Bayes' theorem. Inequality (III) is an equality in the case of uniformly distributed X, and the final step follows from the fact that each of the summands is less than or equal to 1. □

A comparison of Equation (4.1) and Theorem 5 shows that the size of the range of C is an upper bound on the factor by which the probability of correct guessing increases when the adversary sees the output of the side-channel $C(X)$ and is, in that sense, an upper bound for the amount of information leaked by C. We will often give bounds on $|ran(C)|$ on a log-scale, in which case they represent upper bounds on the number of leaked *bits*. Notice that the guarantees of Theorem 5 fundamentally rely on assumptions about the initial distribution of X: if X is easy to guess to begin with, Theorem 5 does not imply meaningful security guarantees.

For a formal connection to traditional (entropy-based) presentations of quantitative information-flow analysis, observe that the negative logarithm of (4.1) is the min-entropy $H(X)$ of X. Likewise the negative logarithm of (4.2) is the conditional min-entropy $H(X|C)$ of X given C (see [49, 105] for definitions), i.e., (4.2) corresponds to $2^{-H(X|C)}$. The logarithm of the factor by which the terms in (4.1) and Theorem 5 differ is a well-known upper bound for $H(X) - H(X|C)$, that is, for the reduction in uncertainty about X when one learns the output of the channel C e.g. [62, 99].

4.3 Automatic Quantification of Cache Side-Channels

Theorem 5 enables the quantification of side-channels by determining their range. As channels are defined in terms of views on computations, their range can be determined by computing *Col* and applying *view*. However, this entails computing a fixpoint of the *next* operator and is practically infeasible in most cases. Abstract interpretation [106] overcomes this fundamental problem by computing a fixpoint with respect to an efficiently computable over-approximation of *next*. This new fixpoint represents a superset

of all computations, which is sufficient for deriving an upper bound on the range of the channel and thus on the leaked information.

In this section, we describe the interplay of the abstractions used for over-approximating *next* in CacheAudit (namely, those for memory, cache, and events), and we explain how the global soundness of CacheAudit can be established from local soundness conditions. This modularity is key for the future extension of CacheAudit using more advanced abstractions. Our results hold for all adversaries introduced in Section 4.2 and we omit the superscript *adv* from channels and views for readability.

4.3.1 Sound Abstraction of Leakage

We frame a static analysis by defining a set of abstract elements $Traces^\sharp$ together with an abstract transfer function $next^\sharp : Traces^\sharp \to Traces^\sharp$. Here, the elements $a \in Traces^\sharp$ represent subsets of *Traces*, which is formalized by a concretization function

$$\gamma : Traces^\sharp \to \mathcal{P}(Traces) \, .$$

The key requirements for $next^\sharp$ are (a) that it be efficiently computable, and (b) that it over-approximates the effect of *next* on sets of computations, which is formalized as the following local soundness condition:

$$\forall a \in Traces^\sharp : next\,(\gamma(a)) \subseteq \gamma(next^\sharp(a)) \, . \tag{4.4}$$

Intuitively, if we maintain a superset of the set of computations during each step of the transfer function as in (4.4), then this inclusion must also hold for the corresponding fixpoints. More formally, any post-fixpoint of $next^\sharp$ that is greater than an abstraction of the initial states I is a sound over-approximation of the collecting semantics, which is a central result from [106]. We use $Col^\sharp$ to denote any such post-fixpoint.

Theorem 6 (Local soundness implies global soundness)**.** *If local soundness holds, formalized by Equation* (4.4)*, then*

$$Col \subseteq \gamma\left(Col^\sharp\right).$$

The following theorem is an immediate consequence of Theorem 6 and the fact that $view\,(Col) = ran(C)$. It states that a sound abstract analysis can be used for deriving bounds on the size of the range of a channel.

Theorem 7 (Upper bounds on leakage)**.**

$$|ran(C)| \leq \left|view\left(\gamma\left(Col^\sharp\right)\right)\right| \, .$$

With the help of Theorem 5, these bounds immediately translate into security guarantees. The relationship of all steps leading to these guarantees is depicted in Figure 4.3.

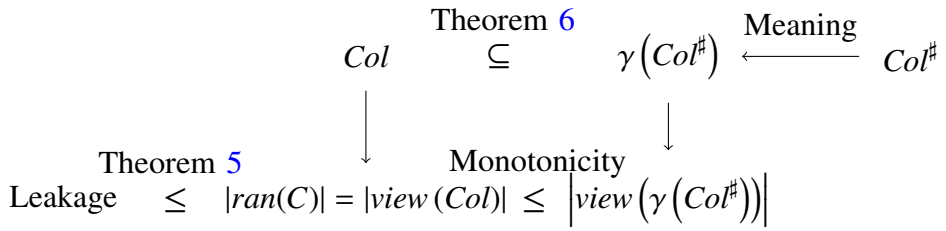


Figure 4.3: Relationship of collecting semantics Col, abstract fixpoint $Col^\sharp$, side-channels C, and leakage bounds.

4.3.2 Abstraction Using a Control Flow Graph

In order to come up with a tractable and modular analysis, we design independent abstractions for cache states, memory, and sequences of events.

- $\mathcal{M}^\sharp$ abstracts memory and $\gamma_\mathcal{M} : \mathcal{M}^\sharp \to \mathcal{P}(\mathcal{M})$ formalizes its meaning.
- $C^\sharp$ abstracts cache configurations and $\gamma_C : C^\sharp \to \mathcal{P}(C)$ formalizes its meaning.
- $\mathcal{E}^\sharp$ abstracts sequences of events and $\gamma_\mathcal{E} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E}^*)$ formalizes its meaning.

However, since cache updates and events depend on memory state, independent analyses would be too imprecise. In order to maintain some of the relations, we link the three abstract domains for memory state, caches, and events through a finite set of labels L so that our abstract domain is

$$Traces^\sharp : L \to \mathcal{M}^\sharp \times C^\sharp \times \mathcal{E}^\sharp \,,$$

where we write $a^\mathcal{M}(l)$, $a^C(l)$ and $a^\mathcal{E}(l)$ for the first, second, and third components of an abstract element $a(l)$.

Labels roughly correspond to nodes in a control flow graph in classical data-flow analyses. One could simply use program locations as labels. But in our setting, we use more general labels, allowing for a more fine-grained analysis in which we can distinguish values of flags or results of previous tests [107]. To capture that, we associate a meaning with each label via a function $\gamma_L : L \to \mathcal{P}(Traces)$. If the labels are program locations, then $\gamma_L(l)$ is the set of traces ending in a state in location l. The analogy with control flow graphs can be extended to edges of that graph: using the *next* operator, we define the successors and predecessors, respectively, of a location l as follows:

$$succ(l) = \{k \mid next(\gamma_L(l)) \cap \gamma_L(k) \neq \emptyset\}$$
$$pred(l) = \{k \mid next(\gamma_L(k)) \cap \gamma_L(l) \neq \emptyset\}$$

With this we can describe the meaning of an abstract element $a \in Traces^\sharp$ by:

$$\gamma(a) = \{\sigma_0 e_0 \sigma_1 \ldots \sigma_n \in Traces \mid \forall i \leq n, \forall l \in L : \quad \sigma_0 e_0 \sigma_1 \ldots \sigma_i \in \gamma_L(l) \Rightarrow \sigma_i^\mathcal{M} \in \gamma_\mathcal{M}(a^\mathcal{M}(l)) \wedge \sigma_i^C \in \gamma_C(a^C(l)) \wedge e_0 \ldots e_{i-1} \in \gamma_\mathcal{E}(a^\mathcal{E}(l))\} \quad (4.5)$$

That is, the meaning of an $a \in \mathit{Traces}^\sharp$ is the set of traces, such that for every prefix of a trace, if it "ends" at program location l, then the memory state, cache state, and the event sequence satisfy the respective abstract elements for that location.

The abstract transfer function $\mathit{next}^\sharp$ will be decomposed into:

$$\mathit{next}^\sharp(a) = \lambda l.\, (\mathit{next}_{\mathcal{M}^\sharp}(a, l), \mathit{next}_{C^\sharp}(a, l), \mathit{next}_{\mathcal{E}^\sharp}(a, l)) \ , \tag{4.6}$$

where each next function over-approximates the corresponding concrete update function defined in the previous section. The effects used for defining the concrete updates are reflected as information flow between otherwise independent abstract domains, which is formalized as a *partial reduction* in the abstract interpretation literature [108].

4.3.3 Local Soundness

The products and powers of sound abstract domains with partial reductions are again sound abstract domains [109]. The soundness of $\mathit{Traces}^\sharp$ hence immediately follows from the local soundness of the memory, cache and event domains. Below we describe those soundness conditions for each domain.

The abstract $\mathit{next}^\sharp$ operation is implemented using local update functions for the memory, cache, and event components. For the memory domain we have, for each label $k \in L$ and each $l \in \mathit{succ}(k)$:

- an abstract memory update $\mathit{upd}_{\mathcal{M}^\sharp,(k,l)} \colon \mathcal{M}^\sharp \to \mathcal{M}^\sharp$, and
- an abstract memory effect $\mathit{eff}_{\mathcal{M}^\sharp,(k,l)} : \mathcal{M}^\sharp \to \mathcal{P}(\mathcal{E}_{\mathcal{M}})$.

For the cache domain, there is no need for separate functions for each pair (k, l), because the cache update only depends on the accessed block, which is delivered by the abstract memory effect. Likewise, the update of the event domain only depends on the abstract cache effect. Thus, we further have:

- an abstract cache update $\mathit{upd}_{C^\sharp} : C^\sharp \times \mathcal{P}(\mathcal{E}_{\mathcal{M}}) \to C^\sharp$,
- an abstract cache effect $\mathit{eff}_{C^\sharp} : C^\sharp \times \mathcal{P}(\mathcal{E}_{\mathcal{M}}) \to \mathcal{P}(\mathcal{E})$, and
- an abstract event $\mathit{upd}_{\mathcal{E}^\sharp} : \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{E}) \to \mathcal{E}^\sharp$.

With these functions, we can approximate the effect of next on each label l, using the abstract values associated with the labels that can lead to l, $\mathit{pred}(l)$. For the example of the cache domain, this yields

$$\mathit{next}_{C^\sharp}(a, l) = \bigsqcup_{k \in \mathit{pred}(l)}^{C^\sharp} \mathit{upd}_{C^\sharp}\left(a^{\mathcal{C}}(k), \mathit{block}(\mathit{eff}_{\mathcal{M}^\sharp,(k,l)}(a^{\mathcal{M}}(k)))\right) \ ,$$

where $\bigsqcup^{C^\sharp}$ refers to the join function and can be thought of as set union, and block is naturally lifted to sets. That is, $\mathit{next}_{C^\sharp}(a, l)$ collects all cache states that can reach l

within one transition when updated with an over-approximation of the corresponding memory blocks.

Now from Equations 4.4, 4.5, and 4.6, we can derive conditions for each domain that are sufficient to guarantee local soundness for the whole analysis:

Definition 9 (Local soundness of abstract domains)**.** *The abstract domains are locally sound if the abstract joins are over-approximations of unions, and if for any function* $f^\sharp \in \{upd_{\mathcal{M}^\sharp,(k,l)}, eff_{\mathcal{M}^\sharp,(k,l)}, upd_{C^\sharp}, eff_{C^\sharp}, upd_{\mathcal{E}^\sharp}\}$ *approximating concrete function* $f \in \{upd_{\mathcal{M}}, eff_{\mathcal{M}}, upd_C, eff_C, next\}$ *and corresponding meaning function* γ_f*, we have for any abstract value* x*:*

$$\gamma_f\left(f^\sharp(x)\right) \supseteq f\left(\gamma_f(x)\right).$$

For example, for the cache abstract domain, we have the following local soundness conditions:

$$\begin{aligned}
\forall c^\sharp \in C^\sharp, M \in \mathcal{P}(\mathcal{B}):\ \gamma_C(upd_{C^\sharp}(c^\sharp, M)) &\supseteq upd_C(\gamma_C(c^\sharp), M),\\
eff_{C^\sharp}(c^\sharp, M) &\supseteq eff_C(\gamma_C(c^\sharp), M),\\
\forall \mathcal{G}^\sharp \subseteq C^\sharp:\ \gamma_C\left(\bigsqcup^{C^\sharp} \mathcal{G}^\sharp\right) &\supseteq \bigcup_{G^\sharp \in \mathcal{G}^\sharp} \gamma_C\left(G^\sharp\right).
\end{aligned}$$

Lemma 5 (Local Soundness Conditions)**.** *If local soundness holds on the abstract memory, cache, and events domains, then the corresponding* $next^\sharp$ *function satisfies local soundness.*

Due to the above lemma, abstract domains for the memory, cache, and events can be separately developed and proven correct. We exploit this fact in this chapter, and we plan to develop further abstractions in the future, targeting different classes of adversaries, more hardware features, or improving precision.

4.3.4 Soundness of Delivered Bounds

We implemented the framework described above in a tool named CacheAudit, which we describe in Chapter 5. We define a number *view*-functions, which capture adversaries with different observational capabilities: adversaries, who can observe the final cache state, the final timing, or a trace of cache hits and misses (see Chapter 5), as well as adversaries who can observe traces of accessed addresses or memory blocks (see Chapter 6). Thanks to the previous results, CacheAudit provides the following guarantees.

Theorem 8. *The bounds derived by CacheAudit soundly over-approximate* $|ran(C)|$*, defined using the respective view-functions, and hence correspond to upper bounds on the maximal amount of leaked information.*

The statement is an immediate consequence of combining Lemma 5 with Theorems 6 and 7, under the assumption that all involved abstract domains satisfy local soundness conditions, and that the corresponding counting procedures are correct. For the novel abstract domains we introduce (see Section 5.5, Section 6.4, and Section 6.5), we include justification of this assumption. For the other domains, corresponding proofs are either standard (e.g. the value domain), or out of scope of this thesis.

5 CacheAudit: A Tool for the Static Analysis of Cache Side-Channels

5.1 Introduction

Processor caches are a particularly rich source of side-channels because their behavior can be monitored in various ways. This is demonstrated by three documented classes of side-channel attacks:

1. In *time-based attacks* [1, 9] the adversary monitors the overall execution time of a victim, which is correlated with the number of cache hits and misses during execution. Time-based attacks are especially daunting because they can be carried out remotely over the network [12].

2. In *access-based attacks* [10, 11, 104] the adversary probes the victim's cache state by timing its own accesses to memory. Access-based attacks require the adversary and the victim to share the same hardware platform, which is common in the cloud and has already been exploited [13, 110].

3. In *trace-based attacks* [111] the adversary monitors the sequence of cache hits and misses. This can be achieved, e.g., by monitoring the CPU's power consumption and is particularly relevant to embedded systems.

A number of proposals have been made for countering cache-based side-channel attacks. Some proposals focus entirely on modifications of the hardware platform; they either solve the problem for specific algorithms such as AES [112], or require modifications to the platform [25] that are so significant that their rapid adoption seems unlikely. The bulk of proposals rely on controlling the interactions between the software and the hardware layers, either through the operating system [104, 113], the client application [9, 11, 114], or both [22, 115]. Reasoning about these interactions can be tricky and error-prone because it relies on the specifics of the binary code and the microarchitecture.

Our approach In this chapter we present CacheAudit, a tool for the automatic, static exploration of the interactions of a program with the cache. CacheAudit takes as input a program binary and a cache configuration and delivers formal security guarantees that cover all possible executions of the corresponding system. The security guarantees are quantitative upper bounds on the amount of information that is contained in the side-channel observations of timing-, access-, and trace-based adversaries, respectively. CacheAudit can be used to formally analyze the effect on the leakage of software countermeasures and cache configurations, such as preloading of lookup tables or increasing the cache's line size. The design of CacheAudit is modular and facilitates extension with any cache model for which efficient abstractions are in place.

We demonstrate the scope of CacheAudit in case studies where we analyze the side-channel leakage of implementations of representative algorithms for symmetric encryption and sorting. We highlight the following results:

- For the PolarSSL implementation of AES, CacheAudit confirms that preloading of tables significantly improves the security of the executable: for most adversary models and replacement policies, we can in fact prove non-leakage of the executable, whenever the tables fit entirely in the cache. However, for access-based adversaries and LRU and PLRU caches, CacheAudit reports small, non-zero bounds. And indeed, with LRU and PLRU (in contrast to, e.g., FIFO), the *ordering* of blocks within a cache set reveals information about the victim's final memory accesses.

- An analysis of the software implementations of the four finalists of the eSTREAM competition [116] yields the following results: the stream ciphers without lookup tables (Rabbit and Salsa20) are secure against all kinds of cache attacks. In particular, CacheAudit can formally establish leakage bounds of zero, on the basis of the binary executable of the reference implementations, for all adversary models and replacement policies. For HC-128, which employs dynamically updated tables, CacheAudit can establish leakage bounds of zero for some adversary models, whenever the tables fit entirely into the cache. This is explained by the regularity of the memory accesses of the dynamic updates, which ensure that the entire table is cached. Indeed, the leakage bounds we obtain strikingly resemble those obtained for AES with preloading. For Sosemanuk, the memory accesses do not exhibit such regularity and, indeed, CacheAudit consistently derives non-zero bounds.

Together, these results show how CacheAudit can help with extracting useful information about the security of the interactions of binary executables with the underlying cache architecture.

Technical contributions On a technical level, our work builds on the fact that the amount of leaked information corresponds to the cardinality of the set of possible side-channel observations (that is, the size of the range of the side-channel). This

set can be over-approximated by abstract interpretation, which is a theory of sound approximation of program semantics [106], and its cardinality can be determined by counting techniques [41]. We elaborate on the formal aspects of analyzing cache side-channels through abstract interpretation in Chapter 4.

To realize CacheAudit based on this insight, we propose three novel abstract domains (that is, data structures that approximate properties of the program semantics) that keep track of the observations of access-based, time-based, and trace-based adversaries, respectively. Moreover, we present counting algorithms that determine the cardinality of the set of observations represented by the abstract states of each of these domains. In particular:

1. We propose an abstract domain that tracks information about the possible cache states, which are represented in terms of the memory blocks that may reside in the cache. We further propose an algorithm that counts the cache states that are represented by an abstract state. The domain and counting procedure are both parametric in the cache update policy, which is described by a permutation [117]. In contrast to existing abstract domains used in worst-case execution time analysis [103, 118] and their counting procedures [66], our novel domain provides increased precision and it enables the abstraction of a large class of update policies in a uniform and simple manner.

2. We propose an abstract domain that tracks the traces of cache hits and misses that may occur during execution. We use a technique based on prefix trees and hash consing to compactly represent such a set of traces, and to determine its cardinality.

3. We propose an abstract domain that tracks the possible execution times of a program. This domain captures timing variations due to control flow and caches by associating hits and misses with their respective latencies and adding the execution time of the respective commands.

We formalize these domains in an abstract interpretation framework that captures the relationship between microarchitectural state and program code. We use this framework to establish the correctness of the derived upper bounds on the leakage to the corresponding side-channel adversaries.

In summary, our main contributions are both theoretical and practical: On a theoretical level, we define novel abstract domains that are suitable for the analysis of cache side-channels, for a rich set of adversary models. On a practical level, we build CacheAudit, the first tool for the automatic, quantitative information-flow analysis of cache side-channels, and we show how it can be used to derive formal security guarantees from binary executables of sorting algorithms and state-of-the-art cryptosystems.

Current scope and future extensions of CacheAudit The current version of CacheAudit offers support for data, instruction, and mixed caches with FIFO, LRU, and PLRU

replacement policies, for programs using a limited subset of 32-bit x86 instructions and CPU flags. The current version does *not* offer support for multiple levels of caches, multiple CPU cores, speculative execution, out-of-order execution, virtual memory, or code using dynamic jump targets. See Section 5.8 for a discussion of the implications of basing security analysis on such imperfect models and the challenges associated with extending CacheAudit accordingly.

The (OCaml) source code and documentation of CacheAudit are available from the project website[1] and on GitHub[2] to facilitate future extension.

Outline The remainder of the chapter is structured as follows. In Section 5.2, we illustrate the power of CacheAudit on a simple example program. We describe the design of CacheAudit, and the novel abstract domains in 5.4 and 5.5, respectively. We present experimental results in Section 5.6, before we discuss prior work in Section 5.7 and conclude in Section 5.9 after discussing challenges for future work in Section 5.8.

5.2 Illustrative Example

In this section, we illustrate on a simple example program the kind of guarantees CacheAudit can derive. Namely, we consider the implementation of BubbleSort shown in Figure 5.1, that receives its input in an array `a` of length `n`. We assume that the contents of `a` are secret and we aim to deduce how much information a cache side-channel adversary can learn about the relative ordering of the elements of `a`.

```
1   void BubbleSort(int a[], int n)
2   {
3    int i, j, temp;
4    for (i = 0; i < n - 1; ++i)
5       for (j = 0; j < n - 1 - i; ++j)
6          if (a[j] > a[j+1])
7          {
8             temp = a[j+1];
9             a[j+1] = a[j];
10            a[j] = temp;
11         }
12  }
```

Figure 5.1: An implementation of the BubbleSort algorithm

To begin with, observe that the conditional swap in lines 6–11 is executed exactly $\frac{n(n-1)}{2}$ times. A *trace-based* adversary that can observe, for each instruction, whether it

[1] http://software.imdea.org/cacheaudit
[2] https://github.com/cacheaudit

corresponds to a cache hit or a miss is likely to be able to distinguish between the two alternative paths in the conditional swap, hence we expect this adversary to be able to distinguish between $2^{\frac{n(n-1)}{2}}$ execution traces. A *timing-based* adversary who can observe the overall execution time is likely to be able to distinguish between $\frac{n(n-1)}{2} + 1$ possible execution times, corresponding to the number of times the swap has been carried out. For an *access-based* adversary who can probe the final cache state upon termination, the situation is more subtle: evaluating the guard in line 6 requires accessing both `a[j]` and `a[j+1]`, which implies that both will be present in the cache when the swap in lines 8–10 is carried out. Assuming we begin with an empty cache, we expect that there is only one possible final cache state.

CacheAudit enables us to perform such analyses (for a particular `n`) formally and automatically, based on actual x86 binary executables and different cache types. CacheAudit achieves this by tracking compact representations of supersets of possible cache states and traces of hits and misses, and by counting the corresponding number of elements. For the above example, CacheAudit was able to precisely confirm the intuitive bounds, for a selection of several `n` in $\{2, \ldots, 64\}$.

In terms of security, the number of possible observations corresponds to the factor by which the cache observation increases the probability of correctly guessing the secret ordering of inputs. Hence, for `n` = 32 and a uniform distribution on this order (i.e. an initial probability of $\frac{1}{32!} = 3.8 \cdot 10^{-36}$), the bounds derived by CacheAudit imply that the probability of determining the correct input order from the side-channel observation is 1 for a trace-based adversary, $3.7 \cdot 10^{-33}$ for a time-based adversary, and remains $\frac{1}{32!}$ for an access-based adversary.

5.3 Adversary Model

In this section, we introduce four adversary models, based on the notion of *channels* defined using *view*-functions, which capture adversaries with different observational capabilities, see Chapter 4.

5.3.1 Adversary Views

The view of an *access-based* adversary that shares the memory space with the victim is defined by

$$view^{acc} : (m_0, c_0)e_0 \ldots e_{n-1}(m_n, c_n) \mapsto c_n$$

and captures that the adversary can determine which memory blocks are contained in the cache upon termination of the victim. In practice, this is achieved by probing the cache, which changes the cache state and hence leads to information loss; the assumption that the adversary can determine the cache state is a safe over-approximation of a real adversary. An adversary that does *not* share the memory space also sees the cache state, but cannot distinguish between the different blocks the victim has loaded in each cache

set. We denote this view by $view^{accd}$. The view of a *trace-based* adversary is defined by

$$view^{tr} : \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto e_0 \dots e_{n-1}$$

and captures that the adversary can determine whether each instruction results in a hit, miss, or does not access memory. The view of a *time-based* adversary is defined by

$$view^{time} : \sigma_0 e_0 \dots e_{n-1} \sigma_n \mapsto t_{hit} \cdot |\{i \mid e_i = hit\}| + t_{miss} \cdot |\{i \mid e_i = miss\}| + t_{\perp} \cdot |\{i \mid e_i = \perp\}|$$

and captures that the adversary can determine the overall execution time of the program. Here, t_{hit}, t_{miss}, and $t_{\perp}$ are the execution times (e.g. in clock cycles) of instructions that imply cache hits, cache misses, or no memory accesses at all. While the view of the time-based adversary as defined above is rather simplistic, e.g. disregarding effects of pipelining and out-of-order execution, notice that our semantics and our tool can be extended to cater for a more fine-grained, instruction- and context-dependent modeling of execution times, thanks to its modular design. We denote the side-channels corresponding to the four views by C^{acc}, C^{accd}, C^{tr}, and C^{time}, respectively.

5.3.2 Adversarially Chosen Input

In Section 4.2 we have assumed that the entire initial state is secret. Now we consider the case that initial states are pairs consisting of *high* components that are meant to be kept secret and *low* components that may be provided by the adversary, i.e., $I = I_{hi} \times I_{lo}$. For example, for a decryption algorithm, the high component of the initial state is the key and the low component is the cache state and the ciphertext.

With low inputs, a program and a view define a *family* of channels $C_{\sigma_{lo}} : I_{hi} \to O$, one for each low component $\sigma_{lo} \in I_{lo}$. In this case we strive for an upper bound on $\left|ran(C_{\sigma_{lo}})\right|$, for *all* $\sigma_{lo} \in I_{lo}$. Such a bound enables us to use Theorem 5 to bound the probability of correctly guessing the high component σ_{hi} of the initial state, regardless of the specific choice of σ_{lo}. Note, however, that in multiple program executions with a fixed high input σ_{hi} and different low inputs, information about σ_{hi} may aggregate. A safe upper bound for the range of the corresponding channel is obtained by taking the product of the ranges of the individual channels or, equivalently, by adding the bounds on the number of leaked bits.

We conclude this section by considering the special case in which only the cache state is adversarially chosen, i.e., we consider $I_{lo} = C$ and $I_{hi} = \mathcal{M}$. We show that the size of the range of the channel corresponding to *one* specific initial cache state can be used as a bound for the size of the range of channels for a larger class of initial cache states. Based on this insight we only have to apply our analysis to the case of that one specific state to obtain sound results for *all* cases of that class. This is particularly useful as our analysis is more precise for known than for unknown initial cache states.

Lemma 6. *1. For adversaries* $adv \in \{acc, accd\}$*, all permutation-based replacement policies, and all* $c_1, c_2 \in C$ *such that* c_1 *does not contain empty cache lines: If no program execution accesses blocks in* c_1 *or* c_2*, then* $\left|ran(C^{adv}_{c_1})\right| \geq \left|ran(C^{adv}_{c_2})\right|$.

2. *For adversaries* $adv \in \{time, tr\}$*, all permutation-based replacement policies, and all* $c \in C$*: If no program execution accesses blocks in* c*, then* $\left|ran(C_{\emptyset}^{adv})\right| = \left|ran(C_{c}^{adv})\right|$.

3. *For adversaries* $adv \in \{acc, accd\}$*, the LRU replacement policy, and all* $c \in C$*:* $\left|ran(C_{\emptyset}^{adv})\right| \geq \left|ran(C_{c}^{adv})\right|$.

Here, $\emptyset$ *is a shorthand for the empty cache state.*

Proof. For the proof of (1) and (2) we rely on two properties of permutation-based policies. First, newly inserted blocks have age 0. Second, upon a cache hit, the permutation that is applied to the ages of memory blocks is determined by the age of the requested block. As we assume that the program does not touch any blocks from c_1 or c_2, the ages of the blocks that are loaded during each execution—as well as cache effects—are entirely determined by the sequence of memory accesses of the program. In particular, $ran(C_{c_1}^{adv}) = ran(C_{c_2}^{adv})$ for $adv \in \{time, tr\}$, where c_1 may be empty. For $adv \in \{acc, accd\}$, we define a function that maps a cache in $ran(C_{c_1}^{adv})$ to a cache in $ran(C_{c_2}^{adv})$ by replacing each block *block* that is also contained in c_1 (i.e., $b = c_1(i)$, for some $i < k$), with the block b' of the same age in c_2 (i.e., $b' = c_2(i)$). This function is well-defined because all lines of c_1 are filled with distinct blocks. The mapping is always surjective, and it is also injective if c_2 does also not contain empty lines. The non-injective case corresponds to the fact that an access-based adversary cannot distinguish between the empty lines in c_2.

For (3) we define a mapping $f_c\colon\ ran(C_{\emptyset}^{adv}) \to ran(C_{c}^{adv})$ and show that it is surjective. For simplicity, we consider only one cache set, which we view as a sequence of blocks that are indexed by their ages. Then $f_c(d)$ is obtained by appending to the end of d the subsequence of blocks in c that do not appear in d. For showing surjectivity of f_c, pick a state d' in $ran(C_{c}^{adv})$ and consider any sequence of memory accesses that leads to d' from c. When applied to the empty cache state, that sequence leads to a cache state d such that $f_c(d) = d'$. Note that f_c is well-defined because, for LRU, the ordering of the blocks in c that do not appear in d does not depend on the sequence of memory accesses that leads to d (which is, e.g., not true for PLRU). Also note that f_c is in general not injective: A program that either accesses block b or no block at all will produce two possible final cache states when run on an empty initial cache, but only one possible final cache state when run on an initial cache that contains only block b. □

5.4 Tool Design and Implementation

In this section we describe the architecture and implementation of CacheAudit.

We take advantage of the compositionality of the framework described in Section 4.3 and use a generic iterator module to compute fixpoints, where we rely on independent modules for the abstract domains that correspond to the components of the $next^{\sharp}$ operation. Figure 5.2 depicts the overall architecture of CacheAudit, with the individual modules described below.

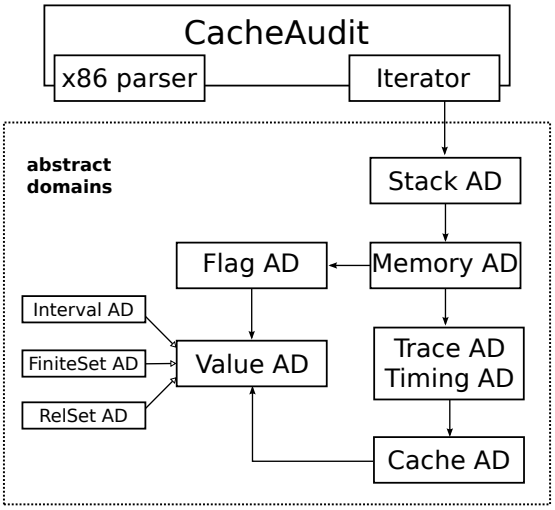


Figure 5.2: The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships.

The current version of CacheAudit allows analyzing a first level cache that is parametric in the cache size, the line size, the associativity, and the replacement policy. We currently support the permutation-based policies LRU, FIFO, and PLRU. We implement a *write-through* cache with *no write-allocate*, i.e., cache writes are directly written to main memory, and when a write-miss occurs, no data is loaded to cache.

5.4.1 Control Flow Reconstruction

The first stage of the analysis is similar to a compiler front end. The main challenge is that we directly analyze x86 executables with no explicit control flow graph, which we need for guiding the fixpoint computation.

For the parsing phase, we rely on Chlipala's parser for x86 executables [119], which we extend to a set of instructions that is sufficient for our case studies (but not complete). For the control-flow reconstruction, we consider only programs without dynamically computed jump and call targets, which is why it suffices to identify the basic blocks and link them according to the corresponding branching conditions and (static) branch targets. We plan to integrate more sophisticated techniques for control-flow reconstruction [120] in the future.

5.4.2 Iterator

The iterator module is responsible for the computation of the $next^\sharp$ operator and of the approximation of its fixpoint using adequate iteration strategies [109]. Our analysis uses the *iterative strategy* [121], i.e., it stabilizes components of the abstract control flow

graph according to a weak topological ordering, which we compute using Bourdoncle's algorithm.

The iterator also implements parts of the reduced cardinal power, based on the labels computed according to the control-flow graph: Each label is associated with an initial abstract state. The analysis computes the effect of the commands executed from that label to its successors on the initial abstract state, and propagates the resulting final states using the abstract domains described below. To increase precision, we expand locations using loop unfolding, so that we have a number of different initial and final abstract states for each label inside loops, depending on a parameter describing the number of unfoldings we want to perform. Most of our examples (e.g. cryptographic algorithms) require only a small, constant number of loop iterations, and we can choose unfolding parameters that avoid joining states stemming from different iterations.

5.4.3 Abstract Domains

As described in Section 4.3, we decompose the abstract domain used by the iterator into mostly independent domains describing different aspects of the concrete semantics.

Value Abstract Domains A value abstract domain represents sets of mappings from variables to (integer) values. Value domains are used by the cache abstract domain to represent ages of blocks in the cache (in that case, the variables are the ages of blocks), and by the flag abstract domain to represent values stored at the addresses used in the program. We have implemented different value abstract domains, such as the interval domain, an exact finite sets domain (where the sets become intervals when they are growing too large) and a relational set domain, which is described in [122].

Flag Abstract Domain In x86 binaries, there are no high level guards: instead, most operations modify flags that are then queried in conditional branches. In order to deal precisely with such branches, we need to record relational information between the values of variables and the values of these flags. To that end, for each operation that modifies the flags, we compute an over-approximation of the values of the arguments that may lead to a particular flag combination. The flag abstract domain works in conjunction with a value abstract domain to store the state of registers and memory other than flags. It represents an abstract state as a mapping from values of flags to elements of the value abstract domain. When the analysis reaches a conditional branch, it can identify which combination of flag values corresponds to the branch and propagate the appropriate abstract values.

Memory Abstract Domain The memory abstract domain associates memory addresses and registers with variables and translates machine instructions into the corresponding operations on those variables, which are represented using flag abstract

domains as described above. One important aspect for efficiency is that variables corresponding to addresses are created dynamically during the analysis whenever they are needed. The memory abstract domain further records all accesses to main memory using a cache abstract domain, as described below.

Stack Abstract Domain Operations on the stack are handled by a dedicated stack abstract domain. In this way the memory abstract domain does not have to deal with stack operations such as procedure calls, for which special techniques can be implemented to achieve precise interprocedural analysis.

Cache, Trace, and Timing Abstract Domains The cache abstract domain tracks information about the cache state. We represent this state by sets of mappings from blocks to ages in the cache, which we implement using an instance of value abstract domains. Effects from the memory domain are passed to the cache domain through the trace domain. The cache domain tracks which addresses are accessed during computation and returns information about the presence or absence of cache hits and misses to the trace domain. The timings are then obtained as an abstraction from the traces. We describe the details of cache, trace, and timing domains in Section 5.5 below.

5.5 Abstract Domains for Cache Adversaries

5.5.1 Domains for cache states

Abstractions of cache states are at the heart of analyses for all three cache adversaries considered in this chapter. Thus, precise abstraction of cache states is crucial to determine tight leakage bounds.

The current state-of-the-art abstraction for LRU replacement by [118] maintains an upper and a lower bound on the age of every memory block. This abstraction was developed with the goal of classifying memory accesses as cache hits or cache misses. In contrast, our goal is to develop abstractions that yield tight bounds on the maximal leakage of a channel. For access-based adversaries the leakage is bounded by the size of the concretization of an abstract cache state, i.e. the size of the set of concrete cache states represented by the abstract state. To derive tighter leakage bounds, we improve previous work along two dimensions:

1. Instead of *intervals* of ages, we maintain *sets* of ages of memory blocks.
2. Upon each cache update, we apply a reduction that eliminates impossible combinations of the ages of the blocks within each cache set.

In addition to increasing precision, these improvements enable us to analyze caches with LRU, FIFO and PLRU replacement in a simple and uniform manner. Interval-based analysis of FIFO and PLRU has been shown to be rather imprecise in the context of worst-case execution time analysis [123].

Representation and Concretization In our domain, an abstract cache state $c^\sharp : \mathcal{B} \to \mathcal{P}(A)$ maintains a set of possible ages for each memory block. We update the ages of the blocks belonging to different cache sets independently; in particular, the concretization of an abstract cache state is the Cartesian product of the concretizations of the individual sets. We present a formalization of the case in which the cache has only one set that is initially empty, and we informally discuss the extension to multiple cache sets that do not contain empty lines (as required for Lemma 6(1)).

At its core, the concretization of a cache set corresponds to the Cartesian product of the ages of the blocks it contains. However, we filter out states that are unreachable in real caches, namely (a) those in which two distinct blocks have the same age, unless that age is k; (b) those with invalid age combinations. For example, for LRU and FIFO replacement, a block of age $a \in \{1, \ldots, k-1\}$ is necessarily preceded by a block of age $a-1$, i.e. cache sets never contain "holes". For PLRU replacement, such holes are possible at certain positions, but not at others.

For a given replacement policy, we represent the set of valid age combinations as a subset $V \subseteq \mathcal{P}(A)$. For example, in a 4-way LRU cache, $\{0,1,2,4\} \in V_{\text{LRU}}$ but $\{0,1,3,4\} \notin V_{\text{LRU}}$ because the missing age 2 constitutes an impossible hole. We use the following algorithm to compute the set V of valid age combinations for a replacement policy defined by permutations Π_i:

(1) $V := \{\emptyset\}$; $R := \emptyset$;

(2) Choose $S \in V \setminus R$; $R := R \cup \{S\}$

   (a) Simulate a cache miss by incrementing ages in S and adding 0:
   $S_M := \{\min(a+1, k) \mid a \in S\} \cup \{0\}$

   (b) Simulate cache hits to blocks of ages $i \in S$:
   $S_i := \{\Pi_i(a) \mid a \in S\}$

(3) $V := V \cup \{S_M\} \cup \bigcup_{i \in S} \{S_i\}$

(4) If $V \setminus R \neq \emptyset$ then go to step (2);

(5) Return V;

Technically, the algorithm computes the fixpoint that is reached when updating the empty state with arbitrary sequences of hits and misses. It follows by construction that the final set V contains all possible age combinations for the replacement policy represented by the permutations Π.

With this, we define the concretization $\gamma_C(c^\sharp)$ of an abstract cache (with one cache set) as the Cartesian product of the sets of ages of memory blocks, from which we remove states that map different blocks to the same age and states whose age combinations are not represented in V:

$$\gamma_C(c^\sharp) = \{c \in \mathcal{B} \to A \mid \forall b \in \mathcal{B} : c(b) \in c^\sharp(b) \ \wedge \forall a, b \in \mathcal{B} : a \neq b \Rightarrow (c(a) \neq c(b) \vee c(a) = c(b) = k) \wedge \{c(b) \mid b \in \mathcal{B}\} \in V\}$$

So far we have assumed that the cache is initially empty. For *non-empty* initial caches, the holes in valid age combinations will contain blocks from the initial state, which can be distinguished. To account for this, we augment V to represent the identities of those blocks and extend the fixpoint computation accordingly. As there are only k possible blocks in the initial state, the fixpoint can still be effectively computed. The concretization $\gamma_C(c^\sharp)$ then ensures that every concrete cache is an extension of a cache in that fixpoint.

Abstract Cache Update We implement two algorithms for abstract cache updates, where each offers a different trade-off between precision and performance. First, we implement an abstract transformer that updates the possible ages of each memory block considering only the possible ages of the accessed block, but without considering ages other blocks in the same cache set. This corresponds to a *direct product* [109]. For LRU and FIFO we gain precision by additionally performing a reduction after each cache update, which makes sure that no impossible states with holes are represented. The reduction works by restricting the maximal age of blocks in a cache set to the number of currently cached blocks. The following example shows imprecisions we avoid when using the reduction, and points out additional imprecisions that motivate the use of the second algorithm we use for abstract update. It is based on actual allocations we encountered when analyzing AES (see Section 5.6).

Example 4. *Let a, b, c, d be blocks that fall into the same set of a 4-way LRU cache, and e be a block that falls into a different cache set. We write $x \in \{n_1, n_2, n_3\}$ if block x has possible ages n_1, n_2, n_3. At a point of the program execution, the analysis has reached one of the following states:*

(i) $a \in \{0, 1\}$; $b \in \{0, 4\}$; $c, d \in \{4\}$

(ii) $a \in \{0, 1\}$; $b \in \{0, 4\}$; $c \in \{1, 2\}$; $d \in \{4\}$

(iii) $a \in \{0, 1, 2\}$; $b \in \{0, 4\}$; $c \in \{2, 3\}$; $d \in \{0, 1, 2\}$

In state (i), if ages are updated without a reduction, then an access to b will result in $a \in \{1, 2\}$. This would mean that a possible allocation is $(b = 0, a = 2)$, however it cannot occur with LRU because it has a hole, as there is no element with age 1. If we perform the reduction described above, as only two elements may be cached, we will only allow ages 0 or 1, which solves the problem. This reduction, however, cannot solve the following problems. Consider a program reaching the state (ii), where we know that either b or e is accessed, after performing reduction, the updated states will be $a \in \{0, 1, 2\}$; $b \in \{0, 4\}$; $c \in \{1, 2\}$. Disallowing c to grow to 3 eliminated a possible hole, however here we obtain a possible allocation $(b = 0, c = 1, a = 2)$, which cannot be reached from state (ii), because it would be only reachable from the impossible allocation $(c = 0, a = 1)$. In state (iii), if we access b or e as in state (ii), the resulting reduced state will be $a \in \{0, 1, 2, 3\}$; $b \in \{0, 4\}$; $c \in \{2, 3, 4\}$; $d \in \{0, 1, 2, 3\}$. Now an additional problem can be observed: the allocation $c = 4$ is possible, i.e., c can be outside of the cache; however when starting at state (iii), we know that c must remain in the cache. This imprecision can then propagate, because if c is accessed at a later

point, instead of a sure hit, we will record a hit or a miss.

To avoid the above-mentioned problems, we additionally implement an abstract transformer that concretizes the abstract cache set, updates each concrete state, and then abstracts again. This corresponds to the explicit computation of the *best abstract transformer* [109]. We use it in cases where its computation is feasible, which was the case with most experiments from our case study. The soundness of the best abstract transformer follows by construction. The direct product is sound because (a) the transformer assumes no knowledge on other blocks' ages and thus excludes fewer states than the best abstract transformer, and (b) the reduction removes only impossible configurations which are also removed when concretizing.

The cache join and abstract cache effect are implemented in a straightforward fashion. Two cache states are joined by a set union of the possible ages of all blocks. The abstract cache effect is a union of the effects of all possibly accessed blocks, for all possible ages. The soundness of these operations follows directly.

Lemma 7. *The cache domains are locally sound.*

Counting Cache States We describe the counting of observations for abstract cache states with one cache set; for cache states with more than one cache set, we compute the product of the number of concretizations of the individual sets.

For counting the observations a shared-memory access-based adversary C^{acc} can make, we simply enumerate the concretizations $\gamma_C(c^\sharp)$ and count their number. For a disjoint-memory access-based adversary C^{accd}, we also enumerate concretizations, but we take into account that a disjoint-memory adversary cannot distinguish between different blocks that have been loaded during execution. That is, we only need to track the number of elements of the fixpoint V that we observe during enumeration of $\gamma_C(c^\sharp)$.

While each counting procedure takes exponential time in the associativity of the cache, this is not a bottleneck in practice, where associativities 2,4,8 are common.

5.5.2 A Domain for Traces

We devise an abstract domain for keeping track of the sets of event traces that may occur during the execution of a program. Following the way events are computed in the concrete, namely as a function from cache states and memory effects (see Section 4.1.3), the abstract cache domain provides abstract cache effects.

In our current implementation of CacheAudit, we use an exact representation for sets of event traces: we can represent any finite set of event traces, and assuming an incoming set of traces $\mathcal{S}$ and a set of cache effects E, we compute the resulting event set precisely as $upd_{\mathcal{E}^\sharp}(\mathcal{S}, E) = \{\sigma.e \mid \sigma \in \mathcal{S} \wedge e \in E\}$.

Then soundness is obvious, since the abstract operation is the same as its concrete counterpart. Due to complete loop unfolding, we do not require widenings, even though the domain contains infinite ascending chains (see Section 5.4.2).

Lemma 8. *The trace domain is locally sound.*

Efficient Implementation of the Event Trace Domain We represent sets of finite event traces corresponding to a program location by a directed acyclic graph (DAG) with vertices V, a dedicated root $r \in V$, and a node labeling $\ell \colon V \to \mathcal{P}(\mathcal{E}) \cup \{\sqcup\}$. In this graph, every node $v \in V$ represents a set of traces $\gamma(v) \in \mathcal{P}(\mathcal{E}^*)$ in the following way:

1. For the root r, $\gamma(r) = \{\epsilon\}$
2. For v with $L(v) = \sqcup$ and predecessors $u_1, \ldots, u_n$, $\gamma(v) = \bigcup_{i=1}^{n} \gamma(u_i)$.
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1, \ldots, u_n$,

$$\gamma(v) = \left\{ t.u \mid u \in L(v) \wedge t \in \bigcup_{i=1}^{n} \gamma(u_i) \right\}$$

Intuitively, every $v \in V$ represents a set of event traces, namely the sequences of labels of paths from r to v.

In the context of CacheAudit, we need to implement two operations on this data structure, namely (a) the join $\sqcup^{\mathcal{E}^\sharp}$ of two sets of traces and the (b) addition $upd_{\mathcal{E}^\sharp}(\mathcal{S}, E)$ of an event to a particular set of traces. For the join of two sets of traces represented by v and w, we add a new vertex u with label $\sqcup$ and add edges from v and w to u. For the extension of a set of traces represented by a vertex v by a set of events E, we first check whether v already has a child w labeled with E. If so, we use w as a representation of the extended set of traces. If not, we add a new vertex u with label E and add an edge (u, v). In this way we make maximal use of sharing and obtain a prefix DAG. The correctness of the representation follows by construction. In CacheAudit, we use hash consing for efficiently building the prefix DAG.

Counting Sets of Traces The following algorithm $count_{tr}$ overapproximates the number of traces that are represented by a given graph.

1. For the root r, $count_{tr}(r) = 1$
2. For v with $L(v) = \sqcup$ and predecessors $u_1, \ldots, u_n$, $count_{tr}(v) = \sum_{i=1}^{n} count_{\tau}(u_i)$
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1, \ldots, u_n$,

$$count_{tr}(v) = |L(v)| \cdot \sum_{i=1}^{n} count_{tr}(u_i)$$

The soundness of this counting, i.e. the fact that $|\gamma(v)| \leq count_{tr}(v)$, follows by construction. Notice that this counting procedure is precise if the labels represent singleton events (because then every trace is uniquely represented in the graph), but that its precision decreases dramatically with larger sets of labels. In our case, labels contain at most three events and the counting is sufficiently precise.

5.5.3 A Domain for Time

We currently model execution time as a simple abstraction of traces, see Section 4.1. In particular, timing is computed from a trace over $\mathcal{E} = \{hit, miss, \perp\}$ by multiplying the number of occurrences of each event by the time they consume: t_{hit}, t_{miss}, and $t_{\perp}$, respectively. The following algorithm $count_{time}$ over-approximates the set of timing behaviors that are represented by a given graph.

1. For the root r, $count_{time}(r) = \{0\}$
2. For v with $L(v) = \sqcup$ and predecessors $u_1, \ldots, u_n$, $count_{time}(v) = \bigcup_{i=1}^{n} count_{time}(u_i)$
3. For v with $L(v) \neq \sqcup$ and predecessors $u_1 \ldots, u_n$,

$$count_{time}(v) = \left\{ t_x + t \;\middle|\; x \in L(v) \wedge t \in \bigcup_{i=1}^{n} count_{time}(u_i) \right\}$$

The soundness of $count_{time}$, i.e. the fact that it delivers a superset of the set of possible timing behaviors, follows by construction.

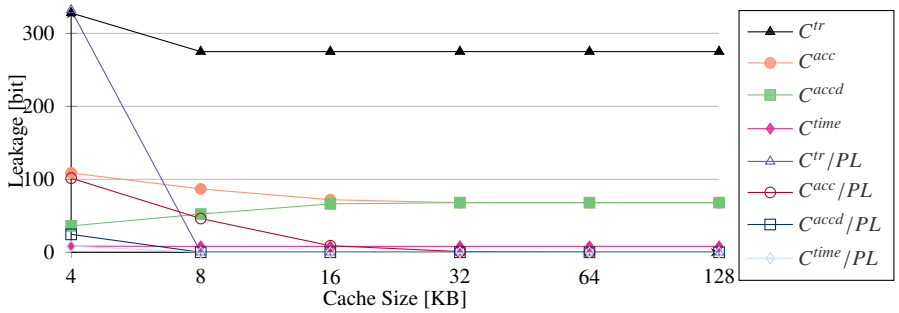
5.6 Case Studies

In this section we demonstrate the capabilities of CacheAudit in case studies where we use it to analyze the cache side-channels of implementations of algorithms for symmetric encryption and sorting. All results are based on the automatic analysis of corresponding 32-bit x86 Linux executables that we compiled using *gcc*.

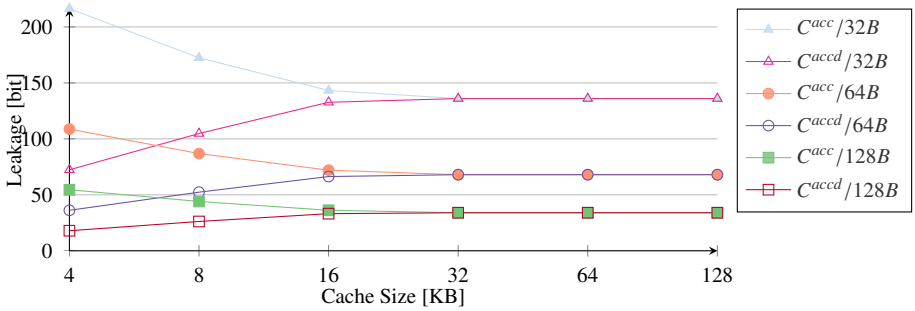
5.6.1 AES

We analyze the AES implementation from the PolarSSL 1.3.7 library with keys of $n \in \{128, 192, 256\}$ bits, where we consider the implementation with and without preloading of lookup tables. We analyze with respect to the $C^{acc}, C^{accd}, C^{tr}, C^{time}$ adversary models, with the LRU, FIFO, and PLRU replacement policies, for a set of cache sizes, associativities, and line sizes. All results are presented as upper bounds of the leakage in bits; for their interpretation see Theorem 5. In some cases, CacheAudit reports upper bounds that exceed the key size n, which corresponds to an imprecision of the static analysis. We opted against truncating to n bits to illustrate the degree of imprecision. We highlight some of our findings.

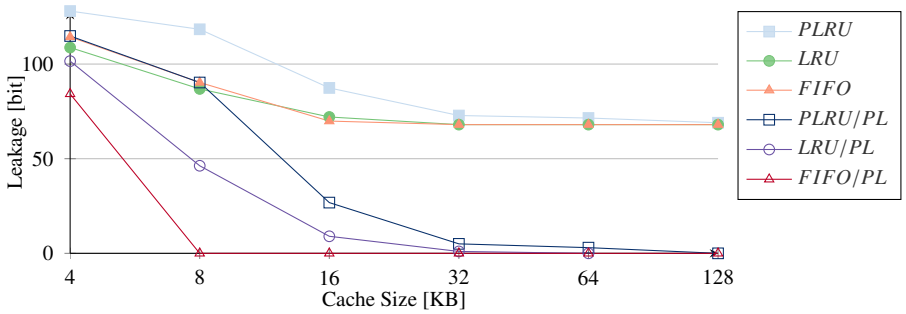
In the following, we first present results for 256-bit keys before we discuss the effect of varying the key size. The base case we consider is an LRU cache with associativity 4 and line size of 64B. We also explore the effect of varying each of these parameters.



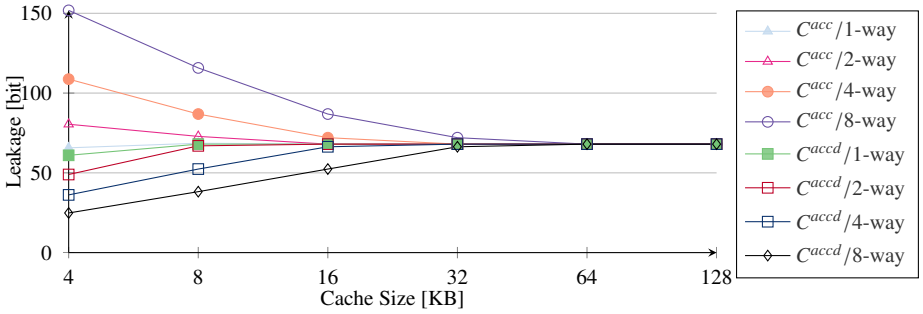
(a) Effect of the attacker model and preloading (PL)



(b) Effect of the cache line size



(c) Effect of the replacement policy



(d) Effect of the associativity

Figure 5.3: Security guarantees for PolarSSL's AES implementation with 256-bit keys. The base case considers a 4-way set associative cache with a line size of 64B, and the LRU replacement policy, and varying cache sizes.

Preloading Preloading the lookup tables almost consistently leads to better security guarantees in all scenarios (see e.g. Figure 5.3a). However, the effect becomes clearly more apparent for cache sizes beyond 8KB, which is explained by the PolarSSL AES tables exceeding the size of the 4KB cache by 256B. For cache sizes that are larger than the preloaded tables, we can prove noninterference for C^{acc} and FIFO, C^{accd} and LRU, and for C^{tr} and C^{time} on LRU, FIFO, and PLRU. For C^{acc} with LRU and PLRU, this result does *not* hold because the adversary can obtain information about the order of memory blocks in the cache.

Line size A larger line size consistently leads to better security guarantees for access-based adversaries (see e.g. Figure 5.3b). This follows because more array indices map to a line, which decreases the resolution of the attacker's observations.

Replacement policy In terms of replacement policies, for C^{accd}, C^{tr}, and C^{time} we consistently derive the lowest bounds for LRU. For C^{acc} and preloading, FIFO exhibits the lowest leakage, with significantly lower bounds than the other policies, as shown in Figure 5.3c. The reason for this is that with LRU and PLRU (but not with FIFO), consecutive cache hits can lead to reordering of the cached elements, and thus the access-based adversary can obtain information about the ordering of memory blocks in the cache.

Associativity When increasing associativity, we observe opposing effects on the leakage of C^{acc} and C^{accd} (see Figure 5.3d). This is explained by the fact that, for a fixed cache size, increasing associativity means decreasing the number of sets. For C^{accd}, which can only observe the number of blocks that have been loaded into each set, this corresponds to a decrease in observational capability; for C^{acc}, which can observe the ordering of blocks, this corresponds to an increase. This difference vanishes for larger cache sizes because then each set contains at most one unique block of the AES tables.

Cache size In terms of *cache size*, we consistently derive lower bounds for larger caches, with the exception of C^{accd}. For C^{accd}, the bounds increase because larger caches correspond to distributing the table to more sets, which increases its possibilities to observe variations. The guarantees we obtain for C^{accd} and C^{acc} converge for caches of 4 ways and sizes beyond 16KB (see e.g. Figure 5.3b). This is due to the fact that each cache set can contain at most one unique block of the 4KB table. In that way, the ability to observe ordering of blocks within a set does not give C^{acc} any advantage.

Key size The choice of key size increases the leakage significantly for C^{tr} and C^{time}, and leads to small variations for C^{acc}, as exemplified in Figure 5.4. The increase in leakage for C^{tr} and C^{time} can be explained by the longer computations for bigger keys: A key size of 128, 192, or 256 bits results in 10, 12, or 14 rounds of transforming the input, respectively. For bigger keys, as more rounds are performed, there are more

accesses to the lookup tables, and each access that cannot be statically predicted as a cache hit or miss doubles the number of possible traces.

	8KB cache				16KB cache			
AES key size	C^{tr}	C^{time}	C^{acc}	C^{accd}	C^{tr}	C^{time}	C^{acc}	C^{accd}
128 bit	199	7.64	89.09	52.79	199	7.64	72.85	66.93
192 bit	223	7.81	87.79	52.79	223	7.81	72.85	66.93
256 bit	279	8.13	90.07	52.79	279	8.13	73.44	66.93

Figure 5.4: Leakage in bits with AES when varying the key size, for a configuration with an 8KB and 16KB 4-way cache, with line size of 64B, with the LRU replacement policy.

The variations for C^{acc} are more subtle and have to do with two contradictory effects. The first effect results from the key expansion process, during which the key is expanded to round keys of 128 bits per round. For an n-bit key, the round keys are computed in a loop, which generates n bits per iteration. Thus, for smaller keys this loop requires *more* iterations: 10 for 128-bit keys, 8 for 192-bit keys, and 7 for 256-bit keys. The leakage differs because within the i-th iteration, an integer round-constant `rcon[i]` is read from an array in memory, from which more values are read if the key is small, thus more blocks may compete with the lookup tables for the same cache sets. This explains why in Figure 5.4, for the 8KB cache, there is less C^{acc}-leakage with 192-bit keys than with 128-bit keys.

The second effect results from the size of the expanded key: when a smaller encryption key is being used, there are *less* round keys, and less blocks corresponding to the round keys compete with the lookup tables for their position in the cache sets. Thus, the end-state can contain less possible ages for those blocks, corresponding to less leakage. This explains the increased C^{acc}-leakage for 256-bit keys in Figure 5.4.

For C^{accd}, no difference is observed between key sizes for the analyzed cache configurations, as the above-described effects affect the ordering of blocks in cache, but not whether those blocks are in cache or not.

5.6.2 The eSTREAM Portfolio

The goal of the eSTREAM project [116] was to foster the creation of novel practical stream ciphers. The project concluded in 2008 with the announcement of the final eSTREAM portfolio, which consists of four ciphers that are particularly suited for implementation in software (Profile 1) and of three ciphers that are suited for implementation in hardware (Profile 2).

We applied CacheAudit to analyse the reference implementations of the ciphers from the eSTREAM Profile 1 portfolio, namely HC-128, Rabbit, Salsa20, and Sosemanuk. We tested the versions of the algorithms with 128-bit keys for the encryption of a 512-byte message, for a 4-way cache with a line size of 64B. The results for LRU are summarized in Figure 5.5a for C^{acc}, Figure 5.5b for C^{accd}, Figure 5.5c for C^{tr},

Figure 5.5d for C^{time}, and are briefly discussed in the following paragraphs. The effects on which we elaborate are observable with all replacement policies, and we restrict the presentation to LRU for brevity.

5.6.2.1 HC-128

HC-128 is a stream cipher by [124] that relies on a 128-bit key, a 128-bit initialization vector, and an internal state of 4KB, which is stored in two S-boxes of 512 entries of 32-bit values each. During keystream generation, new S-box values are generated every 512 steps.

Cache attacks against the HC series of ciphers have been demonstrated by [125] and [126], for a different (but similar) adversary model. For small caches, CacheAudit confirms this, and for all considered adversary models we obtain a non-zero leakage. When increasing the size of the cache, CacheAudit shows that the leakage disappears; varying the cache size was not considered by the mentioned attacks. The effect on HC-128 leakage when varying the cache size is similar to AES leakage with preloading (see Section 5.6.1 and compare with Figure 5.3a and Figure 5.3c). The reason for this is that (a) AES and HC-128 rely on lookup tables of similar sizes; (b) dynamically generating HC-128 S-boxes makes sure that they are freshly loaded in cache as a whole, similarly to preloading AES lookup tables. Non-zero leakage is observed when the cache is small, as some of the memory blocks containing S-box values are evicted from cache or other memory competes with them for the same cache sets. For tested configurations with a bigger cache, we obtain zero-leakage.

5.6.2.2 Rabbit

Rabbit is a stream cipher by [127] that relies on a 128-bit key and a 64-bit initialization vector. A set of eight 32-bit state registers and eight 32-bit counters is used to perform encryption based on basic arithmetic and bit-operations. The lack of key-dependent memory lookups intends to avoid any leakage to the cache. This is reflected by the results we obtained with CacheAudit: for all adversary models and all tested cache configurations, we obtain zero-leakage.

It should be noted that [128] observes the possibility of a timing leak due to operand-dependent timing of integer multiplication on platforms such as the Motorola PowerPC G4e 7450. The current version of CacheAudit does not support operand-dependent timing and hence does not detect this kind of leak.

5.6.2.3 Salsa20

Salsa20 is a stream cipher by [129]. Internally, the cipher uses XOR, addition mod 2^{32}, and constant-distance rotation operations on an internal state of 16 32-bit words. The lack of key-dependent memory lookups intends to avoid any leakage to the cache. With CacheAudit we could formally confirm this intuition, and we consistently obtain upper bounds of 0 for the leakage.

5.6.2.4 Sosemanuk

Sosemanuk is a stream cipher by [130] that relies on keys of length ranging from 128 to 256 bits and an initialization vector of 128 bits. Sosemanuk uses a 10-word linear feedback shift register, a finite-state machine, and an output function for combining both of their outputs into the keystream. The reference implementation we analyze relies on static tables of 4 KB for fast implementation of the feedback register, which have been shown to be susceptible to cache attacks [131]. The analysis using CacheAudit confirms this weakness. In particular, we obtain non-zero leakage for all tested configurations and adversary models, with higher leakage observed for smaller cache. In this respect, the results resemble those for AES without preloading (see Section 5.6.1).

Summary With CacheAudit, we can prove zero-leakage for Rabbit and Salsa20 for all tested configurations. For HC-128, we prove zero-leakage only for bigger cache sizes. For Sosemanuk we obtain non-zero leakage bounds for all tested configurations; for small caches, Sosemanuk's leakage bounds are lower than the bounds for HC-128.

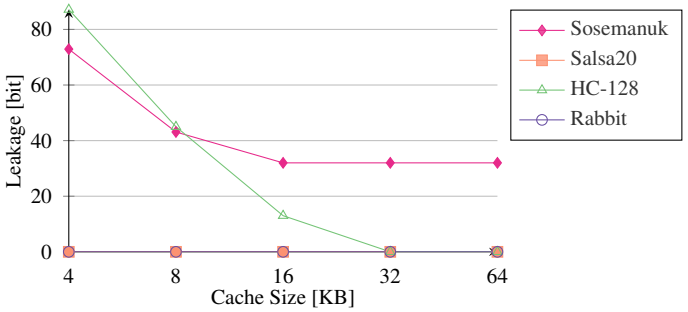
5.6.3 Sorting Algorithms

In this section we use CacheAudit to establish bounds on the cache side-channels of different sorting algorithms. This case study is inspired by an early investigation of secure sorting algorithms [132]. While the authors of [132] consider only time-based adversaries and noninterference as a security property, CacheAudit allows us to give quantitative answers for a comprehensive set of side-channel adversaries, based on the binary executables and concrete cache models.

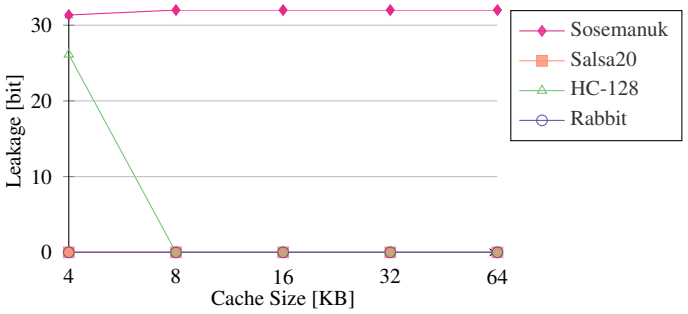
As examples, we use implementations from [133] of the sorting algorithms BubbleSort (see Figure 5.1), InsertionSort, and SelectionSort. We use integer arrays of lengths between 8 and 64.

The results of our analysis are summarized in Figure 5.6. In the following we highlight some of our findings.

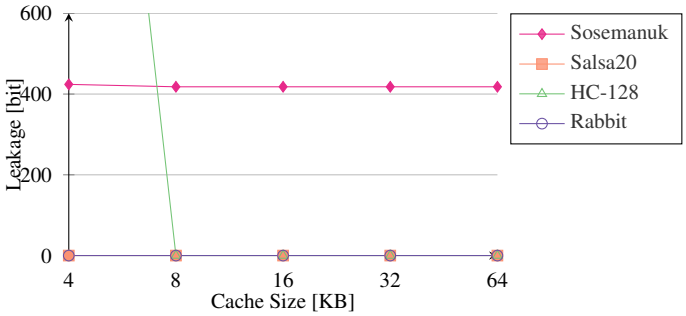
- We obtain the same bounds for BubbleSort and SelectionSort, which is explained by the similar structure of their control flow. A detailed explanation of those bounds is given in Section 5.2. InsertionSort has a different control flow structure, which is reflected by our data. In particular, InsertionSort has only $n!$ possible execution traces due to the possibility of leaving the inner loop, which leads to better bounds w.r.t. trace-based adversaries. However, InsertionSort leaks more information to timing-based adversaries, because the number of iterations in the inner loop varies and thus fewer executions have the same timing.
- For access-based adversaries we obtain zero bounds for all algorithms. For trace-based adversaries, the derived bounds do not imply meaningful security guarantees: the bounds reported for InsertionSort are in the order of $\log_2(n!)$, which corresponds to the maximum information contained in the ordering of the elements; the bounds



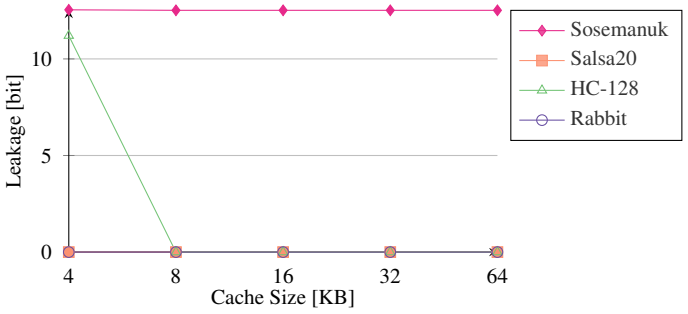
(a) C^{acc}



(b) C^{accd}



(c) C^{tr}



(d) C^{time}

Figure 5.5: Security guarantees of eSTREAM finalists for $C^{acc}, C^{accd}, C^{tr}, C^{time}$, for varying cache sizes. The results are given for a 4-way set associative cache with a line size of 64B and the LRU replacement policy.

len	8			16			32			64		
	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}	C^{tr}	C^{time}	C^{acc}
BS	28	4.9	0	120	6.9	0	496	9	0	2016	11	0
IS	15.3	6.9	0	44.3	10.1	0	118	12.5	0	296	14.6	0
SS	28	4.9	0	120	6.9	0	496	9	0	2016	11	0

Figure 5.6: The table illustrates the security guarantees derived by CacheAudit for the implementations of BubbleSort (BS), InsertionSort (IS), and SelectionSort (SS), for trace-based, timing-based, and access-based adversaries, for LRU caches of 4KB and line sizes of 32B, for array length (len) between 8 and 64.

reported for the other sorting algorithms exceed this maximum, which is caused by the imprecision of the static analysis.

- We performed an analysis of the sorting algorithms for smaller (256B) and larger (64KB) cache sizes and obtained the exact same bounds as in Figure 5.6, with the exception of the case of arrays of 64 entries and 256B caches: there the leakage increases because the arrays do not fit entirely into the cache due to their misalignment with the memory blocks.

5.6.4 Discussion

A number of comments are in order when interpreting the bounds delivered by CacheAudit.

Meaning of Bounds The quantities computed by CacheAudit are upper bounds on the leaked information that are not necessarily tight, that is, they may be pessimistic. There are two reasons why the bounds may be pessimistic: First, CacheAudit may over-estimate the amount of leaked information due to imprecision of the static analysis. Second, the secret input may not be effectively recoverable from the leaked information by an adversary that is computationally bounded.

The fact that CacheAudit delivers upper bounds has two consequences. First, the results can only be used for certifying that a system is secure; they cannot be used for proving that it is *not*. Second, the natural ordering on bounds cannot be directly used for comparing the real-world security of systems. For example, "at most two bits leak" is a correct (but pessimistic) bound for a system that does not leak any information, and "at most one bit leaks" is a correct (and tight) bound for a system that leaks one bit. The first bound is lower than the second, even though the first system is more secure than the second.

Instead, lower bounds represent a better state of affairs in systematic reasoning about the security of a system, which is a desirable goal for implementors of (cryptographic) algorithms and side-channel countermeasures.

Use of Imperfect Models The guarantees delivered by CacheAudit are only valid to the extent to which the models used accurately capture the relevant aspects of the

execution platform. A recent empirical study of OS-level side-channels on different platforms [134] shows that advanced microarchitectural features may interfere with the cache, which can render countermeasures ineffective — and formal guarantees invalid. See Section 5.8 for a discussion of the challenges associated with extending CacheAudit with such advanced features.

Multiple Executions For the case of the cryptosystems we analyzed, the bounds hold for the leakage about the key in *one* execution, with respect to *any* payload. For the case of zero leakage (i.e., noninterference), the bounds trivially extend to bounds for multiple executions and imply strong security guarantees. For the case of non-zero leakage, the bounds can add up when repeatedly running the victim process with a fixed key and varying payload, leading to a decrease in security guarantees. One of our prime targets for future work is to derive security guarantees that hold for multiple executions of the victim process. One possibility to achieve this is to employ leakage-resilient cryptosystems [81, 82], where our work can be used to bound the range of the leakage functions, as demonstrated in [84].

Initial Cache State We obtained all bounds in our experiments for initial states that do not contain blocks that are accessed by the program. As described in Section 5.3.2, they immediately extend to bounds for initial cache states containing empty lines. This is relevant, e.g. for an adversary who can fill the initial cache state only with lines from its own disjoint memory space. For LRU and access-based adversaries, our bounds extend to arbitrary initial cache states without further restriction, as justified by Lemma 6(3).

5.7 Related Work

Existing work on mitigation techniques for cache side-channels can be classified as hardware-based, OS-based, code-based, or mixed:

- Hardware-based techniques include [135], who present a novel microarchitecture that facilitates information-flow tracking by design, where they use noninterference as a baseline confidentiality property. [136] propose non-monopolizable caches, which is a hardware defense against access-based attacks that puts a bound on the number of lines in each cache set that can be used by a process. Depending on the degree of "non-monopolization", an adversary cannot evict any or only some of the victim's data from the cache, which eliminates or at least weakens access-based attacks. [137] propose novel cache architectures that achieve attractive trade-offs between security and performance. In particular, they rely on randomized cache replacement policies that are designed to achieve security.
- OS-based techniques include StealthMem [22], a system-level defense against cache-timing attacks in virtualized environments. The core of StealthMem is a software-based mechanism that locks pages of a virtual machine into the cache and prevents

their eviction by other VMs. StealthMem can be seen as a lightweight variant of flushing/preloading countermeasures. A formalization of StealthMem is provided in [138]. [139] propose CloudFlow, which is a cloud-wide information-control layer based on OpenStack, which relies on a novel, fast VM introspection mechanism. [140] propose system-level defenses for isolating machines in software-as-a-service environments, such as cache-aware CPU core assignment and cache-aware memory-management. [141] proposes information-flow control based on explicit timing labels, together with operating system support for its enforcement.

- Code-based techniques include the program counter security model [142], which is to assume that an adversary can observe the value of the program counter at every step. The authors also propose a program transformation that achieves security in this model. Security implies resistance against control-flow based timing attacks, but does not account for leaks through secret-dependent memory lookups. [143] propose a code transformation for Java Bytecode to eliminate control-flow based attacks in Java Bytecode, together with proofs of soundness. [144] extend this timing model with execution histories, offering a hook for reasoning about cache state. [29] use bitslicing to avoid the use of data caches and show that this leads to efficient software implementations of AES. Finally, [114] discuss practical coding techniques for mitigating cache attacks on x86 CPUs.
- Mixed techniques include [113], who propose an approach for mitigating timing side-channels that is based on contracts between software and hardware. The contract is enforced on the software side using a type system, and on the hardware side, e.g., by using dedicated hardware such as partitioned caches. The analysis ensures that an adversary cannot obtain any information by observing public parts of the memory; any confidential information the adversary obtains must be via timing, which is controlled using dedicated mitigate commands that reduce the number of possible timing observations.

The goal of our work is orthogonal to those approaches in that we focus entirely on the *analysis* of microarchitectural side-channels rather than on their mitigation. Our approach does not rely on a specific platform; rather it can be applied to any language and hardware architecture for which abstractions are in place.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [60], and the development of CacheAudit is inspired by a feasibility study [66]. Those points are covered in Chapter 2.

5.8 Challenges for Future Work

While CacheAudit relies on more accurate models of cache and timing than any information-flow analysis we are aware of, there are several timing-relevant features of microarchitectures that it does not yet capture (and make assertions about), including *second and third level caches*; *shared caches* in multi cores; *DRAM*, commonly used as main memory, which—just like caches—exhibits varying access latencies depending

on the history of memory accesses; *speculation*, which may introduce memory accesses that are not part of the "logical" execution of the program; *out-of-order execution*, which may reorder memory accesses; *translation lookaside buffers (TLBs)* and other mechanisms related to the implementation of *virtual memory*.

There are two immediate challenges regarding the features mentioned above:

1. To obtain detailed models that faithfully capture the behavior of these features in modern microarchitectures.

2. To devise abstractions of these models that enable precise, yet efficient analysis.

Challenge (1) is daunting, as modern microarchitectures are extremely complex and at the same time poorly documented, at least when it comes to documentation that is publicly available. One promising approach to deal with this challenge is to apply techniques from machine learning to reverse engineer microarchitectural models, as recently demonstrated in [117]. Such approaches will, however, never be able to provide absolute certainty about the correctness of the models.

Challenge (2) is equally daunting. The worst-case execution time (WCET) community has gathered experience in the design of analyses for some of the features mentioned above, and it has been observed that speculation and out-of-order execution dramatically increase analysis complexity. Current consumer microarchitectures are at least an order of magnitude more complex than the most advanced microarchitectures used in safety-critical systems for which WCET analyses have been devised. Thus, breakthroughs in analysis technology will be required to solve Challenge (2).

Maybe a more viable approach than to attack Challenges (1) and (2) by devising ever more complex models and analyses, is to actually make *stronger abstractions* that are based upon *fewer* assumptions about the microarchitecture. This would have two beneficial effects:

- An increase in confidence in the analysis results, as they would be based on fewer assumptions that may or may not hold in reality.

- The fewer assumptions are made, the more microarchitectures satisfy the assumptions. Security statements could possibly be made for large classes of architectures.

As an example, in the context of cache side-channels one could base the side-channel analysis on a lower bound on the cache capacity and a lower bound on the number of cache sets, rather than the cache's exact geometry. The challenge is to reduce assumptions without sacrificing precision.

In a similar spirit, side-channel analysis may be performed at a higher level of abstraction. Proposals such as StealthMem [22] introduce a security layer that enables the construction of secure programs without having to know about microarchitectural details. Can we characterize the guarantees that such *security APIs* provide and analyze applications based on these guarantees? Also, can we analyze the implementations of these *security APIs* to prove that they deliver the promised guarantees?

5.9 Conclusions

We presented CacheAudit, the first automatic tool for the static derivation of formal, quantitative security guarantees against cache side-channel attacks. We demonstrate the usefulness of CacheAudit by establishing formal security guarantees for binary executables of sorting algorithms and state-of-the-art cryptosystems.

6 Rigorous Analysis of Software Countermeasures against Cache Attacks

6.1 Introduction

A large number of techniques have been proposed to counter cache-based side-channel attacks. Some proposals work at the level of the operating system [22], others at the level of the hardware architecture [137] or the cryptographic protocol [81]. However, so far only software countermeasures have seen wide-spread adoption in practice.

The most defensive countermeasure is to forbid control flow, memory accesses, and execution time of individual instructions to depend on secret data. While such code is easily seen to prevent leaks through instruction and data caches, it also prevents the performance gains enabled by such accelerators. More permissive countermeasures are to ensure that both branches of each conditional fit into a single line of the instruction cache, to preload lookup tables, or to permit secret-dependent memory access patterns as long as they are secret-independent at the granularity of cache lines or sets. Such permissive code can benefit from hardware acceleration, however, analyzing its security requires intricate reasoning about the interactions of the program and the hardware platform and has so far only been done for restricted cases.

A major hurdle for reasoning about such interactions is that it requires support for accurate, logical and arithmetic reasoning on pointers (for tracking cache alignment), but it also requires dealing with pointers symbolically (for capturing dynamically allocated memory used by multi-precision integer arithmetic). In this chapter we present novel reasoning techniques that support both features. Based on these techniques we devise novel abstractions that capture a range of adversary models, including those that can observe sequences of accessed addresses, or those that can observe sequences of accessed memory blocks. We frame both contributions in terms of novel abstract domains, which we implement on top of the CacheAudit static analyzer [45].

We evaluate the effectiveness of our techniques in a case study where we perform the first formal analysis of commonly used software countermeasures for protecting modular exponentiation algorithms against cache side-channels. The chapter contains a

detailed description of our case study; here we highlight the following results:

- We analyze the security of the scatter/gather countermeasure used in OpenSSL 1.0.2f for protecting window-based modular exponentiation. Scatter/gather ensures that the pattern of data cache accesses is secret-independent at the level of granularity of cache lines and, indeed, our analysis of the binary executables reports security against adversaries that can monitor only cache line accesses.

- Our analysis of the scatter/gather countermeasure also reports a leak with respect to adversaries that can monitor memory accesses at a more fine-grained resolution. This weakness that has been exploited in the CacheBleed attack [145], where the adversary observes accesses to the individual banks within a cache line. We analyze the variant of scatter/gather published in OpenSSL 1.0.2g as a response to the attack and prove its security with respect to powerful adversaries that can monitor the full address trace.

- Our analysis detects the side-channel in the square-and-multiply based algorithm in libgcrypt 1.5.2 that has been exploited in [14, 146], but can prove the absence of instruction and data cache leaks in the square-and-*always*-multiply algorithm used in libgcrypt 1.5.3, for some compiler optimization levels.

Overall, our results illustrate the dependency of software countermeasures against cache attacks on brittle details of the compilation and the hardware architecture, and they demonstrate how the techniques developed in our chapter can effectively support rigorous analysis of software countermeasures.

In summary, our contributions are to devise novel techniques that enable cache-aware reasoning about dynamically allocated memory, and to put these techniques to work in the first rigorous analysis of widely deployed permissive countermeasures against cache side-channel attacks.

The remainder of this chapter is structured as follows. In Section 6.2 we illustrate the scope of our techniques by example. In Section 6.3 we define the adversary models. In Section 6.4 and 6.5 we define our novel abstract domains. We present our case studies in Section 6.6 before we revisit prior art and conclude in Sections 7.5 and 5.9, respectively.

6.2 Illustrative Example

We illustrate the scope of the techniques developed in this chapter using a problem that arises in implementations of windowed modular exponentiation. There, powers of the base are pre-computed and stored in a table for future lookup. Figure 6.1 shows an example memory layout of two such pre-computed values p_1 and p_2, each of 3072 bits. An adversary that observes accesses to the six memory blocks starting at `80eb140` knows that p_2 was requested, which can give rise to effective key-recovery attacks [146].

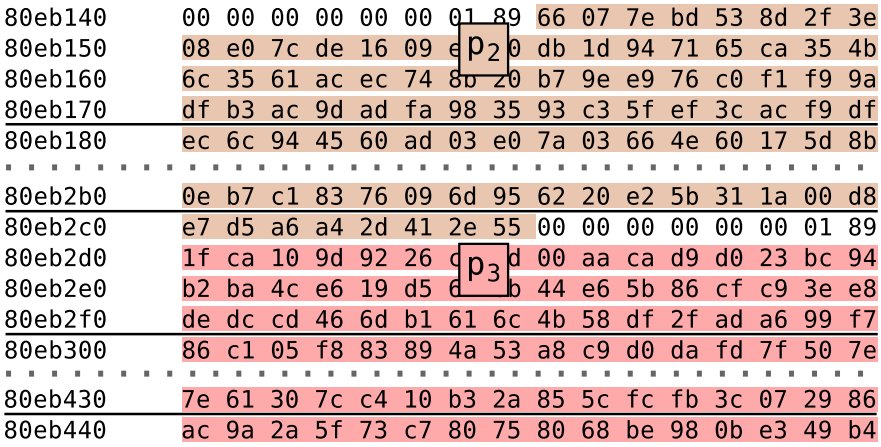


Figure 6.1: Layout of pre-computed values in main memory, for the windowed modular exponentiation implementation from libgcrypt 1.6.1. Data highlighted in different colors correspond to the pre-computed values p_2 and p_3, respectively. Black lines denote the memory block boundaries, for an architecture with blocks of 64 bytes.

Defensive approaches for table lookup, as implemented in NaCl or libgcrypt 1.6.3, avoid such vulnerabilities by accessing *all* table entries, in a constant order. OpenSSL 1.0.2f instead uses a more permissive approach that accesses only *one* table entry, however it uses a smart layout of the tables to ensure that the requested memory blocks are loaded into cache in a constant order. An example layout for storing 8 pre-computed values is shown in Figure 6.2. The code that manages such tables consists of three functions, which are given in Figure 6.3.

- To create the layout, the function `align` aligns a buffer of memory with the memory block boundary by ensuring the least-significant bits of the buffer address are zeroed, see Figure 6.3a.
- To write a value into the array the function `scatter` ensures that the bytes of the precomputed values are stored `spacing` bytes apart, see Figure 6.3a.
- Finally, to retrieve a pre-computed value from the buffer, the function `gather` assembles the value by accessing its bytes in the same order they were stored, see Figure 6.3c.

Reasoning about the effectiveness of such countermeasures is a daunting task, involving reasoning about the interactions of the program and the hardware platform. First, one must ensure that the compiler does not perform unexpected optimizations. Second, one must ensure that the data is aligned correctly, considering the memory and cache geometry, e.g. considering the size of the cache lines. Third, one must ensure that the assertions hold when considering unknown dynamic locations in memory, as practical implementations place the pre-computed values in heap memory.

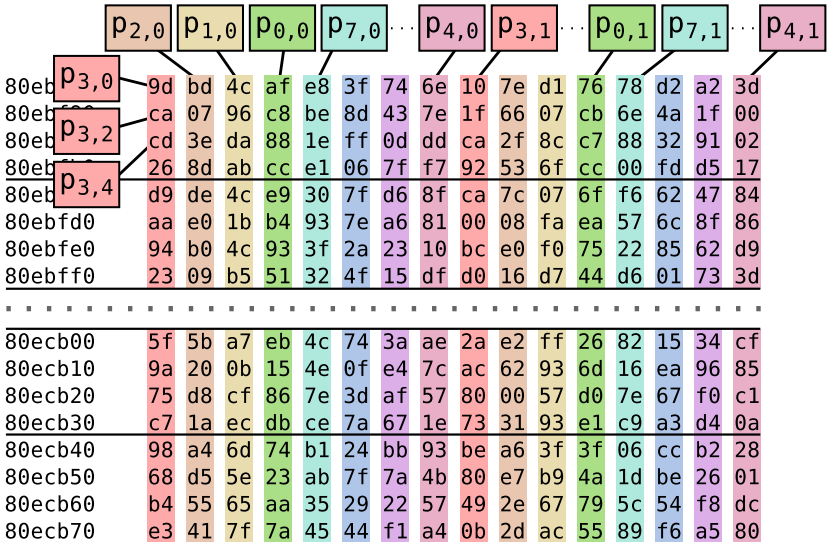


Figure 6.2: Layout of pre-computed values in main memory, achieved with the scatter/-gather countermeasure. Data highlighted in different colors correspond to pre-computed values $p_0, \ldots, p_7$, respectively. Black lines denote the memory block boundaries, for an architecture with blocks of 64 bytes.

The techniques we develop in this chapter enable, for the first time, the rigorous security analysis of permissive countermeasures against side-channel attacks, such as the one in Figure 6.3, based on executable code. Specifically, our techniques enable the static analysis of basic logical and arithmetic operations on symbolic values, which is required for reasoning about cache alignment in the presence of dynamically allocated memory in functions such as `align`, `scatter`, and `gather`. We further develop techniques for tracking sets of observations of cache adversaries, and for computing their size, which enables the quantification of leaks. The results we derive are quantitative upper bounds on the information leaked to different kinds of adversaries; they include formal proofs of non-leakage, but go beyond them in that they help shed insights into the severity of leaks, should they exist.

6.3 Security Against Memory Trace Attacks

In this section we define three kinds of adversaries that can monitor a program's accesses to main memory, ranked by their observational capabilities.

6.3.1 A Hierarchy of Memory Trace Observers

We define three notions of security against memory trace attacks, corresponding to different observational capabilities of the adversary. We express these capabilities in terms of *views*, which are functions that map traces in *Col* to the set of observations

```
1 align ( buf ):
2     return buf − ( buf & ( block_sz − 1) + block_sz
```

(a) Aligning a buffer to block boundary.

```
1 scatter ( buf , p , k):
2     for i := 0 to N − 1 do
3         buf [k + i ∗ spacing ] := p[k][i]
```

(b) Storing a pre-computed value into a buffer.

```
1 gather ( r , buf , k):
2     for i := 0 to N − 1 do
3         r [i] := buf [k + i ∗ spacing ]
```

(c) Retrieving a pre-computed value from a scattered buffer.

Figure 6.3: Scatter/gather method for storing and retrieving pre-computed values.

an adversary can make. The views can be derived by successively applying lossy transformations to the sequence of memory accesses of a program, which leads to a hierarchy of security definitions.

Address-trace observer The first adversary we consider is one that can observe the full sequence of memory locations that are accessed. Security against this adversary implies resilience to many kinds of microarchitectural side-channels, through cache, TLB, DRAM, and branch prediction buffer.[1] This observer, when restricted to addresses of instructions, is equivalent to the program counter security model [142].

Formally, we define the address-trace observer by the view, which takes the initial state and returns the exact sequence of accessed addresses in memory:

$$view^{ato} : (m_0, c_0)e_0 \dots e_{n-1}(m_n, c_n) \mapsto eff_{\mathcal{M}}(m_0) \dots eff_{\mathcal{M}}(m_n)$$

Block-trace observer The second adversary is one that can observe the sequence of memory blocks loaded by the user to the cache. Security against this adversary implies resilience against adversaries that can monitor memory accesses at the level of granularity of cache lines.

For the formalization, recall that the cache logic splits the bit-representation of an n-bit memory address in two parts: the least significant $\log_2 b$ bits represent the position of the address within a memory block of b bytes, and the most significant $n - \log_2 b$ bits

[1] We do not model, or make assertions about, the influence of advanced features such as out-of-order-execution.

represent the set index (identifying the cache set) and the tag (identifying data within a cache set).

tag	set index	block offset

The projection of an address a to the most significant $n - \log_2 b$ bits, denoted by $block(a)$, hence gives a formal account of the observation an adversary makes by observing memory addresses at the granularity of blocks. The block-trace observer is then defined by

$$view^{bto} = (map\ block) \circ view^{ato} ,$$

where *map block* is the natural lifting of *block* to sequences.

Note that security against the block-trace observer does not exclude side-channel attacks by adversaries that can make more precise observations about memory accesses, e.g. by observing DRAM or cache bank accesses. Our analysis is easily adapted to capture such adversaries by including more or less bits into the projection of the address, see the discussion of the CacheBleed attack in Section 6.6.

B-block trace observer Finally we consider a class of adversaries that can observe a sequence of memory blocks, but that can only make limited observations about repeated accesses to the same memory block (which we call *stuttering*). This captures that the adversary cannot count the number of executed instructions, as long as they are guaranteed not to access main memory[2]; it is motivated by the fact that the latency of cache misses dwarfs that of cache hits and is hence easier to observe.

We formalize this intuition in terms of a function $view^{bbto}$ that takes as input a sequence w of blocks and maps maximal subsequences b^r to the block b. That is, repetitions of any block are removed.

Example 5. *The function $view^{bbto}$ maps both aabcddc and abbbccddcc to the sequence abcdc, making them indistinguishable to the adversary.*

We also consider a more powerful observer adversary that can distinguish between repetitions of blocks if they exceed a certain number. We omit the technical details for brevity.

6.3.2 Quantifying Leaks

The approach we describe in Chapter 4 quantifies the degree of confidentiality provided by a program by deriving bounds on the number of observations an adversary can make. Following this approach, we quantify the information leakage (in bits) as

$$\log_2 |view(Col)| \ . \tag{6.1}$$

[2]Here we rely on the (weak) assumption that the second b in any access sequence $\cdots bb \cdots$ is guaranteed to hit the cache.

An advantage of this approach is that it can be automated using standard program analysis techniques and that it comes with different interpretations in terms of security: For example, it can be related to a lower bound on the expected number of guesses an adversary has to make for successfully recovering the secret [52], or to an upper bound for the probability of successfully guessing the secret in one shot [49]. A disadvantage of the application of this approach in previous works is that they have not distinguished between variations in adversary observations that are due to secret data and those that are due to public data. Rather, all unknown data has been considered to be secret (see Section 5.3.2), which can lead to severe imprecisions of the analysis.

In the language of information flow analysis, secret data is called *high*, and public data is called *low*. In this chapter, we propose a novel approach to handle low data by introducing *symbolic values*, which are values that are not known in advance but do not contain secrets. If a symbolic value is part of the adversary's view, the adversary will observe the concrete valuation of the value without learning additional information about the secret data. Thus, symbols take the role of low data, and all remaining variations of the output are potentially due to high data. In that sense, our analysis can be seen as the first quantitative information-flow analysis that can deal with low inputs symbolically. In Section 6.4, we describe our use of symbolic values to capture pointer arithmetic with unknown (but non-secret) memory locations.

6.4 Abstract Domain for Cache-Aware Pointer Arithmetic

Cache-aware code often uses Boolean and arithmetic operations on pointers in order to achieve favorable memory alignment. In this section we devise the *masked symbol domain*, which is a simple abstract domain that enables the static analysis of such code in the presence of dynamically allocated memory.

6.4.1 Representation

The masked symbol domain is based on finite sets of what we call *masked symbols*, namely pairs (s, m) consisting of a *symbol* $s \in \mathit{Syms}$ and a *mask* $m \in \{0, 1, \top\}^n$. The idea is that the symbol s represents an unknown base address and m represents the pattern of known and unknown bits. Here, $\{0, 1\}$ stand for known bits and $\top$ stands for unknown bits. The i-th bit of a masked symbol (s, m) is hence equal to m_i, unless $m_i = \top$, in which case it is unknown. In the first case we call the bit *masked*, in the second *symbolic*. We abbreviate the mask $(\top, \ldots, \top)$ by $\top$.

Two special cases of masked symbols are worth pointing out:

1. $(s, \top)$ represents an unknown constant, and
2. (s, m) with $m \in \{0, 1\}^n$ represents the bit-vector m.

That is, pairs of the form (s, m) generalize both bitvectors and unknown constants.

6.4.2 Concretization and Counting

We now give a semantics to elements of the masked symbol domain. This semantics is parametrized w.r.t. instantiations of the symbols. For the case where masked symbols represent partially known heap addresses, a valuation corresponds to one specific layout.

Technically, we define the concretization of finite sets $F \subseteq \mathit{Syms} \times \{0, 1, \top\}^n$ w.r.t. a mapping $\sigma\colon \mathit{Syms} \to \{0, 1\}^n$ taking symbols to bit-vectors:

$$\gamma_\sigma(F) = \{\sigma(s) \oplus m \mid (s, m) \in F\}$$

Here $\oplus$ is defined bitwise by $c_i \oplus m_i = m_i$ whenever $m_i \in \{0, 1\}$, and c_i otherwise.

The following proposition shows that precise valuation of the constant symbols can be ignored for deriving upper bounds on the numbers of values that the novel domain represents. It enables the quantification of information leaks in the absence of exact information about locations on the heap.

Proposition 3. *For every valuation $\sigma\colon \mathit{Syms} \to B$ we have $|\gamma_\sigma(F)| \leq |F|$*

Note that, with this definition we do not assume any relationship between symbols. Below we will increase its precision by tracking basic arithmetic relations between symbols.

6.4.3 Update

We support two kinds of operations on elements of the masked symbol domain. The first tracks patterns on bit-vectors and is needed for reasoning about memory alignment. The second tracks the arithmetic relationship between masked symbols and is needed for basic pointer arithmetic and equality checks. Here we describe only operations between individual masked symbols, the lifting to sets is obtained by performing the operations on all pairs of elements.

Tracking Bits An important class of operations for cache-aware coding are those that allow the alignment of data to memory blocks without knowing the precise pointer value.

Example 6. *The following code snippet allocates* 1000 *bytes of heap memory and stores a pointer to this chunk in x.*

```
x = malloc(1000);
y = (x & 0xFFFFFFC0) + 0x40;
```

The second line ensures that the 6 least significant bits of that pointer are set to 0*, thereby aligning it with cache lines of* 64 *bytes. Finally, adding* `0x40` *ensures that the resulting pointer points into the allocated region while keeping the alignment.*

s	&	$0 = 0$	s	&	$1 = \top$
s	|	$1 = 1$	s	|	$0 = \top$
s	^	$s = 0$	s	^	$0 = \top$
s	-	$s = 0$			

Table 6.1: Bit operations and their effect on masked symbols. The left column shows operations that recover mask bits from symbols. The right column contains operations that leave symbolic bits unmodified.

To reason about this kind of code, we distinguish between operations that allow the introduction of a mask and those that maintain a symbol: The left column of Table 6.1 lists logical bit operations that translate symbolic bits into masked bits, i.e. creating a mask. For example, the operation `x & 0xFFFFFFC0` in Example 6 results in a masked symbol

$$(s_x, (\top \cdots \top 000000)) \,. \tag{6.2}$$

The right column in Table 6.1 lists logical bit operations that leave symbolic bits unmodified, thus maintaining the symbol. Information about the mask can also be maintained throughout arithmetic operations such as additions, as long as all carry bits can be absorbed within the mask. For example, the addition of `0x3F` to (6.2) results in a masked symbol

$$(s_x, (\top \cdots \top 111111)) \,,$$

for which we can statically determine containment in the same cache line as $(s_x, (\top \cdots \top 000000))$.

In cases when an operation does affect the symbolic bits, we take a conservative approach and drop information about the symbols. Formally,

$$(s_1, m_1) \circ (s_2, m_2) = ((s_3), \top) \,,$$

where s_3 is a fresh symbol. For example, the addition of `0x40` to (6.2) in Example 6 results in the masked symbol

$$(s_y, (\top \cdots \top 000000)) \,,$$

which points to the beginning of some (unknown) cache line.

Tracking Arithmetic Relationships Besides providing support for bit-operations on symbolic addresses we also require support for basic pointer arithmetic.

Example 7. *The following code snippet sets the values of array* `A` *for indices* $i = 0, \ldots, 9$.

```
int *x = A + 10;
int *y;
for (y = A; y < x; y++)
  *y = get_value (...);
```

The loop terminates whenever pointer `y` *points at or beyond* `x`*. While pointer arithmetic here can be avoided by an implementation using an integer counter in the for-condition, modern compilers often optimize such implementations to a form using pointer arithmetic.*

The operations on symbols defined so far are not sufficient for a precise analysis of the code in example 7 in case that `A` has a symbolic value. For example, if the value of `A` is $(s, \top)$, at iteration $i = 2, 3, 4, \ldots$ of the loop, `y` will be assigned a fresh symbolic value $(s_i, \top)$; thus, the equality or inequality between s and s_i cannot be determined, and the loop termination cannot be established.

To address this shortcoming, we add support for tracking simple arithmetic congruences between masked symbols. For masked symbols s_1, s_2, those congruences are terms of the form $s_1 = s_2 + o$, where o is an integer constant. Building up a set of constraints C between masked symbols allows establishing equalities between masked symbols. In the example above, at the first iteration of the loop we determine that $y = (s, \top)$ and $x = (s', \top) = (s, \top) + 10$. After the tenth iteration of the loop, we determine that $y = (s', \top) = x$. The termination of the loop is established by precisely determining the CPU flag values, which we explain below.

Technically, we model congruences by defining the semantics $\gamma_\sigma(F)$ of a set of masked symbols only w.r.t. symbol valuations $\sigma\colon \mathit{Syms} \to \{0,1\}^n$ that satisfy

$$\sigma(s_1) \oplus m_1 = \sigma(s_2) \oplus m_2 + o\,,$$

for all $(s_1, m_1) = (s_2, m_2) + o \in S$. We implement congruences by storing a function $eq\colon \mathit{Syms} \to \mathit{Syms} \times \{0,1\}^n$. We count $eq(F)$ instead of F, which increases the precision of counting.

CPU Flag Values When performing an update involving masked symbols, CPU flag values may be affected. By default we safely assume that all flag combinations are possible. To improve precision of the analysis, we identify several cases where flags are determined even though values are symbolic. Among those cases, we identify the following:

1. If at least one masked bit of the result is non-zero, then `ZF = 0`.
2. If the operation does not affect the (possibly symbolic) most-significant bits of the operands, then `CF = 0`.
3. For operations `sub src, dst` (or equivalently, `cmp src, dst`), if we can deduce that `src` = `dst` + c with $c \neq 0$, then `ZF = 0`.

We apply the information learned about flags in combination with arithmetic congruences, to establish loop termination. For example, at the end of the loop in Example 7, the termination of the loop is determined by the operation `cmp x, y` followed by a conditional jump. After loop iteration $i \in \{1, \ldots, 9\}$, case 3 allows establishing that the

conditional jump is taken, and that the loop is not terminated. At iteration $i = 10$, `x` and `y` have the same symbolic value, which allows establishing that because `ZF = 1`, the loop is terminated.

Local Soundness To establish the local soundness (see (4.4) in Section 4.3) of the masked symbol domain, we consider the abstract $next^\sharp$-operator for the domain, which is implemented by the update-function described in this section.

Lemma 9. *The masked symbol domain is locally sound.*

Informally, the soundness follows from the soundness of the ingredients of its update: (a) fresh symbols are used whenever exact values are not known; (b) the trivial congruence $x = x + 0$ is used whenever no congruence between x and another masked symbol is known; (c) all flag combinations are considered whenever the operation does not determine the exact flags.

6.5 Abstract Domains for Memory Access Traces

In this section, we present data structures for representing the set of possible memory accesses a program can make, and for computing the number of observations that different side-channel attackers can make. For this we devise the *access-trace domains*, which capture the adversary models introduced in Section 6.3.1: address-trace (adversary who can observe addresses), block-trace (adversary who can observe memory blocks), and b-block-trace (adversary who can observe memory blocks modulo repetitions). We describe those domains, and elaborate on developments that allow us to precisely capture cases where a b-block-trace adversary learns less information than an address- and block-trace adversary, as is the case in the following example.

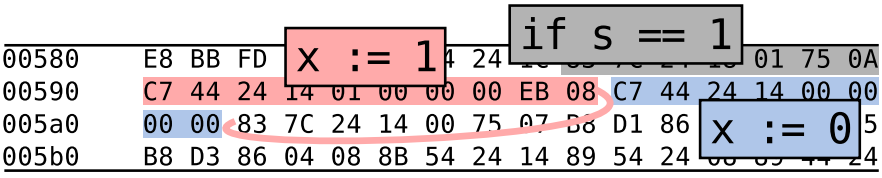
Example 8. *The following code snippet presents implicit information flow from the secret value* `s` *to the public output* `x`*.*

```
if ( s == 1 ) x = 1; else x = 0;
```

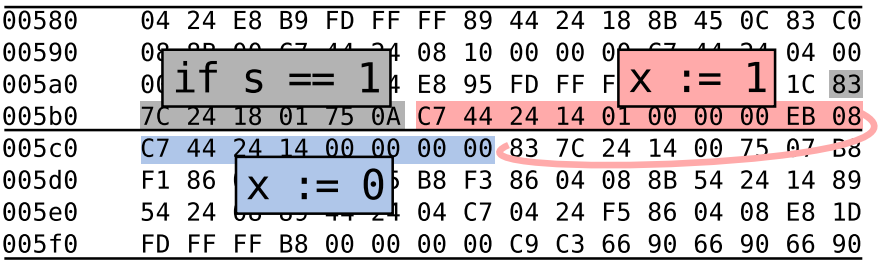
Figure 8 shows layout of the code snippet in the binary. Regardless whether the layout of the executable causes the code to fit into one memory block or two, the same sequence of blocks will be loaded into cache.

6.5.1 Representation

We use a directed acyclic graph (DAG) to compactly represent traces of memory accesses. While a DAG representation can precisely represent a set of traces [45], a counting procedure that does not require enumerating all traces may lose precision. The representation we devise is precise for the b-block observer, and provides an easy counting procedure.



(a) The if-branch causes two I-cache accesses to addresses within block `580`; The else-branch causes one I-cache access to an address within block `580`.



(b) The if-branch causes two I-cache accesses to addresses within block `580`; The else-branch causes one I-cache access to an address within block `5c0`. Afterwards, in both cases instructions within block `5c0` will be executed.

Figure 6.4: Two possible layouts of if-then-else code, compiled with gcc. The highlighted code corresponds to the if-check and jump, the if-branch and the else-branch, respectively. The red curve represents the jump target in the end of the if-branch. Black lines denote block boundaries, for an architecture with 64-byte memory blocks.

The DAG has a set of nodes N representing memory accesses, with a unique root r and a set of edges $E \subseteq N \times N$. We equip each node $n \in N$ with a label $L(n)$ that represents the adversary's view of the node (i.e. addresses or blocks), and a repetition count $R(n)$ that represents the number of times each address may be repeatedly accessed. We represent labels $L(n)$ using the masked symbols abstract domain (see Section 6.4), and use a finite set abstraction to represent $R(n)$.

6.5.2 Concretization and Counting

In an access-trace domain, each node n represents the set of traces of the program up to this point of the analysis. Its concretization function is defined as

$$\gamma(n) = \bigcup_{\substack{r=n_0 \cdots n_k = n \text{ is a} \\ \text{path from } r \text{ to } n}} \{\ell_0^{t_0} \cdots \ell_k^{t_k} \mid \ell_i \in L(n_i), t_i \in R(n_i)\} \tag{6.3}$$

For quantitatively assessing leaks we are interested in the number of observations $|\gamma(n)|$ an adversary can make. An upper bound on this number is easily computed from

the access-trace domains using the following recursive equation:

$$count(n) = |R(n)| \cdot |L(n)| \cdot \sum_{(s,n)\in E} count(s) \tag{6.4}$$

For the b-block trace observer, we replace the factor $|R(n)|$ from the expression in (6.4) by 1, which captures that this observer cannot distinguish between repetitions of accesses to the same memory block.

6.5.3 Update and Join

The access-trace domains are equipped with functions for update and join, which govern how sets of traces are extended and merged, respectively. The domains are defined with respect to an observer modeled by the $view \in \{view^{ato}, view^{bto}, view^{bbto}\}$, for the different views introduced in Section 6.3.2. Below we overload the notation of *view* to take as input sets of addresses, and return the respective set of observations (i.e. addresses or blocks).

The *update* receives a node n representing a set of traces of memory accesses, and it extends n by a new access to a potentially unknown address that represented a finite set of masked symbols F. Technically:

1. If the set of masked symbols is not a repetition (i.e. if $L(n) \neq view(F)$) the update function appends a new node n' to n (adding (n, n') to E) and it sets $L(n') = view(F)$ and $R(n') = \{1\}$.

2. Otherwise (i.e. if $L(n) = view(F)$) it increments the possible repetitions in $R(n)$ by one.

The local soundness (see (4.4) in Section 4.3) of the access-trace domains follows directly because the abstract and the concrete updates are the same, with the difference that the abstract update results in a more compact representation in case of repetitions.

Lemma 10. *The access-trace domains are locally sound.*

The *join* for two nodes n_1, n_2 first checks whether those nodes have the same parents and the same label, in which case n_1 is returned, and their repetitions are joined. Otherwise a new node n' with $L(n') = \{\epsilon\}$ is generated and edges (n_1, n') and (n_2, n') are added to E.

To increase precision in counting for the b-block adversary, joins are delayed until the next update is performed. This captures cases where an if-then-else statement spans two memory blocks, both of which are accessed regardless whether if-condition is taken or not, as demonstrated in Figure 6.4b.

6.6 Case Study

In this section we present a case study in which we leverage the techniques developed in this chapter for the first rigorous analysis of several countermeasures against cache side-channel attacks against modular exponentiation algorithms from versions of libgcrypt and OpenSSL from April 2013 to March 2016. We report on results for leakage to the adversary models presented in Section 6.3.1 due to instruction-cache (I-cache) accesses and data-cache (D-cache) accesses.[3] As the adversary models are ordered according to their observational capabilities, this sheds light into the level of provable security that different protections offer.

6.6.1 Tool building

We implement the novel abstract domains described in Sections 6.4 and 6.5 on top of the CacheAudit open source static analyzer [45]. CacheAudit provides infrastructure for parsing, control-flow reconstruction, and fixed point computation. Our novel domains extend the scope of CacheAudit by providing support for (1) the analysis of dynamically allocated memory, and for (2) adversaries who can make fine-grained observations about memory accesses. The resulting extensions to CacheAudit will be made publicly available.

6.6.2 Target Implementations

For libgcrypt we consider versions 1.5.2 to 1.6.3. In those versions of libgcrypt, four variants of modular exponentiation are implemented, with different protections against side-channel attacks. In addition to those implementations, we analyze two side-channel protections introduced in OpenSSL versions 1.0.2f and 1.0.2g, respectively. For fairness of comparison of security and performance we implement all protections on top of libgcrypt 1.6.3 and analyze this code instead of OpenSSL.

To generate the target executables of our experiments, we use ElGamal decryption routines based on each of the above-described implementations of modular exponentiation and the respective countermeasures, resulting in 5 implementations. We compile them using `GCC 4.8.4`, on a 32-bit Linux machine running kernel 3.13. For ElGamal decryption, we use a key size of 3072 bits.

The current version of CacheAudit supports only a subset of the x86 instruction set and CPU flags, which we extend on demand. To bound the required extensions we focus our analysis on the regions of the executables that were targeted by exploits and to which the corresponding countermeasures were applied, rather than the whole executables. As a consequence, the formal statements we derive only hold for those regions. In particular, we do not analyze the code of the libgcrypt's multi-precision integer multiplication and

[3] We also analyzed the leakage from accesses to shared instruction- and data-caches; for the analyzed instances, the leakage results were consistently the maximum of the I-cache and D-cache leakage results.

modulo routines, and we specify that the output of the memory allocation functions (e.g. `malloc()`) is symbolic (see Section 6.4).

6.6.3 Square-and-Multiply Modular Exponentiation

The first target of our analysis is modular exponentiation by square-and-multiply. The algorithm is depicted in Figure 6.5 and is implemented, e.g., in libgcrypt version 1.5.2. Line 5 of the algorithm contains a conditional branch whose condition depends on a bit of the secret exponent. An attacker who can observe the victim's accesses to instruction or data caches may learn which branch was taken and identify the value of the exponent bit. This weakness has been shown to be vulnerable to key-recovery attacks based on prime+probe [13, 146] and flush+reload [14].

```
1  r := 1
2  for i := |e| − 1 downto 0 do
3      r := mpi sqr(r)
4      r := mpi mod(r, m)
5      if e_i = 1 then
6          r := mpi_mul(b, r)
7          r := mpi_mod(r, m)
8      return r
```

Figure 6.5: Pseudocode for square-and-multiply modular exponentiation

```
1  r := 1
2  for i := |e| − 1 downto 0 do
3      r := mpi_sqr(r)
4      r := mpi_mod(r, m)
5      tmp := mpi_mul(b, r)
6      tmp := mpi_mod(tmp , m)
7      if e_i = 1 then
8          r := tmp
9      return r
```

Figure 6.6: Pseudocode for square-and-always-multiply modular exponentiation

In response to these attacks, libgcrypt 1.5.3 implements a countermeasure that makes sure that the squaring operation is always performed, see Figure 6.6) for the pseudocode. It is noticeable that this implementation still contains a conditional branch that depends on the bits of the exponent in Line 7, namely the copy operation that selects the outcome of both multiplication operations. However, this has been considered a

minor problem because the branch is small and is expected to fit into the same cache line as preceding and following code, or to be always loaded in cache due to speculative execution [14]. In the following we apply the techniques developed in this chapter to analyze whether the expectations on memory layout are met.

Observer	address	block	b-block
I-Cache	1 bit	1 bit	1 bit
D-Cache	1 bit	1 bit	1 bit

(a) Square-and-multiply from libgcrypt 1.5.2

Observer	address	block	b-block
I-Cache	1 bit	1 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

(b) Square-and-*always*-multiply from libgcrypt 1.5.3

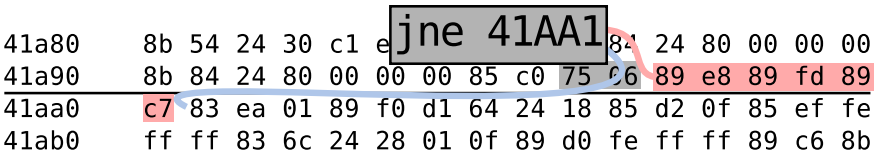
Figure 6.7: Leakage of modular exponentiation algorithms to observers of instruction and data caches, with cache line size of 64 bytes and compiler optimization level `-O2`.

Observer	address	block	b-block
I-Cache	1 bit	1 bit	1 bit
D-Cache	1 bit	1 bit	1 bit

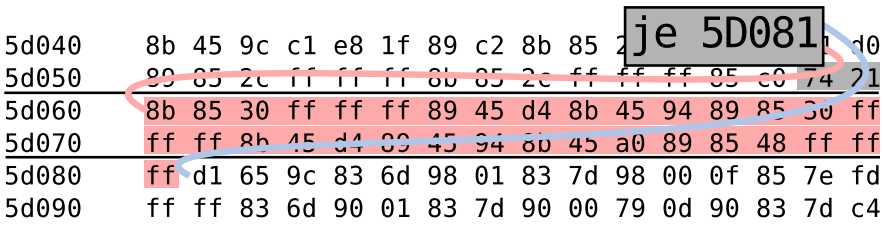
Figure 6.8: Leakage of square-and-always-multiply from libgcrypt 1.5.3, with cache line size of 32 bytes and compiler optimization level `-O0`.

Results The results of our analysis are given in Figure 6.7 and Figure 6.9

- Our analysis identifies a 1-bit data cache leak in square-and-multiply exponentiation, see line 2 in Figure 6.7a, which is due to memory accesses in the conditional branch in that implementation. Our analysis confirms that this data cache leak is closed by square-and-always-multiply, see line 2 in Figure 6.7b
- Line 1 of Figures 6.7a and Figure 6.7b show that both implementations leak through instruction cache to powerful adversaries who can see each access to the instruction cache. However, for weaker, b-block observers that cannot distinguish between repeated accesses to a block, square-and-always-multiply does *not* leak, confirming the intuition that the conditional copy operation is indeed less problematic than the conditional multiplication.
- The data in Figure 6.8 demonstrates that the advantages of conditional copy over conditional multiplication depend on details such as cache line size and compilation strategy. Figure 6.9 illustrates this effect for the conditional copy operation, where more aggressive compilation leads to more compact code that fits into single cache lines. The same effect is observable for data caches, where more aggressive compilation avoids data cache accesses altogether.



(a) Compiled with the default gcc optimization level -O2. Regardless whether the jump is taken or not, first block `41a80` is accessed, followed by block `41aa0`. This results in a 0-bit b-block leak.



(b) Compiled with gcc optimization level -O0. The memory block `5d060` is only accessed when the jump is taken. This results in a 1-bit b-block leak.

Figure 6.9: Layout of libgcrypt 1.5.3 executables with 32-byte memory blocks (black lines denote block boundaries). The highlighted code corresponds to the conditional branching in lines 7–8 in Figure 6.6. The red region corresponds to the executed instructions in the if-branch. The blue curve points to the jump target, where the jump is taken if the if-condition does not hold.

6.6.4 Windowed Modular Exponentiation

In this section we analyze windowed algorithms for modular exponentiation. These algorithms differ from algorithms based on square-and-multiply in that they process multiple exponent bits in one shot. For this they commonly rely on tables filled with precomputed powers of the base. For example, libgcrypt 1.6.1 precomputes 7 multi-precision integers and handles the power 1 in a branch, see Figure 6.10. For moduli of 3072 bits, each precomputed value requires 384 bytes of storage, which amounts to 6-7 memory blocks in architectures with cache lines of 64 bytes. Key-dependent accesses to those tables can be exploited for mounting cache side-channel attacks [146].

```
if (e0 == 0) {
    base_u = bp;
    base_u_size = bsize;
} else {
    base_u = b_2i3[e0 - 1];
    base_u_size = b_2i3size[e0 - 1];
}
```

Figure 6.10: Sliding window table lookup from libgcrypt 1.6.1. Variable `e0` represents the window, right-shifted by 1. The lookup returns a pointer to the first limb of the multi-precision integer in `base_u`, and the number of limbs in `base_u_size`. The first branch deals with powers of 1 by returning pointers to the base.

We consider three countermeasures, which are commonly deployed to defend against this vulnerability. They have in common that they all *copy* the table entries instead of returning a pointer to the entry.

```
1  // Retrieves r from p[k]
2  secure_retrieve ( r , p , k):
3        for i := 0 to n − 1 do
4           for j := 0 to N − 1 do
5              v := p[i][j]
6              s := (i == k)
7              r[j] := r[j] ^ ((0 − s) & ( r[j] ^ v))
```

Figure 6.11: A defensive routine for array lookup with a constant sequence of memory accesses, as implemented in libgcrypt 1.6.3.

- The first countermeasure ensures that in the copy process, a constant sequence of memory locations is accessed, see Figure 6.11 for pseudocode. The expression on line 7 ensures that only the k-th pre-computed value is actually copied to `r`. This countermeasure is implemented, e.g. in NaCl and libgcrypt 1.6.3.

- The second countermeasure stores precomputed values in such a way that the i-th byte of all precomputed values reside in the same memory block. This ensures that when the precomputed values are retrieved, a constant sequence of memory blocks will be accessed. This so-called scatter/gather technique is described in detail in Section 6.2, with code in Fig 6.3, and is deployed, e.g. in OpenSSL 1.0.2f.

- The third countermeasure is a variation of scatter/gather, and ensures that the gather-procedure performs a constant sequence of memory accesses (see Fig-

ure 6.12). This countermeasure was recently introduced in OpenSSL 1.0.2g, as a response to the CacheBleed attack [145].

```
1 defensive_gather( r, buf, k )
2     for i:=0 to N−1 do
3         r[i] := 0
4         for j:= 0 to spacing − 1 do
5             v := buf[j + i∗spacing]
6             s := (k == j)
7             r[i] := r[i] | (v & (0 − s))
```

Figure 6.12: A defensive implementation of gather (compare to Figure 6.3c) from OpenSSL 1.0.2g.

Results Our analysis of the different versions of the table lookup yields the following results:[4]

- Figure 6.13a shows the results of the analysis of the unprotected table lookup of Figure 6.10. The leakage of one bit for most adversaries is explained by the fact that they can observe which branch is taken. The layout of the conditional branch is demonstrated in Figure 6.14a; lowering the optimization level results in a different layout (see Figure 6.14b), and in this case our analysis shows that the I-Cache b-block leak is eliminated.
- More powerful adversaries that can see the exact address can learn $\log_2 7 = 2.8$ bits per access. The static analysis is not precise enough to determine that the lookups are correlated, hence it reports that at most 5.6 bits are leaked.
- Figure 6.13b shows that the defensive copying strategy from libgcrypt 1.6.3 (see Figure 6.11) eliminates all leakage to the cache.
- Figure 6.13c shows that the scatter/gather copying-strategy eliminates leakage for any adversary that can observe memory accesses at the granularity of memory blocks, and this constitutes the first proof of security of this countermeasure. For adversaries that can see the full address-trace, our analysis reports a 3 bit leakage for each memory access, which is again accumulated over correlated lookups because of imprecisions in the static analysis. Below we comment on these results in the context of the recent CacheBleed attack.
- Figure 6.13d shows that defensive gather introduced OpenSSL 1.0.2g (see Figure 6.12) eliminates all leakage to cache.

[4]We note sliding window exponentiation exhibits further control-flow vulnerabilities, some of which we also analyze. To avoid redundancy with Section 6.6.3, we focus the presentation of our results on the lookup-table management.

Observer	address	block	b-block
I-Cache	1 bit	1 bit	1 bit
D-Cache	5.6 bit	2.3 bit	2.3 bit

(a) Instruction- and Data-Cache leakage of secret-dependent table lookup in the modular exponentiation implementation from libgcrypt 1.6.1.

Observer	address	block	b-block
I-Cache	0 bit	0 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

(b) Instruction- and Data-Cache leakage of secret-dependent table lookup in the patch on modular exponentiation from libgcrypt 1.6.3.

Observer	address	block	b-block
I-Cache	0 bit	0 bit	0 bit
D-Cache	1152 bit	0 bit	0 bit

(c) Instruction- and Data-Cache leakage in the scatter/gather technique, applied to libgcrypt 1.6.1.

Observer	address	block	b-block
I-Cache	0 bit	0 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

(d) Instruction- and Data-Cache leakage in the defensive gather technique from OpenSSL 1.0.2g, applied to libgcrypt 1.6.1.

Figure 6.13: Instruction and data cache leaks of different table lookup implementations. Note that the leakage in Fig 6.13a accounts for copying a pointer, whereas the leakage in Fig 6.13b and 6.13c refers to copying multi-precision integers.

```
4b980    8B 84 24 98 00 00 00 83 EE 01 89 84 24 94 00 00
4b990    00 75 [je 4BA58] 89 6C 24 44 8B 54 24 48 83
4b9a0    E2 0F 0F 84 B0 00 00 00 8D 4A FF 8B 94 8C B8 00
4b9b0    00 00 8B 8C 8C F4 00 00 00 8B 74 24 24 89 44 24
4b9c0    04 8B 44 24 40 89 54 24 08 8B 54 24 28 89 4C 24
4b9d0    0C 89 74 24 18 8B 74 24 1C 89 04 24 8B 44 24 44
4b9e0    89 74 24 14 8B 74 24 20 89 74 24 10 E8 CF F6 FF
4b9f0    FF 8B 84 24 98 00 00 00 89 84 24 94 00 00 00 E9
4ba00    84 FE FF FF 8D 74 26 00 83 6C 24 3C 01 0F 88 7B
4ba10    03 00 00 BF 20 00 00 00 8B 6C 24 3C 2B 7C 24 34
4ba20    89 FA 8B 7C 24 58 8D 0C 16 89 54 24 38 89 4C 24
4ba30    30 8B 3C AF 8B 6C 24 2C 89 FA D3 EA 0F B6 4C 24
4ba40    38 D3 ED 8B 4C 24 34 09 EA 29 F1 D3 E7 89 7C 24
4ba50    2C E9 B5 FE FF FF 66 90 8B 4C 24 4C 8B 54 24 54
4ba60    E9 54 FF FF FF 8B 74 24 44 8B 54 24 40 89 74 24
4ba70    40 8D B4 24 98 00 00 00 89 54 24 44 89 74 24 28
```

(a) Compiled with the default gcc optimization level -O2. If the jump is taken, first block `4b980`, followed by block `4ba40`, followed by `4b980` again. If the branch is not taken, only block `4b980` is accessed.

```
47dc0    6C F6 FF FF 8B 84 24 98 00 00 00 89 84 24 94 00
47dd0    00 00 83 EE 01 74 14 89 F0 89 EE 89 C5 EB A9 89
47de0    E8 8B 6C [je 47E0B] 4 3C EB 04 89 74 24 3C 8B
47df0    54 24 44 83 E2 0F 74 13 83 EA 01 8B 84 94 B8 00
47e00    00 00 8B 94 94 F4 00 00 00 EB 08 8B 54 24 50 8B
47e10    44 24 58 8D 8C 24 9C 00 00 00 89 4C 24 18 8B 7C
47e20    24 1C 89 7C 24 14 8B 7C 24 20 89 7C 24 10 89 54
47e30    24 0C 89 44 24 08 8B 84 24 94 00 00 00 89 44 24
```

(b) Compiled with gcc optimization level -O1. Regardless whether the jump is taken or not, first block `47dc0` is accessed, followed by block `47e00`.

Figure 6.14: Layout of executables using libgcrypt 1.6.1. The highlighted code corresponds to a conditional branch. The blue region corresponds to the executed instructions in the if-branch, and the red region corresponds to the executed instructions in the else-branch. Curves represent jump targets.

The CacheBleed Attack The recently disclosed CacheBleed attack [145] against the scatter/gather implementation from OpenSSL 1.0.2g exploits timing differences due to *cache-bank conflicts*. Those are possible in CPUs where cache blocks are divided into banks (e.g. Intel Sandy Bridge), to facilitate concurrent accesses to the data cache. For example, the platform targeted in [145] has 16 banks of 4 bytes; there, bits 2–5 of an address are used to determine the bank, and bits 0–1 are used to determine the offset within the bank. The distribution of the pre-computed values in scatter/gather (see Section 6.2) to different banks will be as shown in Figure 6.15.

The leak leading to CacheBleed is visible in our data when comparing the results of the analysis with respect to address-trace and block-trace adversaries, however, its

severity may be over-estimated due to the powerful address-trace observer. For a more accurate analysis of the effect of cache-bank conflicts, we define the *bank*-trace observer, who cannot distinguish between the elements within a single bank. This observer is weaker than the address-trace observer, but stronger than the block-trace observer.

We perform the analysis of the scatter/gather implementation (see Figure 6.13c) for the bank-trace D-cache observer. The analysis results in 384-bit leak, which corresponds to one bit leak per memory access, accumulated for each accessed byte due to analysis imprecision (see above). The one bit leak in the i-th memory access is explained by the ability of this observer to distinguish between the two banks within which the i-th byte of all pre-computed values fall.

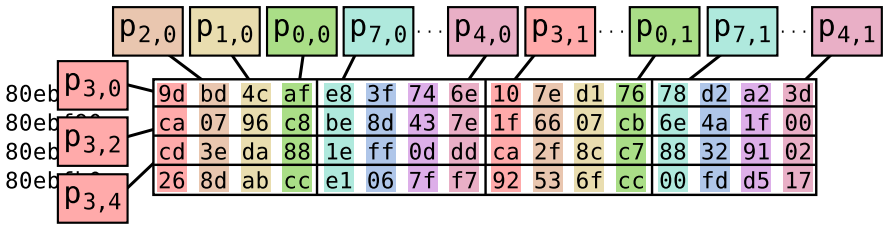


Figure 6.15: Layout of pre-computed values in cache banks, for a platform with 16 banks of 4-bytes. The cells of the grid represent the cache banks.

6.6.5 Discussion

A number of comments are in order when interpreting the bounds delivered by our analysis.

Use of Upper Bounds The results we obtain are upper bounds on the leaked information that are not necessarily tight, that is, they may be pessimistic. This means that while results of zero leakage corresponds to a proof of absence of leaks, positive leakage bounds do not correspond to proofs of the presence of leaks, that is, leaks may be smaller than what is reported by the analysis. The reason for this is that the amount of leaked information may be over-estimated due to imprecisions of the static analysis, as is the case with the D-Cache leak shown on Figure 6.13c.

Practical Detectability of Leaks A reported leak may be practically easier to detect by an adversary in cases where the vulnerable code region produces more cache accesses. This is the case for the control-flow leaks in square-and-multiply, where the vulnerable if-branch includes multiplication and modulo-functions, practically resulting in $\approx 2 \cdot 10^5$ cache accesses. In contrast, the vulnerable if-branch in square-and-always-multiply does not include function calls, practically resulting in a small number of cache accesses, which may be more difficult to detect from noisy observations.

algorithm	square and multiply		sliding window			
countermeasure (CM)	no CM	always multiply	no CM	scatter/ gather	access all bytes	defensive gather
original implementation	libgcrypt 1.5.2	libgcrypt 1.5.3	libgcrypt 1.6.1	openssl 1.0.2f	libgcrypt 1.6.3	openssl 1.0.2g
instructions ($\times 10^6$)	90.32	120.62	73.99	74.21	74.61	75.29
cycles ($\times 10^6$)	75.58	100.73	61.58	61.65	62.20	62.28

Figure 6.16: Performance of the different versions of modular exponentiation, implemented on top of libgcrypt 1.6.3.

6.6.6 Performance

We conclude the case study by considering the effect of the different countermeasures on the performance of modular exponentiation. For this, we use libgcrypt 1.6.3 as a base, and we compile it with the respective `mod_exp.c`, corresponding to each of the considered variants. For performance measurement, we use the time for performing exponentiations on a sample of random bases and exponents, and measure the clock count (through the `rdtsc` instruction), as well as the number of performed instructions (through the `PAPI` library). We follow the approach for performance measurement from [147], performing 100,000 exponentiations with exponents, after a warm-up period of 25,000 exponentiations, and take the minimum over 5 repeated experiments to minimize the noise of background processes. The measurements are performed on a machine with an Intel Q9550 CPU. This architecture does not feature more recent performance-enhancing technologies such as Turbo-Boost Technology and Hyper-Threading Technology.

Figure 6.16 summarizes our measurements. The results show that the applied countermeasure for square and multiply causes a significant slow-down of the exponentiation. A slow-down is observed with sliding-window countermeasures as well, however at a much lower scale.[5] Notable is also the performance gain from using the sliding-window algorithm compared to square-and-multiply.

6.7 Related Work

Agat proposes a program transformation for removing control-flow timing leaks by equalizing branches of conditionals with secret guards [143], with follow-up work in [148, 149]. In an accompanying technical report [150] Agat presents the implementation of the transformation in Java bytecode, which includes an informal discussion of the effect of instruction and data caches on the security of the transformation. Our

[5] We note that performance penalties of countermeasures can be higher when considering a whole cryptographic operation, different platforms, implementations within other crypto libraries, different key sizes. For example, in commit messages prior to OpenSSL 1.0.2g's release, developers report performance penalties of up to 10% for 2048-bit RSA on some platforms.

approach relies on lower-level models, namely x86 executables and simple but accurate cache models, based on which we can prove the security of cache-aware programming.

Molnar et al. [142] propose a program transformation that eliminates branches on secret to remove leaks due to control flow and instruction caches, together with a static check for the resulting x86 executables. Our approach is more permissive than theirs that it can establish the security of code that contains restricted forms of secret-dependent control flow and memory access patterns. It is worth emphasizing that the increased permissiveness of our approach comes from the fact that we rely on models of the hardware architecture for our analysis. If no such models are available, the safe way to go is to forbid all kinds of secret-dependent behavior.

Coppens et al. [114] investigate mitigations for timing-based side-channels on x86 architecture, and they identify new side-channels in programs without secret-dependent memory lookups due to out-of-order execution. In contrast, we prove the security of countermeasures. For this we rely on accurate models of caches, but we do not take into account out-of-order execution.

Bernstein et al. advocate defensive programming [31] that avoids all secret-dependent memory lookups and branching and demonstrate the practicality of their proposal with NaCl. Almeida et al. develop a static analyzer that can automatically confirm that programs follow that regime [86]. Barthe et al. [87] establish that adhering to that policy provides security against very strong adversary models.

Langley [151] shows how to perform a dynamic analysis based on Valgrind and memcheck. His technique flags the secret-dependent (but cache-line independent) memory lookups of the OpenSSL sliding window-based modular exponentiation as a potential leak. Our technique gives more fine-grained insights about its security, including proofs of security for adversaries that can observe memory accesses only at the granularity of memory blocks.

6.8 Conclusions

In this chapter we devise novel techniques that provide support for bit-level and arithmetic reasoning about pointers in the presence of dynamic memory allocation. These techniques enable us to perform the first rigorous analysis of widely deployed software countermeasures against cache attacks on modular exponentiation, based on executable code.

Part III

Web-Traffic Attack Protection

The third part of this thesis develops a framework for the derivation of formal guarantees against side-channels in web traffic. We propose a model that captures important characteristics of web traffic, and we define measures of security based on quantitative information flow. We propose novel techniques for the efficient derivation of security guarantees for web applications. The key novelty of those techniques is that they provides guarantees that cover *all* execution paths in a web application, i.e. it achieves completeness. We demonstrate the utility of our techniques in two case studies, where we derive formal guarantees for the security of a medium-sized regional-language Wikipedia and an auto-complete input field.

The work presented in this chapter provides the first scalable approach for formal quantification of web traffic leaks, and addresses research question 2 (Q2) in Section 1.2. The models presented in Sections 7.2 and 7.3.1 were initially developed in the author's Master thesis [48].

7 Automatic Evaluation of Protections against Web Side-Channels

7.1 Introduction

Internet traffic is exposed to potential eavesdroppers. To limit disclosure of secret information, security-aware users can protect their traffic by accessing web services that offer TLS encryption, or by sending their data through encrypted tunnels. While today's encryption mechanisms hide the secret payload from unauthorized parties, they cannot hide lower-level traffic features such as packet sizes, numbers, and delays. These features contain information about the payload that can be extracted by traffic profiling, side-stepping the protection offered by cryptography. The relevance of this threat is demonstrated by a large number of side-channel attacks against encrypted web traffic, e.g. [4, 16, 17, 152, 153, 154, 155, 156, 157, 158].

A number of approaches for the mitigation and analysis of side-channel leaks have been proposed, e.g. [23, 24, 152, 154]. A common pattern in these approaches is the following relationship between attacks, countermeasures, and their security analysis: An *attack* corresponds to a classification of a sample of network traffic, where classes correspond to secret aspects of user behavior. A correct classification corresponds to a successful attack, i.e. one in which the secret is correctly recovered. A *countermeasure* modifies the shape of the network traffic with the goal of making the classification of samples more difficult or even impossible. A *security analysis* is based on an evaluation of the performance of a particular classification algorithm.

A security analysis following this pattern enables one to assess a system's vulnerability to a particular classifier; however, the analysis does not make immediate assertions about the vulnerability to attackers using more sophisticated techniques for mining their observations. This limitation is not only unsatisfactory from a theoretical viewpoint, but it also raises significant problems in practice: A recent comprehensive study [159] of traffic analysis countermeasures exhibits that the incompleteness of existing security analyses indeed leaves room for highly effective attacks. It is clear that, ultimately, one strives for security guarantees that hold for *all* realistic adversaries who can observe the

web application's traffic, i.e. formally backed-up security guarantees.

There are two key challenges in deriving formal guarantees against side-channel attacks against web traffic. The first challenge is to devise a *mathematical model* of the web application's traffic, which is indispensable for expressing and deriving security guarantees. Such a model has to be accurate enough to encompass all relevant traffic features, and at the same time, the model must be simplistic enough to allow for tractable reasoning about realistic web applications.

The second challenge is to develop *techniques for the computation* of security guarantees for a given web application. The main obstacle is that the derivation of security guarantees requires considering all execution paths of a web application. As the number of paths may grow exponentially with their length, their naive enumeration is computationally infeasible. Previous approaches deal with this problem by resorting to a subset of the possible execution paths [160, 161], introducing incompleteness into the analysis.

In this chapter, we develop a novel framework for the derivation of formal guarantees against traffic side-channels in web applications. We first provide a model that captures the effect of user actions on web-traffic. In particular, we cast web applications as labeled, directed graphs where vertices correspond to the states of the web application, edges correspond to possible user actions, and (vertex) labels are observable features of encrypted traffic induced by moving to the corresponding state. These features can include packet sizes and numbers, and their source and destinations, which is sufficient to encompass relevant examples.

We then interpret this graph as an information-theoretic *channel* mapping sequences of (secret) user actions to (public) traffic observations. Casting the graph as a channel allows us to apply established measures of confidentiality (Shannon entropy, min-entropy, g-leakage) from quantitative information-flow analysis and make use of their well-understood properties. Leveraging those properties, we obtain the following results.

To put our model into action, we propose a novel technique for the efficient derivation of security guarantees for web applications. The key advantage of this technique is that it allows considering *all* execution paths of the web application, which is fundamental for deriving formal security guarantees. We achieve this result for the important special case of a user following the *random surfer model* [162] and for Shannon entropy as a measure: As a first step, we use PageRank to compute the stationary distribution of a random surfer, which we take as the a priori probability of a user visiting a vertex. As a second step, we apply the chain rule of entropy to reduce the computation of the uncertainty about a path of length ℓ to that about path of length $\ell - 1$, and transitively to that about a single transition. As computing the uncertainty about a single transition can be done efficiently, this is the key to avoiding the enumeration of all (exponentially many) paths.

We use this algorithm to study the trade-off between security and performance (in terms of overhead) of different countermeasures. A key observation is that countermeasures based on making individual vertices indistinguishable (e.g. [23]) fail to protect

paths of vertices. To ensure protection of paths, we devise *path-aware* countermeasures, which strengthen countermeasures for vertices by making sure that indistinguishable vertices also have indistinguishable sets of successors. Formally, we achieve this by coarsening indistinguishability on vertices to a probabilistic bisimulation relation, which we compute using a novel algorithm based on randomization and partition refinement.

We demonstrate the applicability of the proposed techniques in two case studies, where we analyze the traffic of (1) a regional-language Wikipedia consisting of 5,000 articles, and that of (2) an auto-complete input field for a dictionary of 1,100 medical terms. Our approach delivers a number of countermeasures that provide different trade-offs between security guarantees and traffic overhead, spreading from good security at the price of a high overhead, to low overhead at the price of lower security guarantees.

In summary, our main contribution is algorithms and techniques that enable the efficient derivation of such guarantees for real systems. We demonstrate how they can be used for adjusting the trade-off between security and performance of practical instances of web applications.

7.2 Web-Traffic as an Information-Theoretic Channel

In this section, we cast web-traffic as an information-theoretic channel (i.e. a conditional probability distribution) from user input to observable traffic patterns. We specify the threat scenario in Section 7.2.1 and present our basic model of web applications and their traffic in Section 7.2.2.

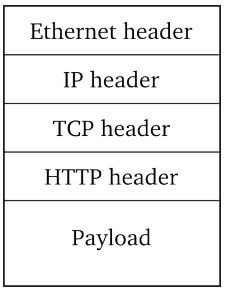
7.2.1 Threat Scenario

The threat scenario we are considering is a user performing confidential actions (such as following hyperlinks, or typing characters) in a web application, and a passive attacker who inspects network traffic and wants to obtain information about the user's actions. and

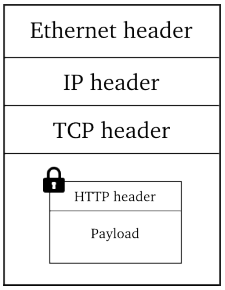
When a user performs an action on the web application, this causes messages (which we call *web-objects*) to be exchanged between the client- and the server-side of the web application, and the attacker observes bursts of network packets corresponding to each web-object. Traffic is protected by encrypting the data at a certain layer of the protocol stack. Commonly used protection mechanisms are HTTPS, an encrypted connection to proxies (e.g. an SSH tunnel), and encryption of wireless traffic (e.g. using WPA2). Depending on the used protection mechanism, attackers will have a different view of the traffic, as illustrated in Figure 7.1.

7.2.2 Basic Model

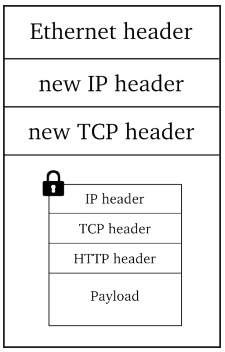
We model the network traffic corresponding to a web application using a directed labeled graph. In this graph, vertices correspond to the states of the web application, edges



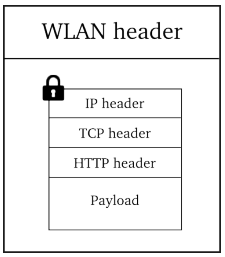
(a) An Ethernet frame without encryption



(b) An Ethernet frame with HTTPS encryption



(c) An Ethernet frame with an encrypted tunnel to a proxy server



(d) A WLAN frame with WPA2 encryption

Figure 7.1: Encrypted packets with commonly used protection mechanisms. When HTTPS is applied (Figure 7.1b), the attacker may use the information contained in the TCP header to reassemble the packets and infer the approximate size of the web-object. When traffic is forced through an SSH tunnel (Figure 7.1c) and when using an encrypted wireless connection (Figure 7.1d), the original headers are encrypted and direct reassembly is not possible any more.

correspond to possible user actions, and vertex labels correspond to induced traffic patterns.

Definition 10. *A* web application *is a directed graph $G = (V, E)$, together with an* application fingerprint $f_{app} \colon V \to o$ *that maps vertices to* observations $o \subseteq (A \times \{\uparrow, \downarrow\})^*$, *where A is the set of observable objects. We denote the set of paths in G by Paths(G).*

An application fingerprint of a vertex is intended to capture an eavesdropper's view of transmitted packets, requests and responses, which necessarily includes the packets' directions. If traffic is not encrypted, we set the observable objects to be bit-strings, i.e. $A = \{0, 1\}^*$. If (a part of) an object is encrypted, only its size remains visible, i.e., we assume that encryption is perfect and length-preserving. Formally, encryption is defined as a function $enc : \{0, 1\}^* \to \mathbb{N}$, and if all objects are encrypted, then we set $A = \mathbb{N}$.

For example, a sequence $(10, \uparrow), (10, \uparrow), (20, \downarrow), (32, \downarrow)$ captures an exchange of encrypted objects, two of size 10 sent from the client to the server, and two of size 20 and 32, respectively, sent from the server to the client.

We next show how Definition 10 can be instantiated to capture two representative scenarios, which we use as running examples throughout the chapter. The first scenario models web navigation, where user actions correspond to following hyperlinks. The second scenario models an auto-complete form, where user actions correspond to typing characters into an input field. For ease of presentation, in the following examples we cast the states of a web application as the requested web-objects, and assume that all objects are encrypted.

Example 9. *Consider a user who is navigating the Web by following hyperlinks. The web application is a* web-graph $G = (V, E)$ *, where each vertex $v = (w_1, w_2, w_3, \dots)$ is a sequence of web-objects, i.e $V \subseteq W^*$ for a set W of web-objects. For example, we may have a vertex v = (a.html, style.css, script.js, image.jpg, video.flv), which corresponds to a particular webpage. An edge (u, v) models that the webpage v can be reached from webpage u by following a hyperlink. If we additionally allow users to jump to arbitrary webpages, the resulting graph will be complete. For the definition of the application fingerprint f_{app}, let the size of a web-request corresponding to a web-object, and the size of the web-object be given as functions $r \colon W \to \mathbb{N}$ and $s \colon W \to \mathbb{N}$, respectively. Then, the application fingerprint of a webpage $v = (w_1, \dots, w_n)$ is given by*

$$f_{app}(v) = ((r(w_1), \uparrow), (s(w_1), \downarrow), \dots, (r(w_n), \uparrow), (s(w_n), \downarrow)) \,.$$

Example 10. *Consider an auto-complete input field where, after each typed character, a list of suggestions is generated by the server and displayed to the user (see Figure 7.2). Let Σ be the input alphabet (i.e. the set of possible user actions) and Σ^* the set of possible words. We define the corresponding web application as a graph $G = (V, E)$ with $V = \Sigma^*$, the root $r = \varepsilon$ is the empty word, and $(u, v) \in E$ if and only if $v = u\sigma$, for some $\sigma \in \Sigma$. Note that G is in fact a prefix tree [163], and the leafs of the tree form the input dictionary $D \subseteq \Sigma^*$. Let the auto-complete functionality be implemented by a function suggest$\colon V \to S^*$ returning a list of suggestions from the dictionary of*

possible suggestions $S \subseteq \Sigma^*$. *Let the sizes of a suggestion request and the corresponding suggestion list be given as functions* $r\colon V \to \mathbb{N}$ *and* $s\colon S^* \to \mathbb{N}$, *respectively. Then, given a word* $v \in V$, *we define its application fingerprint as*

$$f_{app}(v) = ((r(v), \uparrow), (s(suggest(v)), \downarrow)).$$

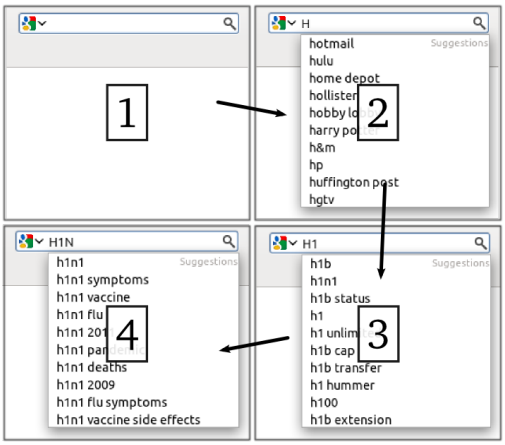


Figure 7.2: An auto-complete input field

To capture modifications of observations when traffic passes through different network protocols, as well as the effect of (potentially randomized) countermeasures, we introduce the notion of a network fingerprint.

Definition 11. *A* network fingerprint *is a function*

$$f_{net}\colon o \to (o \to [0, 1]),$$

such that for each $o \in o$, $\sum_{o' \in o} f_{net}(o)(o') = 1$.

A network fingerprint $f_{net}(o)(o')$ models the conditional probability of outputting a burst of network packets o', given as input a burst of network packets o. We cast network fingerprints using function notation because this allows for a clean combination with application fingerprints.

We now show how the combination of the web application with the network fingerprint f_{net} can be cast as an information-theoretic channel mapping execution paths $Paths(G)$ to sequences of network observations o^*.

Definition 12. *Let* G *be a web application with fingerprints* f_{app} *and* f_{net}. *Let* X *be a random variable with* $ran(()X) = Paths(G)$, *and* Y *a random variable with* $ran(()Y) = o^*$. *Then the* traffic channel *induced by* (G, f_{app}, f_{net}) *is the conditional distribution*

$$P[Y = o_1 \dots o_\ell | X = v_1 \dots v_\ell] = \prod_{i=1}^{\ell} f_{net}(f_{app}(v_i))(o_i)$$

With Definition 12, we make two implicit assumptions: First, we assume that when a user traverses a path of length ℓ in a web application, the attacker can see a sequence $o_1, \ldots, o_\ell \in o^\ell$ of sequences of packet sizes and directions. This corresponds to an attacker that can distinguish between bursts of traffic corresponding to different vertices, but that cannot observe the timing of the packets. Second, by multiplying the probabilities of observations, we assume that the network bursts corresponding to individual vertices are pairwise independent. Note that this assumption can be weakened by conditioning on multiple vertices, which we forgo for simplicity of presentation. Finally, note that Definition 12 does not make any assumptions on the distribution of X, which models the user's behavior. We will only make such assumptions in Section 7.3, where we need them for the algorithmic derivation of security guarantees.

7.3 Algorithms for Practical Evaluation of Web Applications

In this section we devise techniques for the derivation of *absolute* security guarantees, i.e. concrete numbers for the remaining uncertainty $\mathcal{H}(X|Y)$ about the user input. For this, two challenges need to be addressed.

The first is that the derivation of absolute guarantees requires making assumptions about (the attacker's initial uncertainty $\mathcal{H}(X)$ about) the user's behavior. The second is that the direct computation of $\mathcal{H}(X|Y)$ requires enumerating the values of X (i.e. all paths) and Y (i.e. all possible traffic patterns), which quickly becomes infeasible. In this section we present algorithms that enable the efficient computation of formal security guarantees. Our algorithms assume Markov models of user behavior and rely on the chain rule of entropy, which is only satisfied by Shannon entropy $H(X)$. As a consequence, their application is restricted to scenarios where such assumptions can be justified and where guarantees based on Shannon entropy (see Chapter 2) are adequate.

We begin by showing how, under these assumptions, the initial uncertainty about a user's navigation in a website can be computed. We then present a generic approach for constructing countermeasures based on bisimulations and for computing the uncertainty induced by such countermeasures.

7.3.1 Modeling User Behavior as a Markov chain

We capture user behavior by a random variable X, where $P[X = (v_1, \ldots, v_\ell)]$ describes the probability of the user taking execution path $(v_1, \ldots, v_\ell)$, see Section 7.2. The random variable X can be decomposed into the components $X_1, \ldots, X_\ell$ corresponding to the vertices on the path. When the user's choice of the next vertex depends only on the currently visited vertex, then $X_1, \ldots, X_\ell$ form a *Markov chain*. Formally: $P[X_{i+1} =$

$v_{i+1}|X_i = v_i, \ldots, X_1 = v_1] = P[X_{i+1} = v_{i+1}|X_i = v_i]$ for $i \in \{1, \ldots, \ell - 1\}$. Then, we obtain

$$P[X = (v_1, \ldots, v_\ell)] = P[X_1 = v_1] \prod_{i=1}^{\ell-1} P[X_{i+1} = v_{i+1}|X_i = v_i] \,.$$

This decomposition enables one to specify the probability of execution paths in terms of probabilities of individual transitions. The Markov property is clearly valid in the auto-complete input fields from Example 10 where the execution paths form a tree, and it also is a commonly made assumption for predicting navigation behavior of users [164, 165].

7.3.2 Computing the Initial Uncertainty

We next devise an algorithm for computing the initial uncertainty $H(X)$ about the behavior of a user, based on the stationary distribution of X and the chain rule of entropy. We then show how X can be instantiated and how its stationary distribution can be computed using the PageRank algorithm.

7.3.2.1 Initial Uncertainty Based on Stationary Distributions

From the Fundamental theorem of Markov chains [166] it follows that each finite, irreducible, and aperiodic Markov chain converges to a unique stationary distribution. Formally, if the Markov chain is given in terms of a transition matrix Q, where $q_{i,j}$ denotes the probability of moving from v_i to v_j, there is a row vector (the *stationary distribution*) π with $\pi = \pi Q$.

Under the above conditions, the user's choice of a next vertex will converge to π, which we use to capture the probability of the user choosing the *first* vertex $P[X_1 = v_1]$. The following theorem gives a handle on efficiently computing the initial uncertainty about the user's execution path.

Theorem 9. *Let $X_1, \ldots, X_\ell$ be a Markov chain with $P[X_1] = \pi$, where π is the stationary distribution. Then,*

$$H(X_1, \ldots, X_\ell) = H(X_1) + (\ell - 1)H(X_2|X_1).$$

Proof.

$$\begin{aligned} & H(X_1, \ldots, X_\ell) \\ \overset{(*)}{=} \; & H(X_\ell|X_{\ell-1}, \ldots, X_1) + \cdots + H(X_2|X_1) + H(X_1) \\ \overset{(**)}{=} \; & H(X_\ell|X_{\ell-1}) + H(X_{\ell-1}|X_{\ell-2}) + \cdots + H(X_2|X_1) + H(X_1) \end{aligned}$$

Here $(*)$ follows from the chain rule for Shannon entropy and $(**)$ follows from the Markov property. As $P[X_1]$ is the stationary distribution, we have $H(X_k|X_{k-1}) = H(X_j|X_{j-1})$ for all $j, k \in \{2, \ldots, n\}$, which concludes the proof. □

Given the stationary distribution, Theorem 9 enables the computation of the initial uncertainty $H(X)$ about the user's navigation in time $O(|V|^2)$, for any fixed ℓ. We next show how the Markov chain representation of X can be instantiated, and how its stationary distribution can be obtained, using the PageRank algorithm.

7.3.2.2 Using PageRank for Practical Computation of the Initial Uncertainty

PageRank [162] is a link analysis algorithm that has been applied for predicting user behavior in web usage mining [165]. It relies on the *random surfer model*, which captures a user who either follows a link on the currently visited webpage, or jumps to a random webpage.

Formally, the probability of following links is given by a transition matrix Q', and the probability of random jumping is $1 - \alpha$, where the *damping factor* $\alpha \in [0, 1]$ is usually set to 0.85. The user's behavior is then captured by the transition matrix $Q = \alpha Q' + (1 - \alpha)p\mathbf{1}$, where p is a row vector representing the probability of jumping to pages, and $\mathbf{1}$ is the column vector with all 1 entries. One typically assumes uniform distributions for the rows q'_i of Q' and for p, however the probabilities can also be biased according to additional knowledge, e.g. obtained by mining usage logs [167].

Given a transition matrix Q of a website, the PageRank algorithm computes the corresponding stationary distribution π, and π_i is called the PageRank of a webpage i. Notice that the conditions for the existence of π are usually fulfilled: irreducibility is guaranteed by a damping factor $\alpha < 1$, and in practice websites are aperiodic.

7.3.3 Constructing Path-Aware Countermeasures

Most traffic analysis countermeasures aim to provide protection by making multiple states of a web-application indistinguishable for an attacker who can only observe their traffic fingerprints (e.g. see [23]). In our model, we say that two vertices v_1 and v_2 are *(f)-indistinguishable* for some countermeasure f whenever $f(v_1) = f(v_2)$, i.e. the observations produced by v_1 and v_2 have the same probability distribution. Clearly, f-indistinguishability induces a partition $P = \{B_1, \ldots, B_m\}$ on the state space V of a web-application, where the blocks B_i correspond to the equivalence classes of f-indistinguishability.

Unfortunately, indistinguishability of individual states does not protect information gained by observing a sequence of states. Consider, e.g., a scenario where an attacker observes traffic produced by typing a word in English. Assume that the first typed character is known to be t and the second character is either h or s, which we assume to be f-indistinguishable. From the first action t, the attacker can deduce that the second user action is most likely h, i.e. secret information is revealed.

7.3.3.1 Ensuring Indistinguishability of Paths

To protect the information contained in the transition between states, we devise *path-aware countermeasures*, which require that the induced partition be in fact a *bisimulation* [168] (also called *lumping* [169]), a property that ensures behavioral equivalence of states.

Definition 13. *A partition $P = \{B_1, \ldots, B_m\}$ of the state space V of a Markov chain $X = (X_1, \ldots, X_\ell)$ is a* (probabilistic) bisimulation *if for any blocks $B_i, B_j \in P$ and for any $v_i, v_h \in B_i$, $\sum_{v_j \in B_j} q_{i,j} = \sum_{v_j \in B_j} q_{h,j}$. We define the corresponding* quotient process *$Y = (Y_1, \ldots, Y_\ell)$ as the random variable with state space P, such that $Y_k = B_i$ iff $X_k = v$ and $v \in B_i$.*

A fundamental characteristic of bisimulations is that they preserve the Markov property of the resulting quotient process.

Theorem 10 (Kemeny-Snell [169])**.** *A partition of the state space of a Markov chain is a probabilistic bisimulation iff the corresponding quotient process is a Markov chain.*

To measure the strength of a path-aware countermeasure, we need to calculate the remaining uncertainty $H(X|Y)$, for which we have

$$H(X|Y) \geq H(X) - H(Y), \tag{7.1}$$

with equality whenever Y is determined by X (e.g., if the corresponding countermeasure is deterministic). By Theorem 10, the resulting quotient process Y is a Markov chain. This enables us to use Theorem 9 for computing $H(Y)$, leading to an efficient algorithm returning a lower bound for $H(X|Y)$.

7.3.3.2 Implementing Path-Aware Countermeasures

For a given countermeasure f, we seek to group together vertices in V to blocks of indistinguishable vertices, such that the resulting partition is path-aware, i.e. a bisimulation. This could be trivially achieved by padding all possible observations to a maximal element o^*. While the corresponding (1-block) partition achieves maximal security, it may also induce an unacceptably large traffic overhead. Our goal is hence to find a bisimulation that coarsens the partition induced by f and that offers a more attractive trade-off between security and performance.

While there are efficient algorithms for computing bisimulations that *refine* a given partition to a bisimulation, we are not aware of existing algorithms for obtaining bisimulations by coarsening. We tackle the problem by the following two-step approach.

In a first step, we compute a set of *random* bisimulations on V. To achieve this, we select random initial partitions of V with only two blocks, e.g. by flipping a coin for each vertex and selecting a block accordingly. For each of those two-block partitions, we compute the coarsest bisimulation that refines it using the efficient algorithm by

Derisavi et al. [170], which we call `CoarsestBisimulation` in the remainder of this chapter.

In a second step, we coarsen the partition given by f to the partition $P = \{B_1, \ldots, B_m\}$ given by each bisimulation computed in the previous step. We achieve this by modifying f to a new countermeasure that is a constant function on each B_i. We consider two such modifications:

1. The first countermeasure, which we call f_{limit}, returns for each vertex $v \in B_i$ the maximum over all vertices $w \in B_i$ of the sum of the sizes of the observable objects for each countermeasure $f_{\text{limit}}(v) = \max_{w \in B_i} \sum_i |o_{i,w}|$, where $f(w) = (o_{1,w}, \ldots, o_{t,w})$. If observations consist of one object each, f_{limit} can be implemented by padding. If observations consist of multiple objects, an implementation may not be possible without additional overhead, and thus f_{limit} is a theoretical concept representing the smallest possible size that can be achieved without losing information about f, and hence represents a lower limit for the cost required by any countermeasure inducing a partition $\{B_1, \ldots, B_m\}$.

2. The second countermeasure, which we call f_{order}, first orders for each vertex $v \in B_i$ the components in $f(v)$ according to their (descending) size, and then pads the k-th component to the maximum size of the k-th components over all $w \in B_i$.

In our experiments in Section 7.4, we demonstrate the overhead induced by the practical f_{order}, as well as by the cost-limiting f_{limit} countermeasures. The results show that there is room for designing more cost-effective countermeasures, which induce an overhead closer to f_{limit}; for this to be achieved, further basic countermeasures can be utilized, e.g. splitting of objects.

7.4 Case Studies

In this section we report on two case studies in which we apply the approach presented in this chapter to evaluate the security of countermeasures for a website and an auto-complete field. Moreover, we analyze the overhead in traffic induced by instantiations of the countermeasure to different strengths, which gives a formal account of the ubiquitous trade-off between security and performance in side-channel analysis.

7.4.1 Web Navigation

We analyze a web navigation scenario as described Example 9. As target we use the Bavarian-language Wikipedia[1] of more than 5,000 articles. We instantiate user models and network fingerprints as follows.

[1] `http://bar.wikipedia.org`

User model We assume the random surfer model (see Section 7.3.2), where a user is navigating through Wikipedia articles. To instantiate the web-graph G, we crawled the website only following links leading to the Wikipedia article namespace, where the crawling was performed using a customized `wget`[2]. Excluding redirect pages, we obtained a graph with 3,496 vertices.

Application fingerprint For each vertex v, we took the application fingerprint $f_{app}(v)$ to be the tuple containing the size of the source .html file and the sizes of the contained image files. On average, the total size of each vertex was 104 kilobytes, with a standard deviation of 85 kilobytes. Taking sizes of web-objects instead of sizes of packets is a safe approximation whenever encryption happens at the object-level. We did not account for the sizes of requests because they are almost of equal size in our example, and padding them to a uniform constant size is cheap.

Results without countermeasure We compute the initial uncertainty $H(X)$ about the user's navigation using Theorem 9, where we obtain the stationary distribution of the user's starting point using PageRank, as described in Section 7.3.2. The first row in Figure 7.3 gives the values of $H(X)$ for varying path lengths ℓ. An interpretation of the data in the sense of Proposition 1 shows that for $\ell = 5$ we already obtain a lower bound of around 2^{30} for the expected number of guesses to determine the correct execution path. For the corresponding sequence of network fingerprints Y (without any countermeasure applied) and $\ell > 1$, we consistently obtain a remaining uncertainty $H(X|Y)$ of 0. This follows because in our example almost every page is determined by its fingerprint, and whenever there is ambiguity, the page can be recovered from observing the path on which it occurs. Also note that a naive computation of $H(X)$ quickly becomes infeasible, because the number of paths is $|V|^{\ell}$, due to the completeness of the graph induced by the random surfer model.

ℓ	1	3	5	9	15	25	40
$H(X)$	10.1	21	31.8	53.4	85.9	139.9	221
# paths	3496	$2^{36.5}$	$2^{59.8}$	2^{106}	2^{176}	2^{295}	2^{472}

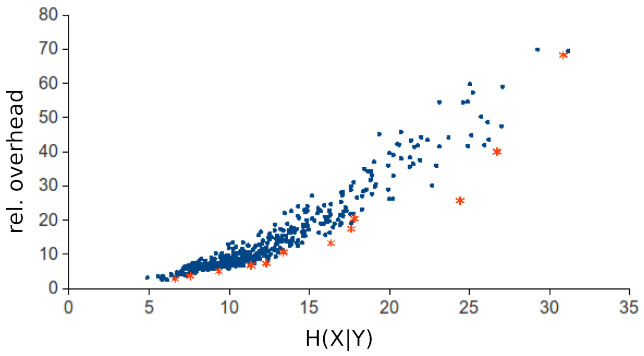
Figure 7.3: Web navigation: the initial uncertainty $H(X)$ about the user's navigation in the regional-language Wikipedia for a varying path length ℓ. # paths denotes the number of execution paths.

Results with countermeasure As described in Section 7.3.3, we obtain a number of path-aware countermeasures by refining randomly chosen initial partitions. For each of them, we evaluate the effect on security and performance: (1) we compute the remaining uncertainty $H(X|Y)$ using Equation 7.1 and Theorem 9; (2) we compute the relative

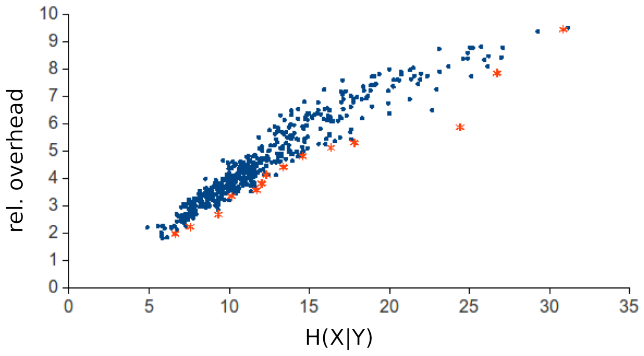
[2] `http://www.gnu.org/software/wget/`

expected overhead of a countermeasure f_{net} as $E_v[|f_{net}(v)| - |f_{app}(v)|]/E_v[|f_{app}(v)|]$, where $|\cdot|$ denotes the size of the payload, i.e. the sum of the number of bytes used for the individual observations. The results below are obtained for paths of length $\ell = 5$ and the practical f_{order} and cost-limiting f_{limit} countermeasures (see Section 7.3.3).

As reference point we use the countermeasure that makes all states indistinguishable and induces maximal security, i.e. $H(X|Y) = H(X)$. On this countermeasure, f_{order} produces a relative expected overhead of 73.5, while f_{limit} produces a relative expected overhead of 9.7. Trading security for performance, we randomly chose 500 initial partitions, and refined them to path-aware countermeasures using `CoarsestBisimulation` (see Section 7.3.3). The remaining uncertainty and the relative expected overhead for each of those countermeasures are depicted in Figure 7.4a (for f_{order}), and Figure 7.4b (for f_{limit}). The results spread from partitions delivering strong security at the price of a high overhead, to smaller overhead at the price of weaker security.



(a) Results with the f_{order} countermeasure



(b) Results with the f_{limit} countermeasure

Figure 7.4: Web navigation: the remaining uncertainty $H(X|Y)$ about the user's navigation in the regional-language Wikipedia versus the relative expected overhead, using the f_{limit} and f_{order} countermeasures, for 500 randomly chosen initial partitions. Red stars represent favored resulting partitions, i.e. resulting in a better security-versus-overhead trade-off than close-by points.

Figure 7.5 details on 8 of the 500 computed bisimulation partitions. The first and the last lines present the trivial coarsest and finest partitions, respectively. The data shows

that, when performance is mandatory, one can still achieve 6.63 bits of uncertainty (corresponding a lower bound of 25 expected guesses, according to Proposition 1) with an overhead of factor 2.75. On the other hand, stronger security guarantees come at the price of overhead of more than factor 10. The high required overheads in the web-navigation scenario can be explained by the large diversity of the sizes and numbers of the web-objects in the studied pages. As the next case study shows, the required overheads are much lower for web applications with more homogeneous web-objects.

$H(X\|Y)$	overhead f_{order}	overhead f_{limit}
31.77	73.47	9.71
26.7	40.01	7.83
24.4	25.68	5.86
16.35	13.29	5.11
12.32	7.31	4.11
9.35	5.04	3.18
6.63	2.78	1.97
0	0	0

Figure 7.5: Web navigation: the remaining uncertainty $H(X|Y)$ about the user's navigation and the relative expected overhead for 8 selected bisimulation partitions.

7.4.2 Auto-Complete Input Field

We analyze the scenario of an auto-complete input field as described in Example 10. Unlike graphs corresponding to websites, the graph corresponding to an auto-complete field is a tree with a designated root and a small number of paths. This allows us to compute the remaining uncertainty by enumerating the paths in the tree, which is why we can also consider min-entropy H_∞. In this case study, we instantiate user models and network fingerprints as follows.

User model As described in Section 7.3.1, user behavior is characterized by a probability distribution $P[X = (v_1, \ldots, v_\ell)]$ of possible paths of length ℓ, which for auto-complete input fields corresponds to the set of words of length ℓ. To obtain such a distribution, we take as an input dictionary D the 1,183 hyponyms of (i.e., terms in an *is-a* relationship with) the word "illness", contained in the WordNet English lexical database [171]. We use the number of results of a corresponding Google query as an approximation of the probability for each term in the dictionary: We issue such queries for all 1,183 words in the dictionary D and count their relative frequencies, thereby establishing probabilities for the leaves in the prefix tree corresponding to D. We traverse the tree towards the root, instantiating the probability of each *vertex* as the sum of the probabilities of its children. Then we instantiate the probabilities of

the outgoing *edges* of each vertex with values proportional to the probabilities of its children.

Application fingerprint We instantiate the application fingerprints as follows. The size of the request $r(v)$ (see Example 10) is given by the length of the typed word v. The size of the response $s(suggest(v))$ is characterized by the size of the suggestion list given by the Google's auto-complete service on query v. We issued queries for all 11,678 vertices in the prefix tree to instantiate these numbers. The responses in all cases consisted of only one packet with average size of 243 bytes and a standard deviation of 97 bytes.

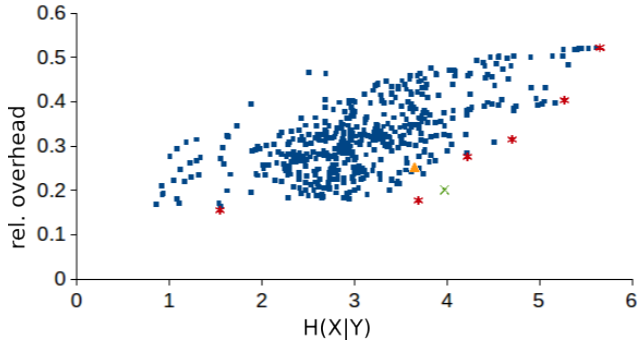
Results without countermeasure For a varying path length $\ell = 1, \ldots, 7$, the initial uncertainty is between 3.79 and 5.65 bits of Shannon Entropy, and between 1.61 and 2.85 of min-entropy (see Figure 7.6). Unlike graphs corresponding to websites, the graph corresponding to an auto-complete field has a tree-structure, and the number of paths is bounded by the number of terms in the dictionary D. The initial uncertainty is highest for paths of a smaller length, and the decrease of the uncertainty for longer paths occurs because the transition relation reveals more information: there are less words in D sharing the same prefix. In all cases the remaining uncertainty after observing the traffic was 0, meaning that in those cases the web application is highly vulnerable to attacks: an attacker can infer the secret input word with probability 1.

ℓ	1	2	3	4	5	6	7
$H(X)$	4.22	5.09	5.52	5.65	5.52	5.51	5.32
$H_\infty(X)$	2.7	2.85	2.65	2.39	1.96	1.71	1.61
# paths	47	238	538	657	710	773	804

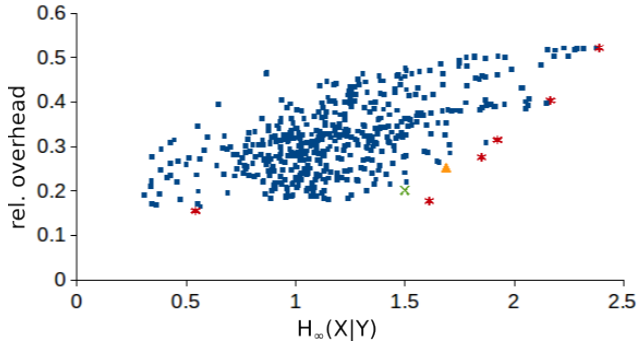
Figure 7.6: Auto-complete: the initial uncertainty (in terms of Shannon entropy $H(X)$ and min-entropy $H(X|Y)$) about the typed word in the auto-complete input field for a varying path length ℓ. # paths denotes the number of execution paths.

Results with countermeasure We also measured the vulnerability of the auto-complete field with countermeasures applied. States were made indistinguishable by applying f_{limit}, which here can be practically implemented because the observations consist of single files (see Section 7.3.3). In the following, we report on our experiments for a path length of $\ell = 4$, where we obtain the following results:

First, padding all vertices to a uniform size results in a maximal uncertainty of 5.65 bits of Shannon entropy and 2.39 bits of min-entropy, respectively, with an relative expected overhead of 2.9. Second, padding only vertices at the same depth to a uniform size, the relative expected overhead drops to 0.52. The resulting uncertainty remains maximal because, in our model, the depth of a vertex is common knowledge. Finally, trading security for performance, we construct path-aware countermeasures



(a) Shannon entropy



(b) Min-entropy

Figure 7.7: Auto-complete: the remaining uncertainty about the typed word in the auto-complete input field versus the relative expected overhead, for 500 randomly chosen initial parititions, using Shannon entropy and min-entropy as a measure. Red stars represent favored resulting partitions, i.e. resulting in a better security-versus-overhead trade-off than close-by points. The green × denotes a partition that is only favored by the Shannon entropy; the yellow triangle denotes a partition that is only favored by the min-entropy.

using `CoarsestBisimulation`, for 500 randomly chosen initial partitions (see Section 7.3.3). Figure 7.7 depicts the trade-off between remaining uncertainty and relative expected overhead in this case. For example, the data show that the overhead needed for maximal protection can be decreased by 46%, for the price of leaking of 1.44 bits of Shannon entropy (0.54 bits of min-entropy), and by 65%, for the price of leaking 1.97 bits of Shannon entropy (0.78 bits of min-entropy). Note that in most cases the two considered entropy measures favor the same partitions, as depicted by the partitions corresponding to the red stars in Figure 7.7a and Figure 7.7b coincide. This is not always the case: the partition denoted as a green × is only favored by the Shannon entropy, and the partition denoted as a yellow triangle is only favored by the min-entropy.

7.5 Related Work

There is a long history of attacks that exploit visible patterns in encrypted web traffic. The first attacks for extracting user information from the volume of encrypted web traffic were proposed in by Cheng and Avnur [16]; since then there have been several published attacks of this kind, e.g. [4, 17, 152, 153, 154, 155, 156, 157, 158, 159]. There has been an evolution of the goals that such attacks pursue, and the of settings of those attacks. While the goal of the attack presented in [16] is to identify a webpage within a website accessed through HTTPS, later attacks aim at identifying the website a user is visiting when browsing through an encrypted tunnel. Those attacks target HTTPS connections to anonymizing proxies [17, 152, 156], SSH or a VPN connections to anonymizing proxies [153, 154, 157, 158], data contained in anonymized NetFlow records [155], or onion routing and web mixes [157, 172]. Chen et al. [4] turn the focus of their attacks to web applications accessed through HTTPS and WPA, and the goal of those attacks is the extraction of sensitive information about user's health conditions and financial status.

Several works propose countermeasures against those attacks. *Padding*, i.e., adding noise to observations, is the standard countermeasure and has been proposed by a large number of authors [4, 16, 17, 152, 154, 156, 158]. A different approach proposed by [152] and implemented by [23] changes patterns of observations corresponding to one website to look like patterns corresponding to another website, which allows sensitive traffic to be camouflaged as traffic coming from popular services. Features of the TCP and HTTP protocols were utilized in the techniques proposed by Luo et al. [24] used to build the HTTPOS system, which offers browser-side protection from traffic analysis. We have shown how parts of HTTPOS and other previously proposed countermeasures can be cast in our model, and how we can use it to formally reason about them.

A number of works empirically evaluate the security of proposed countermeasures [16, 23, 24, 152, 154, 172]. There, the security is measured by comparing the performance of attacks before and after the application of the countermeasures. In recent work, Dyer et al. [159] exhibit shortcomings in this kind of security evaluation. In particular, they demonstrate the inability of nine previously known countermeasures to mitigate information leaks, even when only coarse traffic features are exploited, such as total bandwidth or total time. In contrast, we propose a formal framework that does not assume the use of particular attacks, but rather measures security in terms of the amount of information leaked from the produced observations.

Sidebuster [160] is a language-based approach to countering side-channel attacks in web applications. It uses a combination of taint-based analysis and repeated sampling to estimate the information revealed through traffic patterns, however without an explicitly defined system model. Liu et al. [173] present a traffic padding technique that is backed up by formal guarantees. In contrast to our work, their model does not account for an attacker's prior knowledge or for the probabilistic nature of traffic patterns. Finally, [161] propose a black-box web application crawling system that interacts with the web application as an actual user while logging network traffic. They view an

attacker as a classifier and quantify information leaks in terms of classifier performance, using metrics based on entropy and on the Fisher criterion. Their metrics are computed from a small number of samples, which can deliver imprecise results for entropy [174]. In contrast, we apply our metric to the entirety of paths, which is possible due to the random surfer assumption. Finally, work in quantitative information-flow analysis has been applied for assessing the information leakage of anonymity protocols [175] and cryptographic algorithms [39].

7.6 Conclusions and Future Work

We have presented a formal model that enables reasoning about side-channel attacks against web applications. In our model, the web application's encrypted traffic is cast as an information-theoretic channel, which gives a clean interface to the growing body of research in quantitative information-flow analysis and allows us to use established notions of quantitative security. We have demonstrated that our model is expressive enough to encompass web browsing, simple web applications, and several countermeasures from the recent literature. Furthermore, we have demonstrated algorithms allowing the efficient derivation of security guarantees for real systems.

A potential goal for future work is to use the presented model as a semantic basis for language-based approaches for reasoning about side-channel attacks against web applications, such as [160, 161]. Progress along these lines could lead to principled analysis techniques with soundness guarantees that hold for the whole protocol stack. Intermediate steps for this are investigating methods for approximating f_{net} by sampling, investigating how quality guarantees for the sampling affect the final security guarantees.

8 Conclusions

8.1 Summary

In this thesis, we present a number of tools that enable the security quantification and the choice of practical protections against side-channel attacks. For this, we develop novel models that capture several types of side-channel adversaries: (1) timing adversaries, who can perform attacks combining online (i.e., side-channel) and offline (i.e., computational) steps; (2) cache adversaries of several types depending on their observational capabilities, e.g. ones who can observe the cache state after a program's execution, and ones who can observe the sequence of memory addresses accessed by a program; (3) web-traffic adversaries, who can observe traffic corresponding to sequences of visited webpages. Based on these models, we develop the following tools.

First, we present a systematic approach for choosing an optimal protection against timing attacks, where we make use of a number of simple but powerful tools from game theory, information theory, and cryptography. The results we obtain are rigorous but practical enough to justify the use of a fast but leaky implementation of ElGamal over a defensive constant-time implementation.

Second, we present CacheAudit, the first automatic tool for the static derivation of formal, quantitative security guarantees against cache side-channel attacks. We demonstrate the usefulness of CacheAudit by establishing formal security guarantees for binary executables of sorting algorithms and state-of-the-art cryptosystems such as AES and the finalists of the eSTREAM stream cipher competition. Furthermore, we present an extension to CacheAudit, which utilizes novel techniques that provide support for bit-level and symbolic reasoning about pointers in the presence of dynamic memory allocation. We apply these techniques to reason about the effectiveness of several widely deployed side-channel countermeasures from the libgcrypt and OpenSSL libraries, based on executable code.

Third, we present algorithms for reasoning about side-channel attacks against web applications, as well as for generating of protections against such attacks. We apply these algorithms on practical instances of web applications.

8.2 Limitations and Future Work

8.2.1 Systematic Choice of Protections against Further Side-Channel Attacks

A key motivation of this thesis is to enable the systematic choice of protections against side-channel attacks, offering a balance between the security guarantees and the performance penalties of these protections (see research question Q1 in Section 1.2). In Chapter 3, we demonstrate an approach for choosing protections against a class of timing attacks, using a game-theoretic model. This approach does not directly extend to further types of side-channel attacks we investigate in this thesis: cache attacks and web-traffic attacks. The reason for this is that the approach from Chapter 3 is facilitated by the bounds on the security level we derive, which are not directly available for further types of side-channel attacks. Namely, these bounds capture the probability of a successful attack, for an adversary who collects multiple side-channel observations, and uses the obtained information in a computational attack. Advantages of these bounds are that they (1) capture a realistic attack scenario, and (2) have a direct economic application: the probability of outcomes can be used for obtaining expected utilities in decision and game theory. In the following, we discuss challenges for obtaining such bounds for further types of side-channel attacks.

Cache Attacks In Chapters 4 to 6, the obtained bounds on cache leakage can be used for deriving bounds on the probability of recovering the cryptographic key, for an adversary who can make *one* side-channel observation (see Section 4.2). A limitation that hinders the direct application of such bounds in the approach presented in Chapter 3 is that realistic side-channel adversaries would be able to perform multiple observations. In cases where we obtain 0 bits of leakage, our bounds directly translate to attacks with multiple observations; however, in cases of non-zero leakage, reasoning about accumulation of leakage over multiple side-channel observations remains out of reach for our analysis, and is left to future work.

Web-Traffic Attacks In Chapter 7, the proposed algorithms allow obtaining bounds in terms of Shannon entropy, which can be used for deriving the expected number of guesses an adversary has to make in order to determine the (secret) browsing behavior, by passively observing the corresponding encrypted traffic (see Section 2.1). Several limitations hinder the direct application of these bounds in the approach presented in Chapter 3. First, the economic application of such guarantees is not directly clear. Second, the justification of an attack consisting of multiple guesses is not clear in this scenario: multiple guesses are justified in attacks with an offline phase, in which the correctness of a guess can be determined, e.g. in attacks aiming at recovering cryptographic keys, where a known plaintext-ciphertext pair can be used to determine the correctness of a key. Third, for determining the initial probability of correctly guessing the secret,

assumptions on the adversaries' prior knowledge need to be made, which may be difficult to justify. A possible direction for addressing these limitations is to investigate the effect of side-channel leakage on further measures for confidentiality [176].

8.2.2 Performance Overhead of Countermeasures

The countermeasures we investigate in the case studies in Chapter 3 and Chapter 7 come with provable security guarantees, however have high performance overhead compared to the original, unprotected systems. This leads to the following open questions and directions for future work.

- A reason for the high overhead is that we rely on conservative bounds on the security against side-channel attacks, which overapproximate the adversary's capabilities. Possible ways to reduce these overheads is by deriving tighter bounds on side-channel leakage, or by relaxing the underlying models.
- In Chapter 3, we reason about the choice of protections, requiring that they guarantee a fixed security level, which leads to a high performance overhead. A question that stays open is: How to reason about the more general problem of choosing protections without requirements on the security level? An answer to this question has to take into consideration that we rely on *bounds* on the security level of protections, and deciding to reduce the security level may be unsound, e.g. leaving a system unprotected if the defender falsely believes that the adversary has 100% probability of breach (see discussion in Section 3.2.5).
- A further question we leave to future work is: Can the performance of state-of-the-art fast constant-time cryptographic implementations (e.g., [29, 30, 31, 32, 33]) be improved by allowing controlled side-channel leaks, without sacrificing their level of provable security?

Bibliography

[1] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO*, pp. 104–113, Springer, 1996.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*, Springer, 1999.

[3] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems—CHES 2001*, pp. 251–261, Springer, 2001.

[4] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: a reality today, a challenge tomorrow," in *IEEE Symposium on Security and Privacy (SSP)*, pp. 191–206, IEEE, 2010.

[5] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 35–49, IEEE, 2008.

[6] D. Asonov and R. Agrawal, "Keyboard acoustic emanations," in *null*, p. 3, IEEE, 2004.

[7] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers.," in *USENIX Security Symposium*, pp. 307–322, 2010.

[8] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[9] D. Bernstein, "Cache-timing attacks on AES." http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[10] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[11] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, vol. 3860 of *LNCS*, pp. 1–20, Springer, 2006.

[12] O. Aciiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the AES," in *CT-RSA*, pp. 271–286, Springer, 2007.

[13] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, ACM, 2012.

[14] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, USENIX Association, 2014.

[15] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1406–1418, ACM, 2015.

[16] H. Cheng, H. Cheng, and R. Avnur, "Traffic analysis of ssl encrypted web browsing," 1998.

[17] A. Hintz, "Fingerprinting websites using traffic analysis," in *Privacy Enhancing Technologies (PET)*, 2002.

[18] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, "Touching from a distance: Website fingerprinting attacks and defenses," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 605–616, ACM, 2012.

[19] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas, "Circuit fingerprinting attacks: Passive deanonymization of tor hidden services," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 287–302, USENIX Association, Aug. 2015.

[20] J. Rizzo and T. Duong, "The CRIME attack," in *EKOparty Security Conference*, vol. 2012, 2012.

[21] Y. Gluck, N. Harris, and A. Prado, "BREACH: reviving the CRIME attack," *Unpublished manuscript*, 2013.

[22] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: System-level protection against cache-based side channel attacks in the cloud," in *19th USENIX Security Symposium*, USENIX, 2012.

[23] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic morphing: An efficient defense against statistical traffic analysis," in *Proc. Network and Distributed Systems Symposium (NDSS)*, The Internet Society, 2009.

[24] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, and R. K. C. Chang, "HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows," in *Proc. Network and Distributed Systems Symposium (NDSS)*, The Internet Society, 2011.

[25] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, pp. 494–505, ACM, 2007.

[26] S. Gueron, "Intel advanced encryption standard (aes) instructions set," *Intel White Paper, Rev*, vol. 3, 2009.

[27] S. Gueron, "Intel's new aes instructions for enhanced performance and security," in *Fast Software Encryption*, pp. 51–66, Springer, 2009.

[28] Intel Corporation, "ARK Processor Feature Filter." http://ark.intel.com/search/advanced?s=t&AESTech=false. Accessed: 15 December 2015.

[29] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *CHES*, pp. 1–17, 2009.

[30] M. Hamburg, "Accelerating aes with vector permute instructions," in *Cryptographic Hardware and Embedded Systems-CHES 2009*, pp. 18–32, Springer, 2009.

[31] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *LATINCRYPT*, pp. 159–176, Springer, 2012.

[32] D. J. Bernstein, T. Chou, and P. Schwabe, "Mcbits: fast constant-time code-based cryptography," in *Cryptographic Hardware and Embedded Systems-CHES 2013*, pp. 250–272, Springer, 2013.

[33] J. W. Bos, C. Costello, H. Hisil, and K. Lauter, "High-performance scalar multiplication using 8-dimensional glv/gls decomposition," in *Cryptographic Hardware and Embedded Systems-CHES 2013*, pp. 331–348, Springer, 2013.

[34] OpenSSL Software Foundation, "OpanSSL Changelog." https://www.openssl.org/news/changelog.txt. Accessed: 16 December 2015.

[35] Moritz Schulte, "Libgcrypt 1.1.42 released." https://lists.gnupg.org/pipermail/gnupg-announce/2003q3/000155.html. Accessed: 16 December 2015.

[36] ARM Limited, "PolarSSL 1.2.9 released." https://polarssl.org/tech-updates/releases/polarssl-1.2.9-released. Accessed: 16 December 2015.

[37] W. Schindler, "Exponent blinding may not prevent timing attacks on rsa,"

[38] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Information Flow, Relations and Polymorphic Types," *Journal of Logic and Computation*, vol. 18, no. 2, pp. 181–199, 2005.

[39] B. Köpf and D. Basin, "An Information-Theoretic Model for Adaptive Side-Channel Attacks," in *CCS*, pp. 286–296, ACM, 2007.

[40] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *SSP*, pp. 141–153, IEEE, 2009.

[41] B. Köpf and A. Rybalchenko, "Approximation and randomization for quantitative information-flow analysis," in *CSF*, pp. 3–14, IEEE, 2010.

[42] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *ACSAC*, pp. 261–269, ACM, 2010.

[43] G. Doychev and B. Köpf, "Rational Protection Against Timing Attacks," in *Proc. 28th IEEE Computer Security Foundations Symposium (CSF'15)*, IEEE, 2015.

[44] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *22nd USENIX Security Symposium*, USENIX, 2013.

[45] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security*, vol. 18, pp. 4:1–4:32, June 2015.

[46] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," *CoRR arXiv*, vol. abs/1603.02187, March 2016.

[47] M. Backes, G. Doychev, and B. Köpf, "Preventing Side-Channel Leaks in Web Traffic: A Formal Approach," in *Proc. 20th Network and Distributed Systems Security Symposium (NDSS)*, Internet Society, 2013.

[48] G. Doychev, "Analysis and mitigation of information leaks in web applications," Master's thesis, Saarland University, Germany, 2012.

[49] G. Smith, "On the foundations of quantitative information flow," in *FoSSaCS*, Springer, 2009.

[50] C.-Y. Hsiao, C.-J. Lu, and L. Reyzin, "Conditional computational entropy, or toward separating pseudoentropy from compressibility," in *EUROCRYPT*, pp. 169–186, Springer, 2007.

[51] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[52] J. L. Massey, "Guessing and Entropy," in *Proc. 1994 IEEE Symposium on Information Theory (ISIT 1994)*, p. 204, IEEE, 1994.

[53] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *CSF*, pp. 324–335, IEEE, 2009.

[54] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudorandom bits," *SIAM journal on Computing*, vol. 13, no. 4, pp. 850–864, 1984.

[55] D. Aggarwal and U. Maurer, "The leakage-resilience limit of a computational problem is equal to its unpredictability entropy," in *ASIACRYPT*, pp. 686–701, Springer, 2011.

[56] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT*, pp. 256–266, Springer, 1997.

[57] U. Maurer, "Abstract models of computation in cryptography," in *Cryptography and Coding*, pp. 1–12, Springer, 2005.

[58] S. Krenn, K. Pietrzak, A. Campus, A. Wadia, and D. Wichs, "A counterexample to the chain rule for conditional hill entropy," 2014.

[59] D. E. Denning, *Cryptography and Data Security*. Addison-Wesley, 1982.

[60] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *JCS*, vol. 15, no. 3, pp. 321–371, 2007.

[61] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden, "Anonymity protocols as noisy channels," *Information and Computation*, vol. 206, pp. 378–401, 2008.

[62] C. Braun, K. Chatzikokolakis, and C. Palamidessi, "Quantitative notions of leakage for one-try attacks," *Electronic Notes in Theoretical Computer Science*, vol. 249, pp. 75–91, 2009.

[63] M. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith, "Measuring information leakage using generalized gain functions," in *CSF*, pp. 265–279, IEEE, 2012.

[64] J. K. Millen, "Covert Channel Capacity," in *Proc. 1987 IEEE Symposium on Security and Privacy (Oakland 1987)*, pp. 60–66, IEEE, 1987.

[65] G. Lowe, "Quantifying Information Flow," in *Proc. 15th IEEE Computer Security Foundations Symposium (CSFW 2002)*, pp. 18–31, IEEE, 2002.

[66] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *CAV*, pp. 564–580, Springer, 2012.

[67] AbsInt Angewandte Informatik GmbH, "AbsInt aiT Worst-Case Execution Time Analyzers." http://www.absint.com/ait/, January 2015.

[68] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *PLAS*, pp. 73–85, ACM, 2009.

[69] Z. Meng and G. Smith, "Calculating bounds on information leakage using two-bit patterns," in *PLAS*, ACM, 2011.

[70] M. Boreale and F. Pampaloni, "Quantitative multirun security under active adversaries," in *QEST*, pp. 158–167, IEEE, 2012.

[71] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '08)*, pp. 193–205, ACM, 2008.

[72] P. Mardziel, M. S. Alvim, M. W. Hicks, and M. R. Clarkson, "Quantifying information flow for dynamic secrets," in *SSP*, 2014.

[73] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *CCS*, pp. 563–574, ACM, 2011.

[74] M. Tambe, *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press, 2011.

[75] S. Bhattacharya, V. Conitzer, and K. Munagala, "Approximation algorithm for security games with costly resources," in *Internet and Network Economics*, pp. 13–24, Springer, 2011.

[76] J. Blocki, N. Christin, A. Datta, A. D. Procaccia, and A. Sinha, "Audit games," in *IJCAI*, pp. 41–47, AAAI Press, 2013.

[77] R. Shokri, G. Theodorakopoulos, C. Troncoso, J.-P. Hubaux, and J.-Y. Le Boudec, "Protecting location privacy: optimal strategy against localization attacks," in *CCS*, pp. 617–627, ACM, 2012.

[78] R. Shokri, "Privacy games: Optimal user-centric data obfuscation," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 299–315, 2015.

[79] M. Khouzani, P. Mardziel, C. Cid, and M. Srivatsa, "Picking vs. guessing secrets: A game-theoretic analysis," in *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pp. 243–257, IEEE, 2015.

[80] J. A. Garay, J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Rational protocol design: Cryptography against incentive-driven adversaries," in *FOCS*, pp. 648–657, 2013.

[81] S. Dziembowski and K. Pietrzak, "Leakage-resilient cryptography," in *FOCS*, IEEE, 2008.

[82] Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung, "Practical leakage-resilient pseudorandom generators," in *CCS*, pp. 141–151, ACM, 2010.

[83] P. C. Kocher, "Leak-resistant cryptographic indexed key update," Mar. 25 2003. US Patent 6,539,092.

[84] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa, "Leakage Resilience against Concurrent Cache Attacks," in *Proc. 3rd Conference on Principles of Security and Trust (POST '14)*, Springer, 2014.

[85] S. Belaïd, V. Grosso, and F.-X. Standaert, "Masking and leakage-resilient primitives: One, the other (s) or both?," *Cryptography and Communications*, vol. 7, no. 1, pp. 163–184, 2015.

[86] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Sci. Comput. Program.*, vol. 78, no. 7, pp. 796–812, 2013.

[87] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1267–1279, ACM, 2014.

[88] J. B. Almeida12, M. Barbosa13, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc," in *Foundations of Software Engineering (FSE)*, 2016.

[89] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *SSP*, pp. 143–157, IEEE, 2012.

[90] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[91] D. Fudenberg and J. Tirole, *Game Theory*. MIT Press, 1991.

[92] "Functional safety of electrical, electronic and programmable electronic safety related systems–IEC 61508." www.iec.ch/functionalsafety/.

[93] "ECRYPT II Yearly Report on Algorithms and Key Lengths (2011)," June 2011.

[94] E. Zermelo, "Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels," in *Proc. Fifth International Congress of Mathematicians*, vol. 2, pp. 501–504, II, Cambridge UP, Cambridge, 1913.

[95] H. W. Kuhn, "Extensive games and the problem of information," *Contributions to the Theory of Games*, vol. 2, no. 28, pp. 193–216, 1953.

[96] V. Conitzer and T. Sandholm, "Computing the optimal strategy to commit to," in *EC*, pp. 82–90, ACM, 2006.

[97] E. Kiltz and K. Pietrzak, "Leakage resilient elgamal encryption," in *ASIACRYPT*, Springer, 2010.

[98] N. Heninger and H. Shacham, "Reconstructing rsa private keys from random key bits," in *CRYPTO*, pp. 1–17, Springer, 2009.

[99] B. Köpf and G. Smith, "Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks," in *CSF*, pp. 44–56, IEEE, 2010.

[100] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *DoD HPCMP Users Group Conference*, 1999.

[101] J. W. Bos, "Constant time modular inversion," *Journal of Cryptographic Engineering*, vol. 4, no. 4, pp. 275–281, 2014.

[102] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[103] D. Grund, *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012.

[104] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *SSP*, pp. 490–505, IEEE, 2011.

[105] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM J. Comput.*, vol. 38, no. 1, pp. 97–139, 2008.

[106] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints," in *POPL*, pp. 238–252, 1977.

[107] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *ESOP*, vol. 3444 of *LNCS*, pp. 5–20, Springer, 2005.

[108] P. Cousot, R. Cousot, and L. Mauborgne, "Theories, solvers and static analysis by abstract interpretation," *Journal of the ACM*, vol. 59, no. 6, p. 31, 2012.

[109] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *POPL*, 1979.

[110] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *CCS*, pp. 199–212, ACM, 2009.

[111] O. Aciiçmez and Ç. K. Koç, "Trace-driven cache attacks on AES," in *ICICS*, pp. 112–121, Springer, 2006.

[112] S. Gueron, "Intel Advanced Encryption Standard (AES) Instructions Set." http://software.intel.com/file/24917, 2010.

[113] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *PLDI*, pp. 99–110, ACM, 2012.

[114] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *SSP*, pp. 45–60, IEEE, 2009.

[115] Ú. Erlingsson and M. Abadi, "Operating system protection against side-channel attacks that exploit memory latency," tech. rep., 2007.

[116] ECRYPT, "The eSTREAM portfolio in 2012." http://www.ecrypt.eu.org/documents/D.SYM.10-v1.pdf, 2012.

[117] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pp. 65–74, IEEE, 2013.

[118] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt, "Cache behavior prediction by abstract interpretation," *Science of Computer Programming*, vol. 35, no. 2, pp. 163 – 189, 1999.

[119] A. Chlipala, "Modular development of certified program verifiers with a proof assistant," in *ICFP*, pp. 160–171, ACM, 2006.

[120] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *VMCAI*, pp. 214–228, 2009.

[121] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *FMPA*, Springer, 1993.

[122] D. Feld, "Relational Domains for the Quantification of Cache Side Channels," Master's thesis, Saarland University, 2013.

[123] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *IEEE Proceedings on Real-Time Systems*, vol. 91, no. 7, pp. 1038–1054, 2003.

[124] H. Wu, "The Stream Cipher HC-128." http://www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf, 2004.

[125] E. Zenner, "A cache timing analysis of hc-256," in *Selected Areas in Cryptography*, Springer, 2009.

[126] G. Paul and S. Raizada, "Impact of extending side channel attack on cipher variants: a case study with the hc series of stream ciphers," in *Security, Privacy, and Applied Cryptography Engineering*, pp. 32–44, Springer, 2012.

[127] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, "The Stream Cipher Rabbit." http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf, 2005.

[128] D. Bernstein, "Leaks." http://cr.yp.to/streamciphers/leaks.html, January 2015.

[129] D. Bernstein, "Snuffle 2005: the Salsa20 encryption function." http://cr.yp.to/snuffle.html, January 2015.

[130] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert, "Sosemanuk, a fast software-oriented stream cipher." http://www.ecrypt.eu.org/stream/p3ciphers/sosemanuk/sosemanuk_p3.pdf, 2005.

[131] G. Leander, E. Zenner, and P. Hawkes, "Cache timing analysis of lfsr-based stream ciphers," in *Cryptography and Coding*, pp. 433–445, Springer, 2009.

[132] J. Agat and D. Sands, "On confidentiality and algorithms," in *SSP*, pp. 64–77, IEEE, 2001.

[133] Code Beach, "Sorting algorithms." http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html, 2008. Accessed: 7 January 2015.

[134] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *CCS*, ACM, 2014.

[135] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security," in *ISCA*, pp. 189–200, ACM, 2011.

[136] L. Domnitser, A. Jaleel, J. Loew, N. B. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *TACO*, vol. 8, no. 4, p. 35, 2012.

[137] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *41st IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, pp. 83–93, 2008.

[138] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography." Cryptology ePrint Archive, Report 2014/422, 2014.

[139] M. B. Baig, C. Fitzsimons, S. Balasubramanian, R. Sion, and D. E. Porter, "Cloudflow: Cloud-wide policy enforcement using fast vm introspection," in *IC2E*, IEEE, 2014.

[140] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proc. ACM Cloud Computing Security Workshop, (CCSW)*, pp. 77–84, 2009.

[141] B. Ford, "Plugging side-channel leaks with timing information flow control," in *HotCloud*, USENIX, 2012.

[142] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology-ICISC 2005*, pp. 156–168, Springer, 2006.

[143] J. Agat, "Transforming out timing leaks," in *POPL 2000*, pp. 40–53, ACM, 2000.

[144] D. Hedin and D. Sands, "Timing aware information flow security for a JavaCard-like bytecode," *ENTCS*, vol. 141, no. 1, pp. 163–182, 2005.

[145] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa." Cryptology ePrint Archive, Report 2016/224, March 2016. http://eprint.iacr.org/.

[146] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*, pp. 605–622, IEEE Computer Society, 2015.

[147] S. Gueron, "Efficient software implementations of modular exponentiation," *J. Cryptographic Engineering*, vol. 2, no. 1, pp. 31–43, 2012.

[148] G. Barthe, T. Rezk, and M. Warnier, "Preventing Timing Leaks Through Transactional Branching Instructions," in *Proc. 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL 2006)*, Electronic Notes in Theoretical Computer Science (ENTCS), pp. 33–55, Elsevier, 2005.

[149] H. Mantel and A. Starostin, "Transforming out timing leaks, more or less," in *ESORICS*, pp. 447–467, Springer, 2015.

[150] J. Agat, "Transforming out timing leaks in practice: An experiment in implementing programming language-based methods for confidentiality," 2000.

[151] A. Langley, "Checking that functions are constant time with valgrind." https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010. Accessed: 6 March 2016.

[152] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *IEEE Symposium on Security and Privacy (SSP)*, pp. 19–30, IEEE, 2002.

[153] G. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy Vulnerabilities in Encrypted HTTP Streams," in *Proc. Privacy Enhancing Technologies (PET)*, pp. 1–11, Springer, 2005.

[154] M. Liberatore and B. N. Levine, "Inferring the Source of Encrypted HTTP Connections," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, pp. 255–263, ACM, 2006.

[155] S. E. Coull, M. P. Collins, C. V. Wright, F. Monrose, and M. K. Reiter, "On web browsing privacy in anonymized netflows," in *Proc. 16th USENIX Security Symposium*, pp. 23:1–23:14, USENIX Association, 2007.

[156] G. Danezis, "Traffic analysis of the http protocol over tls," 2007.

[157] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive-bayes classifier," in *Proc. ACM Workshop on Cloud Computing Security (CCSW)*, pp. 31–42, ACM, 2009.

[158] L. Lu, E.-C. Chang, and M. C. Chan, "Website Fingerprinting and Identification Using Ordered Feature Sequences," in *Proc. 15th European Symposium on Research in Computer Security (ESORICS)*, Springer, 2010.

[159] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-Boo, I still see you: Why Traffic Analysis Countermeasures Fail," in *IEEE Symposium on Security and Privacy (SSP)*, pp. 332–346, IEEE, 2012.

[160] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: automated detection and quantification of side-channel leaks in web application development," in *Proc. ACM Conference on Computer and Communication Security (CCS)*, pp. 595–606, ACM, 2010.

[161] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, pp. 263–274, ACM, 2011.

[162] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999.

[163] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[164] R. Sarukkai, "Link prediction and path analysis using markov chains," *Computer Networks*, vol. 33, no. 1-6, pp. 377–386, 2000.

[165] M. Eirinaki, M. Vazirgiannis, and D. Kapogiannis, "Web path recommendations based on page ranking and markov models," in *Proc. 7th ACM Workshop on Web Information and Data Management (WIDM)*, pp. 2–9, ACM, 2005.

[166] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

[167] M. Eirinaki and M. Vazirgiannis, "Usage-Based PageRank for Web Personalization," in *Proc. 5th IEEE Intl. Conference on Data Mining (ICDM)*, pp. 130–137, IEEE, 2005.

[168] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Inf. Comput.*, vol. 94, no. 1, pp. 1–28, 1991.

[169] J. Kemeny and J. Snell, *Finite Markov Chains*. Undergraduate Texts in Mathematics, Springer-Verlag, 1960.

[170] S. Derisavi, H. Hermanns, and W. H. Sanders, "Optimal state-space lumping in markov chains," *Inf. Process. Lett.*, vol. 87, no. 6, pp. 309–315, 2003.

[171] C. Fellbaum, ed., *WordNet: an electronic lexical database*. MIT Press, 1998.

[172] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website Fingerprinting in Onion Routing Based Anonymization Networks," in *Proc. ACM Workshop on Privacy in the Electronic Society (WPES)*, ACM, 2011.

[173] W. M. Liu, L. Wang, K. Ren, P. Cheng, and M. Debbabi, "k-indistinguishable traffic padding in web applications," in *Proc. Privacy Enhancing Technologies (PET)*, pp. 79–99, Springer, 2012.

[174] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld, "The complexity of approximating entropy," in *Proc. 34th ACM Symposium on Theory of Computing (STOC 2002)*, pp. 678–687, ACM, 2002.

[175] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden, “Anonymity protocols as noisy channels,” *Inf. Comput.*, vol. 206, no. 2-4, pp. 378–401, 2008.

[176] C. Dwork, “Differential Privacy,” in *Proc. 33rd Intl. Colloquium on Automata, Languages and Programming (ICALP)*, pp. 1–12, Springer, 2006.