MORGAN & CLAYPOOL PUBLISHERS

Principles of Secure Processor Architecture Design

Jakub Szefer

Synthesis Lectures on Computer Architecture

Margaret Martonosi, *Series Editor*

Principles of Secure Processor Architecture Design

Synthesis Lectures on Computer Architecture

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Space-Time Computing with Temporal Neural Networks
James E. Smith
2017

Hardware and Software Support for Virtualization
Edouard Bugnion, Jason Nieh, and Dan Tsafrir
2017

Datacenter Design and Management: A Computer Architect's Perspective
Benjamin C. Lee
2016

A Primer on Compression in the Memory Hierarchy
Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A. Wood
2015

Research Infrastructures for Hardware Accelerators
Yakun Sophia Shao and David Brooks
2015

Analyzing Analytics
Rajesh Bordawekar, Bob Blainey, and Ruchir Puri
2015

Customizable Computing
Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao
2015

Die-stacking Architecture
Yuan Xie and Jishen Zhao
2015

Single-Instruction Multiple-Data Execution
Christopher J. Hughes
2015

Power-Efficient Computer Architectures: Recent Advances
Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras
2014

FPGA-Accelerated Simulation of Computer Systems
Hari Angepat, Derek Chiou, Eric S. Chung, and James C. Hoe
2014

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong
2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts and John Kim
2011

Processor Microarchitecture: An Implementation Perspective
Antonio González, Fernando Latorre, and Grigorios Magklis
2010

Transactional Memory, Second Edition
Tim Harris, James Larus, and Ravi Rajwar
2010

Computer Architecture Performance Evaluation Methods
Lieven Eeckhout
2010

Introduction to Reconfigurable Supercomputing
Marco Lanzagorta, Stephen Bique, and Robert Rosenberg
2009

On-Chip Networks
Natalie Enright Jerger and Li-Shiuan Peh
2009

The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It
Bruce Jacob
2009

Fault Tolerant Computer Architecture
Daniel J. Sorin
2009

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines
Luiz André Barroso and Urs Hölzle
2009

Computer Architecture Techniques for Power-Efficiency
Stefanos Kaxiras and Margaret Martonosi
2008

Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency
Kunle Olukotun, Lance Hammond, and James Laudon
2007

Transactional Memory
James R. Larus and Ravi Rajwar
2006

Quantum Computing for Computer Architects
Tzvetan S. Metodi and Frederic T. Chong
2006

Principles of Secure Processor Architecture Design

Jakub Szefer

www.morganclaypool.com

ISBN: 9781681730011 paperback
ISBN: 9781681730028 ebook
ISBN: 9781681734040 hardcover

DOI 10.2200/S00864ED1V01Y201807CAC045

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #45
Series Editor: Margaret Martonosi, *Princeton University*
Founding Editor Emeritus: Mark D. Hill, *University of Wisconsin, Madison*
Series ISSN
Print 1935-3235 Electronic 1935-3243

Principles of Secure Processor Architecture Design

Jakub Szefer
Yale University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #45

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

With growing interest in computer security and the protection of the code and data which execute on commodity computers, the amount of hardware security features in today's processors has increased significantly over the recent years. No longer of just academic interest, security features inside processors have been embraced by industry as well, with a number of commercial secure processor architectures available today. This book aims to give readers insights into the principles behind the design of academic and commercial secure processor architectures. Secure processor architecture research is concerned with exploring and designing hardware features inside computer processors, features which can help protect confidentiality and integrity of the code and data executing on the processor. Unlike traditional processor architecture research that focuses on performance, efficiency, and energy as the first-order design objectives, secure processor architecture design has security as the first-order design objective (while still keeping the others as important design aspects that need to be considered).

This book aims to present the different challenges of secure processor architecture design to graduate students interested in research on architecture and hardware security and computer architects working in industry interested in adding security features to their designs. It aims to educate readers about how the different challenges have been solved in the past and what are the best practices, i.e., the principles, for design of new secure processor architectures. Based on the careful review of past work by many computer architects and security researchers, readers also will come to know the five basic principles needed for secure processor architecture design. The book also presents existing research challenges and potential new research directions. Finally, this book presents numerous design suggestions, as well as discusses pitfalls and fallacies that designers should avoid.

KEYWORDS

secure processor design, processor architecture, computer security, trustworthy computing, computer hardware security

Dla ukochanej Injoong i najwspanialszej Adusi.

Contents

Preface

Recent years have brought increased interest in hardware security features that can be added to computer processors to protect sensitive code and data. It has been realized that new hardware security features can be used to provide, for example, means of authentication or protection of confidentiality and integrity. The hardware offers a very high level of immutability, helping to ensure that it is difficult to change the hardware security protections (unlike with software-only protections). Hardware cannot be as easily bypassed or subverted as software, as it is the ultimate lowest layer of a computer system. Finally, dedicated hardware for providing security protections can potentially offer energy efficiency and minimal impact on system performance.

Yet, adding security features in hardware has many challenges. Defining what has to be secured, and how, is often a subjective choice based on qualitative arguments—unlike the quantitative choices that computer architects are used to making. Moreover, once made, the hardware cannot be easily changed, which necessitates careful design of the security features in the first place. The secure architecture design process also requires foresight to include features and algorithms that will be suitable for many years to come. Perhaps the biggest challenges are the attacks and various information leaks that the system should protect against. Not only random errors or faults need to be considered, but the system also needs to defend against "smart" attackers who can intelligently manipulate inputs or probe the hardware to try to maximize their chances of subverting the computer system's protections.

This book assumes readers may be at the level of a first- or second-year graduate student in computer architecture. The book is also suitable for more senior students or for practicing computer architects who are interested in starting work on the design of secure processor architectures. The book provides a chapter on security topics such as encryption, hashing, confidentiality, and integrity, to name a few—consequently a background in computer security is not required. It is the hope that this book will get computer architects excited about security and help them work on secure processor architectures.

The chapters of this book are based on research ideas developed by the author and also ideas gleaned from papers that a variety of researchers have presented in conferences such as ISCA, ASPLOS, HPCA, CCS, S&P, and Usenix Security. Information is also included about recent commercial architectures, such as Intel SGX, ARM TrustZone, and AMD memory encryption technologies. The book, however, is not meant as a manual or tutorial about any one specific security architecture. Rather, it uses past academic and industry research to derive and present the principles behind design of such secure processor architectures.

This book is divided into ten chapters. Chapter 1 focuses on motivating the need for secure processor architectures and gives an overview of the book's organization. Chapter 2 covers

basics of computer security needed for understanding secure processor architecture designs. It can be considered an optional chapter for those already familiar with major computer security topics. Chapter 3 discusses main features of secure processor architectures, such as extending processors with new privilege levels, or breaking the traditional linear hierarchy of the privilege levels. Chapter 4 focuses on the Trusted Execution Environments which are created by the hardware and software Trusted Computing Base, and discusses various protections that secure architectures can offer to the Trusted Execution Environments. Chapter 5 introduces the Root of Trust from which most of the security features of a secure processor architecture are derived. Chapter 6 is an in-depth discussion of protections that secure architectures use to protect main memory, usually DRAM. Chapter 7 overviews security features that target designs with many processors or many processor cores. Chapter 8 gives extended review of side channel threats, processor features that contribute to existence of side channels, and ideas for eliminating various side channels. Chapter 9 is an optional chapter, which can be considered a mini survey of work on security verification of processor architectures and hardware. Chapter 10 concludes the book by presenting the five principles for secure processor architecture design, along with research challenges and future trends in secure processor designs.

After finishing this book, readers should be familiar with the five design principles for secure processor architecture design, numerous design suggestions, as well as become educated about pitfalls and fallacies that they should avoid when working on secure processor designs. Most importantly, security at the processor and hardware level is a crucial aspect of today's computers, and this book aims to educate and excite readers about this research area and its possibilities.

Jakub Szefer
October 2018

Acknowledgments

The ideas and principles derived in this book are based not only on my own research, but also on research and ideas explored over many years by numerous researchers and gleaned from their academic papers presented in top architecture and security conferences. I would like to especially acknowledge my former Ph.D. adviser, Prof. Ruby B. Lee, and others with whom I learned about, and worked on, secure processor architectures. The principles and ideas presented here reflect the hard work of many researchers and of the broader computer architecture and security communities.

I would like to thank Prof. Margaret Martonosi, the editor of the Synthesis Lectures on Computer Architecture series, and Michael B. Morgan, President and CEO of Morgan & Claypool Publishers, for their support and deadline extensions. I hope this book is a valuable addition to the series, and it was made much better through their input and encouragement. In addition, this book was improved thanks to the feedback and reviews from Margaret Martonosi, Caroline Trippel, and Chris Fletcher. Further, I would like to thank Dr. C.L. Tondo, Christine Kiilerich, and the copyeditors for helping bring this book to reality.

Work on this book was made possible in part through generous support from the National Science Foundation, through grants number 1716541, 1524680, and an NSF CAREER award number 1651945, and through support by Semiconductor Research Corporation (SRC). It was further made possible through support from Yale University.

Special thanks to my current Ph.D. students: Wenjie Xiong, Wen Wang, Shuwen Deng, and Shanquan Tian. It is a pleasure to work with them on secure processor architectures, hardware security, and other topics related to improving computer hardware security; our work forces me to constantly learn new ideas and push the boundaries on these exciting research topics.

I would like to thank my parents, Ewa and Krzysztof, for their constant encouragement, especially during my years in graduate school, and now in my academic career. Their unwavering love and support can always be counted on.

Most importantly, I would like to thank my amazing wife, Injoong, for all she does. Without her, my research, work, and this book would not be possible. She is the most loving wife and my best friend. And last, but not least, many hugs and kisses to our baby daughter, Adriana, for being the cutest and smartest baby ever! Every day is a surprise and she brings nothing but joy to me.

Jakub Szefer
October 2018

CHAPTER 1

Introduction

This chapter provides motivation for research and work on secure processor architectures. It also provides an outline of the organization of this book and, in particular, highlights the core and the optional chapters.

1.1 NEED FOR SECURE PROCESSOR ARCHITECTURES

Secure processor architectures by design provide extra hardware features which enhance commodity processors with new security capabilities. The new security features may be purely in hardware, or they may be implemented in both hardware and software. The former are the so-called hardware security architectures, and the latter are the so-called hardware-software architectures. Unlike hardware security modules or dedicated security accelerators (see Section 3.6.5), secure processor architectures are mainly designed as extensions of commodity processors, and are based on architectures such as such x86 or RISC. The increased need for implementing security features in processor architectures has been driven in recent years by three factors.

- **Software Complexity and Bugs**: Increased complexity and size of the software code running on commodity processors, especially the operating system or the hypervisor code, makes it impractical, or even impossible, to provide security solely based in software—more and more lines of software code lead to increased number of software bugs and potential exploits. New features (e.g., new protection levels or new hardware features for creating trusted software executing environments) are needed to provide an execution environment wherein a small, trusted code can execute separated from the rest of the untrusted code.

- **Side-Channel Attacks**: Computation is today often done in settings such as in cloud computing where many different users share the same physical hardware. Co-residency of potential victims and attackers on same hardware can allow the attackers to learn sensitive information through shared hardware and the side channels. Timing-based side channels, and also power, RF, or EM-based ones, exploit known side effects of the behavior of commodity processors when different types of computations are performed. Only modifications at the architecture and hardware levels can mitigate different types of side channels and the resulting side-channel attacks.

- **Physical Attacks**: Attacks including physical probing of memory busses or even memory chips have necessitated protections against not just software, but hardware or physical

attacks, especially as cloud computing has increased in popularity, where users no longer have physical control over the hardware on which their code runs, new mechanisms are needed to protect the code and data against physical attacks. Likewise, embedded devices or Internet-of-Things devices are prone to physical attacks as users may not have physical control over the devices at all times.

The first motivating factor for the need for secure processor architectures listed above is the constantly increasing code size of the software, and consequent number of bugs and exploits. According to some estimates there can be as many as 20 bugs per 1000 lines of software code [69]. Even if it is an overestimate, an operating system or hypervisor with millions of lines of code will have thousands of bugs. The operating system is typically in charge of managing applications, and as long as the operating system is running correctly, applications are protected from one another. Today, however, the code base of an average operating system has grown to tens of millions of lines of code and it is no longer possible for operating system to be bug free—the operating system is no longer trustworthy to protect the applications. Similarly, after introduction of the new hypervisor privilege level, hypervisors were supposed to be small and trustworthy to isolate different operating systems form one another. Yet today, hypervisors are more like operating systems with millions of lines of code. The bloated code base makes the operating systems and hypervisors less trustworthy. To secure applications (from each other and from the untrustworthy operating system) or to secure virtual machines (VMs) (from each other and from the untrustworthy hypervisor), a variety of secure processor architectures, and design ideas, have been proposed that leverage changes in the architecture. New privilege levels or mechanisms for separation of code and data can help protect trusted code from the rest of the code running on the computer system.[1]

The second motivating factor for the need for secure processor architectures is side-channel attacks. Side-channel attacks that leverage timing, power, RF, or EM changes or emanations as processors execute different programs and instructions. The side-channel attacks have been known and explored for a long time. Defending against such types of attacks, however, has gained new urgency after Spectre [120] and Meltdown [138] attacks were publicized (while this book was being written), which partly leverage cache timing side channels, for example.

The third factor driving the need for secure processor architectures is the physical attacks. Lack of physical control over the computer system where the code is executing means that users can no longer be sure that there is physical security and that nobody is tampering with their system. For example, in cloud computing, users rent servers or VMS that are located in far away data centers where rogue employees or hosting company compelled by the government can probe the server hardware while it is running. In another scenario, embedded devices, such as in now popular Internet-of-Things computing paradigm, are spread out in variety of locations where the owner may not be able to ensure that they are physically secure.

[1] Hardware designs can suffer form bugs in the hardware description code, just as there are bugs in software. Chapter 9 gives information about approaches for security verification of processor architectures and designs.

1.2 BOOK ORGANIZATION

Organization of this book's chapters is shown in Figure 1.1 below. The overarching design goal of secure architectures is to protect integrity and confidentiality of user applications, operating system, hypervisor, or other software components, depending on the threat model and assumptions of the particular architecture, and prevent software or hardware attacks (again, within limits of the particular threat model). To help students and practitioners learn about, and create, such solutions and protections, the remainder of the book is divided into a number of chapters focusing on major topic areas, culminating in the last chapter that presents the principles for secure processor architecture design.

The remaining chapters of this book can be divided into three groups. Chapter 2 covers basics of computer security needed for understanding secure processor architecture designs. It can be considered an optional chapter for those already familiar with topics such as: confidentiality, integrity, availability, symmetric and public-key cryptography, secure hashing, hash trees, etc. Chapters 3–8 and also Chapter 10 are the core of this book. Chapter 9 is again an optional chapter, which can be considered a mini survey of work on security verification of processor architectures and hardware.

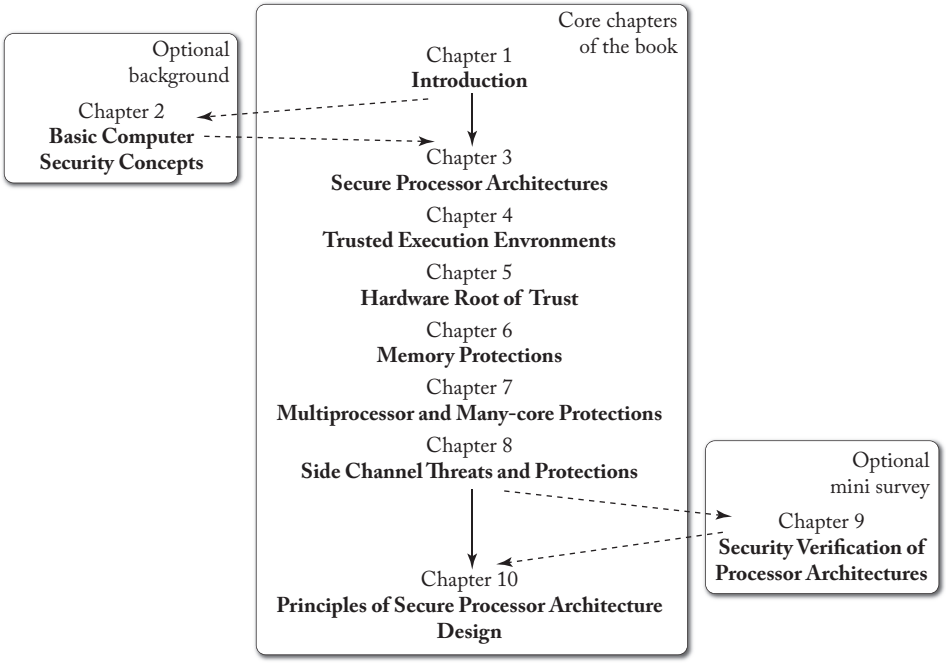

Figure 1.1: Organization of this book's chapters.

CHAPTER 2

Basic Computer Security Concepts

This chapter covers basics of computer security needed for understanding secure processor architecture designs. It begins with a discussion of the trusted computing base, security threats to a system, and discussion of threat models. A brief overview of information leaks and side channels is also included in the chapter as it will be needed to understand the side-channel attack protections that secure processor architectures needs. This chapter then dives into security concepts of confidentiality, integrity, and availability. It also explains basics of symmetric-key cryptography, public-key cryptography, and secure hashes. A short section on importance of good sources of randomness is included. The chapter closes with a short overview of physically unclonable functions (PUFs) and their applications.

2.1 TRUSTED COMPUTING BASE

In a secure processor architecture, the Trusted Computing Base (TCB) is formed by the hardware components and the software components that work together and provide some security guarantees, as specified by the architecture. There is a distinction that can be made between hardware secure architectures where the hardware is protecting the software but all security mechanisms are solely in hardware, or hardware-software secure, architectures where hardware works with some software (usually privileged software such as the operating system or the hypervisor) to protect other software. The latter can provide more flexibility, but typically increases the size of the TCB. Larger TCB is usually considered less desirable as more lines of software code (and likewise lines of hardware description code) are assumed to be correlated with more potential bugs [69] and consequently security vulnerabilities.

A typical computer system can be broken down into a number of distinct hardware components, e.g., processor cores, processor caches, interconnect, Dynamic Random Access Memory (DRAM), etc. These hardware components interact with each other as well as with the different software components as the system executes. The hardware components which are dependent upon to provide security form the hardware TCB. Meanwhile, the software components (if any) which are dependent upon to provide security form the software TCB. These hardware and software components that work together and provide some security guarantees are assumed to be trusted, and together form the whole TCB.

Correct operation of the system depends on the correctness of the TCB. The goal of secure processor architectures is then to ensure that the trusted hardware components can work together with the trusted software components to provide the desired security properties and protections for the software (e.g., trusted software modules or enclaves, whole user applications, or even containers or VMs) running on the system.

In addition to the trusted hardware and software components, there are the untrusted parts of the system. The untrusted parts (hardware or software) need not be overtly malicious, but are simply not trusted for the correct operation of the system. During the design of a secure processor architecture, the trusted computing base has to be constructed such that, regardless of the actions taken by the untrusted parts, the security properties of the system will be maintained. Effectively, each untrusted entity can be a potential attacker that tries to break the security of the system—the designer has to then consider all possible attacks by all the untrusted entities, unless a specific threat is explicitly not protected against, as specified in the threat model (discussed in Section 2.2.6). Beyond the untrusted entities which are part of the system, there are external attackers which should also be considered, i.e., physical attacks on the computer system.

It should be emphasized that trusted parts are ones on which the correct operation of the system explicitly depends on. If something happens to a trusted part (e.g., a software function is altered or a hardware module is modified) then the protections of the whole system can no longer be assumed. The trusted part, however, could be malicious to begin with or buggy due to poor design. A trusted software or a trusted hardware part thus may or may not be trustworthy.

Trustworthiness is a qualitative designation indicating whether the entity will behave as expected, is free of bugs and vulnerabilities, and is not malicious. The designation of an entity as trustworthy is separate from the designation of the entity as trusted. A secure processor architecture is designed with explicit assumption about which hardware or software entities need to be trusted, i.e., these entities from the TCB. During the design and implementation of these entities, it needs to be ensured that they are indeed trustworthy. Techniques such as formal security verification [52] should be applied to make sure the design is correct. However, even beyond design, bugs, or malicious modifications can be introduced during manufacturing time, e.g., hardware trojans can be added [213].

Architecture designers typically focus on ensuring that the protocols, interactions, interfaces, and use of encryption and hashing among the trusted components are such that there can be no attack. Once there is confidence that the system cannot be attacked due to a logical design in the flaw, the focus can move to the implementation details. Implementation details[1] focus on issues such as malicious foundries [231] or supply chain security [178]. Even when the lifetime of a processor ends, issues of trustworthiness can continue (e.g., make sure the system permanently destroys the encryption keys).

[1]It should be noted that side channels are related to both the design process, e.g., timing side channels due to the way the cache is designed are independent of the physical implementation details of the cache, and to the implementation process, e.g., different types of transistors or logic gates may create thermal or EM side channels. Side channels are discussed in Section 2.2.4.

2.1.1 KERCKHOFFS'S PRINCIPLE: AVOID SECURITY THROUGH OBSCURITY

Kerckhoffs' principle is well known and was first created in the context of cryptographic systems [167]. The principle can be paraphrased as stating that operation of the TCB of a security system should be publicly known and should have no secrets other then the cryptographic keys. Thus, even if an attacker knows everything about the operation of the TCB, he or she still cannot break the system unless he or she knows the cryptographic keys.

Many failed security systems practice the opposite of this principle, which is security through obscurity. Security through obscurity can be paraphrased as attempting to secure the system by making the operation of the TCB secret and hoping that any potential attackers are not able to reverse engineer the system and break it. Designers should not underestimate the cleverness of attackers and they should never practice security through obscurity. Security through obscurity has led to many real attacks, such as on metro cards [44], or could potentially lead to attacks on computer processors, as with researchers recently breaking into Intel's Management Engine [60] that runs secretive code which is in charge of the computer platform.

2.2 SECURITY THREATS TO A SYSTEM

The processor, the whole hardware of the computer system, and the software executing on it can be vulnerable to a number of security threats. The attackers can exploit the attack surface to mount different types of attacks, and these need to be protected against.

2.2.1 THE ATTACK SURFACE

The attack surface is the combination of all the attack vectors that can be used against a system. Individual attack vectors are different ways that an attacker can try to break system's security. Figure 2.1 shows different types of attack vectors (left side of the figure), along with potential parts of the system which can be targets of the attacks (bottom of the figure). Attack vectors could be through hardware or software, and in both cases could be due to so-called external attackers (attacker is not executing code on the target computer nor physically near the computer system they are attacking) or so-called internal attackers (the attacker is running code on the system he or she is trying to attack, or has physical access to the system). Hardware attacks could come from untrusted hardware (not in the TCB), or external physical attacks (e.g., physical probing of the memory or data buses). Software attacks could come from untrusted software (not in the TCB) or the software that is supposed to be protected (but due to bugs or malicious behavior tries to attack the system on which it is running). The targets of the attacks, shown at the bottom of Figure 2.1, can be hardware which is in the TCB, software which is in the TCB, or the software that is supposed to be protected. It does not make sense to try to attack the untrusted hardware or the untrusted software as by definition it does not provide any security mechanisms nor hold any sensitive data.

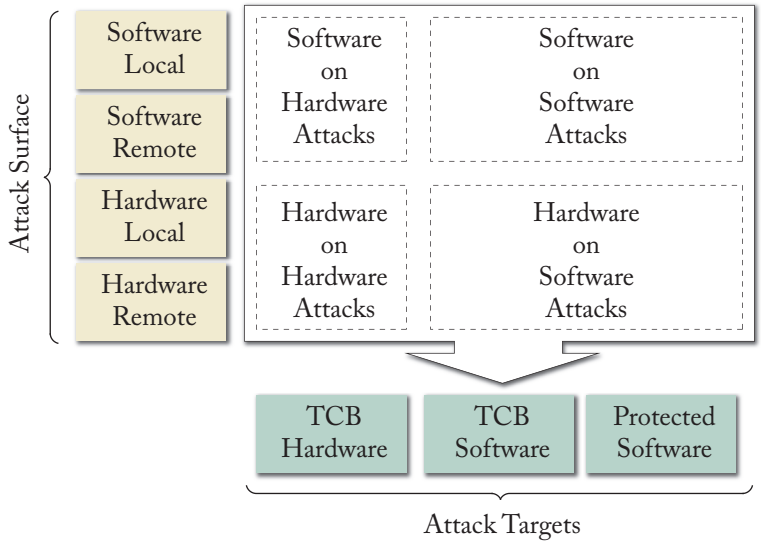

Figure 2.1: The potential attack surface of a secure processor. The terms external and internal refer to whether the potential attack is from within the system or from outside. The targets of the attacks can be either the TCB or the software that is being protected (by the TCB).

There are numerous examples of the different classes of attacks. Software-on-software attacks could be untrusted operating system attacking software that is being protected. A software-on-hardware attack could be untrusted software using cache side-channel attacks to learn secret information from a processor cache's operation. A hardware-on-software attack could be an untrusted memory controller trying to extract information from DRAM memory. A hardware-on-hardware attack could be untrusted peripheral trying to disable memory encryption engine. Note that components inside the software or hardware TCB are never assumed to be sources of attacks, as by definition, they are trusted with ensuring protections for the system.

2.2.2 PASSIVE AND ACTIVE ATTACKS

Passive attacks are the types of attacks where the attacker only observes the operation of the system. He or she does not actively trigger any inputs nor otherwise interact with the system. Passive attackers are most often called an eavesdroppers. They listen in on communication (internal or external to the system) to try to deduce the secret information or code. Side-channel attacks are types of passive attacks where attackers gather timing measurements, or power, EM, thermal, or other emanations from the system to try to learn what it is doing or to learn the secrets (more on side channels is mentioned shortly in Section 2.2.4).

Active attacks involve the attacker trying to modify code or data of the system. An attacker may try to write into some memory location (e.g., to change encryption key stored there) or

execute a series of instructions (e.g., to heat up the chip and cause it to fail). Physical attacks on processor chips such as fault injection are also active attacks as the attacker actively manipulates the (physical) state of the system.

The attacks can be further divided into: snooping, spoofing, splicing, replay, and disturbance attacks. Snooping attacks are passive attacks where the attacker simply tries to observe or read some information. Spoofing attacks are active attacks that, for example, involve injecting new memory commands to try to read or modify data in memory. Splicing attacks are also active attacks, which, for example, involve combining portions of legitimate memory commands (observed before through a snooping attack) into new memory commands (to read or modify data). Typically, splicing involves combining requests from one command (e.g., a memory read from, or a write to, a specific address) with authentication information from another command. Replay attacks are another example of active attacks which involve, for example, re-sending previously observed memory command again in the future. Disturbance attacks are the last type of active attack and they include attacks such as Denial of Service (DoS) on memory bus [153], using repeated memory accesses to age circuits [110], repeated memory access to trigger Rowhammer bug [117], etc.

2.2.3 MAN-IN-THE-MIDDLE ATTACKS

Man-in-the-middle attacks are attacks on communication (internal external to the system). As the name implies, man-in-the-middle attempts to intercept communication between two trusted components, and the goal is for neither entity to recognize that someone is intercepting the communication. Man-in-the-middle can be passive (receive data, read it, and forward to original destination without any changes) or active (modify data after receiving it or inject some new data).

2.2.4 SIDE AND COVERT CHANNELS AND ATTACKS

A covert channel [67] is a communication channel that was not intended or designed to transfer information between a sender and a receiver. Covert channels typically leverage unusual methods for communication of information, never intended by the system's designers. These channels can include use of timing, power, thermal emanations, electro-magnetic (EM) emanations, acoustic emanations, and possibly others. With the exception of timing channels, most channels require some physical proximity and sensors to detect the transmitted information, e.g., use of EM probe to sense EM emanations. Meanwhile, many timing-based channels are very powerful as they do not require physical access, only that sender and receiver run some code on the same system. Some covert channels can be prevented at the design time if they are known to exist, e.g., through proper isolation of processes or by partitioning caches, but many go unnoticed until the system is deployed.

Covert channels are important when considering intentional information exfiltration where one program manipulates the state of the system according to some protocol and an-

other observers the changes to read the "messages" that are sent to it. Covert channels are a concern because even when there is explicit isolation, e.g., each program runs in its own address space and cannot directly read and write another program's memory, a covert channel may allow the isolation mechanisms to be bypassed.

A side channel [128] is similar to a covert channel, but the sender does not intend to communicate information to the receiver, rather the sending (i.e., leaking) of information is a side effect of the implementation and the way the computer hardware or software is used. Side channels can use same means as covert channels, e.g., timing, to transmit information. Typically, covert channels and attacks are analyzed or presented first as both sender and receiver is under control of the potential attacker and it is easier to create a covert channel. Next, side channels are usually explored as they are more difficult to create since the victim (i.e., sender) is not under control of the attacker. Some side channels, like covert channels, can be prevented at the design time, e.g., through constant-time software implementations or by randomizing caches, but many also go unnoticed until the system is deployed.

Side channels are important when considering unintentional information leaks. In a side channel, there is usually a victim process that uses a computer system and the way the system is used can be observed by an attacker process. Processor-based side and covert channels can be generally categorized as timing-based, access-based, or trace-based channels. Timing-based channels rely on the timing of various operations to leak information, e.g., [4, 21, 122]. For example, one process performs many memory accesses so that memory accesses of another process are slowed down. Access-based channels rely on accessing some information directly, e.g., [83, 161, 165, 174, 253]. For example, one process probes the cache state by timing its own memory accesses. Trace-based channels rely on measuring exact execution of a program, e.g., [1]. For example, attacker obtains sequence of memory accesses and whether they are cache hits or misses based on the power measurements. Besides processor-based side and covert channels, others exist, such as through power [121], EM radiation [6], thermal [106], etc.

Both covert and side channels can be created as a result of the logical design of the system, or due to the implementation of the system. Timing channels are often due to design, e.g., caches have different timing for a cache hit vs. cache miss, thus observation of the timing of memory access can reveal some information about state of the system. Timing channels can often be fixed or at least mitigated by changing the design, e.g., disable caches or use partitioned or randomized caches. Implementation-related channels can be, for example, EM channels, where the logical design does not leak any information (e.g., no timing cache channel exists), but EM radiation can reveal some information (e.g., memory address used to perform a memory access). Implementation-related channels, similar to hardware trojans, cannot often be easily defended at the architecture level. Meanwhile, timing channels or other due to the logical design of the system can be defended by changing the system design.

2.2.5 INFORMATION FLOWS AND ATTACK BANDWIDTHS

The information flow refers to the transfer of information between different entities. Information flow can be explicit, e.g., $a = b$ where data or information in b is moved to a, or it can be implicit, e.g., $b = 0$; *if* (a) *then* $b = 1$, where the value of b reflects whether a is true, but there was never a direct assignment, or copying of data, from a to b. Typically when discussing information flow there is a low-security entity that interacts with a high-security entity. A system could have a desired property such as "there is no information flow between the components x and y" or "component x's file z is never accessible by component y." Information flow can happen through data or through timing information.

If there is a possible information flow between the system and an attacker, then there is a potential attack vector. This has been formalized in security verification by number of tools which can try to check whether there are information flows between different entities in the system [52]. Side and covert channels can also be expressed in terms of information flow between an attacker and a victim.

When there is information flow, one needs to also consider bandwidth and the probabilistic nature of the information flow. Bandwidth simply means how much data can be transferred in a unit of time. Naturally, higher bandwidth information flows are more dangerous from security perspective. Although, in many cases to goal of an attacker is to exfiltrate a cryptographic key, e.g., 128–4096 bits, and the actual bandwidth does not need to be very high to get useful information (i.e., the key). The other aspect is the probabilistic nature of the information flow. As an example, in a cache side-channel attack, there is interaction between attacker process and victim process, who both affect the state of the cache, which leads to the side-channel attack. However, the information flow depends on the behavior of the attacker and the victim. If victim and attacker always access mutually exclusive cache sets, there is no information leak; if they access or contend for the same cache set, there can be information leak. Thus, there is not always an actual information flow, even it it is possible to have one.

Whether there is an information flow between a potential attacker and victim can be analyzed using non-interference. Non-interference is a property that typically refers to how untrusted entities interact with trusted entities. The interference between these entities can be analyzed to check that the untrusted entity is not able to observe any differences in its own behavior, or the behavior of the system, in response to, or as a byproduct of, a trusted entity processing sensitive or non-sensitive inputs. The trusted entity, however, may observe differences in the behavior of system or the untrusted entity. Non-interference essentially means that information about the operation of the secure entity does not leak to the insecure entity.

2.2.6 THE THREAT MODEL

A threat model is a concise specification of the threats that a given secure processor architecture protects against. It is unlikely that an architecture can be designed to protect against all possible hardware and software attacks. Also, it may be economically infeasible to try to provide

protections against certain, unlikely attacks. At the very least, a threat model should specify the assumptions and threats that the architecture considers. It should specify the following.

1. TCB: the set of trusted hardware and software components.
2. Security properties: properties that the TCB aims to guarantee.
3. Attacker assumptions: capabilities of potential attackers.
4. Potential vulnerabilities: attackers and attack vectors on the trusted computing base, including untrusted entities and also any external attackers which will be defended against.

As part of the threat model, it is beneficial to also list any groups of potential attackers, attackers' capabilities, or attack vectors which are out-of-the-scope of the protections the trusted computing base of the secure architecture aims to ensure. This can clarify what is not being protected by the design.

Different secure processor architectures will provide protections under assumptions of different possible attacks, thus direct comparison of the architectures is often difficult. For example, some may protect from physical attacks on DRAM memory but not side channels, but others may protect against timing side channels, but not against physical attacks.

When there is an attack, since each architecture is designed for a specific threat model, it is important to distinguish whether the attack is indeed within the scope of the threat model. Sometimes the problem with the architecture is actually with the threat model. For example, recent Intel SGX architecture does not aim to protect against side-channel attacks due to caches [104], while there are many researchers presenting such attacks against SGX [75]. Architects need to consider needs and expectations of the users and make sure the threat model matches what is assumed by the users.

2.2.7 THREATS TO HARDWARE AFTER THE DESIGN PHASE

Secure processor design focuses on minimizing the TCB, and protecting against a variety of attacks. All the protections depend on the trustworthiness of the TCB, and at the design time the designer should attempt to verify the TCB [52]. When the hardware and software is actually manufactured, however, there are a number of threats that may still undermine the design:

Bugs or Vulnerabilities in the TCB—By definition, the TCB is fully trusted and assumed to be bug and vulnerability free. This is usually not explicitly stated, but should still be kept in mind while discussing security of any given architecture. Software (and hardware) design security verification [52] are themselves large research areas that can inform assumptions about the dangers of bugs and vulnerabilities in the TCB, and how to avoid them.

Hardware Trojans and Supply Chain Attacks—With the growing globalization of the supply chain, a single processor may include intellectual property (IP) blocks from multiple vendors from different countries, the whole system may be manufactured and processed in many different facilities in many countries before the final product is delivered to the customers. All

the different parties who supply IP blocks or are part of the manufacturing and supply chain can potentially alter or modify the design to insert hardware trojans. When discussing the secure processor architecture design, it is usually implied that as long as the design is correct, the actual processor will be properly manufactured according to the design and not altered. Hardware trojan detection and prevention [231] as well as supply chain issues [178] are themselves large research topics that are well studied, and they can inform the assumptions about the threats of hardware trojans or possible supply chain attacks on the TCB.

Physical Probing and Invasive Attacks—Once a processor is manufactured and used in a real device, the device can be potentially easily probed through physical means. Some architectures may assume no physical attacks (e.g., processor is used in a system that is located in a secure facility) or they may assume limited physical attacks (e.g., the memory chip can be probed as it is separate from the main processor, but the processor itself cannot be probed). Typical physical attacks may involve reading out data from device via standard interfaces (e.g., remove DRAM chip, place in another computer, and read out contents of the DRAM memory chip [86]). They may also involve invasive attacks, such as decapsulating the package of the processor or memory to get access to the circuits on the chip. Such invasive attacks may use etching or drilling and use an optical microscope and a small metal probes to inspect the circuits. More sophisticated attacks can use focused ion beam (FIB) for probing of deep metal and polysilicon lines on the chip, or they can even be used to modify of the chip structure by adding interconnect wires or even creating new transistors [92]. Different secure processor architectures typically aim to protect against a subset of these attacks, e.g., assume DRAM memory can be removed and probed. There is a large body of ongoing research relating to physical attacks [213] that can be used to contextualize the assumptions about physical probing and attacks.

2.3 BASIC SECURITY CONCEPTS

Analyzing and designing a secure processor architecture involves deciding about what properties the system will provide for the protected software, within the limitations of the threat model. The basic properties are: confidentiality, integrity, and availability. Furthermore, authentication mechanisms are important to consider. As part of integrity checking and also of authentication, understanding freshness and nonces is a crucial aspect. Finally, it is important to distinguish security from reliability, and keep in mind that security assumes reliability is already in place.

2.3.1 CONFIDENTIALITY, INTEGRITY, AND AVAILABILITY

One of most basic objectives of any computer security system, whether hardware or software, is to protect code and data stored or executing on the system. There are three well-known security properties of the code or data with regard to which the code or data can be protected: confidentiality, integrity, and availability. These properties are defined by [128] as follows.

- "Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities."
- "Integrity is the prevention of unauthorized modification of protected information without detection."
- "Availability is the provision of services and systems to legitimate users when requested or needed."

Confidentiality of code or data can be ensured if there is no action, or set of actions, that an untrusted entity can make to directly read, or otherwise deduce, contents of the confidential code or data. In terms of information flow, confidentiality can be modeled as existence of channels for information flow from the trusted entities to the untrusted entities. Such channels could be modeled as noisy channels or lossy channels if the untrusted entity only gets partial information about the confidential code or data. If the attacker is able to find a confidentiality breach, then the attacker can learn some or all information about sensitive code or data that the architecture aimed to protect, and the architecture in question needs to be redesigned and fixed.

It should be stressed that even partial information lean can be dangerous. Often attackers will have access to some external information or an ability to brute-force and make educated guesses once they have some partial information. The external information can be public information known to everybody, or some information attacker has obtained (e.g., through a different attack). This can help deduce information (e.g., combine known memory layout of a program with timing information for the cache). The ability to brute-force and check all possibilities, especially when guessing cryptographic keys, means that attackers do not actually need to get all the 128 bits for 128-bit AES, for example, they may get 96 or even 64 bits, and try to guess the rest in matter of days on a powerful computer. Finding an attack that requires no specific outside information, or no brute-forcing to work, is much more powerful and damaging than finding a very specific attack that only works in certain scenarios with lots of external information or requires brute-forcing some information.

Integrity of code and data can be ensured if there is no action, or set of actions, which allow untrusted entity to modify the protected code or data. Integrity attacks do not require an attacker to learn any information, but only the ability to modify something. Integrity attacks focus on code or data related to authentication or integrity checks, so as to allow the attacker to bypass these checks, and breach the system.

Integrity attacks can vary in the amount of modification the attacker attempts to do. On one end of the spectrum, an attacker may want to modify just one bit of information, e.g., enable/disable protections bit, that will allow him or her to later damage the system through a different attack. On the other end of the spectrum, he or she may try to modify whole memory contents to rewrite some code or data in the system.

Availability of the code and data can be ensured if there is no way for an attacker to deny service to the users of the system. Availability almost never can be achieved through use of one

single secure processor design, e.g., the attacker can always smash the processor with a hammer to destroy it and thus deny it access to anybody. In less drastic approaches, attackers can slowly use up memory or other resources of the system, making it impossible for the protected code and data to execute in reasonable time. Availability can be achieved by using many secure processor working together, such as through redundancy.

2.3.2 AUTHENTICATION

Authentication relates to determining who a user or system is [128]. One approach to implementing authentication is for the parts of a system, or for a user and a system, to exchange information something that each knows, such as a password. This is usually referred to as proving "what you know," other approaches use proving "what you have" or "what you are" [128]. Authentication inherently requires integrity, as an attackers should neither be able to modify the authentication information nor make up their own.

2.3.3 FRESHNESS AND NONCES

When dealing with integrity or authentication, it is not only important that the information is correct but that it is "fresh." A good example of need for freshness are replay attacks where some previously correct and good data or information is re-sent at a future time when the information is no longer up to date. To ensure freshness, nonces are used. A nonce is a "number used once" and is a common way to ensure freshness in cryptographic protocols. Often, there is not a trusted, global clock that a components of a system can reference to find out if some event has happened before or after some time. An alternative is to use some indicator, the nonce, which can be referenced. As each value of the nonce is used once during a lifetime of the system, once a number has been used, it can be remembered. A practical way to implement nonces is to use a monotonic counter, thus only one, latest, value needs to be stored securely for reference by each component. An alternative could be to use random numbers as nonces (e.g., sender sends a random n and receiver replies with $n + 1$). When using random numbers as nonces, there is danger of two random number repeating themselves. Consequently, it may be best to use monotonic counters as nonces when it is possible to store the state (i.e., store the last nonce value."

2.3.4 SECURITY VS. RELIABILITY

From a security perspective, confidentiality, integrity, and availability assume a sophisticated attacker who attempts to maximize their chance of breaking the system. Security assumes that reliability, i.e., protection from random faults or errors, is already provided by the system, and focuses instead on the deliberate attacks by a smart adversary. Thus, reliability is about random errors, e.g., cosmic rays striking DRAM and causing a fault, while security is about deliberate attacks, e.g., attacker modifying exactly the memory location storing the secret key. When considering system availability, if the reliability is not maintained (e.g., the system shuts down), it

still should not allow disclosure of any information to a potential attacker, nor allow information to be modified.

2.4 SYMMETRIC-KEY CRYPTOGRAPHY

To ensure data confidentiality, symmetric- or private-key cryptography is needed. In symmetric-key cryptography, data encryption and decryption uses the same secret key. When protecting data, there is the plaintext p which is encrypted into a resulting ciphertext c by the encryption function which also uses some private key k. Thus, the encryption process is: $c = Enc(k, p)$. To get back the plaintext data, decryption is needed: $p = Dec(k, c)$. Note, both encryption and decryption use the same key in symmetric-key cryptography. It is required that the encrypted ciphertext looks almost random to someone who does not posses the key k. Given ciphertext, an attacker should not be able to learn neither the key nor the plaintext data. Symmetric-key algorithms can be broken down into block ciphers and stream ciphers.

2.4.1 SYMMETRIC-KEY ALGORITHMS: BLOCK CIPHERS

Block ciphers work on blocks of data, e.g., AES uses a 16-byte block. Plaintext to be encrypted has to be a multiple of the block size. Data smaller than the block needs to be padded to the block size, while data bigger than the block size is encrypted in block-sized chunks. For data bigger than one block size, there are different modes of operation of the block cipher that determine how the blocks are encrypted [128]. Electronic Code Book (ECB) mode simply encrypts one block at a time. The danger of ECB is that encryption of same data blocks will result in same ciphertext blocks when using same keys, which can reveal patterns about the input plaintext data. Other modes include Cipher Block Chaining (CBC) or Counter Mode (CTR). Here, the same input data blocks will results in different ciphertext blocks, even when using same key.

Encryption only provides confidentiality, but integrity is often also desired. For integrity protection, there are dedicated modes of operation [128], e.g., Hash-based Message Authentication Code (HMAC), Cipher-based Message Authentication Code (CMAC), or Galois Message Authentication Code (GMAC). In these modes, the last ciphertext block is effectively a secure hash (fingerprint) of the whole data, and its value depends on the encryption key.

To avoid having to separately do encryption and hashing, there are authenticated encryption modes which combine the two operations into one. One of the most recent recommended modes is the Galois/Counter Mode (GCM) [148]. Given a key, it can encrypt data and generate a keyed-hash value of the data.

Decryption works similar to encryption, where one has to work with block-sized chunks of ciphertext. Some modes of operation, e.g., CTR, allow for easy decryption of random blocks inside the ciphertext. With CTR, each block is xored with an encryption of a counter, knowing which counter corresponds to which block, a random block can be decrypted by encrypting the counter value and xoring it with the ciphertext block. Meanwhile, others may require to decrypt

multiple blocks together or a whole ciphertext (due to the chaining, in a serial fashion, among the individual blocks).

2.4.2 SYMMETRIC-KEY ALGORITHMS: STREAM CIPHERS

A different approach to symmetric-key encryption is taken by stream ciphers. Here, the plaintext is encrypted bit by bit by combining it (using xor operation) with a pseudorandom keystream. The keystream is typically generated from a random seed using shift registers, and the seed is the cryptographic key needed for decryption. Some modes of operation of block ciphers behave as stream ciphers, but still the main distinction is that they operate on blocks of data, while stream ciphers operate on bits.

Stream ciphers can be faster in hardware than block ciphers. One disadvantage is that is hard to provide ability to do random data access inside the encrypted ciphertext as one has to re-generate the pseudorandom keystream up to the point where the to-be-access data is located in the ciphertext stream.

2.4.3 STANDARD SYMMETRIC: KEY ALGORITHMS

While a number of symmetric-key algorithms exist, designers should use the AES [173] which is well studied and considered secure. There are also some "lightweight" block ciphers, such as PRESENT [27]. Older algorithms such as RC4, DES, or 3DES should no longer be used as their key sizes are too small or they are considered insecure due to cryptographic attacks. Moreover, custom-designed algorithms should be avoided. There are numerous examples of security-by-obscurity where custom, publicly untested algorithms have been deployed in products such as metro cards, only to be found to have serious flaws [44]. Among stream ciphers, there are Salsa29 and its variant ChaCha [22].

2.5 PUBLIC-KEY CRYPTOGRAPHY

In public-key cryptography, also called asymmetric-key cryptography, data encryption and decryption uses different keys. For confidentiality protection, the input is a plaintext p which is encrypted into a resulting ciphertext c by the encryption function which uses the public key pk. Thus the encryption process is: $c = Enc(pk, p)$. To get back the plaintext data, decryption is needed: $p = Dec(sk, c)$. Here, the decryption uses the secret key sk. Given a pk it should be infeasible to find out what is the secret key sk, which depends on hardness of certain mathematical problems, such as factoring of large numbers, e.g., RSA [175]. The advantage of public-key cryptography is that pk can be given to anybody, and they can encrypt the data or code using this pk. Meanwhile, only the user, program, or hardware module in possession of the sk can decrypt the data.

For integrity, public-key cryptography can be used in "reverse" direction when used in digital signatures or message authentication codes. The sk can be used to create a digital signa-

ture, and anybody with access to the pk can verify the signature—but cannot make a new valid signature as they do not have sk nor can they get the sk from knowing pk.

2.5.1 KEY ENCAPSULATION MECHANISMS

Key Encapsulation Mechanisms (KEMs) are methods for securely transmitting a symmetric encryption key using public-key encryption. Public-key encryption is typically significantly slower and more costly in terms of computations. As result, in most applications which require encryption of a lot of data in a public-key setting, the encryption key is secured and transferred using the (relatively slow) public-key encryption, while the actual data is later encrypted (relatively fast) using that symmetric key. Upon receiving a message, first the public-key algorithm is used to decrypt the key, then the symmetric-key algorithm is used to decrypt the actual data.

2.5.2 STANDARD PUBLIC-KEY ALGORITHMS

The most well-known public-key algorithm is the RSA algorithm [175], which derives its security from hardness of problem of factoring large numbers. More recently, Elliptic Curve Cryptography (ECC) [87] has gained popularity as well, which is based on the algebraic structure of elliptic curves over finite fields.

2.5.3 POST-QUANTUM CRYPTOGRAPHY

Most recently, there is active interest in the so-called Post-Quantum Cryptographic (PQC) algorithms, as the promise of practical quantum computers nears [37]. In the 1990s, Shor proposed algorithms that can solve both the integer-factorization problem and the discrete-logarithm problem in polynomial time on a quantum computer [197, 198]. Cryptosystems based on the hardness assumptions of the integer-factorization problem and the discrete-logarithm problem can be broken using Shor's algorithm and a quantum computer. For example, public-key algorithms such as RSA may not be secure for long-term use and other algorithms need to be standardized and used in their place.

As of 2018, there are five categories of mathematical problems that are under investigation as candidates for PQC: code-based systems, lattice-based systems, hash-based systems, systems based on multivariate polynomial equations, and systems based on supersingular isogenies of elliptic curves [23, 179].

In addition, Grover's algorithm [78] gives a square-root speedup on brute-force attacks that check every possible key. This does not break symmetric-key algorithms, such as AES, but does necessitate the use of larger keys. For example, a 256-bit pre-quantum security level corresponds to 128-bit post-quantum security level.

2.6 RANDOM NUMBER GENERATION

Most security and cryptographic algorithms and protocols depend on good sources of random numbers, especially for key generation. There are pseudo-random number generators (PRNGs), which use an algorithm to expand a seed into a long string of random-looking numbers. The numbers are not truly random, as given knowledge of the seed and the algorithm; anybody can re-generate the same sequence of random-looking numbers. There are also true random number generators (TRNGs), which generate truly random numbers. For example, physical phenomenon like electrical noise or temperature variations can be used as sources of randomness. True random numbers are hard to obtain at high rate, thus many times TRNGs generate a small true random number, the seed, and a PRNG is used to expand that seed into a long string of random numbers. As long as the seed is truly random, and never accessible to potential attackers, then the resulting PRNG output can be used in secure manner. Computer architects often assume existence of sufficient randomness, and hardware and circuit designers are ones focusing on how to create such circuits. Designers should however realize that TRNGs can be manipulated, such as by inserting backdoors into processor's random number generator [166].

2.7 SECURE HASHING

To ensure data integrity, secure hashes are needed. A secure hash algorithm is a cryptographic hash function. A secure hash maps input data, m, of variable size to a fixed size output, h—the output is simply called the hash of the input data. Secure hash is a one-way function, and it should be infeasible to mathematically invert it and deduce the input data given the hash value. Furthermore, there are three properties that strong cryptographic hashes should have. They are as follows.

- Pre-image resistance—given a hash value h it should be infeasible to find any message m such that $h = hash(m)$.
- Second pre-image resistance—given a specific input m_1 it should be infeasible to find different input m_2 such that $hash(m_1) = hash(m_2)$.
- Collision resistance—it should be difficult to find two random messages m_3 and m_4, where $m_3 \neq m_4$, such that $hash(m_3) = hash(m_4)$; due to the birthday paradox [128] it is possible to find two such random messages that hash to the same value much more readily than one may expected.

Hash functions are used to compute the hash value, sometimes called digest or fingerprint, of some input data. Given the hash size is fixed, and much smaller than the data size in most cases, it is easier to store and protect the hash value, rather than the original data. Given same input m, the hash will always be the same h, and anybody with access to m can compute h.

Common application of hashes is in authentication and integrity checking. For example, a hash can be computed for a large file, then the file can be sent to an untrusted storage while the

hash is kept in a safe location. Later, when the large file is read again, its hash can be recomputed and checked against the stored value to make sure there was no modification to the file (note this does not protect against replay attacks, or if someone is able to change the hash value stored in the secure location). When checking integrity or authentication, freshness needs to be ensured; often a nonce is included as part of the hash (i.e., hash data concatenated with the nonce).

2.7.1 USE OF HASHES IN MESSAGE AUTHENTICATION CODES

When using Message Authentication Codes (MACs) only the entity with the correct cryptographic key can generate or check the hash value. Unlike secure hashes where anybody can generate the hash value, MACs have added requirement that only select entities who have the correct cryptographic key can do so. The advantage of MACs over plain secure hashes is that the hash value cannot be generated by anybody, just the entity that has the key. MACs can be sent over untrusted communication channels along with the data. The attacker may try to change the data, but he or she cannot generate a new MAC that matches the data as long as he or she does not have the cryptographic key. To check the integrity, data can be hashed again by the receiver, and then hash compared with the decrypted hash from the MAC. MAC can be realized by using keyed-hash message authentication codes, block ciphers, or based on universal hashing. Both sender and receiver need to share the same key when using MACs.

2.7.2 USE OF HASHES IN DIGITAL SIGNATURES

Digital signatures are similar to MACs, in that only the user or program with the correct cryptographic key can generate or check the hash value. Unlike MACs, digital signature use a private signing key to generate the signatures, and a public verification key to check the signatures. Akin to public-key cryptography, the parties generating and verifying the digital signatures have different keys. Digital signatures can leverage public-key cryptographic algorithms, e.g., on the sender's end securely hash a message then encrypt the hash with the private key which allows the verifier to re-generate the hash, on the receiver's end decrypt the received value using the public key, and check if the two match. The public key infrastructure can be leveraged to distribute certificates so that sender and receiver need not to have direct contact in order to be able to authenticate the messages that were digitally signed.

2.7.3 USE OF HASHES IN HASH TREES

A hash tree, also called a Merkle tree [150] and shown in Figure 2.2, is usually a binary tree data structure. In the hash tree, a parent node contains hash value of its children nodes. At the very bottom of the tree are the leaf nodes. Thus, as one proceeds up the tree, integrity of the lower tree nodes can be checked by computing their hash value and comparing with the hash stored in the parent nodes; finally the root node contains the hash value which is dependent on the values of all the intermediate nodes, and thus the leaf nodes.

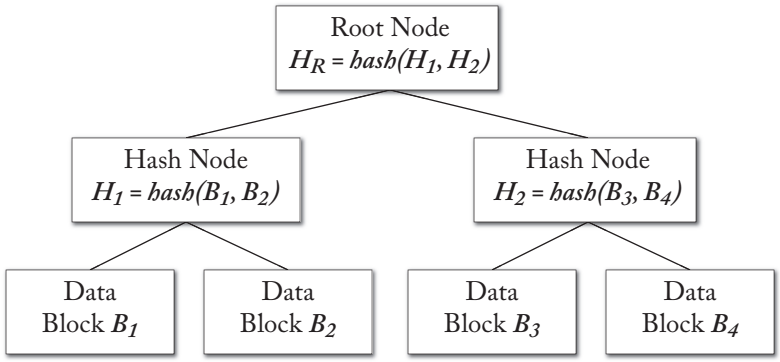

Figure 2.2: Example of a hash tree with four data nodes, showing internal hash nodes, and the root hash. The value of the root hash depends on all the data nodes' values.

The hash tree was invented to speed up the verification of integrity of large sets of data. Assuming leaf nodes correspond to parts of a file, or parts of the computer memory, if there is a change in part of the file, or part of the memory, one only needs to check the hash value of the parent nodes of the leaf nodes that were changed—and not compute the hash over the whole file or memory.

2.7.4 APPLICATION OF HASHES IN KEY DERIVATION FUNCTION

Often, a system uses many symmetric keys, or needs to generate a symmetric key in a specific format. Key Derivation Function (KDF), is used to derive one or more secret keys from a master secret key. Keyed cryptographic hash functions are one example of pseudo-random functions that can be used for key derivation [32].

2.7.5 STANDARD SECURE HASH ALGORITHMS

There exist numerous secure hash algorithms. Most common, and most recommended, ones today are the SHA-2 [202] and SHA-3 [57] algorithms. Use of older algorithms such as MD4, MD5, or SHA-1 show be avoided as they are no longer secure as a commodity computer can be used to perform a brute-force search to find a message that gives the desired hash value. A hash digest size of 256 bits is likely a good choice today as it makes it impossible launch a brute-force attack, even with a quantum computer (due to Grover's algorithm, pre-quantum security level of 256 is equivalent to post-quantum 128-bit security level).

2.8 PUBLIC KEY INFRASTRUCTURE

Public key infrastructure (PKI) is a set of policies and protocols for managing, storing, and distributing certificates used in public-key encryption [14]. In PKI, there is a trusted third party

which can distribute digital certificates that vouch for correctness of public keys *pk* of different entities, and allows for verification and decryption of public-key encrypted data without having to directly talk to each sender to get their key.

The trusted third party is the certificate authority. The authority is responsible for verification of the certificates it receivers. It then distributes the certificates to other users, signed with its own private keys. Certificates for the certificate authorities are usually pre-distributed (e.g., browsers come with built-in list of certificates for certificate authorities). If there is a problem with a certificate authority, then the PKI infrastructure will break. For example, a user can by mistake trust a certificate authority they should not. Or, a malicious certificate authority could equivocate and give different information to different users, enabling man-in-the-middle type attacks, for example.

2.8.1 DIGITAL CERTIFICATES

Digital certificates are a central part of the PKI. In a simplified form, a digital certificate contains some identifying information about a system or a user and their public key. This information is encrypted with a private key of the certificate authority. Users are given known-good public keys of the certificate authorities, which they can use to check authenticity of the certificates. Once authenticity of certificate is verified (i.e., the digital signature of the certificate was indeed made by the trusted authority) then the users can be sure that the public key in the certificate belongs to the system or user that the certificate is for. With this information, users can verify messages sent by the entity that has the private key corresponding to the public key in the certificate.

2.8.2 DIFFIE–HELLMAN KEY EXCHANGE

Diffie–Hellman (DH) is an algorithm used to establish a shared secret between two entities. Its main application is as a way of exchanging cryptography keys for use in symmetric-key algorithms. Symmetric-key encryption and decryption is much more efficient than public-key encryption. Thus, often it is desired to use public-key protected means of communication to generate a shared, symmetric secret key for actual transfer of data between two entities. Diffie-Hellman requires use of authenticated channel, i.e., the two communicating parties need to authenticate each other, otherwise there is a man-in-the-middle attack.

2.8.3 APPLICATION OF PKI IN SECURE PROCESSOR ARCHITECTURES

As will be seen in later chapters, a PKI is not directly used by secure processor hardware. However, for users of such processors, a PKI is needed to distribute certificates vouching for the cryptographic secrets embedded in these secure processors (such as a private key *pk* of the processor). Each secure processor typically needs at least one private key (either burned-in in some registers, stored on non-volatile memory, or derived from a PUF). To ensure users that they are indeed communicating with the expected processor, the manufacturer typically needs a PKI to distribute certificates vouching for the public keys assigned to each processor.

Having a publicly known signing key may be a privacy issue, allowing adversaries to de-anonymize communication if the see the same key being used over and over again. This issue has been addressed by work such as on Direct Anonymous Attestation (DAA), which is a cryptographic primitive used to enable remote authentication of a trusted computer while preserving privacy of the platform's user [29].

2.9 PHYSICALLY UNCLONABLE FUNCTIONS

Each secure processor should be uniquely identified and have a unique set of cryptographic keys. This can be achieve by "burning in" the unique information at the factory. However, such an approach requires extra cost (since each chip has to be written with the unique information), and a potentially malicious manufacturer may keep information about all the secret keys they have burned into their products. As an alternative, researchers have recently proposed PUFs [143], and they could be used to generate unique information per-chip.

PUFs leverage the unique behavior of a device, due to manufacturing variations, as a hardware-based fingerprint. A PUF instance is extremely difficult to replicate, even by the manufacturer. Many uses of PUFs have been presented in literature: authentication and identification [119, 206, 224], hardware-software binding [81, 82, 123, 186, 187], remote attestation [125, 189], and secret key storage [225, 226]. PUFs can also be leveraged for random number generation [144] as well as for recently proposed virtual proofs of reality [182]. PUF implementation is mostly domain of hardware and circuit designers, but architects can leverage them in their security architectures. At the architecture level, PUFs are most often abstracted away as modules that give a unique, unclonable fingerprint of the hardware.