17 Security and Privacy

Security and **privacy** are two of the foremost design concerns for cyber-physical systems today. Security, broadly speaking, is the state of being protected from harm. Privacy is the state of being kept away from observation. With embedded and cyber-physical systems being increasingly networked with each other and with the Internet, security and privacy concerns are now front and center for system designers.

In a formal sense, there are two primary aspects that differentiate security and privacy from other design criteria. First, the operating environment is considered to be significantly more adversarial in nature than in typical system design. Second, the kinds of properties, specifying desired and undesired behavior, are also different from traditional system specifications (and often impose additional requirements on top of the traditional ones). Let us consider each of these aspects in turn.

The notion of an **attacker** or an **adversary** is central to the theory and practice of security and privacy. An attacker is a malicious agent whose goal is to subvert the operation of the system in some way. The exact subversion depends on characteristics of the system, its goals and requirements, and the capabilities of the attacker. Typically, these characteristics are grouped together into an entity called the **threat model** or **attacker model**. For example, while designing an automobile with no wireless network connectivity, the designers may assume that the only threats arise from people who have physical access to the automobile and knowledge of its components, such as a well-trained mechanic. Transforming an informal threat model (such as the preceding sentence) into a precise, mathematical statement of an attacker's objectives is a challenging task, but one that is essential for principled model-based design of secure embedded systems.

The second defining characteristic of the field of security and privacy are the distinctive properties it is concerned with. Broadly speaking, these properties can be categorized into the following types: confidentiality, integrity, authenticity, and availability. **Confidentiality** is the state of being secret from the attacker. A good example of confidential data is a password or PIN that one uses to access one's bank account. **Integrity** is the state of being unmodified by an attacker. An example of integrity is the property that an attacker, who does not have authorized access to your bank account, cannot modify its contents. **Authenticity** is the state of knowing with a level of assurance the identity of agents you are communicating or interacting with. For instance, when you connect to a website that purports to be your bank, you want to be sure that it indeed is your bank's website and not that of some malicious entity. The process of demonstrating authenticity is known as **authentication**. Finally, **availability** is the property that a system provides a sufficient quality of service to its users. For example, you might expect your bank's website to be available to you 99% of the time.

It is important to note that security and privacy are *not* absolute properties. Do not believe anyone who claims that their system is "completely secure"! Security and privacy can only be guaranteed for a specific threat model and a specific set of properties. As a system designer, if security and privacy are important concerns for you, you must first

define your threat model and formalize your properties. Otherwise, any solution you adopt is essentially meaningless.

In this chapter, we seek to give you a basic understanding of security and privacy with a special focus on concepts relevant for cyber-physical system design. The field of security and privacy is too broad for us to cover it in any comprehensive manner in a single chapter; we refer the interested reader to some excellent textbooks on the topic, e.g., (Goodrich and Tamassia, 2011; Smith and Marchesini, 2007). Instead, our goal is more modest: to introduce you to the important basic concepts, and highlight the topics that are specific to, or especially relevant for, embedded, cyber-physical systems.

17.1 Cryptographic Primitives

The field of cryptography is one of the cornerstones of security and privacy. The word "cryptography" is derived from the Latin roots "crypt" meaning *secret* and "graphia" meaning *write*, and thus literally means "the study of secret writing."

We begin with a review of the basic **cryptographic primitives** for encryption and decryption, secure hashing, and authentication. Issues that are particularly relevant for the design and analysis of embedded and cyber-physical systems are highlighted. The reader is warned that examples given in this chapter, particularly those listing code, are given at a high level of abstraction, and omit details that are critical to developing secure cryptographic implementations. See books such as (Menezes et al., 1996; Ferguson et al., 2010) for a more in-depth treatment of the subject.

17.1.1 Encryption and Decryption

Encryption is the process of translating a message into an encoded form with the intent that an adversary cannot recover the former from the latter. The original message is typically referred to as the **plaintext**, and its encoded form as the **ciphertext**. **Decryption** is the process of recovering the plaintext from the ciphertext.

The typical approach to encryption relies on the presence of a secret called the **key**. An encryption algorithm uses the key and the plaintext in a prescribed manner to obtain the ciphertext. The key is shared between the parties that intend to exchange a message securely. Depending on the mode of sharing, there are two broad categories of cryptography.

In **symmetric-key cryptography**, the key is a single entity that is known to both sender and receiver. In **public-key cryptography**, also termed **asymmetric cryptography**, the key is split into two portions, a public part and a private part, where the **public key** is known to everyone (including the adversary) and the **private key** is known only to the receiver. In the rest of this section, we present a brief introduction to these two approaches to cryptography.

One of the fundamental tenets of cryptography, known as **Kerckhoff's principle**, states that a cryptographic system (algorithm) should be secure even if everything about the system, except the key, is public knowledge. In practical terms, this means that even if the adversary knows all details about the design and implementation of a cryptographic algorithm, as long as he does not know the key, he should be unable to recover the plaintext from the ciphertext.

Symmetric-Key Cryptography

Let Alice and Bob be two parties that seek to communicate with each other securely. In symmetric-key cryptography, they accomplish this using a shared secret key K. Suppose Alice wishes to encrypt and send Bob an n-bit plaintext message, $M = m_1 m_2 m_3 \dots m_n \in \{0,1\}^n$. We wish to have an encryption scheme that, given the shared key K, should encode M into ciphertext C with the following two properites. First, the intended recipient Bob should be able to easily recover M from C. Second, any adversary who does not know K should not, by observing C, be able to gain any more information about M.

We present intuition for the operation of symmetric-key cryptography using a simple, idealized scheme known as the **one-time pad**. In this scheme, the two parties Alice and Bob share an n-bit secret key $K = k_1 k_2 k_3 \dots k_n \in \{0,1\}^n$, where the n bits are chosen independently at random. K is known as the one-time pad. Given K, encryption works by taking the bit-wise XOR of M and K; i.e., $C = M \oplus K$ where $\oplus$ denotes XOR. Alice then sends C over to Bob, who decrypts C by taking the bit-wise XOR of C with K; using properties of the XOR operation, $C \oplus K = M$.

Suppose an adversary Eve observes C. We claim that Eve has no more information about M or K than she had without C. To see this, fix a plaintext message M. Then, every unique ciphertext $C \in \{0,1\}^n$ can be obtained from M with a corresponding unique choice of key K — simply set $K = C \oplus M$ where C is the desired ciphertext. Put another way, a uniformly random bit-string $K \in \{0,1\}^n$ generates a uniformly random ciphertext

$C \in \{0,1\}^n$. Thus, looking at this ciphertext, Eve can do no better than guessing at the value of K uniformly at random.

Notice that Alice cannot use the key K more than once! Consider what happens if she does so twice. Then Eve has access to two ciphertexts $C_1 = M_1 \oplus K$ and $C_2 = M_2 \oplus K$. If Eve computes $C_1 \oplus C_2$, using properties of XOR, she learns $M_1 \oplus M_2$. Thus, Eve receives partial information about the messages. In particular, if Eve happens to learn one of the messages, she learns the other one, and can also recover the key K. Thus, this scheme is not secure if the same key is used for multiple communications, hence the name "one-time pad." Fortunately, other, stronger symmetric key cryptography schemes exist.

Most common symmetric key encryption methods use a building block termed a **block cipher**. A **block cipher** is an encryption algorithm based on using a k-bit key K to convert an n-bit plaintext message M into an n-bit ciphertext C. It can be mathematically captured in terms of an encryption function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$; i.e., $E(K, M) = C$. For a fixed key K, the function E_K defined by $E_K(M) = E(K, M)$ must be a permutation from $\{0,1\}^n$ to $\{0,1\}^n$. Decryption is the inverse function of encryption; note that the inverse exists for each K as E_K is invertible. We denote the decryption function corresponding to E_K by $D_K = E_K^{-1}$. Note that for simplicity this model of encryption abstracts it to be (only) a function of the message and the key; in practice, care must be taken to use a suitable block cipher mode that, for example, does not always encrypt the same plaintext to the same ciphertext.

One of the classic block ciphers is the **data encryption standard** (**DES**). Introduced in the mid-1970s, DES was the first block cipher based on modern cryptographic techniques, considered to be "commercial-grade," with an open specification. While the details of DES are beyond the scope of this chapter, we note that versions of DES are still used in certain embedded devices. For instance, a version of DES, known as **3DES**, which involves applying the DES algorithm to a block three times, is used in certain "chip and PIN" payment cards around in the world. The basic version of the DES involves use of a 56-bit key and operates on 64-bit message blocks. While DES initially provided an acceptable level of security, by the mid-1990s, it was getting easier to break it using "brute-force" methods. Therefore, in 2001, the U.S. National Institute of Standards and Technology (NIST) established a new cryptographic standard known as the **advanced encryption standard** or **AES**. This standard is based on a cryptographic scheme called *Rijndael* proposed by Joan Daemen and Vincent Rijmen, two researchers from Belgium. AES uses a message block length of 128 bits and three different key lengths of 128, 192, and 256 bits. Since its adoption, AES has proved to be a strong cryptographic algorithm amenable to efficient implementation both in hardware and in software. It is estimated

that the current fastest supercomputers could not successfully mount a brute-force attack on AES within the estimated lifetime of the solar system!

Public-Key Cryptography

In order to use symmetric-key cryptography, Alice and Bob need to have already set up a shared key in advance. This is not always easy to arrange. Public-key cryptography is designed to address this limitation.

In a public-key cryptosystem, each principal (Alice, Bob, etc.) has two keys: a *public key*, known to everyone, and a *private key*, known only to that principal. When Alice wants to send Bob a secret message, she obtains his public key K_B and encrypts her message with it. When Bob receives the message, he decrypts it with his private key k_B. In other words, the encryption function based on K_B must be invertible using the decryption function based on k_B, and moreover, must not be invertible without k_B.

Let us consider the last point. Encryption using a public key must effectively be a **one-way function**: a publicly-known function F such that it is easy to compute $F(M)$ but (virtually) impossible to invert without knowledge of a suitable secret (the private key). Remarkably, there is more than one public-key cryptographic scheme available today, each based on a clever combination of mathematics, algorithm design, and implementation tricks. We focus our attention here on one such scheme proposed in 1978 (Rivest et al.), known as **RSA** for the initials of its creators: Rivest, Shamir and Adleman. For brevity, we mostly limit our discussion to the basic mathematical concepts behind RSA.

Consider the setting where Bob wishes to create his public-private key pair so that others can send him secret messages. The RSA scheme involves three main steps, as detailed below:

1. *Key Generation:* Select two large prime numbers p and q, and compute their product $n = pq$. Then compute $\varphi(n)$ where φ denotes **Euler's totient function**. This function maps an integer n to the number of positive integers less than or equal to n that are relatively prime to n. For the special case of the product of primes, $\varphi(n) = (p-1)(q-1)$. Select a random integer e such that $1 < e < \varphi(n)$ such that the greatest common divisor of e and $\varphi(n)$, denoted $GCD(e, \varphi(n))$ equals 1. Then compute d, $1 < d < \varphi(n)$, such that $ed \equiv 1 \pmod{\varphi(n)}$. The key generation process is now complete. Bob's public key K_B is the pair (n, e). The private key is $k_B = d$.

2. *Encryption:* When Alice wishes to send a message to Bob, she first obtains his public key $K_B = (n, e)$. Given the message M to transmit, she computes the ciphertext C as $C = M^e \pmod{n}$. C is then transmitted to Bob.

3. *Decryption:* Upon receipt of C, Bob computes $C^d \pmod{n}$. By properties of Euler's totient function and modular arithmetic, this quantity equals M, allowing Bob to recover the plaintext message from Alice.

We make two observations on the above scheme. First, the operations in the various steps of RSA make heavy use of non-linear arithmetic on large numbers, especially modular multiplication of large integers and modular exponentiation. These operations must be implemented carefully, especially on embedded platforms, in order to operate efficiently. For instance, modular exponentiation is typically implemented using a version of the square-and-multiply technique outlined in Example 16.1 of Chapter 16. Second, the effectiveness of the approach depends critically on infrastructure to store and maintain public keys. Without such a **public-key infrastructure**, an attacker Eve can masquerade as Bob, advertising her own public key as Bob's, and thus fooling Alice into sending her messages intended for Bob. While such a public-key infrastructure has now been established for servers on the Web using so-called "certificate authorities," it is more challenging to extend that approach for networked embedded systems for a variety of reasons: the large scale of devices that may act as servers, the ad-hoc nature of networks, and the resource constraints on many embedded devices.[1] For this reason, public-key cryptography is not as widely used in embedded devices as is symmetric-key cryptography.

17.1.2 Digital Signatures and Secure Hash Functions

The cryptographic primitives for encryption and decryption help in providing confidentiality of data. However, by themselves, they are not designed to provide integrity or authenticity guarantees. Those properties require the use of related, but distinct, primitives: digital signatures, secure hash functions, and message authentication codes (MACs). This section provides a brief overview of these primitives.

[1] *Let's Encrypt*, an ongoing effort to develop a free, automated, and open certificate authority might mitigate some of these challenges; see `https://letsencrypt.org/` for more details.

Secure Hash Functions

A **secure hash function** (also known as *cryptographic hash function*) is a deterministic and keyless function $H : \{0,1\}^n \rightarrow \{0,1\}^k$ that maps n-bit messages to k-bit hash values. Typically, n can be arbitrarily large (i.e., the message can be arbitrarily long) while k is a relatively small fixed value. For example, SHA-256 is a secure hash function that

Implementation Matters for Cryptography

Embedded platforms typically have limited resources such as memory or energy, and must obey physical constraints, for instance on real-time behavior. Cryptography can be computationally quite demanding. Such computations cannot simply be offloaded "to the cloud" as secret data must be secured at the end point before being sent out on the network. Thus, cryptographic primitives must be implemented in a manner that obeys the constraints imposed by embedded platforms.

Consider, for example, public-key cryptography. These primitives involve modular exponentiation and multiplications that can have large timing, memory, and energy requirements. Consequently, researchers have developed efficient algorithms as well as software and hardware implementations. A good example is **Montgomery multiplication** (Montgomery, 1985), a clever way of performing modular multiplication by replacing the division (implicit in computing the modulus) with addition. Another promising technology is **elliptic curve cryptography** which allows for a level of security comparable with the RSA approach with keys of smaller bit-width, reducing the memory footprint of implementations; see, for example, Paar and Pelzl (2009) for details. Due to the costs of public-key cryptography, it is often used to exchange a shared symmetric key that is then used for all subsequent communication.

Another important aspect of implementing cryptographic schemes is the design and use of **random number generation** (**RNG**). A high-quality RNG requires as input a source of randomness with high entropy. Many recommended high entropy sources involve physical processes, such atmospheric noise or atomic decay, but these may not be easy to use in embedded systems, especially in hardware implementations. Alternatively, one can use on-chip thermal noise and the timing of input events from the physical world and network events. Polling the physical world for many such sources of randomness may consume power, leading to implementation trade-offs; see Perrig et al. (2002) for an example.

maps messages to a 256-bit hash, and is being adopted for authenticating some software packages and for hashing passwords in some Linux distributions. For a message M, we term $H(M)$ as the "hash of M."

A secure hash function H has certain important properties that distinguishes it from more traditional hash functions used for non-security purposes:

- *Efficient to compute:* Given a message M, it should be efficient to compute $H(M)$.
- *Pre-image resistance:* Given a hash (value) h, it should be computationally infeasible to find a message M such that $h = H(M)$.
- *Second pre-image resistance:* Given a message M_1, it should be computationally infeasible to find another message M_2 such that $H(M_1) = H(M_2)$. This property prevents attackers from taking a known message M_1 and modifying it to match a desired hash value (and hence find the corresponding message M_2).
- *Collision resistance:* It should be computationally infeasible to find two different messages M_1 and M_2 where $H(M_1) = H(M_2)$. This property, a stronger version of second pre-image resistance, prevents an attacker from taking *any* starting message and modifying it to match a given hash value.

A common use of secure hash functions is to verify the integrity of a message or a piece of data. For example, before you install a software update on your computer you might want to verify that the update you downloaded is a valid copy of the package you are updating and not something modified by an attacker. Providing a secure hash of the software update on a separate channel than the one on which the update file is distributed can provide such assurance. After separately downloading this hash value, you can verify, by computing the hash yourself on the software, that the provided hash indeed matches the one you compute. With the growing trend for embedded systems to be networked and to receive software patches over the network, the use of secure hash functions is expected to be a central component of any solution that maintains system integrity.

Digital Signatures

A **digital signature** is a cryptographic mechanism, based on public-key cryptography, for the author of a digital document (message) to authenticate himself/herself to a third party and to provide assurance about the integrity of the document. The use of digital signatures involves the following three stages:

1. *Key generation:* This step is identical to the key generation phase of public-key cryptosystems, resulting in a public key-private key pair.

2. *Signing:* In this step, the author/sender digitally signs the document to be sent to the receiving party.

3. *Verification:* The recipient of a signed document verifies, using the digital signature, that the sender of the document is indeed who he/she purports to be.

We now illustrate how the basic scheme works for the RSA cryptosystem. Suppose Bob wishes to digitally sign a message M before sending it to Alice. Recall from Section 17.1.1 that, using RSA, Bob can generate his public key as $K_B = (n, e)$ and private key $k_B = d$. One simple authentication scheme is for Bob to use his key-pair in reverse: to sign M, he simply encrypts it with his private key, computing $S = M^d \pmod{n}$, and sends S to Alice along with M. When Alice receives this signed message, she uses K_B to recover M from the signature S and verifies it by comparing it with the received message. If they match, the message has been authenticated; otherwise, Alice has detected tampering either in the message or its signature.

The above scheme is simple, but flawed. In particular, given the signatures S_1 and S_2 for two messages M_1 and M_2, notice that an attacker can construct the signature for message $M_1 \cdot M_2$ by simply multiplying together S_1 and $S_2 \pmod{n}$. In order to guard against this attack, one can first compute a secure hash of M_1 and M_2 before computing the signatures on the resultant hashes rather than on the messages.

Message Authentication Codes

A **message authentication code** (**MAC**) is a cryptographic mechanism, based on symmetric-key cryptography, for providing integrity and authenticity to a document. It is thus the symmetric-key analog of a digital signature. As in other symmetric-key schemes, the use of a MAC requires first setting up a shared key between sender and receiver that each of them can use to authenticate messages sent by the other. For this reason, MACs are best suited to settings where such a shared key can be easily set up. For instance, modern automobiles comprise multiple electronic control units (ECUs) communicating with each other on an on-board network such as a **controller-area network** (**CAN**) bus. In this case, if each ECU is pre-programmed with a common key, then each ECU can authenticate messages sent from other ECUs on the CAN bus using MACs.

17.2 Protocol and Network Security

An attacker typically gains access to a system via its connection to a network or some other medium used to connect it with other components. Additionally, it is increasingly the case that a single embedded system comprises many distributed components connected over a network. Thus, one must address fundamental questions about security over a network and using various communication protocols, a field termed **protocol security** or **network security**. In this section, we review basic ideas related to two topics of particular relevance for embedded systems: *key exchange* and *cryptographic protocol design*.

17.2.1 Key Exchange

We have already seen how secret keys are critical components of both symmetric and asymmetric cryptosystems. How are these keys established in the first place? In a symmetric cryptosystem, we need a mechanism for two communicating parties to agree on a single shared key. In an asymmetric cryptosystem, we need infrastructure for establishing and maintaining public keys so that any party on a network can look up the public key of any other party it wishes to communicate with. In this section, we discuss a classic method for **key exchange** and some alternative schemes customized to specific embedded system design problems, with a focus on symmetric cryptography.

Diffie-Hellman Key Exchange

In 1976, Whitfield Diffie and Martin Hellman introduced what is widely considered the first public-key cryptosystem (Diffie and Hellman, 1976). The crux of their proposal was a clever scheme for two parties to agree on a shared secret key using a communication medium observable by anyone. This scheme is commonly referred to as **Diffie-Hellman** key exchange.

Suppose there are two parties Alice and Bob that wish to agree on a secret key. Everything Alice sends to Bob, and vice-versa, can be viewed by an attacker Eve. Alice and Bob wish to agree on a key while ensuring that it is computationally infeasible for Eve to compute that key by observing their communication. Here is how Alice and Bob can use Diffie-Hellman key exchange to achieve this objective.

To begin with, Alice and Bob need to agree on two parameters. In doing so, they can use various methods including a hard-coded scheme (e.g., a parameter hard-coded into the same program they use) or one of them can announce the parameters to the other in some deterministic fashion (e.g., based on a fixed ordering between their network addresses). The first parameter is a very large prime number p. The second is a number z such that $1 < z < p - 1$. Note that an attacker can observe z and p.

After these parameters p and z are agreed upon, Alice randomly selects a number a from the set $\{0, 1, \ldots, p - 2\}$ and keeps it secret. Similarly, Bob randomly selects a number $b \in \{0, 1, \ldots, p - 2\}$ and keeps it secret. Alice computes the quantity $A = z^a \pmod p$ and sends it to Bob. Likewise, Bob computes $B = z^b \pmod p$ and sends it to Alice. In addition to z and p, the attacker Eve can observe A and B, but not a and b.

On receiving B, Alice uses her secret number a to compute $B^a \pmod p$. Bob performs the analogous step, computing $A^b \pmod p$. Now, observe that

$$A^b \pmod p = z^{ab} \pmod p = B^a \pmod p$$

Thus, amazingly, Alice and Bob have agreed on a shared key $K = z^{ab} \pmod p$ simply by communicating on a public channel. Note that they have revealed neither of the secret numbers a or b to the attacker Eve. Without knowledge of a or b, Eve cannot reliably compute K. Moreover, it is computationally very difficult for Eve to compute a or b simply by knowing p, z, A or B, since the underlying problem is one known as the **discrete logarithm** problem, for which no efficient (polynomial-time) algorithm is known. Put another way, the function $f(x) = z^x \pmod p$ is a one-way function.

Diffie-Hellman is simple, elegant, and effective: unfortunately, it is typically not practical for resource-constrained embedded systems. Computing it involves modular exponentiation for large primes (typical key sizes range to 2048-bits), and thus impractical for energy-constrained platforms that must obey real-time constraints. We therefore discuss next a few alternative schemes.

Timed Release of Keys

Many networked embedded systems use a **broadcast** medium for communication — one where senders send data over a public channel where all receivers, intended and unintended, can read that data. Examples include on-board automotive networks using the CAN bus and wireless sensor networks. A broadcast medium has many advantages: it is simple, requires little infrastructure, and a sender can quickly reach a large number of

receivers. However, it also has certain disadvantages, notably that malicious parties can eavesdrop and inject malicious packets with ease. Reliability of the broadcast medium can also be concern.

How can one achieve secure key exchange in such a broadcast medium, under constraints on timing and energy consumption? This question was studied in the early 2000s, when the first work was done on deploying wireless sensor networks (e.g., see Perrig et al. (2004); Perrig and Tygar (2012)). One of the novel ideas is to leverage *timing properties* of these networks, an idea whose roots go back to a paper by Anderson et al. (1998).

The first property is that of **clock synchronization**, where different nodes on the network have clocks such that the difference between the values of any two clocks is bounded by a small constant. Many sensor nodes can have synchronized clocks via GPS or protocols such as the **precision time protocol** (**PTP**) (Eidson, 2006).

The second property is that of *scheduled transmission*. Each node in the network transmits packets at pre-determined times, following a schedule.

The combination of the above two properties permits the use of a secure broadcast protocol called μTESLA (Perrig et al., 2002). This protocol is designed for broadcast networks comprising two types of nodes, *base stations* and *sensor nodes*. A key property of μTESLA is *secure authentication* — a node receiving a message should be able to determine who the authentic sender is, discarding spoofed messages. It performs this by using a message authentication code (MAC) in each message, which is computed using a secret key. This secret key is broadcast some time after the original message was sent, allowing the receiver to compute the MAC for itself, and determine the authenticity of the received message. Since each message is timestamped, upon receiving a message the receiver verifies that the key used to compute the MAC has not yet been disclosed based on this timestamp, its local clock, the maximum clock skew, and the time schedule at which keys are disclosed. If the check fails, the message is discarded; otherwise, when the key is received, the authenticity of the message is checked by computing the MAC and checking against the received MAC. In this way, the *timed release* of keys is used to share keys between a broadcast sender and its receivers, for the purpose of authenticating messages.

A similar approach has been adopted for CAN-based automotive networks. Lin et al. (2013) have shown how to extend the μTESLA approach for the CAN bus where the nodes do not necessarily share a common global notion of time. Timed delayed release

of keys has also been implemented for Time Triggered Ethernet (TTE) (Wasicek et al., 2011).

Other Schemes

In certain application domains, customized schemes have been developed for key exchange using special characteristics of those domains. For instance, Halperin et al. (2008) discuss a key exchange scheme for **implantable medical devices** (**IMD**s) such as pacemakers and defibrillators. They propose a "zero-power" security mechanism on the implanted device based on radio-frequency (RF) power harvesting. The novel key exchange scheme involves a "programmer" device initiating communication with the IMD (and supplying an RF signal to power it). The IMD generates a random value that is communicated as a weak modulated sound wave that is perceptible by the patient. The security

Reverse Engineering Systems to Improve Security

Several embedded systems were designed at a time when security was a secondary design concern, if one at all. In recent years, researchers have shown how this lack of attention to security can lead to compromised systems in various domains. The chief approach taken is to reverse engineer protocols that provide access to systems and the working of their components.

We present here three representative examples of vulnerabilities demonstrated in real embedded systems. Halperin et al. (2008) show how implantable medical devices (IMDs) can be read and reprogrammed wirelessly. Koscher et al. (2010) demonstrate for automobiles how the on-board diagnostics bus (OBD-II) protocol can be subverted to adversarially control a wide range of functions, including disabling the brakes and stopping the engine. Ghena et al. (2014) study traffic light intersections and show how their functioning can be partially controlled wirelessly to perform, for example, a denial-of-service attack causing traffic congestion.

Such investigations based on reverse engineering "legacy" embedded systems to understand their functioning and uncover potential security flaws form an important part of the process of securing our cyber-physical infrastructure.

of this scheme relies on a threat model based on physical security around the patient's immediate vicinity, one that is plausible for this particular application.

17.2.2 Cryptographic Protocol Design

Some of the most-publicized vulnerabilities in networked embedded systems, at the time of writing, arise from the fact that the communication protocols were designed with little or no security in mind (see the box on page 472). However, even protocols designed for security using cryptographic primitives can have vulnerabilities. Poor protocol design can compromise confidentiality, integrity, and authenticity properties even if one assumes perfect cryptography and the presence of key exchange and key management infrastructure. We explain this point with an example.

Example 17.1: A **replay attack** on a protocol is one where an attacker is able to violate a security property by replaying certain messages in the protocol, possibly with modifications to accompanying data.

We describe here a replay attack on a fictitious wireless sensor network protocol. This network comprises a base station S and several energy-constrained sensor nodes $N_1, N_2, \ldots, N_k$ communicating via broadcast. Each node spends most of its time in a sleep mode, to conserve energy. When S wants to communicate with a particular node N_i to take a sensor reading, it sends it a message M encrypted with a suitable shared key K_i pre-arranged between S and N_i. N_i then responds with an encrypted sensor reading R. In other words, the protocol has a message exchange as follows:

$$\begin{aligned} S &\rightarrow N_i : E(K_i, M) \\ N_i &\rightarrow S : E(K_i, R) \end{aligned}$$

This protocol, on the face of it, seems secure. However, it has a flaw that can hinder operation of the network. Since the nodes use broadcast communication, an attacker can easily record the message $E(K_i, M)$. The attacker can then replay this message multiple times at a low power so that it is detected by N_i, but not by S. N_i will keep responding with encrypted sensor readings, possibly causing it to run down its battery much faster than anticipated, and disabling its operation.

Notice that there was nothing wrong with the individual cryptographic steps used in this protocol. The problem was with the temporal behavior of the system —

that N_i had no way of checking if a message is "fresh." A simple countermeasure to this attack is to attach a fresh random value, called a **nonce**, or a timestamp with each message.

This attack is also an example of a so-called **denial-of-service attack**, abbreviated **DoS attack**, indicating that the impact of the attack is a loss of service provided by the sensor node.

Fortunately, techniques and tools exist to formally model cryptographic protocols and verify their correctness with respect to specific properties and threat models before deploying them. More widespread use of these techniques is needed to improve protocol security.

17.3 Software Security

The field of **software security** is concerned with how errors in the software implementation can impact desired security and privacy properties of the system. Bugs in software can and have been used to steal data, crash a system, and worse, allow an attacker to obtain an arbitrary level of control over a system. Vulnerabilities in software implementations are one of the largest categories of security problems in practice. These vulnerabilities can be especially dangerous for embedded systems. Much of embedded software is programmed in low-level languages such as C, where the programmer can write arbitrary code with few language-level checks. Moreover, the software often runs "bare metal" without an underlying operating system or other software layer that can monitor and trap illegal accesses or operations. Finally, we note that in embedded systems, even "crashing" the system can have severe consequences, since it can completely impair the interaction of the system with the physical world. Consider, for example, a medical device such as a pacemaker, where the consequences of a crash may be that the device stops functioning (e.g. pacing), leading to potentially life-threatening consequences for the patient.

In this section, we give the reader a very brief introduction to software security problems using a few illustrative examples. The reader is referred to relevant books on security (e.g., Smith and Marchesini (2007)) for further details.

Our examples illustrate a vulnerability known as a **buffer overflow** or **buffer overrun**. This class of errors is particularly common in low-level languages such as C. It arises from

the absence of automatic bounds-checking for accessing arrays or pointers in C programs. More precisely, a buffer overflow is a error arising from the absence of a bounds check resulting in the program writing past the end of an array or memory region. Attackers can use this out-of-bounds write to corrupt trusted locations in a program such as the value of a secret variable or the return address of a function. (Tip: material in Chapter 9 on Memory Architectures might be worth reviewing.) Let us begin with a simple example.

Example 17.2: The sensors in certain embedded systems use communication protocols where data from various on-board sensors is read as a stream of bytes from a designated port or network socket. The code example below illustrates one such scenario. Here, the programmer expects to read at most 16 bytes of sensor data, storing them into the array `sensor_data`.

```
1  char sensor_data[16];
2  int secret_key;
3
4  void read_sensor_data() {
5    int i = 0;
6
7    // more_data returns 1 if there is more data,
8    // and 0 otherwise
9    while(more_data()) {
10     sensor_data[i] = get_next_byte();
11     i++;
12   }
13
14   return;
15 }
```

The problem with this code is that it implicitly trusts the sensor stream to be no more than 16 bytes long. Suppose an attacker has control of that stream, either through physical access to the sensors or over the network. Then an attacker can provide more than 16 bytes and cause the program to write past the end of the array `sensor_data`. Notice further how the variable `secret_key` is defined right after `sensor_data`, and assume that the compiler allocates them adjacently. In this case, an attacker can exploit the buffer oveflow vulnerability to provide a stream of length 20 bytes and overwrite `secret_key` with a key of his choosing. This exploit can then be used to compromise the system in other ways.

The example above involves an out-of-bounds write in an array stored in global memory. Consider next the case when the array is stored on the stack. In this case, a buffer overrun vulnerability can be exploited to overwrite the return address of the function, and cause it to execute some code of the attacker's choosing. This can lead to the attacker gaining an arbitrary level of control over the embedded system.

Example 17.3: Consider below a variant on the code in Example 17.2 where the `sensor_data` array is stored on the stack. As before, the function reads a stream of bytes and stores them into `sensor_data`. However, in this case, the read sensor data is then processed within this function and used to set certain globally-stored flags (which can then be used to take control decisions).

```
1  int sensor_flags[4];
2
3  void process_sensor_data() {
4    int i = 0;
5    char sensor_data[16];
6
7    // more_data returns 1 if there is more data,
8    // and 0 otherwise
9    while(more_data()) {
10     sensor_data[i] = get_next_byte();
11     i++;
12   }
13
14   // some code here that sets sensor_flags
15   // based on the values in sensor_data
16
17   return;
18 }
```

Recall from Chapter 9 how the stack frame of a function is laid out. It is possible for the return address of function `process_sensor_data` to be stored in memory right after the end of the array `sensor_data`. Thus, an attacker can exploit the buffer overflow vulnerability to overwrite the return address and cause execution to jump to a location of his choice. This version of the buffer overflow exploit is sometimes (and rather aptly) termed as **stack smashing**.

Moreover, the attacker could write a longer sequence of bytes to memory and include arbitrary code in that sequence. The overwritten return address could be

tailored to be *within* this overwritten memory region, thus causing the attacker to control the code that gets executed! This attack is often referred to as a **code injection attack**.

How do we avoid buffer overflow attacks? One easy way is to explicitly check that we never write past the end of the buffer, by keeping track of its length. Another approach is to use higher-level languages that enforce **memory safety** — preventing the program from reading or writing to locations in memory that it does not have privileges for or that the programmer did not intend it to. Exercise 1 gives you some practice with writing code so as to avoid a buffer overflow.

17.4 Information Flow

Many security properties specify restrictions on the flow of information between principals. Confidentiality properties restrict the flow of secret data to channels that are readable by an attacker. For example, your bank balance should be viewable only by you and authorized bank personnel and not by an attacker. Similarly, integrity properties restrict the flow of untrusted values, controlled by an attacker, to trusted channels or locations. Your bank balance should be writable only by trusted deposit or withdrawal operations and not arbitrarily by a malicious party. Thus, **secure information flow**, the systematic study of how the flow of information in a system affects its security and privacy, is a central topic in the literature.

Given a component of an embedded system it is important to understand how information flows from secret locations to attacker-readable "public" locations, or from attacker-controlled untrusted channels to trusted locations. We need techniques to detect illegal information flows and to quantify their impact on the overall security of the system. Additionally, given security and privacy policies, we need techniques to enforce those policies on a system by suitably restricting the flow of information. Our goal, in this section, is to equip the reader with the basic principles of secure information flow so as to be able to develop such techniques.

17.4.1 Examples

In order to motivate the secure information flow problem, we present a series of examples. Although these examples focus on confidentiality, the same approaches and concerns apply for integrity properties as well.

Example 17.4: Medical devices are increasingly software-controlled, personalized, and networked. An example of such a device is a blood glucose meter, used by doctors and patients to monitor a patient's blood glucose level. Consider a hypothetical glucose meter that can take a reading of a patient's glucose level, show it on the device's display, and also transmit it over the network to the patient's hospital. Here is a highly-abstracted version of the software that might perform these tasks.

```
1  int patient_id; // initialized to the
2                  // patient's unique identifier
3  void take_reading() {
4    float reading = read_from_sensor();
5
6    display(reading);
7
8    send(network_socket, hospital_server,
9         reading, patient_id);
10
11   return;
12 }
```

The function `take_reading` records a single blood glucose reading (e.g., in mg/dL) and stores it in a floating-point variable `reading`. It then writes this value, suitably formatted, to the device's display. Finally, it transmits the value along with the patient's ID without any encryption over the network to the hospital server for analysis by the patient's doctor.

Suppose that the patient wishes to have his glucose level kept private, in the sense that its value is known only to him and his doctor, but no one else. Does this program achieve that objective? It is easy to see that it does not, as the value of `reading` is transmitted "in the clear" over the network. We can formalize this violation of privacy as an illegal information flow from `reading` to `network_socket`, where the former is a secret location whereas the latter is a public channel visible to an attacker.

With the knowledge of cryptographic primitives presented earlier in this chapter, we know that we can do better. In particular, let us assume the presence of a shared key between the patient's device and the hospital server that permits the use of symmetric-key encryption. Consider the following new version of the above example.

Example 17.5: In this version of the program, we employ symmetric-key cryptography, denoted by `AES`, to protect the reading. The encryption process is abstractly represented by the function `enc_AES`, and `send_enc` differs from `send` only in the type of its third argument.

```
1  int patient_id; // initialized to the
2                  // patient's unique identifier
3  long cipher_text;
4
5  struct secret_key_s {
6    long key_part1; long key_part2;
7  }; // struct type storing 128-bit AES key
8
9  struct secret_key_s secret_key; // shared key
10
11 void take_reading() {
12   float reading = read_from_sensor();
13
14   display(reading);
15
16   enc_AES(&secret_key, reading, &cipher_text);
17
18   send_enc(network_socket, hospital_server,
19            cipher_text, patient_id);
20
21   return;
22 }
```

In this case, there is still a flow of information from `reading` to `network_socket`. In fact, there is also a flow from `secret_key` to `network_socket`. However, both these flows passed through an encryption function denoted by `enc_AES`. In the security literature, such a function is termed a **declassifier** since it encodes the secret data in such a way that it becomes acceptable to transmit the encoded, "declassified" result to an attacker-readable channel. In other words, the declassifier serves as a "dam" blocking the flow of secret information and releasing only information that, by the properties of the cryptographic

primitives, provides attackers with no more information than they had before the release.

While the above fix to the code blocks the flow of information about the reading of the patient's blood glucose, note that it does not completely protect the program. In particular, notice that the patient's ID is still being sent in the clear over the network! Thus, even though the attacker cannot tell, without the secret key, what the patient's reading is, he/she knows who transmitted a reading to the hospital (and possibly also at what time). Some private data is still being leaked. One solution to this leak is fairly obvious – we can encrypt both variables `reading` and `patient_id` using the secret key before sending them over the network.

So far we have assumed an implicit threat model where the attacker can only read information sent over the network. However, for many embedded systems, attackers may also have physical access to the device. For instance, consider what happens if the patient loses his device and an unauthorized person (attacker) gains access to it. If the patient's readings are stored on the device, the attacker might be able to read the data stored on it if such an illegal information flow exists. One approach to guard against this attack is to have the patient create a password, much like one creates a password for other personal devices, and have the device prompt one for the password when turned on. This idea is discussed in the example below.

Example 17.6: Suppose that a log of the past 100 readings is stored on the device. The code below sketches an implementation of a function `show_readings` that prompts the user for an integer password, and displays the stored readings only if the current password stored in `patient_pwd` is entered, otherwise displaying an error message that has no information about the stored readings.

```
1 int patient_id; // initialized to the
2                 // patient's unique identifier
3 int patient_pwd; // stored patient password
4
5 float stored_readings[100];
6
7 void show_readings() {
```

```
8     int input_pwd = read_input(); // prompt user for
9                                  // password and read it
10    if (input_pwd == patient_pwd) // check password
11      display(&stored_readings);
12    else
13      display_error_mesg();
14
15    return;
16  }
```

Assuming the attacker does not know the value of `patient_pwd`, we can see that the above code does not leak any of the 100 stored readings. However, it does leak one bit of information: whether the input password provided by the attacker equals the correct password or not. In practice, such a leak is deemed acceptable since, for a strong choice of patient password, it would take even the most determined attacker an exponential number of attempts to log in successfully, and this can be thwarted by disallowing more than a small, fixed number of attempts to provide the correct password.

The above example illustrates the concept of **quantitative information flow** (QIF). QIF is typically defined using a function measuring the amount of information flowing from a secret location to a public location. For instance, one might be interested in computing the number of bits leaked, as illustrated by the example above. If this quantity is deemed small enough, the information leak is tolerated, otherwise, the program must be rewritten.

However, the code in Example 17.6 has a potential flaw. Once again, this flaw stems from a corresponding threat model. In particular, consider the case where an attacker not only has physical possession of the device, but also has the knowledge and resources to perform *invasive* attacks by opening up the device, and reading out the contents of main memory as it executes. In this case, the attacker can read the value of `patient_pwd` simply by booting up the device and reading out the contents of memory as the system is initialized.

How can we protect against such invasive attacks? One approach involves the use of secure hardware or firmware coupled with secure hash functions. Suppose the hardware provides for secure storage where a secure hash of the user password can be written out, and any tampering detected. Then, only the secure hash need be stored in main memory, not the actual password. By the properties of a secure hash function, it would be compu-

tationally infeasible for an attacker to reverse engineer the password simply by knowing the hash.

A similar flaw exists in Example 17.5, where the secret key is stored in memory in the clear. However, in this case, simply taking a secure hash of the key is insufficient as the actual key is required to perform encryption and decryption to facilitate secure communication with the server. How do we secure this function? In general, this requires an additional component such as a secure key manager, e.g., implemented in hardware or by a trusted operating system, one that uses a master key to encrypt or decrypt the secret key for the application.

Finally, although our examples illustrate the flow of information through values of variables in software, information can also be leaked through other channels. The term **side channel** is used to refer to a channel that involves a non-traditional way of communicating information, typically using a physical quantity such as time, power consumption, or the amplitude and frequency of an audio signal, or a physical modification to the system, such as a fault, that induces a traditional information leak. We cover some basic concepts on side-channel attacks in Section 17.5.2.

17.4.2 Theory

So far we have spoken of confidentiality and integrity properties only informally. How can we state these properties precisely, similar to the notation introduced in Chapter 13? It turns out that specifying security and privacy properties formally can be quite tricky. Various formalisms have been proposed in the literature over the years, and there is no consensus on a common definition of notions such as confidentiality or integrity. However, some foundational concepts and principles have emerged. We introduce these basic concepts in this section.

The first key concept is that of **non-interference**. The term **non-interference** is used to refer to any of a family of (security) properties that specify how actions taken by one or more principals can or cannot affect ("interfere with") actions taken by others. For instance, for integrity properties, we require that actions of an attacker cannot affect the values of certain trusted data or computations. Similarly, for confidentiality properties, we require that the actions taken by an attacker cannot depend on secret values (thus implying that the attacker has no information about those secrets).

There are several variants of non-interference that are useful for expressing different kinds of security or privacy properties. In defining these, it is customary to use the terms *high* and *low* to denote two security levels. For confidentiality, the high level denotes secret data/channels and the low level denotes public data/channels. For integrity, the high level denotes trusted data/channels while the low level denotes untrusted data/channels. Non-interference is typically defined over traces of a system, where each input, output, and state element is classified as being either low or high.

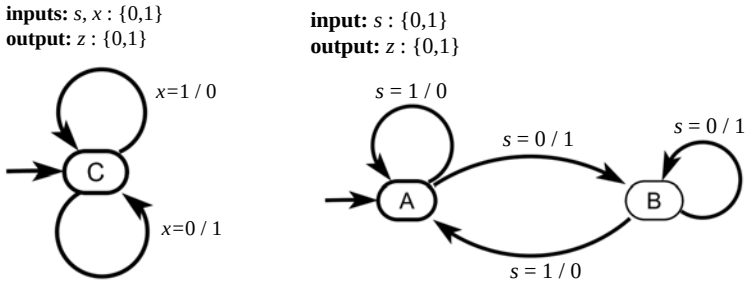
The first variant we introduce is **observational determinism** (McLean, 1992; Roscoe, 1995) which is defined as follows: if two traces of a system are initialized to be in states in which the low elements are identical, and they receive the same low inputs, then that implies that the low elements of *all* states and outputs in that trace must be identical. Put another way, the low parts of a system's trace are deterministic functions of the low initial state and the low inputs, and *nothing else*. This, in turn, implies that an attacker who only controls or observes the low parts of a system's trace cannot infer anything about the high parts.

Another variant is **generalized non-interference** (McLean, 1996) which requires a system to exhibit a set of traces with a special property: for any two traces τ_1 and τ_2, there must exist a newly-constructed trace τ_3 such that, at each time step in τ_3, the high inputs are the same as in τ_1 and the low inputs, states, and outputs are the same as in τ_2. Intuitively, by observing the low states and outputs, an attacker cannot tell if she is seeing the high inputs as in τ_1 or as they occur in τ_2.

The different variants of non-interference, however, share something in common. Mathematically, they are all **hyperproperties** (Clarkson and Schneider, 2010). Formally, a **hyperproperty** is a set of sets of traces. Contrast this definition with the definition of a property as a set of traces, as introduced in Chapter 13. The latter is more accurately called a **trace property**, as its truth value can be evaluated on a single, standalone trace of a system. The truth value of a hyperproperty, in contrast, typically[2] cannot be determined by observing a single trace. One needs to observe multiple traces, and compute a relation between them, in order to determine if they together satisfy or violate a hyperproperty.

Example 17.7: Consider the two state machines in Figure 17.1. In both cases, the state machine has one secret input s and one public output z. Additionally,

[2]Clearly, every (trace) property has an equivalent hyperproperty, which is why we use the adverb "typically."



(a) M_1 (b) M_2

Figure 17.1: An example of a state machine satisfying observational determinism (a) and one that does not (b). Input s is a secret (high) input, and output z is a public (low) output.

M_1 has a public input x. An attacker can directly read the value of x and z but not s.

State machine M_1 satisfies observational determinism (OD). For any input sequence, M_1 produces a binary output sequence that depends solely on the values of the public input x and not on s.

However, M_2 does not satisfy OD. For example, consider the input sequence 1010^ω. Then, the corresponding output sequence of M_2 will be 0101^ω. If the input sequence is switched to 0101^ω, then the output will correspondingly change to 1010^ω. The adversary can gain information about the input s simply by observing the output z.

17.4.3 Analysis and Enforcement

Several techniques have been developed to analyze and track the flow of information in software systems, and, in some cases, to enforce information flow policies either at design time or run time. We mention some of these methods below, with the caveat that this is a rapidly evolving area and our selection is meant to be representative, not exhaustive.

Taint analysis. In taint analysis, the flow of information is tracked using labels ("taint") attached to data items and by monitoring memory operations. When an illegal information flow is detected, a warning is generated. Taint analysis has been mainly developed for software systems. It can be performed statically, when software is compiled, or dynamically, at run time. For instance, for the code in Example 17.4, taint analysis can detect the flow of secret data from `reading` to `network_socket`. At run time, this can be detected before the write to the network socket, and used to raise a run-time exception or other preventive action. Static taint analysis is a simple concept, and easy to apply, but can generate false alarms. Dynamic taint analysis is just as easy and more precise, but imposes a run-time overhead. System designers must evaluate for themselves the suitability of either mechanism for their contexts.

Formal verification of information flow. Techniques for formal verification, such as the model checking methods covered in Chapter 15, can also be adapted to verify secure information flow. These methods usually operate after the original analysis problem has been reduced to one of checking a safety property on a suitably constructed model (Terauchi and Aiken, 2005). Compared to taint analysis, such methods are much more precise, but they also come at a higher computational cost. As these methods are still rapidly evolving, detailed coverage of these methods is beyond the scope of this chapter at the present time.

Run-time enforcement of policies. The notion of taint allows one to specify simple information flow policies, such as disallowing the flow of secret data to any public location at all times. However, some security and privacy policies are more complex. Consider, for example, a policy that a person's location is public some of the time (e.g., while they are at work or in a public space such as an airport), but is kept secret at selected other times (e.g., when they visit their doctor). Such a policy involves a time-varying secret tag on the variable storing the person's location. The problem of specifying, checking, and enforcing such expressive policies is an active area of research.

17.5 Advanced Topics

Certain problems in security and privacy gain special significance in the context of cyber-physical systems. In this section, we review two of these problems and highlight some of the key issues.

17.5.1 Sensor and Actuator Security

Sensors and actuators form the interface between the cyber and physical worlds. In many cyber-physical systems, these components are easily observed or controlled by an attacker. Detecting attacks on these components and securing them is therefore an important problem. Central to both efforts is to develop realistic threat models and mechanisms to address those threats. We review two representative recent efforts in the area of **sensor security**.

Recent work has focused on attacks on analog sensors, i.e., developing threat models for these attacks as well as countermeasures. The main mode of attack is to employ **electromagnetic interference** (EMI) to modify the sensed signal. Two recent projects have studied EMI attacks in different applications. Foo Kune et al. (2013) investigate EMI attacks at varying power and distances on implantable medical devices and consumer electronics. Shoukry et al. (2013) study the possibility of EMI attacks that spoof sensor values for certain types of automotive sensors. We present here some basic principles from both projects.

Threat Models

In the context of sensor security, EMI can be classified along multiple dimensions. First, such interference can be *invasive*, involving modification to the sensor components, or *non-invasive*, involving observation or remote injection of spurious values. Second, it can be *unintentional* (e.g., arising from lightning strikes or transformers) or *intentional* (injected by an attacker). Third, it can be *high-power*, potentially injecting faults into or disabling the sensor, or *low-power*, which simply injects false values or modifies sensor readings. These dimensions can be used to define informal threat models.

Foo Kune et al. (2013) describe a threat model for *intentional, low-power, non-invasive* attacks. This combination of characteristics is amongst the hardest to defend against, since the low-power and non-intrusive nature of the attacks makes it potentially hard to detect. The researchers design two kinds of EMI attacks. *Baseband attacks* inject signals within the same frequency band in which the generated sensor readings lie. They can be effective on sensors that filter signals outside the operating frequency band. *Amplitude-modulated attacks* start with a carrier signal in the same frequency band of the sensor and modulate it with an attack signal. They can match the resonant frequency of a sensor, thus amplifying the impact of even a low-power attack signal. They demonstrate how

these attacks can be performed on implantable cardiac devices, injecting forged signals in leads, inhibiting pacing, causing defibrillation, etc. from 1 to 2 meters away from the device.

The threat model considered by Shoukry et al. (2013) is for *intentional, non-invasive* attacks, with a focus on anti-lock braking systems (ABS) in automobiles. The magnetic wheel speed sensors are attacked by placing a *malicious actuator* in close proximity of the sensor (mountable from outside the vehicle) that modifies the magnetic field measured by the ABS sensor. One form of the attack, in which the actuator disrupts the magnetic field but is not precisely controllable by the attacker, can be "merely" disruptive. Its impact is similar to unintentional, high-power EMI. The trickier form of the attack is a *spoofing* attack, where the ABS system is deceived into measuring an incorrect but precise wheel speed for one or more of the wheels. This is achieved by implementing an active magnetic shield around the actuator. The reader is referred to the paper for additional details. The authors show how their attack can be mounted on real automotive sensors, and demonstrate, in simulation, how the ABS system can be tricked into making an incorrect braking decision in icy road conditions, causing the vehicle to slip off-road.

Countermeasures

Both of the projects mentioned above also discuss potential countermeasures to the attacks.

Foo Kune et al. (2013) consider a range of defenses, both hardware-based and software-based. Hardware or circuit-level approaches include shielding the sensor, filtering the input signals, and common mode noise rejection. However, these methods are reported to have limited effectiveness. Thus, software-based defenses are also introduced, including estimating the EMI level in the environment, adaptive filtering, cardiac probing to see if the sensed signals follow an expected pattern, reverting to safe defaults, and notifying the victim (patient) or physician about the possibility of an attack to allow them to take suitable actions to move away from the EMI source.

Shoukry et al 2015; 2016 take a somewhat different approach that is rooted in control theory and formal methods. The idea is to create a mathematical model of the system and of sensor attacks, and devise algorithms to identify subsets of sensors that are attacked, isolate them, and use the remaining (unattacked) sensors to perform state estimation and control.

17.5.2 Side-channel Attacks

Many embedded systems are accessible to an attacker not just over a network but also physically. These systems are therefore exposed to classes of attacks not possible in other settings. One such class are known as **side-channel attacks**, which involve illegal information flows through side channels. Since the seminal work by Kocher (1996), attacks have been demonstrated using several types of side channels, including timing (Brumley and Boneh, 2005; Tromer et al., 2010), power (Kocher et al., 1999), faults (Anderson and Kuhn, 1998; Boneh et al., 2001) memory access patterns (Islam et al., 2012), acoustic signals (Zhuang et al., 2009), and data remanence (Anderson and Kuhn, 1998; Halderman et al., 2009). **Timing attacks** involve the observation of the timing behavior of a system rather than the output values it generates. **Power attacks** involve the observation of power consumption; a particularly effective variant, known as **differential power analysis**, are based on a comparative analysis of power consumption for different executions. In **memory access pattern attacks**, observing addresses of memory locations accessed suffices to extract information about encrypted data. **Fault attacks** induce information leaks by modifying normal execution via fault injection.

While an exhaustive survey of side-channel attacks is outside the scope of this book, we illustrate, using the example below, how information can leak through a timing side channel.

Example 17.8: Consider the C implementation of modular exponentiation given as the `modexp` function of Example 16.1. In Figure 17.2 we show how the execution time of `modexp` varies as a function of the value of `exponent`. Recall how the exponent usually corresponds to the secret key in implementations of public-key cryptosystems such as RSA or Diffie-Hellman. Execution time is depicted in CPU cycles on the y axis, while the value of `exponent` ranges from 0 to 15. The measurements are made on a processor implementing the ARMv4 instruction set.

Notice that there are five clusters of measurements around 600, 700, 750, 850, and 925 cycles respectively. It turns out that each cluster corresponds to values of `exponent` that, when represented in binary, have the same number of bits set to 1. For example, the execution times for values 1, 2, 4 and 8, each corresponding to a single bit set to 1, are clustered around 700 CPU cycles.

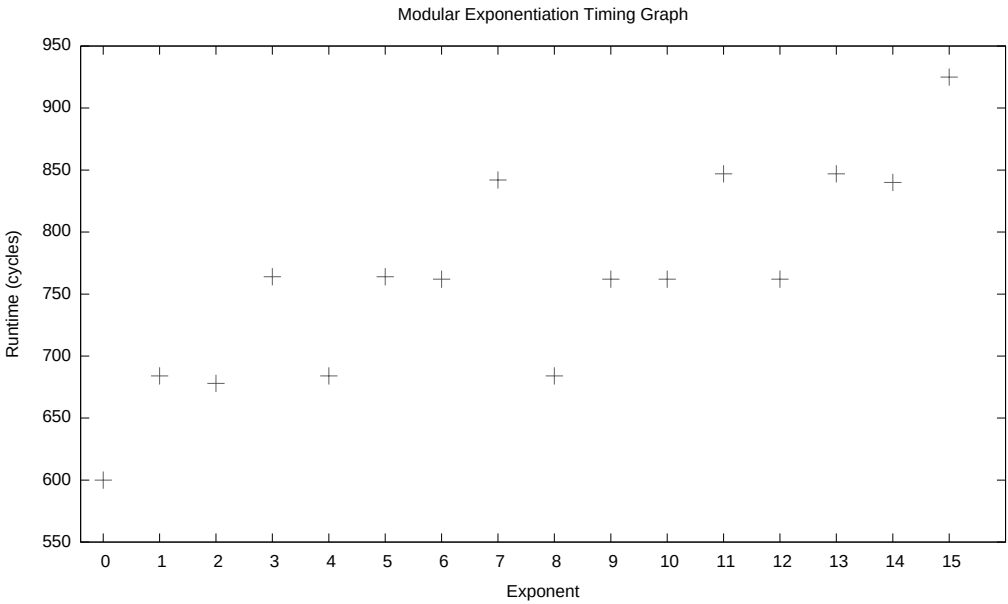


Figure 17.2: Timing data for the `modexp` function of Example 16.1. The graph shows how execution time (in CPU cycles on the y-axis) for `modexp` varies as a function of the value of the 4-bit `exponent` (listed on the x-axis).

> Thus, from simply observing the execution time of `modexp` and *nothing else*, and with some knowledge of the underlying hardware platform (or access to it), an attacker can infer the number of bits set to 1 in `exponent`, i.e., in the secret key. Such information can significantly narrow down the search space for brute-force enumeration of keys.

In Example 17.8, it is assumed that an attacker can measure the execution time of a fairly small piece of code such as `modexp`, embedded within a larger program. Is this realistic? Probably not, but it does not mean timing attacks cannot be mounted. More sophisticated methods are available for an attacker to measure execution time of a procedure. For instance, Tromer et al. (2010) show how AES can be broken using a timing attack where an attacker simply induces cache hits or misses in another process (by writing to carefully chosen memory locations within its own memory segment), along with an indirect way of measuring whether the victim process suffered a cache hit or a cache miss. This can allow the attacker to know if certain table entries were looked up during the AES computation, thereby allowing him to reconstruct the key.

Side-channel attacks remind us that security compromises are often achieved by breaking assumptions made by system designers. In designing an embedded system, careful consideration must be given to the assumptions one makes and to plausible threat models, in order to achieve a reasonable level of assurance in the system's security.

17.6 Summary

Security and privacy are now amongst the top design concerns for embedded, cyber-physical systems. This chapter gave an introduction to security and privacy with a focus on topics relevant for cyber-physical systems. We covered basic cryptographic primitives covering techniques for encryption and decryption, secure hash functions, and digital signatures. The chapter also gave overviews of protocol security and software security, as well as secure information flow, a fundamental topic that cuts across many sub-areas in security and privacy. We concluded with certain advanced topics, including sensor security and side-channel attacks.

Exercises

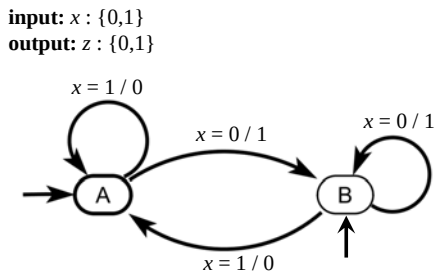
1. Consider the buffer overflow vulnerability in Example 17.2. Modify the code so as to prevent the buffer overflow.

2. Suppose a system M has a secret input s, a public input x, and a public output z. Let all three variables be Boolean. Answer the following TRUE/FALSE questions with justification:

   (a) Suppose M satisfies the linear temporal logic (LTL) property $\mathbf{G}\,\neg z$. Then M must also satisfy observational determinism.

   (b) Suppose M satisfies the linear temporal logic (LTL) property $\mathbf{G}\,[(s \wedge x) \Rightarrow z]$. Then M must also satisfy observational determinism.

3. Consider the finite-state machine below with one input x and one output z, both taking values in $\{0, 1\}$. Both x and z are considered public ("low") signals from a security viewpoint. However, the state of the FSM (i.e., "A" or "B") is considered secret ("high").



   TRUE or FALSE: There is an input sequence an attacker can supply that tells him whether the state machine begins execution in A or in B.

Part IV

Appendices

This part of this text covers some background in mathematics and computer science that is useful to more deeply understand the formal and algorithmic aspects of the main text. Appendix A reviews basic notations in logic, with particular emphasis on sets and functions. Appendix B reviews notions of complexity and computability, which can help a system designer understand the cost of implementing a system and fundamental limits that make certain systems not implementable.

A Sets and Functions

This appendix reviews some basic notation for sets and functions.

A.1 Sets

In this section, we review the notation for sets. A **set** is a collection of objects. When object a is in set A, we write $a \in A$. We define the following sets:

- $\mathbb{B} = \{0, 1\}$, the set of **binary digits**.
- $\mathbb{N} = \{0, 1, 2, \cdots\}$, the set of **natural numbers**.
- $\mathbb{Z} = \{\cdots, -1, 0, 1, 2, \cdots\}$, the set of **integers**.
- $\mathbb{R}$, the set of **real numbers**.
- $\mathbb{R}_+$, the set of **non-negative real numbers**.

When set A is entirely contained by set B, we say that A is a **subset** of B and write $A \subseteq B$. For example, $\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. The sets may be equal, so the statement $\mathbb{N} \subseteq \mathbb{N}$ is true, for example. The **powerset** of a set A is defined to be the set of all subsets. It is written 2^A. The **empty set**, written $\emptyset$, is always a member of the powerset, $\emptyset \in 2^A$.

We define **set subtraction** as follows,

$$A \setminus B = \{a \in A \ : \ a \notin B\}$$

for all sets A and B. This notation is read "the set of elements a from A such that a is not in B."

A **cartesian product** of sets A and B is a set written $A \times B$ and defined as follows,

$$A \times B = \{(a, b) \ : \ a \in A, b \in B\}.$$

A member of this set (a, b) is called a **tuple**. This notation is read "the set of tuples (a, b) such that a is in A and b is in B." A cartesian product can be formed with three or more sets, in which case the tuples have three or more elements. For example, we might write $(a, b, c) \in A \times B \times C$. A cartesian product of a set A with itself is written $A^2 = A \times A$. A cartesian product of a set A with itself n times, where $n \in \mathbb{N}$ is written A^n. A member of the set A^n is called an n**-tuple**. By convention, A^0 is a **singleton set**, or a set with exactly one element, regardless of the size of A. Specifically, we define $A^0 = \{\emptyset\}$. Note that A^0 is not itself the empty set. It is a singleton set containing the empty set (for insight into the rationale for this definition, see the box on page 500).

A.2 Relations and Functions

A **relation** from set A to set B is a subset of $A \times B$. A **partial function** f from set A to set B is a relation where $(a, b) \in f$ and $(a, b') \in f$ imply that $b = b'$. Such a partial function is written $f : A \rightharpoonup B$. A **total function** or simply **function** f from A to B is a partial function where for all $a \in A$, there is a $b \in B$ such that $(a, b) \in f$. Such a function is written $f : A \to B$, and the set A is called its **domain** and the set B its **codomain**. Rather than writing $(a, b) \in f$, we can equivalently write $f(a) = b$.

Example A.1: An example of a partial function is $f : \mathbb{R} \rightharpoonup \mathbb{R}$ defined by $f(x) = \sqrt{x}$ for all $x \in \mathbb{R}_+$. It is undefined for any $x < 0$ in its domain $\mathbb{R}$.

A partial function $f: A \rightharpoonup B$ may be defined by an **assignment rule**, as done in the above example, where an assignment rule simply explains how to obtain the value of $f(a)$ given $a \in A$. Alternatively, the function may be defined by its **graph**, which is a subset of $A \times B$.

Example A.2: The same partial function from the previous example has the graph $f \subseteq \mathbb{R}^2$ given by

$$f = \{(x, y) \in \mathbb{R}^2 \ : \ x \geq 0 \text{ and } y = \sqrt{x}\} \ .$$

Note that we use the same notation f for the function and its graph when it is clear from context which we are talking about.

The **set of all functions** $f: A \to B$ is written $(A \to B)$ or B^A. The former notation is used when the exponential notation proves awkward. For a justification of the notation B^A, see the box on page 500.

The **function composition** of $f: A \to B$ and $g: B \to C$ is written $(g \circ f): A \to C$ and defined by

$$(g \circ f)(a) = g(f(a))$$

for any $a \in A$. Note that in the notation $(g \circ f)$, the function f is applied first. For a function $f: A \to A$, the composition with itself can be written $(f \circ f) = f^2$, or more generally

$$\underbrace{(f \circ f \circ \cdots \circ f)}_{n \text{ times}} = f^n$$

for any $n \in \mathbb{N}$. In case $n = 1$, $f^1 = f$. For the special case $n = 0$, the function f^0 is by convention the **identity function**, so $f^0(a) = a$ for all $a \in A$. When the domain and codomain of a function are the same, i.e., $f \in A^A$, then $f^n \in A^A$ for all $n \in \mathbb{N}$.

For every function $f: A \to B$, there is an associated **image function** $\hat{f}: 2^A \to 2^B$ defined on the powerset of A as follows,

$$\forall \ A' \subseteq A, \quad \hat{f}(A') = \{b \in B \ : \ \exists \, a \in A', \ f(a) = b\}.$$

The image function $\hat{f}$ is applied to *sets* A' of elements in the domain, rather than to single elements. Rather than returning a single value, it returns the set of all values that f would

return, given an element of A' as an argument. We call $\hat{f}$ the **lifted** version of f. When there is no ambiguity, we may write the lifted version of f simply as f rather than $\hat{f}$ (see problem 2(c) for an example of a situation where there is ambiguity).

For any $A' \subseteq A$, $\hat{f}(A')$ is called the **image** of A' for the function f. The image $\hat{f}(A)$ of the domain is called the **range** of the function f.

Example A.3: The image $\hat{f}(\mathbb{R})$ of the function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is $\mathbb{R}_+$.

A function $f: A \to B$ is **onto** (or **surjective**) if $\hat{f}(A) = B$. A function $f: A \to B$ is **one-to-one** (or **injective**) if for all $a, a' \in A$,

$$a \neq a' \Rightarrow f(a) \neq f(a'). \tag{A.1}$$

That is, no two distinct values in the domain yield the same values in the codomain. A function that is both one-to-one and onto is **bijective**.

Example A.4: The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = 2x$ is bijective. The function $f: \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = 2x$ is one-to-one, but not onto. The function $f: \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = xy$ is onto but not one-to-one.

The previous example underscores the fact that an essential part of the definition of a function is its domain and codomain.

Proposition A.1. *If $f: A \to B$ is onto, then there is a one-to-one function $h: B \to A$.*

Proof. Let h be defined by $h(b) = a$ where a is any element in A such that $f(a) = b$. There must always be at least one such element because f is onto. We can now show that h is one-to-one. To do this, consider any two elements $b, b' \in B$ where $b \neq b'$.

We need to show that $h(b) \neq h(b')$. Assume to the contrary that $h(b) = h(b') = a$ for some $a \in A$. But then by the definition of h, $f(a) = b$ and $f(a) = b'$, which implies $b = b'$, a contradiction. □

The converse of this proposition is also easy to prove.

Proposition A.2. *If $h: B \to A$ is one-to-one, then there is an onto function $f: A \to B$.*

Any bijection $f: A \to B$ has an **inverse** $f^{-1}: B \to A$ defined as follows,

$$f^{-1}(b) = a \in A \text{ such that } f(a) = b\,, \tag{A.2}$$

for all $b \in B$. This function is defined for all $b \in B$ because f is onto. And for each $b \in B$ there is a single unique $a \in A$ satisfying (A.2) because f is one-to-one. For any bijection f, its inverse is also bijective.

A.2.1 Restriction and Projection

Given a function $f: A \to B$ and a subset $C \subseteq A$, we can define a new function $f|_C$ that is the **restriction** of f to C. It is defined so that for all $x \in C$, $f|_C(x) = f(x)$.

Example A.5: The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not one-to-one. But the function $f|_{\mathbb{R}_+}$ is.

Consider an n-tuple $a = (a_0, a_1, \cdots, a_{n-1}) \in A_0 \times A_1 \times \cdots \times A_{n-1}$. A **projection** of this n-tuple extracts elements of the tuple to create a new tuple. Specifically, let

$$I = (i_0, i_1, \cdots, i_m) \in \{0, 1, \cdots, n-1\}^m$$

for some $m \in \mathbb{N} \setminus \{0\}$. That is, I is an m-tuple of indexes. Then we define the projection of a onto I by

$$\pi_I(a) = (a_{i_0}, a_{i_1}, \cdots, a_{i_m}) \in A_{i_0} \times A_{i_1} \times \cdots \times A_{i_m}\,.$$

The projection may be used to permute elements of a tuple, to discard elements, or to repeat elements.

Projection of a tuple and restriction of a function are related. An n-tuple $a \in A^n$ where $a = (a_0, a_1, \cdots, a_{n-1})$ may be considered a function of the form $a\colon \{0, 1, \cdots, n-1\} \to A$, in which case $a(0) = a_0$, $a(1) = a_1$, etc. Projection is similar to restriction of this function, differing in that restriction, by itself, does not provide the ability to permute, repeat, or renumber elements. But conceptually, the operations are similar, as illustrated by the following example.

Example A.6: Consider a 3-tuple $a = (a_0, a_1, a_2) \in A^3$. This is represented by the function $a\colon \{0, 1, 2\} \to A$. Let $I = \{1, 2\}$. The projection $b = \pi_I(a) = (a_1, a_2)$, which itself can be represented by a function $b\colon \{0, 1\} \to A$, where $b(0) = a_1$ and $b(1) = a_2$.

The restriction $a|_I$ is not exactly the same function as b, however. The domain of the first function is $\{1, 2\}$, whereas the domain of the second is $\{0, 1\}$. In particular, $a|_I(1) = b(0) = a_1$ and $a|_I(2) = b(1) = a_2$.

A projection may be lifted just like ordinary functions. Given a set of n-tuples $B \subseteq A_0 \times A_1 \times \cdots \times A_{n-1}$ and an m-tuple of indexes $I \in \{0, 1, \cdots, n-1\}^m$, the **lifted projection** is

$$\hat{\pi}_I(B) = \{\pi_I(b) \ : \ b \in B\} \ .$$

A.3 Sequences

A tuple $(a_0, a_1) \in A^2$ can be interpreted as a sequence of length 2. The order of elements in the sequence matters, and is in fact captured by the natural ordering of the natural numbers. The number 0 comes before the number 1. We can generalize this and recognize that a **sequence** of elements from set A of length n is an n-tuple in the set A^n. A^0 represents the set of empty sequences, a singleton set (there is only one empty sequence).

The set of all **finite sequences** of elements from the set A is written A^*, where we interpret $*$ as a wildcard that can take on any value in $\mathbb{N}$. A member of this set with length n is an n-tuple, a **finite sequence**.

The set of **infinite sequences** of elements from A is written $A^{\mathbb{N}}$ or A^{ω}. The set of **finite and infinite sequences** is written

$$A^{**} = A^* \cup A^{\mathbb{N}} .$$

Finite and infinite sequences play an important role in the semantics of concurrent programs. They can be used, for example, to represent streams of messages sent from one part of the program to another. Or they can represent successive assignments of values to a variable. For programs that terminate, finite sequences will be sufficient. For programs that do not terminate, we need infinite sequences.

Exponential Notation for Sets of Functions

The exponential notation B^A for the set of functions of form $f: A \to B$ is worth explaining. Recall that A^2 is the cartesian product of set A with itself, and that 2^A is the powerset of A. These two notations are naturally thought of as sets of functions. A construction attributed to John von Neumann defines the natural numbers as follows,

$$\begin{aligned}
\mathbf{0} &= \emptyset \\
\mathbf{1} &= \{\mathbf{0}\} = \{\emptyset\} \\
\mathbf{2} &= \{\mathbf{0}, \mathbf{1}\} = \{\emptyset, \{\emptyset\}\} \\
\mathbf{3} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\
&\dots
\end{aligned}$$

With this definition, the powerset $\mathbf{2}^A$ is the set of functions mapping the set A into the set $\mathbf{2}$. Consider one such function, $f \in \mathbf{2}^A$. For each $a \in A$, either $f(a) = \mathbf{0}$ or $f(a) = \mathbf{1}$. If we interpret "$\mathbf{0}$" to mean "nonmember" and "$\mathbf{1}$" to mean "member," then indeed the set of functions $\mathbf{2}^A$ represents the set of all subsets of A. Each such function defines a subset.

Similarly, the cartesian product $A^{\mathbf{2}}$ can be interpreted as the set of functions of form $f: \mathbf{2} \to A$, or using von Neumann's numbers, $f: \{\mathbf{0}, \mathbf{1}\} \to A$. Consider a tuple $a = (a_0, a_1) \in A^{\mathbf{2}}$. It is natural to associate with this tuple a function $a: \{\mathbf{0}, \mathbf{1}\} \to A$ where $a(\mathbf{0}) = a_0$ and $a(\mathbf{1}) = a_1$. The argument to the function is the index into the tuple. We can now interpret the set of functions B^A of form $f: A \to B$ as a set of tuples indexed by the set A instead of by the natural numbers.

Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ represent the set of **von Neumann numbers**. This set is closely related to the set $\mathbb{N}$ (see problem 2). Given a set A, it is now natural to interpret A^ω as the set of all infinite sequences of elements from A, the same as $A^{\mathbb{N}}$.

The singleton set $A^{\mathbf{0}}$ can now be interpreted as the set of all functions whose domain is the empty set and codomain is A. There is exactly one such function (no two such functions are distinguishable), and that function has an empty graph. Before, we defined $A^{\mathbf{0}} = \{\emptyset\}$. Using von Neumann numbers, $A^{\mathbf{0}} = \mathbf{1}$, corresponding nicely with the definition of a zero exponent on ordinary numbers. Moreover, you can think of $A^{\mathbf{0}} = \{\emptyset\}$ as the set of all functions with an empty graph.

It is customary in the literature to omit the bold face font for $A^{\mathbf{0}}$, $\mathbf{2}^A$, and $A^{\mathbf{2}}$, writing instead simply A^0, 2^A, and A^2.

Exercises

1. This problem explores properties of onto and one-to-one functions.

   (a) Show that if $f\colon A \to B$ is onto and $g\colon B \to C$ is onto, then $(g \circ f)\colon A \to C$ is onto.

   (b) Show that if $f\colon A \to B$ is one-to-one and $g\colon B \to C$ is one-to-one, then $(g \circ f)\colon A \to C$ is one-to-one.

2. Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ be the von Neumann numbers as defined in the box on page 500. This problem explores the relationship between this set and $\mathbb{N}$, the set of natural numbers.

   (a) Let $f\colon \omega \to \mathbb{N}$ be defined by

   $$f(x) = |x|, \quad \forall\, x \in \omega\ .$$

   That is, $f(x)$ is the size of the set x. Show that f is bijective.

   (b) The lifted version of the function f in part (a) is written $\hat{f}$. What is the value of $\hat{f}(\{0, \{0\}\})$? What is the value of $f(\{0, \{0\}\})$? Note that on page 496 it is noted that when there is no ambiguity, $\hat{f}$ may be written simply f. For this function, is there such ambiguity?

Complexity and Computability

Complexity theory and **computability theory** are areas of Computer Science that study the *efficiency* and the *limits of computation*. Informally, computability theory studies *which problems can be solved* by computers, while complexity theory studies *how efficiently* a problem can be solved by computers. Both areas are *problem-centric*, meaning that they are more concerned with the intrinsic ease or difficulty of problems and less concerned with specific techniques (algorithms) for solving them.

In this appendix, we very briefly review selected topics from complexity and computability theory that are relevant for this book. There are excellent books that offer a detailed treatment of these topics, including Papadimitriou (1994), Sipser (2005), and Hopcroft et al. (2007). We begin with a discussion of the complexity of algorithms. Algorithms are realized by computer programs, and we show that there are limitations on what computer programs can do. We then describe Turing machines, which can be used to define what we have come to accept as "computation," and show how the limitations of programs manifest themselves as undecidable problems. Finally, we close with a discussion of the complexity of *problems*, as distinct from the complexity of the algorithms that solve the problems.

B.1 Effectiveness and Complexity of Algorithms

An **algorithm** is a step-by-step procedure for solving a problem. To be **effective**, an algorithm must complete in a finite number of steps and use a finite amount of resources (such as memory). To be **useful**, an algorithm must complete in a *reasonable* number of steps and use a reasonable amount of resources. Of course, what is "reasonable" will depend on the problem being solved.

Some problems are known to have no effective algorithm, as we will see below when we discuss undecidability. For other problems, one or more effective algorithms are known, but it is not known whether the best algorithm has been found, by some measure of "best." There are even problems where we know that there exists an effective algorithm, but no effective algorithm is known. The following example describes such a problem.

Example B.1: Consider a function $f: \mathbb{N} \to \{\text{YES}, \text{NO}\}$ where $f(n) = \text{YES}$ if there is a sequence of n consecutive fives in the decimal representation of π, and $f(n) = \text{NO}$ otherwise. This function has one of two forms. Either

$$f(n) = \text{YES} \quad \forall\, n \in \mathbb{N},$$

or there is a $k \in \mathbb{N}$ such that

$$f(n) = \begin{cases} \text{YES} & \text{if } n < k \\ \text{NO} & \text{otherwise} \end{cases}$$

It is not known which of these two forms is correct, nor, if the second form is correct, what k is. However, no matter what the answer is, there is an effective algorithm for solving this problem. In fact, the algorithm is rather simple. Either the algorithm immediately returns YES, or it compares n to k and returns YES if $n < k$. We know that one of these is the right algorithm, but we do not know which. Knowing that one of these is correct is sufficient to know that there is an effective algorithm.

For a problem with known effective algorithms, there are typically many algorithms that will solve the problem. Generally, we prefer algorithms with lower complexity. How do we choose among these? This is the topic of the next subsection.

B.1.1 Big O Notation

Many problems have several known algorithms for solving them, as illustrated in the following example.

Example B.2: Suppose we have a list $(a_1, a_2, \cdots, a_n)$ of n integers, arranged in increasing order. We would like to determine whether the list contains a particular integer b. Here are two algorithms that accomplish this:

1. Use a **linear search**. Starting at the beginning of the list, compare the input b against each entry in the list. If it is equal, return YES. Otherwise, proceed to the next entry in the list. In the worst case, this algorithm will require n comparisons before it can give an answer.
2. Use a **binary search**. Start in the middle of the list and compare b to the entry $a_{(n/2)}$ in the middle. If it is equal, return YES. Otherwise, determine whether $b < a_{(n/2)}$. If it is, then repeat the search, but over only the first half of the list. Otherwise, repeat the search over the second half of the list. Although each step of this algorithm is more complicated than the steps of the first algorithm, usually fewer steps will be required. In the worst case, $\log_2(n)$ steps are required.

The difference between these two algorithms can be quite dramatic if n is large. Suppose that $n = 4096$. The first algorithm will require 4096 steps in the worst case, whereas the second algorithm will require only 12 steps in the worst case.

The number of steps required by an algorithm is the **time complexity** of the algorithm. It is customary when comparing algorithms to simplify the measure of time complexity by ignoring some details. In the previous example, we might ignore the complexity of each step of the algorithm and consider only how the complexity grows with the input size n. So if algorithm (1) in Example B.2 takes $K_1 n$ seconds to execute, and algorithm (2) takes $K_2 \log_2(n)$ seconds to execute, we would typically ignore the constant factors K_1 and K_2. For large n, they are not usually very helpful in determining which algorithm is better.

To facilitate such comparisons, it is customary to use **big O notation**. This notation finds the term in a time complexity measure that grows fastest as a function of the size of the input, for large input sizes, and ignores all other terms. In addition, it discards any constant factors in the term. Such a measure is an **asymptotic complexity** measure because it studies only the limiting growth rate as the size of the input gets large.

Example B.3: Suppose that an algorithm has time complexity $5 + 2n + 7n^3$, where n is the size of the input. This algorithm is said to have $O(n^3)$ time complexity, which is read "order n cubed." The term $7n^3$ grows fastest with n, and the number 7 is a relatively unimportant constant factor.

The following complexity measures are commonly used:

1. **constant time**: The time complexity does not depend at all on the size of the input. The complexity is $O(1)$.
2. **logarithmic time**: $O(\log_m(n))$ complexity, for any fixed m.
3. **linear time**: $O(n)$ complexity.
4. **quadratic time**: $O(n^2)$ complexity.
5. **polynomial time**: $O(n^m)$ complexity, for any fixed $m \in \mathbb{N}$.

6. **exponential time**: $O(m^n)$ complexity for any $m > 1$.
7. **factorial time**: $O(n!)$ complexity.

The above list is ordered by costliness. Algorithms later in the list are usually more expensive to realize than algorithms earlier in the list, at least for large input size n.

Example B.4: Algorithm 1 in Example B.2 is a linear-time algorithm, whereas algorithm 2 is a logarithmic-time algorithm. For large n, algorithm (2) is more efficient.

The number of steps required by an algorithm, of course, is not the only measure of its cost. Some algorithms execute in rather few steps but require a great deal of memory. The size of the memory required can be similarly characterized using big O notation, giving a measure of **space complexity**.

B.2 Problems, Algorithms, and Programs

Algorithms are developed to solve some problem. How do we know whether we have found the best algorithm to solve a problem? The time complexity of *known* algorithms can be compared, but what about algorithms we have not thought of? Are there problems for which there is no algorithm that can solve them? These are difficult questions.

Assume that the input to an algorithm is a member of a set W of all possible inputs, and the output is a member of a set Z of all possible outputs. The algorithm computes a function $f : W \to Z$. The function f, a mathematical object, is the **problem** to be solved, and the algorithm is the **mechanism** by which the problem is solved.

It is important to understand the distinction between the problem and the mechanism. Many different algorithms may solve the same problem. Some algorithms will be better than others; for example, one algorithm may have lower time complexity than another. We next address two interesting questions:

- Is there a function of the form $f : W \to Z$ for which there is no algorithm that can compute the function for all inputs $w \in W$? This is a computability question.

- Given a particular function $f: W \to Z$, is there a lower bound on the time complexity of an algorithm to compute the function? This is a complexity question.

If W is a finite set, then the answer to the first question is clearly no. Given a particular function $f: W \to Z$, one algorithm that will always work uses a lookup table listing $f(w)$ for all $w \in W$. Given an input $w \in W$, this algorithm simply looks up the answer in the table. This is a constant-time algorithm; it requires only one step, a table lookup. Hence, this algorithm provides the answer to the second question, which is that if W is a finite set, then the lowest time complexity is constant time.

A lookup table algorithm may not be the best choice, even though its time complexity is constant. Suppose that W is the set of all 32-bit integers. This is a finite set with 2^{32} elements, so a table will require more than four billion entries. In addition to time complexity, we must consider the memory required to implement the algorithm.

The above questions become particularly interesting when the set W of possible inputs is infinite. We will focus on **decision problems**, where $Z = \{\text{YES}, \text{NO}\}$, a set with only two elements. A decision problem seeks a yes or no answer for each $w \in W$. The simplest infinite set of possible inputs is $W = \mathbb{N}$, the natural numbers. Hence, we will next consider fundamental limits on decision problems of the form $f: \mathbb{N} \to \{\text{YES}, \text{NO}\}$. We will see next that for such problems, the answer to the first question above is yes. There are functions of this form that are not computable.

B.2.1 Fundamental Limitations of Programs

One way to describe an algorithm is to give a computer program. A computer program is always representable as a member of the set $\{0,1\}^*$, i.e., the set of finite sequences of bits. A **programming language** is a subset of $\{0,1\}^*$. It turns out that not all decision problems can be solved by computer programs.

Proposition B.1. *No programming language can express a program for each and every function of the form $f: \mathbb{N} \to \{YES, NO\}$.*

Proof. To prove this proposition, it is enough to show that there are strictly more functions of the form $f: \mathbb{N} \to \{\text{YES}, \text{NO}\}$ than there are programs in a programming language. It is sufficient to show that the set $\{\text{YES}, \text{NO}\}^{\mathbb{N}}$ is strictly larger than the

set $\{0,1\}^*$, because a programming language is a subset of $\{0,1\}^*$. This can be done with a variant of **Cantor's diagonal argument**, which goes as follows.

First, note that the members of the set $\{0,1\}^*$ can be listed in order. Specifically, we list them in the order of binary numbers,

$$\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \cdots, \tag{B.1}$$

where λ is the empty sequence. This list is infinite, but it includes all members of the set $\{0,1\}^*$. Because the members of the set can be so listed, the set $\{0,1\}^*$ is said to be **countable** or **countably infinite**.

For any programming language, every program that can be written will appear somewhere in the list (B.1). Assume the first such program in the list realizes the decision function $f_1 : \mathbb{N} \to \{\text{YES}, \text{NO}\}$, the second one in the list realizes $f_2 : \mathbb{N} \to \{\text{YES}, \text{NO}\}$, etc. We can now construct a function $g : \mathbb{N} \to \{\text{YES}, \text{NO}\}$ that is not computed by any program in the list. Specifically, let

$$g(i) = \begin{cases} \text{YES} & \text{if } f_i(i) = \text{NO} \\ \text{NO} & \text{if } f_i(i) = \text{YES} \end{cases}$$

for all $i \in \mathbb{N}$. This function g differs from every function f_i in the list, and hence it is not included in the list. Thus, there is no computer program in the language that computes function g.

□

This theorem tells us that programs, and hence algorithms, are not capable of solving all decision problems. We next explore the class of problems they can solve, known as the **effectively computable** functions. We do this using Turing machines.

B.3 Turing Machines and Undecidability

In 1936, **Alan Turing** proposed a model for computation that is now called the **Turing machine** (Turing, 1936). A Turing machine, depicted in Figure B.1, is similar to a finite-state machine, but with an unlimited amount of memory. This memory has the form of an infinite tape that the Turing machine can read from and write to. The machine comprises

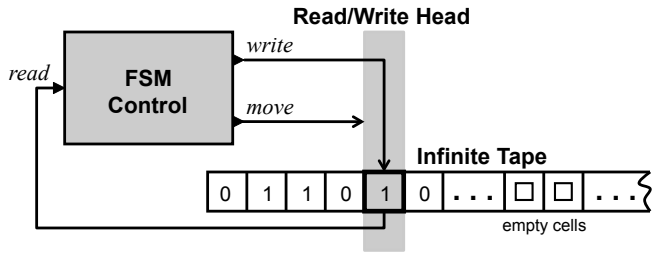


Figure B.1: Illustration of a Turing machine.

a finite-state machine (FSM) controller, a read/write head, and an infinite tape organized as a sequence of cells. Each cell contains a value drawn from a finite set Σ or the special value $\square$, which indicates an **empty cell**. The FSM acts as a control for the read/write head by producing outputs that move the read/write head over the tape.

In Figure B.1, the symbols on the non-empty cells of the tape are drawn from the set $\Sigma = \{0, 1\}$, the binary digits. The FSM has two output ports. The top output port is *write*, which has type Σ and produces a value to write to the cell at the current position of the read/write head. The bottom output port is *move*, which has type $\{L, R\}$, where the output symbol L causes the read/write head to move to the left (but not off the beginning of the tape), and R causes it to move to the right. The FSM has one input port, *read*, which has type Σ and receives the current value held by the cell under the read/write head.

The tape is initialized with an **input string**, which is an element of the set Σ^* of finite sequences of elements of Σ, followed by an infinite sequence of empty cells. The Turing machine starts in the initial state of the FSM with the read/write head on the left end of the tape. At each reaction, the FSM receives as input the value from the current cell under the read/write head. It produces an output that specifies the new value of that cell (which may be the same as the current value) and a command to move the head left or right.

The control FSM of the Turing machine has two final states: an **accepting state** accept and a **rejecting state** reject. If the Turing machine reaches accept or reject after a finite number of reactions, then it is said to **terminate**, and the execution is called a **halting computation**. If it terminates in accept, then the execution is called an **accepting computation**. If it terminates in reject, then the execution is called a **rejecting computation**. It is also possible for a Turing machine to reach neither accept nor reject, meaning that it does not halt. When a Turing machine does not halt, we say that it **loops**.

When the control FSM is deterministic, we say that the Turing machine is also deterministic. Given an input string $w \in \Sigma^*$, a deterministic Turing machine D will exhibit a unique computation. Therefore, given an input string $w \in \Sigma^*$, a deterministic Turing machine D will either halt or not, and if it halts, it will either accept w or reject it. For simplicity, we will limit ourselves in this section to deterministic Turing machines, unless explicitly stated otherwise.

B.3.1 Structure of a Turing Machine

More formally, each **deterministic Turing machine** can be represented by a pair $D = (\Sigma, M)$, where Σ is a finite set of symbols and M is any FSM with the following properties:

- a finite set $States_M$ of states that includes two final states accept and reject;
- an input port *read* of type Σ;
- an output port *write* of type Σ; and
- an output port *move* of type $\{L, R\}$.

As with any FSM, it also must have an initial state s_0 and a transition function $update_M$, as explained in Section 3.3.3. If the read/write head is over a cell containing $\Box$, then the input to the *read* port of the FSM will be *absent*. If at a reaction the *write* output of the FSM is *absent*, then the cell under the read/write head will be erased, setting its contents to $\Box$.

A Turing machine described by $D = (\Sigma, M)$ is a synchronous composition of two machines, the FSM M and a tape T. The tape T is distinctly not an FSM, since it does not have finite state. Nonetheless, the tape is a (extended) state machine, and can be described using the same five-tuple used in Section 3.3.3 for FSMs, except that the set $States_T$ is now infinite. The data on the tape can be modeled as a function with domain $\mathbb{N}$ and codomain $\Sigma \cup \{\Box\}$, and the position of the read/write head can be modeled as a natural number, so

$$States_T = \mathbb{N} \times (\Sigma \cup \{\Box\})^{\mathbb{N}}.$$

The machine T has input port *write* of type Σ, input port *move* of type $\{L, R\}$, and output port *read* of type Σ. The $update_T$ transition function is now easy to define formally (see Exercise 1).

Note that the machine T is the same for all Turing machines, so there is no need to include it in the description $D = (\Sigma, M)$ of a particular Turing machine. The description D can be understood as a program in a rather special programming language. Since all sets in the formal description of a Turing machine are finite, any Turing machine can be encoded as a finite sequence of bits in $\{0, 1\}^*$.

Note that although the control FSM M and the tape machine T both generate output, the Turing machine itself does not. It only computes by transitioning between states of the control FSM, updating the tape, and moving left (L) or right (R). On any input string w, we are only concerned with whether the Turing machine halts, and if so, whether it accepts or rejects w. Thus, a Turing machine attempts to map an input string $w \in \Sigma^*$ to $\{\textit{accept}, \textit{reject}\}$, but for some input strings, it may be unable to produce an answer.

We can now see that Proposition B.1 applies, and the fact that a Turing machine may not produce an answer on some input strings is not surprising. Let $\Sigma = \{0, 1\}$. Then any input string $w \in \Sigma^*$ can be interpreted as a binary encoding of a natural number in $\mathbb{N}$. Thus, a Turing machine implements a partial function of the form $f : \mathbb{N} \to \{\textit{accept}, \textit{reject}\}$. The function is partial because for some $n \in \mathbb{N}$, the machine may loop. Since a Turing machine is a program, Proposition B.1 tells that Turing machines are incapable of realizing all functions of the form $f : \mathbb{N} \to \{\textit{accept}, \textit{reject}\}$. This limitation manifests itself as looping.

A principle that lies at heart of computer science, known as the **Church-Turing thesis**, asserts that every effectively computable function can be realized by a Turing machine. This principle is named for mathematicians Alonzo Church and Alan Turing. Our intuitive notion of computation, as expressed in today's computers, is equivalent to the Turing machine model of computation in this sense. Computers can realize exactly the functions that can be realized by Turing machines: no more, no less. This connection between the informal notion of an algorithm and the precise Turing machine model of computation is not a theorem; it cannot be proved. It is a principle that underlies what we mean by computation.

B.3.2 Decidable and Undecidable Problems

Turing machines, as described here, are designed to solve decision problems, which only have a YES or NO answer. The input string to a Turing machine represents the encoding of a **problem instance**. If the Turing machine *accepts*, it is viewed as a YES answer, and

if it *rejects*, it is viewed as a NO answer. There is the third possibility that the Turing machine might loop.

Example B.5: Consider the problem of determining, given a directed graph G with two nodes s and t in G, whether there is a path from s to t. One can think of writing down the problem as a long string listing all nodes and edges of G, followed by s and t. Thus, an instance of this path problem can be presented to the Turing machine as an input string on its tape. The *instance* of the problem is the particular graph G, and nodes s and t. If there exists a path from s to t in G, then this is a YES problem instance; otherwise, it is a NO problem instance.

Turing machines are typically designed to solve *problem*s, rather than specific problem instances. In this example, we would typically design a Turing machine that, for *any* graph G, nodes s and t, determines whether there is a path in G from s to t.

Probing Further: Recursive Functions and Sets

Logicians make distinctions between the functions that can be realized by Turing machines. The so-called **total recursive functions** are those where a Turing machine realizing the function terminates for all inputs $w \in \Sigma^*$. The **partial recursive functions** are those where a Turing machine may or may not terminate on a particular input $w \in \Sigma^*$. By these definitions, every total recursive function is also a partial recursive function, but not vice-versa.

Logicians also use Turing machines to make useful distinctions between sets. Consider sets of natural numbers, and consider Turing machines where $\Sigma = \{0, 1\}$ and an input $w \in \Sigma^*$ is the binary encoding of a natural number. Then a set C of natural numbers is a **computable set** (or synonymously a **recursive set** or **decidable set**) if there is a Turing machine that terminates for all inputs $w \in \mathbb{N}$ and yields *accept* if $w \in C$ and *reject* if $w \notin C$. A set $E \subset \mathbb{N}$ is a **computably enumerable set** (or synonymously a **recursively enumerable set** or a **semidecidable set**) if there is a Turing machine that terminates if and only if the input w is in E.

Recall that a decision problem is a function $f: W \to \{\text{YES}, \text{NO}\}$. For a Turing machine, the domain is a set $W \subseteq \Sigma^*$ of finite sequences of symbols from the set Σ. A problem instance is a particular $w \in W$, for which the "answer" to the problem is either $f(w) = \text{YES}$ or $f(w) = \text{NO}$. Let $Y \subseteq W$ denote the set of all YES instances of problem f. That is,

$$Y = \{w \in W \mid f(w) = \text{YES}\}.$$

Given a decision problem f, a Turing machine $D = (\Sigma, M)$ is called a **decision procedure** for f if D accepts every string $w \in Y$, and D rejects every $w \in W \setminus Y$, where $\setminus$ denotes set subtraction. Note that a decision procedure always halts for any input string $w \in W$.

A problem f is said to be **decidable** (or **solvable**) if there exists a Turing machine that is a decision procedure for f. Otherwise, we say that the problem is **undecidable** (or **unsolvable**). For an undecidable problem f, there is no Turing machine that terminates with the correct answer $f(w)$ for all input strings $w \in W$.

One of the important philosophical results of 20th century mathematics and computer science is the existence of problems that are undecidable. One of the first problems to be proved undecidable is the so-called **halting problem** for Turing machines. This problem can be stated as follows:

> Given a Turing machine $D = (\Sigma, M)$ initialized with input string $w \in \Sigma^*$ on its tape, decide whether or not M will halt.

Proposition B.2. *(Turing, 1936) The halting problem is undecidable.*

Proof. This is a decision problem $h: W' \to \{\text{YES}, \text{NO}\}$, where W' denotes the set of all Turing machines and their inputs. The proposition can be proved using a variant of Cantor's diagonal argument.

It is sufficient to prove the theorem for the subset of Turing machines with binary tape symbols, $\Sigma = \{0, 1\}$. Moreover, we can assume without loss of generality that every Turing machine in this set can be represented by a finite sequence of binary digits (bits), so

$$W' = \Sigma^* \times \Sigma^*.$$

Assume further that every finite sequence of bits represents a Turing machine. The form of the decision problem becomes

$$h\colon \Sigma^* \times \Sigma^* \to \{\text{YES}, \text{NO}\}. \tag{B.2}$$

We seek a procedure to determine the value of $h(D, w)$, where D is a finite sequence of bits representing a Turing machine and w is a finite sequence of bits representing an input to the Turing machine. The answer $h(D, w)$ will be YES if the Turing machine D halts with input w and NO if it loops.

Consider the set of all effectively computable functions of the form

$$f\colon \Sigma^* \times \Sigma^* \to \{\text{YES}, \text{NO}\}.$$

These functions can be given by a Turing machine (by the Church-Turing thesis), and hence the set of such functions can be enumerated $f_0, f_1, f_2, \cdots$. We will show that the halting problem (B.2) is not on this list. That is, there is no f_i such that $h = f_i$.

Consider a sequence of Turing machines $D_0, D_1, \cdots$ where D_i is the sequence of bits representing the ith Turing machine, and D_i halts if $f_i(D_i, D_i) = \text{NO}$ and loops otherwise. Since f_i is a computable function, we can clearly construct such a Turing machine. Not one of the computable functions in the list $f_0, f_1, f_2, \cdots$ can possibly equal the function h, because every function f_i in the list gives the wrong answer for input (D_i, D_i). If Turing machine D_i halts on input $w = D_i$, function f_i evaluates to $f_i(D_i, D_i) = \text{NO}$, whereas $h(D_i, D_i) = \text{YES}$. Since no function in the list $f_0, f_1, f_2, \cdots$ of computable functions works, the function h is not computable. □

B.4 Intractability: P and NP

Section B.1 above studied asymptotic complexity, a measure of how quickly the cost (in time or space) of solving a problem with a particular algorithm grows with the size of the input. In this section, we consider *problems* rather than *algorithms*. We are interested in whether an algorithm with a particular asymptotic complexity *exists* to solve a problem. This is not the same as asking whether an algorithm with a particular complexity class is *known*.

A **complexity class** is a collection of problems for which there exist algorithms with the same asymptotic complexity. In this section, we very briefly introduce the complexity classes **P** and **NP**.

First recall the concept of a deterministic Turing machine from the preceding section. A **nondeterministic Turing machine** $N = (\Sigma, M)$ is identical to its deterministic counterpart, except that the control FSM M can be a nondeterministic FSM. On any input string $w \in \Sigma^*$, a nondeterministic Turing machine N can exhibit several computations. N is said to **accept** w if *any* computation accepts w, and N **rejects** w if *all* its computations reject w.

A decision problem is a function $f: W \to \{\text{YES}, \text{NO}\}$, where $W \subseteq \Sigma^*$. N is said to be a decision procedure for f if for each input $w \in W$, *all* of its computations halt, no matter what nondeterministic choices are made. Note that a particular execution of a nondeterministic Turing machine N may give the wrong answer. That is, it could yield NO for input w when the right answer is $f(w) = \text{YES}$. It can still be a decision procedure, however, because we define the final answer to be YES if *any* execution yields YES. We do not require that *all* executions yield YES. This subtle point underlies the expressive power of nondeterministic Turing machines.

An execution that accepts an input w is called a **certificate**. A certificate can be represented by a finite list of choices made by the Turing machine such that it accepts w. We need only one valid certificate to know that $f(w) = \text{YES}$.

Given the above definitions, we are ready to introduce P and NP. **P** is the set of problems decidable by a *deterministic* Turing machine in polynomial time. **NP**, on the other hand, is the set of problems decidable by a *nondeterministic* Turing machine in polynomial time. That is, a problem f is in NP if there is a nondeterministic Turing machine N that is a decision procedure for f, and for all inputs $w \in W$, *every* execution of the Turing machine has time complexity no greater than $O(n^m)$, for some $m \in \mathbb{N}$.

An equivalent alternative definition of NP is the set of all problems for which one can check the validity of a certificate for a YES answer in polynomial time. Specifically, a problem f is in NP if there is a nondeterministic Turing machine N that is a decision procedure for f, and given an input w and a certificate, we can check in polynomial time whether the certificate is valid (i.e., whether the choices it lists do indeed result in accepting w). Note that this says nothing about NO answers. This asymmetry is part of the meaning of NP.

An important notion that helps systematize the study of complexity classes is that of **completeness**, in which we identify problems that are "representative" of a complexity class. In the context of NP, we say that a problem A is **NP-hard** if any other problem B in NP can be reduced ("translated") to A in polynomial time. Intuitively, A is "as hard as" any problem in NP — if we had a polynomial-time algorithm for A, we could derive one for B by first translating the instance of B to one of A, and then invoking the algorithm to solve A. A problem A is said to be **NP-complete** if (i) A is in NP, and (ii) A is NP-hard. In other words, an NP-complete problem is a problem in NP that is as hard as any other problem in NP.

Several core problems in the modeling, design, and analysis of embedded systems are NP-complete. One of these is the very first problem to be proved NP-complete, the **Boolean satisfiability** (**SAT**) problem. The SAT problem is to decide, given a propositional logic formula ϕ expressed over Boolean variables $x_1, x_2, \ldots, x_n$, whether there exists a valuation of the x_i variables such that $\phi(x_1, x_2, \ldots, x_n) = \textit{true}$. If there exists such a valuation, we say ϕ is **satisfiable**; otherwise, we say that ϕ is **unsatisfiable**. The SAT problem is a decision problem of the form $f : W \to \{\text{YES}, \text{NO}\}$, where each $w \in W$ is an encoding of a propositional logic formula ϕ.

Example B.6: Consider the following propositional logic formula ϕ:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_2) \wedge (x_1 \vee \neg x_3)$$

We can see that setting $x_1 = x_3 = \textit{true}$ will make ϕ evaluate to *true*. It is possible to construct a nondeterministic Turing machine that takes as input an encoding of the formula, where the nondeterministic choices correspond to choices of valuations for each variable x_i, and where the machine will accept the input formula if it is satisfiable and reject it otherwise. If the input w encodes the above formula ϕ, then one of the certificates demonstrating that $f(w) = \text{YES}$ is the choices $x_1 = x_2 = x_3 = \textit{true}$.

Next consider the alternative formula ϕ':

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$

In this case, no matter how we assign Boolean values to the x_i variables, we cannot make $\phi' = \textit{true}$. Thus while ϕ is satisfiable, ϕ' is unsatisfiable. The

same nondeterministic Turing machine as above will reject an input w' that is an encoding of ϕ'. Rejecting this input means that *all* choices result in executions that terminate in reject.

Another problem that is very useful, but NP-complete, is checking the feasibility of an **integer linear program** (**ILP**). Informally, the feasibility problem for integer linear programs is to find a valuation of integer variables such that each inequality in a collection of linear inequalities over those variables is satisfied.

Given that both SAT and ILP are NP-complete, one can transform an instance of either problem into an instance of the other problem, in polynomial time.

Example B.7: The following integer linear program is equivalent to the SAT problem corresponding to formula ϕ' of Example B.6:

$$\begin{aligned} &\text{find } x_1, x_2 \in \{0, 1\} \\ &\quad \text{such that:} \\ x_1 - x_2 &\geq 0 \\ -x_1 + x_2 &\geq 0 \\ x_1 + x_2 &\geq 1 \\ -x_1 - x_2 &\geq -1 \end{aligned}$$

One can observe that there is no valuation of x_1 and x_2 that will make all the above inequalities simultaneously true.

NP-complete problems seem to be harder than those in P; for large enough input sizes, these problems can become **intractable**, meaning that they cannot be practically solved. In general, it appears that to determine that $f(w) = \text{YES}$ for some w without being given a certificate, we might have to explore *all* executions of the nondeterministic Turing machine before finding, on the last possibility, an execution that accepts w. The number of possible executions can be exponential in the size of the input. Indeed, there are no known polynomial-time algorithms that solve NP-complete problems. Surprisingly, as of

this writing, there is no proof that no such algorithm exists. It is widely believed that NP is a strictly larger set of problems than P, but without a proof, we cannot be sure. The **P versus NP** question is one of the great unsolved problems in mathematics today.

Despite the lack of polynomial-time algorithms for solving NP-complete problems, many such problems turn out to be solvable in practice. SAT problems, for example, can often be solved rather quickly, and a number of very effective **SAT solvers** are available. These solvers use algorithms that have worst-case exponential complexity, which means that for some inputs they can take a very long time to complete. Yet for most inputs, they complete quickly. Hence, we should not be deterred from tackling a problem just because it is NP-complete.

B.5 Summary

This appendix has very briefly introduced two rather large interrelated topics, the theories of complexity and computability. The chapter began with a discussion of complexity measures for algorithms, and then established a fundamental distinction between a problem to be solved and an algorithm for solving the problem. It then showed that there are problems that cannot be solved. We then explained Turing machines, which are capable of describing solution procedures for all problems that have come to be considered "computable." The chapter then closed with a brief discussion of the complexity classes P and NP, which are classes of problems that can be solved by algorithms with comparable complexity.

Exercises

1. Complete the formal definition of the tape machine T by giving the initial state of T and the mathematical description of its transition function $update_T$.

2. *Directed, acyclic graphs* (DAGs) have several uses in modeling, design, and analysis of embedded systems; e.g., they are used to represent precedence graphs of tasks (see Chapter 12) and control-flow graphs of loop-free programs (see Chapter 16).

   A common operation on DAGs is to topologically sort the nodes of the graph. Formally, consider a DAG $G = (V, E)$ where V is the set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is the set of edges. A **topological sort** of G is a linear ordering of vertices $\{v_1, v_2, \ldots, v_n\}$ such that if $(v_i, v_j) \in E$ (i.e., there is a directed edge from v_i to v_j), then vertex v_i appears before vertex v_j in this ordering.

   The following algorithm due to Kahn (1962) topologically sorts the vertices of a DAG:

input : A DAG $G = (V, E)$ with n vertices and m edges.
output: A list L of vertices in V in topologically-sorted order.

```
1  L ← empty list
2  S ← {v | v is a vertex with no incoming edges}
3  while S is non-empty do
4  |   Remove vertex v from S
5  |   Insert v at end of list L
6  |   for each vertex u such that edge (v, u) is in E do
7  |   |   Mark edge (v, u)
8  |   |   if all incoming edges to u are marked then
9  |   |   |   Add u to set S
10 |   |   end
11 |   end
12 end
   L contains all vertices of G in topologically sorted order.
13
```

Algorithm B.1: Topological sorting of vertices in a DAG

State the asymptotic time complexity of Algorithm B.1 using Big O notation. Prove the correctness of your answer.

Bibliography

Abelson, H. and G. J. Sussman, 1996: *Structure and Interpretation of Computer Programs*. MIT Press, 2nd ed.

Adam, T. L., K. M. Chandy, and J. R. Dickson, 1974: A comparison of list schedules for parallel processing systems. *Communications of the ACM*, **17(12)**, 685–690.

Adve, S. V. and K. Gharachorloo, 1996: Shared memory consistency models: A tutorial. *IEEE Computer*, **29(12)**, 66–76.

Allen, J., 1975: Computer architecture for signal processing. *Proceedings of the IEEE*, **63(4)**, 624– 633.

Alpern, B. and F. B. Schneider, 1987: Recognizing safety and liveness. *Distributed Computing*, **2(3)**, 117–126.

Alur, R., 2015: *Principles of Cyber-Physical Systems*. MIT Press.

Alur, R., C. Courcoubetis, and D. Dill, 1991: Model-checking for probabilistic real-time systems. In *Proc. 18th Intl. Colloquium on Automata, Languages and Programming (ICALP)*, pp. 115–126.

Alur, R. and D. L. Dill, 1994: A theory of timed automata. *Theoretical Computer Science*, **126(2)**, 183–235.

Alur, R. and T. A. Henzinger, 1993: Real-time logics: Complexity and expressiveness. *Information and Computation*, **104(1)**, 35–77.

Anderson, R., F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham, 1998: A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, **32(4)**, 9–20.

Anderson, R. and M. Kuhn, 1998: Low cost attacks on tamper resistant devices. In *Security Protocols*, Springer, pp. 125–136.

André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, revision: RR (96–56), University of Sophia-Antipolis. Available from: http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/overview.html.

ARM Limited, 2006: Cortex™- M3 technical reference manual. Tech. rep. Available from: http://www.arm.com.

Audsley, N. C., A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, 2005: Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, **8(2-3)**, 173–198. Available from: http://www.springerlink.com/content/w602g7305r125702/.

Ball, T., V. Levin, and S. K. Rajamani, 2011: A decade of software model checking with SLAM. *Communications of the ACM*, **54(7)**, 68–76.

Ball, T., R. Majumdar, T. Millstein, and S. K. Rajamani, 2001: Automatic predicate abstraction of c programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 36 of *ACM SIGPLAN Notices*, pp. 203–213.

Ball, T. and S. K. Rajamani, 2001: The SLAM toolkit. In *13th International Conference on Computer Aided Verification (CAV)*, Springer, vol. 2102 of *Lecture Notes in Computer Science*, pp. 260–264.

Barr, M. and A. Massa, 2006: *Programming Embedded Systems*. O'Reilly, 2nd ed.

Barrett, C., R. Sebastiani, S. A. Seshia, and C. Tinelli, 2009: Satisfiability modulo theories. In Biere, A., H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability*, IOS Press, vol. 4, chap. 8, pp. 825–885.

Ben-Ari, M., Z. Manna, and A. Pnueli, 1981: The temporal logic of branching time. In *8th Annual ACM Symposium on Principles of Programming Languages*.

Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.

Berger, A. S., 2002: *Embedded Systems Design: An Introduction to Processes, Tools, & Techniques*. CMP Books.

Berry, G., 1999: *The Constructive Semantics of Pure Esterel - Draft Version 3*. Book Draft. Available from: http://www-sop.inria.fr/meije/esterel/doc/main-papers.html.

—, 2003: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152.

Biere, A., A. Cimatti, E. M. Clarke, and Y. Zhu, 1999: Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Springer, vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207.

Boehm, H.-J., 2005: Threads cannot be implemented as a library. In *Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices, vol. 40(6), pp. 261 – 268.

Boneh, D., R. A. DeMillo, and R. J. Lipton, 2001: On the importance of eliminating errors in cryptographic computations. *Journal of cryptology*, **14(2)**, 101–119.

Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.

Brumley, D. and D. Boneh, 2005: Remote timing attacks are practical. *Computer Networks*, **48(5)**, 701–716.

Bryant, R. E., 1986: Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, **C-35(8)**, 677–691.

Bryant, R. E. and D. R. O'Hallaron, 2003: *Computer Systems: A Programmer's Perspective*. Prentice Hall.

Brylow, D., N. Damgaard, and J. Palsberg, 2001: Static checking of interrupt-driven software. In *Proc. Intl. Conference on Software Engineering (ICSE)*, pp. 47–56.

Buck, J. T., 1993: *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.d. thesis, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/.

Burns, A. and S. Baruah, 2008: Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, **2(1)**, 74–97.

Burns, A. and A. Wellings, 2001: *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 3rd ed.

Buttazzo, G. C., 2005a: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd ed.

—, 2005b: Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, **29(1)**, 5–26. doi:10.1023/B:TIME.0000048932.30002.d9.

Cassandras, C. G., 1993: *Discrete Event Systems, Modeling and Performance Analysis*. Irwin.

Cataldo, A., E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, 2006: A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/.

Chapman, B., G. Jost, and R. van der Pas, 2007: *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

Chetto, H., M. Silly, and T. Bouchentouf, 1990: Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, **2(3)**, 181–194.

Clarke, E. M. and E. A. Emerson, 1981: Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pp. 52–71.

Clarke, E. M., O. Grumberg, S. Jha, Y. Lu, and H. Veith, 2000: Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV)*, Springer, vol. 1855 of *Lecture Notes in Computer Science*, pp. 154–169.

Clarke, E. M., O. Grumberg, and D. Peled, 1999: *Model Checking*. MIT Press.

Clarkson, M. R. and F. B. Schneider, 2010: Hyperproperties. *Journal of Computer Security*, **18(6)**, 1157–1210.

Coffman, E. G., Jr., M. J. Elphick, and A. Shoshani, 1971: System deadlocks. *Computing Surveys*, **3(2)**, 67–78.

Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.

Conway, R. W., W. L. Maxwell, and L. W. Miller, 1967: *Theory of Scheduling*. Addison-Wesley.

Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.

Dennis, J. B., 1974: First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science.

Derenzo, S. E., 2003: *Practical Interfacing in the Laboratory: Using a PC for Instrumentation, Data Analysis and Control*. Cambridge University Press.

Diffie, W. and M. E. Hellman, 1976: New directions in cryptography. *Information Theory, IEEE Transactions on*, **22(6)**, 644–654.

Dijkstra, E. W., 1968: Go to statement considered harmful (letter to the editor). *Communications of the ACM*, **11(3)**, 147–148.

Eden, M. and M. Kagan, 1997: The Pentium® processor with MMX™ technology. In *IEEE International Conference (COMPCON)*, IEEE, San Jose, CA, USA, pp. 260–262.

Edwards, S. A., 2000: *Languages for Digital Embedded Systems*. Kluwer Academic Publishers.

Edwards, S. A. and E. A. Lee, 2003: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

Eidson, J. C., 2006: *Measurement, Control, and Communication Using IEEE 1588*. Springer.

Eidson, J. C., E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, 2009: Time-centric models for designing embedded cyber-physical systems. Technical Report UCB/EECS-2009-135, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-135.html.

Einstein, A., 1907: Uber das relativitatsprinzip und die aus demselben gezogene folgerungen. *Jahrbuch der Radioaktivitat und Elektronik*, **4**, 411–462.

Emerson, E. A. and E. M. Clarke, 1980: Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Intl. Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science 85, pp. 169–181.

European Cooperation for Space Standardization, 2002: Space engineering – SpaceWire – links, nodes, routers, and networks (draft ECSS-E-50-12A). Available from: http://spacewire.esa.int/.

Ferguson, N., B. Schneier, and T. Kohno, 2010: *Cryptography Engineering: Design Principles and Practical Applications*. Wiley.

Fielding, R. T. and R. N. Taylor, 2002: Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, **2(2)**, 115–150. doi:10.1145/514183.514185.

Fishman, G. S., 2001: *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag.

Foo Kune, D., J. Backes, S. S. Clark, D. B. Kramer, M. R. Reynolds, K. Fu, Y. Kim, and W. Xu, 2013: Ghost talk: Mitigating EMI signal injection attacks against analog sensors. In *Proceedings of the 34th Annual IEEE Symposium on Security and Privacy*.

Fujimoto, R., 2000: *Parallel and Distributed Simulation Systems*. John Wiley and Sons.

Gajski, D. D., S. Abdi, A. Gerstlauer, and G. Schirner, 2009: *Embedded System Design - Modeling, Synthesis, and Verification*. Springer.

Galison, P., 2003: *Einstein's Clocks, Poincaré's Maps*. W. W. Norton & Company, New York.

Galletly, J., 1996: *Occam-2*. University College London Press, 2nd ed.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, 1994: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334.

Ghena, B., W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman, 2014: Green lights forever: analyzing the security of traffic infrastructure. In *Proceedings of the 8th USENIX conference on Offensive Technologies*, USENIX Association, pp. 7–7.

Ghosal, A., T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido, 2004: Event-driven programming with logical execution times. In *Seventh International Workshop on Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, vol. LNCS 2993, pp. 357–371.

Goldstein, H., 1980: *Classical Mechanics*. Addison-Wesley, 2nd ed.

Goodrich, M. T. and R. Tamassia, 2011: *Introduction to Computer Security*. Addison Wesley.

Graham, R. L., 1969: Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, **17(2)**, 416–429.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, 1991: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, **79(9)**, 1305–1319.

Halderman, J. A., S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, 2009: Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, **52(5)**, 91–98.

Halperin, D., T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, 2008: Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the 29th Annual IEEE Symposium on Security and Privacy*, pp. 129–142.

Hansson, H. and B. Jonsson, 1994: A logic for reasoning about time and reliability. *Formal Aspects of Computing*, **6**, 512–535.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**, 403 – 414. doi:10.1109/32.54292.

Harel, D. and A. Pnueli, 1985: On the development of reactive systems. In Apt, K. R., ed., *Logic and Models for Verification and Specification of Concurrent Systems*, Springer-Verlag, vol. F13 of *NATO ASI Series*, pp. 477–498.

Harter, E. K., 1987: Response times in level structured systems. *ACM Transactions on Computer Systems*, **5(3)**, 232–248.

Hayes, B., 2007: Computing in a parallel universe. *American Scientist*, **95**, 476–480.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch, 2003: Giotto: A time-triggered language for embedded programming. *Proceedings of IEEE*, **91(1)**, 84–99. doi:10.1109/JPROC.2002.805825.

Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.

Hoffmann, G., D. G. Rajnarqan, S. L. Waslander, D. Dostal, J. S. Jang, and C. J. Tomlin, 2004: The Stanford testbed of autonomous rotorcraft for multi agent control (starmac). In *Digital Avionics Systems Conference (DASC)*. doi:10.1109/DASC.2004.1390847.

Holzmann, G. J., 2004: *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Hopcroft, J. E., R. Motwani, and J. D. Ullman, 2007: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd ed.

Horn, W., 1974: Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, **21(1)**, 177 – 185.

Islam, M. S., M. Kuzu, and M. Kantarcioglu, 2012: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium (NDSS)*.

Jackson, J. R., 1955: Scheduling a production line to minimize maximum tardiness. Management Science Research Project 43, University of California Los Angeles.

Jantsch, A., 2003: *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

Jensen, E. D., C. D. Locke, and H. Tokuda, 1985: A time-driven scheduling model for real-time operating systems. In *Real-Time Systems Symposium (RTSS)*, IEEE, pp. 112–122.

Jin, X., A. Donzé, J. Deshmukh, and S. A. Seshia, 2015: Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **34(11)**, 1704–1717.

Joseph, M. and P. Pandya, 1986: Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, **29(5)**, 390–395.

Kahn, A. B., 1962: Topological sorting of large networks. *Communications of the ACM*, **5(11)**, 558–562.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., pp. 993–998.

Kamal, R., 2008: *Embedded Systems: Architecture, Programming, and Design*. McGraw Hill.

Kamen, E. W., 1999: *Industrial Controls and Manufacturing*. Academic Press.

Klein, M. H., T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, 1993: *A Practitioner's Guide for Real-Time Analysis*. Kluwer Academic Publishers.

Kocher, P., J. Jaffe, and B. Jun, 1999: Differential power analysis. In *Advances in CryptologyCRYPTO99*, Springer, pp. 388–397.

Kocher, P. C., 1996: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in CryptologyCRYPTO96*, Springer, pp. 104–113.

Kodosky, J., J. MacCrisken, and G. Rymar, 1991: Visual programming using structured data flow. In *IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 34–39.

Kohler, W. H., 1975: A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Transactions on Computers*, **24(12)**, 1235–1238.

Koopman, P., 2010: *Better Embedded System Software*. Drumnadrochit Education. Available from: http://www.koopman.us/book.html.

Kopetz, H., 1997: *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.

Kopetz, H. and G. Bauer, 2003: The time-triggered architecture. *Proceedings of the IEEE*, **91(1)**, 112–126.

Kopetz, H. and G. Grunsteidl, 1994: TTP - a protocol for fault-tolerant real-time systems. *Computer*, **27(1)**, 14–23.

Koscher, K., A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al., 2010: Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp. 447–462.

Kremen, R., 2008: Operating inside a beating heart. *Technology Review*, October 21, 2008. Available from: http://www.technologyreview.com/biomedicine/21582/.

Kurshan, R., 1994: Automata-theoretic verification of coordinating processes. In Cohen, G. and J.-P. Quadrat, eds., *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, Springer Berlin / Heidelberg, vol. 199 of *Lecture Notes in Control and Information Sciences*, pp. 16–28.

Lamport, L., 1977: Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, **3(2)**, 125–143.

—, 1979: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, **28(9)**, 690–691.

Landau, L. D. and E. M. Lifshitz, 1976: *Mechanics*. Pergamon Press, 3rd ed.

Lapsley, P., J. Bier, A. Shoham, and E. A. Lee, 1997: *DSP Processor Fudamentals – Architectures and Features*. IEEE Press, New York.

Lawler, E. L., 1973: Optimal scheduling of a single machine subject to precedence constraints. *Management Science*, **19(5)**, 544–546.

Le Guernic, P., T. Gauthier, M. Le Borgne, and C. Le Maire, 1991: Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, **79(9)**, 1321 – 1336. doi:10.1109/5.97301.

Lea, D., 1997: *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA.

—, 2005: The java.util.concurrent synchronizer framework. *Science of Computer Programming*, **58(3)**, 293–309.

Lee, E. A., 1999: Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45. doi:10.1023/A:1018998524196.

—, 2001: Soft walls - modifying flight control systems to limit the flight space of commercial aircraft. Technical Memorandum UCB/ERL M001/31, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/01/softwalls2/.

—, 2003: Soft walls: Frequently asked questions. Technical Memorandum UCB/ERL M03/31, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/softwalls/.

—, 2006: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.

—, 2009a: Computing needs time. Tech. Rep. UCB/EECS-2009-30, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-30.html.

—, 2009b: Disciplined message passing. Technical Report UCB/EECS-2009-7, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-7.html.

Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279 –1283. doi:10.1109/GLOCOM.1989.64160.

Lee, E. A., S. Matic, S. A. Seshia, and J. Zou, 2009: The case for timing-centric distributed software. In *IEEE International Conference on Distributed Computing Systems Workshops: Workshop on Cyber-Physical Systems*, IEEE, Montreal, Canada, pp. 57–64. Available from: http://chess.eecs.berkeley.edu/pubs/607.html.

Lee, E. A. and E. Matsikoudis, 2009: The semantics of dataflow with firing. In Huet, G., G. Plotkin, J.-J. Lévy, and Y. Bertot, eds., *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge University Press. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

Lee, E. A. and D. G. Messerschmitt, 1987: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.

Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.

Lee, E. A. and S. Tripakis, 2010: Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, Linköping University Electronic Press, Linköping University, Oslo, Norway, vol. 47, pp. 11–21. Available from: http://chess.eecs.berkeley.edu/pubs/700.html.

Lee, E. A. and P. Varaiya, 2003: *Structure and Interpretation of Signals and Systems*. Addison Wesley.

—, 2011: *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, 2nd ed. Available from: http://LeeVaraiya.org.

Lee, E. A. and H. Zheng, 2005: Operational semantics of hybrid systems. In Morari, M. and L. Thiele, eds., *Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, Zurich, Switzerland, vol. LNCS 3414, pp. 25–53. doi:10.1007/978-3-540-31954-2_2.

—, 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, Salzburg, Austria, pp. 114 – 123. doi:10.1145/1289927.1289949.

Lee, I. and V. Gehlot, 1985: Language constructs for distributed real-time programming. In *Proc. Real-Time Systems Symposium (RTSS)*, San Diego, CA, pp. 57–66.

Lee, R. B., 1996: Subword parallelism with MAX2. *IEEE Micro*, **16(4)**, 51–59.

Lemkin, M. and B. E. Boser, 1999: A three-axis micromachined accelerometer with a cmos position-sense interface and digital offset-trim electronics. *IEEE J. of Solid-State Circuits*, **34(4)**, 456–468. doi:10.1.1.121.8237.

Leung, J. Y.-T. and J. Whitehead, 1982: On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, **2(4)**, 237–250.

Li, X., Y. Liang, T. Mitra, and A. Roychoudhury, 2005: Chronos: A timing analyzer for embedded software. Technical report, National University of Singapore.

Li, Y.-T. S. and S. Malik, 1999: *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers.

Lin, C.-W., Q. Zhu, C. Phung, and A. Sangiovanni-Vincentelli, 2013: Security-aware mapping for can-based real-time distributed automotive systems. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, IEEE, pp. 115–121.

Liu, C. L. and J. W. Layland, 1973: Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, **20(1)**, 46–61.

Liu, J. and E. A. Lee, 2003: Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, **23(1)**, 65–75. doi:10.1109/MCS.2003.1172830.

Liu, J. W. S., 2000: *Real-Time Systems*. Prentice-Hall.

Liu, X. and E. A. Lee, 2008: CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, **409(1)**, 110–125. doi:10.1016/j.tcs.2008.08.044.

Liu, X., E. Matsikoudis, and E. A. Lee, 2006: Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, Springer, Bonn, Germany, vol. LNCS 4137, pp. 1–15. doi:10.1007/11817949_1.

Luminary Micro®, 2008a: Stellaris® LM3S8962 evaluation board user's manual. Tech. rep., Luminary Micro, Inc. Available from: http://www.luminarymicro.com.

—, 2008b: Stellaris® LM3S8962 microcontroller data sheet. Tech. rep., Luminary Micro, Inc. Available from: http://www.luminarymicro.com.

—, 2008c: Stellaris® peripheral driver library - user's guide. Tech. rep., Luminary Micro, Inc. Available from: http://www.luminarymicro.com.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.

Maler, O. and D. Nickovic, 2004: Monitoring temporal properties of continuous signals. In *Proc. International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, Springer, vol. 3253 of *Lecture Notes in Computer Science*, pp. 152–166.

Malik, S. and L. Zhang, 2009: Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM*, **52(8)**, 76–82.

Manna, Z. and A. Pnueli, 1992: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin.

—, 1993: Verifying hybrid systems. In *Hybrid Systems*, vol. LNCS 736, pp. 4–35.

Marion, J. B. and S. Thornton, 1995: *Classical Dynamics of Systems and Particles*. Thomson, 4th ed.

Marwedel, P., 2011: *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2nd ed. Available from: http://springer.com/978-94-007-0256-1.

McLean, J., 1992: Proving noninterference and functional correctness using traces. *Journal of Computer security*, **1(1)**, 37–57.

—, 1996: A general theory of composition for a class of possibilistic properties. *Software Engineering, IEEE Transactions on*, **22(1)**, 53–67.

Mealy, G. H., 1955: A method for synthesizing sequential circuits. *Bell System Technical Journal*, **34**, 1045–1079.

Menezes, A. J., P. C. van Oorschot, and S. A. Vanstone, 1996: *Handbook of Applied Cryptography*. CRC Press.

Milner, R., 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer.

Mishra, P. and N. D. Dutt, 2005: *Functional Verification of Programmable Embedded Processors - A Top-down Approach*. Springer.

Misra, J., 1986: Distributed discrete event simulation. *ACM Computing Surveys*, **18(1)**, 39–65.

Montgomery, P. L., 1985: Modular multiplication without trial division. *Mathematics of Computation*, **44(170)**, 519–521.

Moore, E. F., 1956: Gedanken-experiments on sequential machines. *Annals of Mathematical Studies*, **34(Automata Studies, C. E. Shannon and J. McCarthy (Eds.))**, 129–153.

Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77(4)**, 541–580.

Nemer, F., H. Cass, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, 2006: Papabench: A free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Available from: http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97.

Noergaard, T., 2005: *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier.

Oshana, R., 2006: *DSP Software Development Techniques for Embedded and Real-Time Systems*. Embedded Technology Series, Elsevier.

Ousterhout, J. K., 1996: Why threads are a bad idea (for most purposes) (invited presentation). In *Usenix Annual Technical Conference*.

Paar, C. and J. Pelzl, 2009: *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media.

Papadimitriou, C., 1994: *Computational Complexity*. Addison-Wesley.

Parab, J. S., V. G. Shelake, R. K. Kamat, and G. M. Naik, 2007: *Exploring C for Microcontrollers*. Springer.

Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis.

Patterson, D. A. and D. R. Ditzel, 1980: The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, **8(6)**, 25–33.

Patterson, D. A. and J. L. Hennessy, 1996: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd ed.

Perrig, A., J. Stankovic, and D. Wagner, 2004: Security in wireless sensor networks. *Communications of the ACM*, **47(6)**, 53–57.

Perrig, A., R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, 2002: SPINS: Security protocols for sensor networks. *Wireless networks*, **8(5)**, 521–534.

Perrig, A. and J. D. Tygar, 2012: *Secure Broadcast Communication in Wired and Wireless Networks*. Springer Science & Business Media.

Plotkin, G., 1981: *A Structural Approach to Operational Semantics*.

Pnueli, A., 1977: The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57.

Pottie, G. and W. Kaiser, 2005: *Principles of Embedded Networked Systems Design*. Cambridge University Press.

Price, H. and R. Corry, eds., 2007: *Causation, Physics, and the Constitution of Reality*. Clarendon Press, Oxford.

Queille, J.-P. and J. Sifakis, 1981: Iterative methods for the analysis of Petri nets. In *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, pp. 161–167.

Ravindran, B., J. Anderson, and E. D. Jensen, 2007: On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In *IFIFP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, IEEE ISORC.

Rice, J., 2008: Heart surgeons as video gamers. *Technology Review*, June 10, 2008. Available from: http://www.technologyreview.com/biomedicine/20873/.

Rivest, R. L., A. Shamir, and L. Adleman, 1978: A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, **21(2)**, 120–126.

Roscoe, A. W., 1995: Csp and determinism in security modelling. In *Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on*, IEEE, pp. 114–127.

Sander, I. and A. Jantsch, 2004: System modeling and transformational design refinement in forsyde. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **23(1)**, 17–32.

Schaumont, P. R., 2010: *A Practical Introduction to Hardware/Software Codesign*. Springer. Available from: http://www.springerlink.com/content/978-1-4419-5999-7.

Scott, D. and C. Strachey, 1971: Toward a mathematical semantics for computer languages. In *Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, pp. 19–46.

Seshia, S. A., 2015: Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE*, **103(11)**, 2036–2051.

Seshia, S. A. and A. Rakhlin, 2008: Game-theoretic timing analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 575–582. doi:10.1109/ICCAD.2008.4681634.

—, 2012: Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, **11(S2)**, 55:1–55:27.

Sha, L., T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, 2004: Real time scheduling theory: A historical perspective. *Real-Time Systems*, **28(2)**, 101–155. doi:10.1023/B:TIME.0000045315.61234.1e.

Sha, L., R. Rajkumar, and J. P. Hehoczky, 1990: Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, **39(9)**, 1175–1185.

Shoukry, Y., M. Chong, M. Wakiaki, P. Nuzzo, A. Sangiovanni-Vincentelli, S. A. Seshia, J. P. Hespanha, and P. Tabuada, 2016: SMT-based observer design for cyber physical systems under sensor attacks. In *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*.

Shoukry, Y., P. D. Martin, P. Tabuada, and M. B. Srivastava, 2013: Non-invasive spoofing attacks for anti-lock braking systems. In *15th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pp. 55–72.

Shoukry, Y., P. Nuzzo, A. Puggelli, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, 2015: Secure state estimation under sensor attacks: A satisfiability modulo theory approach. In *Proceedings of the American Control Conference (ACC)*.

Simon, D. E., 2006: *An Embedded Software Primer*. Addison-Wesley.

Sipser, M., 2005: *Introduction to the Theory of Computation*. Course Technology (Thomson), 2nd ed.

Smith, S. and J. Marchesini, 2007: *The Craft of System Security*. Addison-Wesley.

Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

Stankovic, J. A., I. Lee, A. Mok, and R. Rajkumar, 2005: Opportunities and obligations for physical computing systems. *Computer*, 23–31.

Stankovic, J. A. and K. Ramamritham, 1987: The design of the Spring kernel. In *Real-Time Systems Symposium (RTSS)*, IEEE, pp. 146–157.

—, 1988: *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press.

Sutter, H. and J. Larus, 2005: Software and the concurrency revolution. *ACM Queue*, **3(7)**, 54–62.

Terauchi, T. and A. Aiken, 2005: Secure information flow as a safety problem. In *In Proc. of Static Analysis Symposium (SAS)*, pp. 352–367.

Tiwari, V., S. Malik, and A. Wolfe, 1994: Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on VLSI*, **2(4)**, 437–445.

Tremblay, M., J. M. O'Connor, V. Narayannan, and H. Liang, 1996: VIS speeds new media processing. *IEEE Micro*, **16(4)**, 10–20.

Tromer, E., D. A. Osvik, and A. Shamir, 2010: Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, **23(1)**, 37–71.

Turing, A. M., 1936: On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**, 230–265.

Vahid, F. and T. Givargis, 2010: *Programming Embedded Systems - An Introduction to Time-Oriented Programming*. UniWorld Publishing, 2nd ed. Available from: `http://www.programmingembeddedsystems.com/`.

Valvano, J. W., 2007: *Embedded Microcomputer Systems - Real Time Interfacing*. Thomson, 2nd ed.

Vardi, M. Y. and P. Wolper, 1986: Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, **32(2)**, 183–221.

von der Beeck, M., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytopil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.

Wang, Y., S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, 2009: The theory of deadlock avoidance via discrete control. In *Principles of Programming Languages (POPL)*, ACM SIGPLAN Notices, Savannah, Georgia, USA, vol. 44, pp. 252–263. doi:10.1145/1594834.1480913.

Wasicek, A., C. E. Salloum, and H. Kopetz, 2011: Authentication in time-triggered systems using time-delayed release of keys. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 31–39.

Wiener, N., 1948: *Cybernetics: Or Control and Communication in the Animal and the Machine*. Librairie Hermann & Cie, Paris, and MIT Press.Cambridge, MA.

Wilhelm, R., 2005: Determining Bounds on Execution Times. In Zurawski, R., ed., *Handbook on Embedded Systems*, CRC Press.

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstr, 2008: The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, **7(3)**, 1–53.

Wolf, W., 2000: *Computers as Components: Principles of Embedded Computer Systems Design*. Morgan Kaufman.

Wolfe, V., S. Davidson, and I. Lee, 1993: RTC: Language support for real-time concurrency. *Real-Time Systems*, **5(1)**, 63–87.

Wolper, P., M. Y. Vardi, and A. P. Sistla, 1983: Reasoning about infinite computation paths. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 185–194.

Young, W., W. Boebert, and R. Kain, 1985: Proving a computer system secure. *Scientific Honeyweller*, **6(2)**, 18–27.

Zeigler, B., 1976: *Theory of Modeling and Simulation*. Wiley Interscience, New York.

Zeigler, B. P., H. Praehofer, and T. G. Kim, 2000: *Theory of Modeling and Simulation*. Academic Press, 2nd ed.

Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, Bellevue, WA, USA, pp. 259 – 268. doi:10.1109/RTAS.2007.5.

Zhuang, L., F. Zhou, and J. D. Tygar, 2009: Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)*, **13(1)**, 3.

Notation Index

Index

computer science/electrical engineering

Introduction to Embedded Systems
A Cyber-Physical Systems Approach
second edition
Edward Ashford Lee and Sanjit Arunkumar Seshia

The most visible use of computers and software is processing information for human consumption. The vast majority of computers in use, however, are much less visible. They run the engine, brakes, seatbelts, airbag, and audio system in your car. They digitally encode your voice and construct a radio signal to send it from your cell phone to a base station. They command robots on a factory floor, power generation in a power plant, processes in a chemical plant, and traffic lights in a city. These less visible computers are called embedded systems, and the software they run is called embedded software. The principal challenges in designing and analyzing embedded systems stem from their interaction with physical processes. This book takes a cyber-physical approach to embedded systems, introducing the engineering concepts underlying embedded systems as a technology and as a subject of study. The focus is on modeling, design, and analysis of cyber-physical systems, which integrate computation, networking, and physical processes.

The second edition offers two new chapters, several new exercises, and other improvements. The book can be used as a textbook at the advanced undergraduate or introductory graduate level and as a professional reference for practicing engineers and computer scientists. Readers should have some familiarity with machine structures, computer programming, basic discrete mathematics and algorithms, and signals and systems.

Edward Ashford Lee is the Robert S. Pepper Distinguished Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Sanjit Arunkumar Seshia is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley.

"Books titled *Introduction* to *Embedded Systems* traditionally focus on computer hardware and software. By taking *A Cyber-Physical Systems Approach,* Lee and Seshia give students the integrated perspective they need to understand and design the computing systems that make our world function. No other book provides such a comprehensive introduction to embedded systems for real-time applications."
—Bruce H. Krogh, Professor of Electrical and Computer Engineering, Carnegie Mellon University

"*Introduction to Embedded Systems* by Lee and Seshia is an introductory yet rigorous textbook for the future Internet of Things engineer. It provides a unified systems view of computing and the physical world that will be the foundation of the 21st-century Internet of Things revolution."
—George J. Pappas, Joseph Moore Professor, University of Pennsylvania

"Designers of embedded systems are only too often overwhelmed by the many skills and disciplines that have to be mastered: from writing device drivers, to worst case execution time analysis, to formal verification and modeling of continuous time systems. This book by Lee and Seshia is an excellent guide to bringing order into these complexities of design by discerning the fundamental from the detail, the essential property from the accidental aspect. It presents all the indispensable knowledge areas for an embedded systems designer and leaves out what can be delegated to other specialized disciplines."
—Axel Jantsch, Professor of Systems on Chips, Institute of Computer Technology, TU Wien, Vienna; author of *Modeling Embedded Systems and SoC's*

"The outstanding property of this textbook is the combination of mathematical rigor and comprehensiveness. It is presented with numerous examples and with such quality that understanding the material is easy. *Introduction to Embedded Systems* is a must-read for those wanting to master the complexity of what is today the key enabling technology in most every complex system surrounding us: embedded and cyber-physical systems."
—Werner Damm, Director, Interdisciplinary Research Center on Cooperative Critical Systems, Carl von Ossietzky University of Oldenburg

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142
http://mitpress.mit.edu

978-0-262-53381-2