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() {
    int user_pin[] = {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[i]) {
            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.

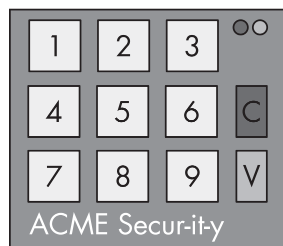

Figure 8-1: A simple keypad

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, 2000, through 9000) while recording the time delay between pressing the V button and the error LED illuminating. Figure 8-2 shows the timing sequence.

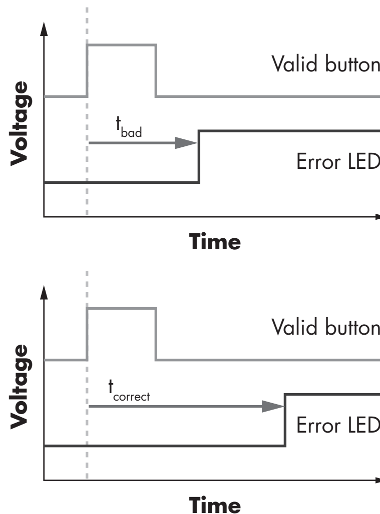

Figure 8-2: Determination of loop delay time

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.

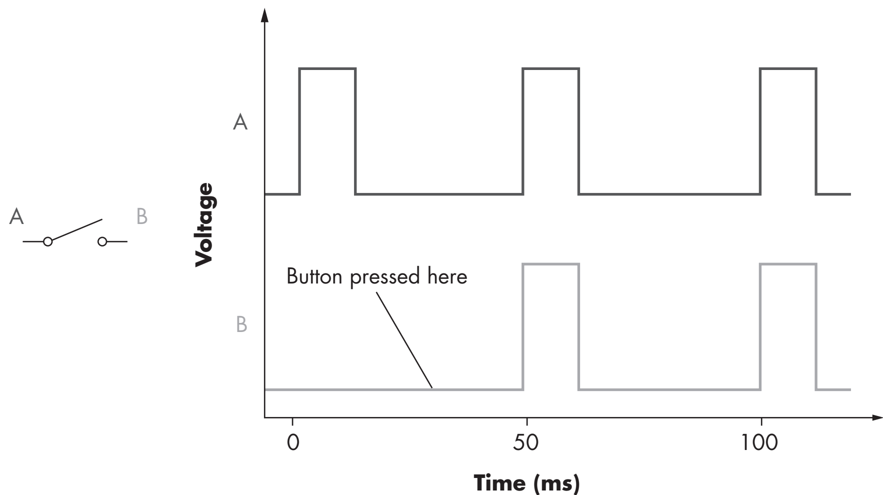

Figure 8-3: Hard drive attack timing diagram

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 top line is the error signal, and the bottom line is the button scan signal. The 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.

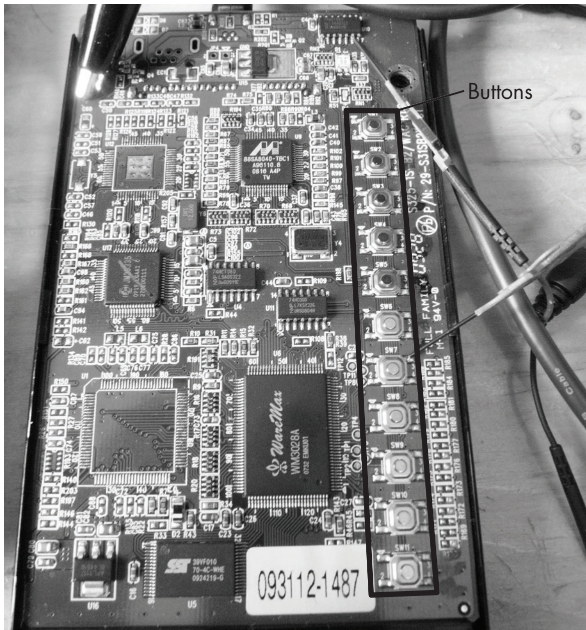

Figure 8-4: Vantec Vault NSTV290S2 hard drive enclosure

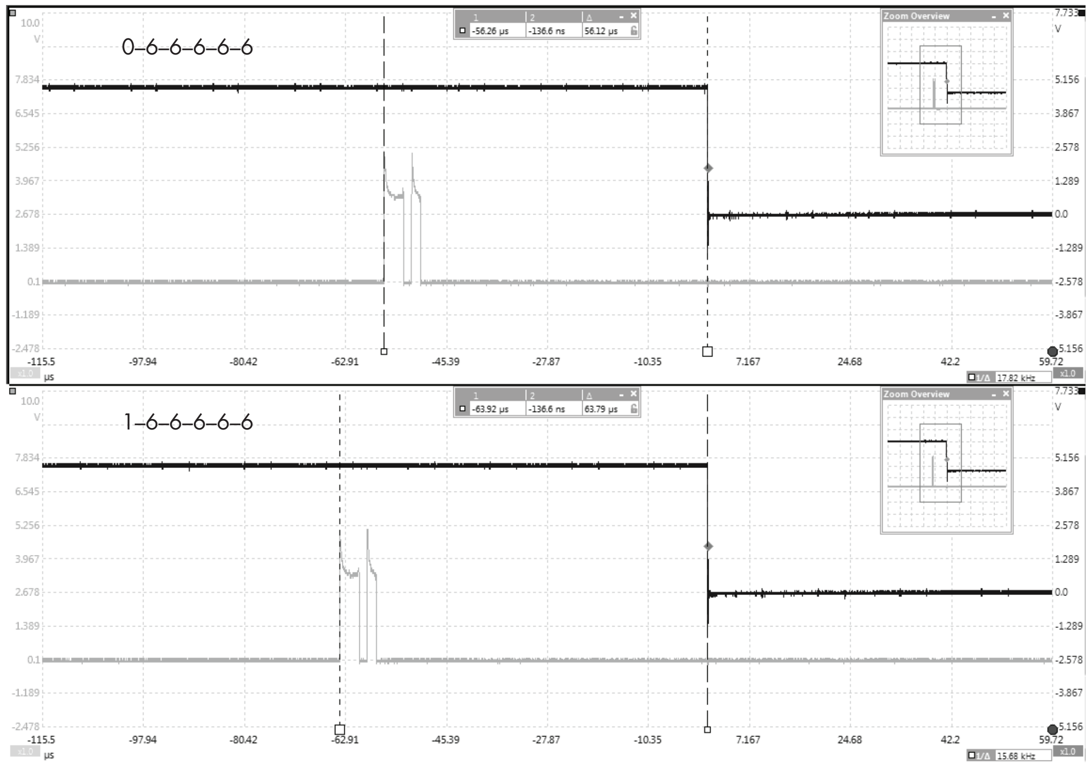

Figure 8-5: Hard drive timing measurement

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 section "Preparing the Oscilloscope" in Chapter 9.) The power consumption might look something like Figure 8-6, which shows the power trace of a device while it's performing an operation.

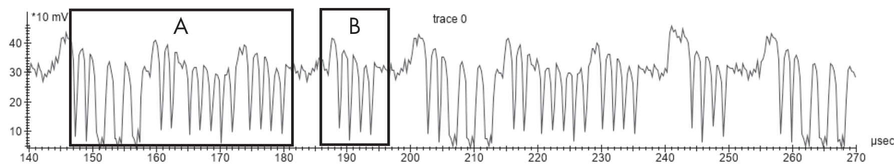

Figure 8-6: A sample power consumption trace of a device performing an operation

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 previous section, 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 mod 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.

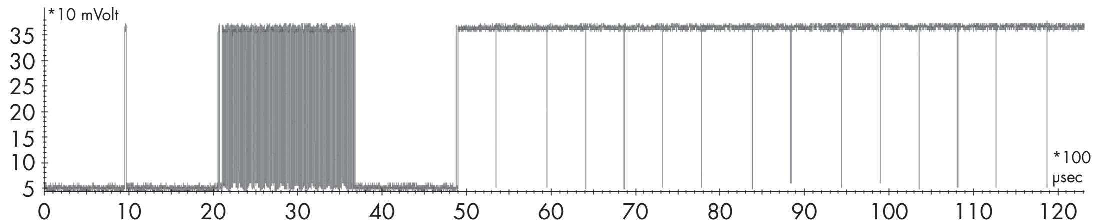

Figure 8-7: Power consumption trace of a square-and-multiply execution

Notice some interesting patterns between roughly 5ms and 12ms (between 50 and 120 on the 100µs unit x-axis): 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 *often* 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 may not work with every compiler) 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 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.

(continued)

Also very relevant in more complex systems are cache timing attacks. Specifically, algorithms that use lookup tables for certain operations can leak information revealing which element is being accessed when a timing variation 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 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 that 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 the function `timeleak_to_trace()`. The start of such a trace will be plotted in the notebook; Figure 8-8 also shows an example.

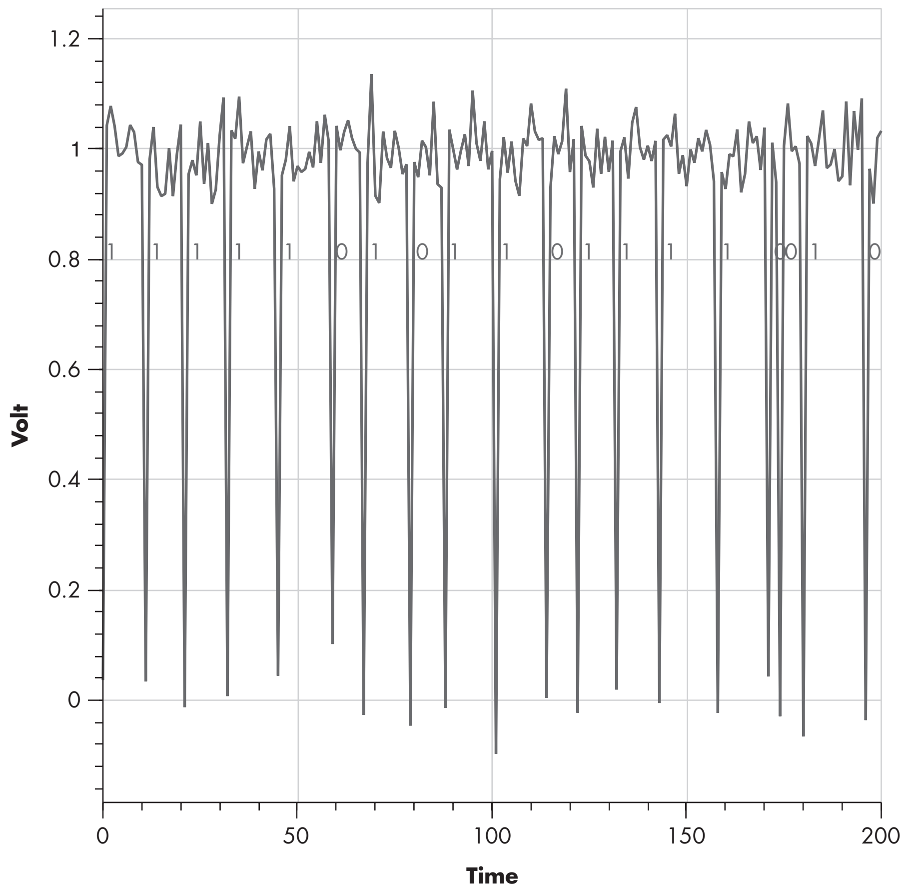

Figure 8-8: Simulated ECDSA power consumption trace showing nonce bits

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.

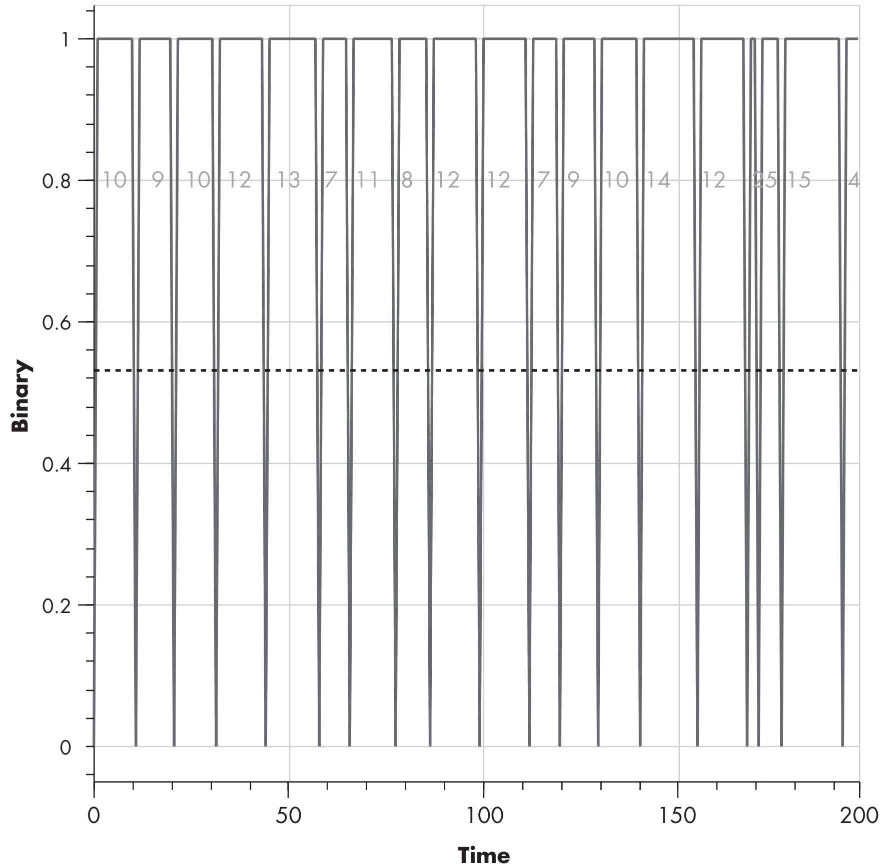

Figure 8-9: Binary ECDSA power consumption trace showing SPA timing leakage

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.

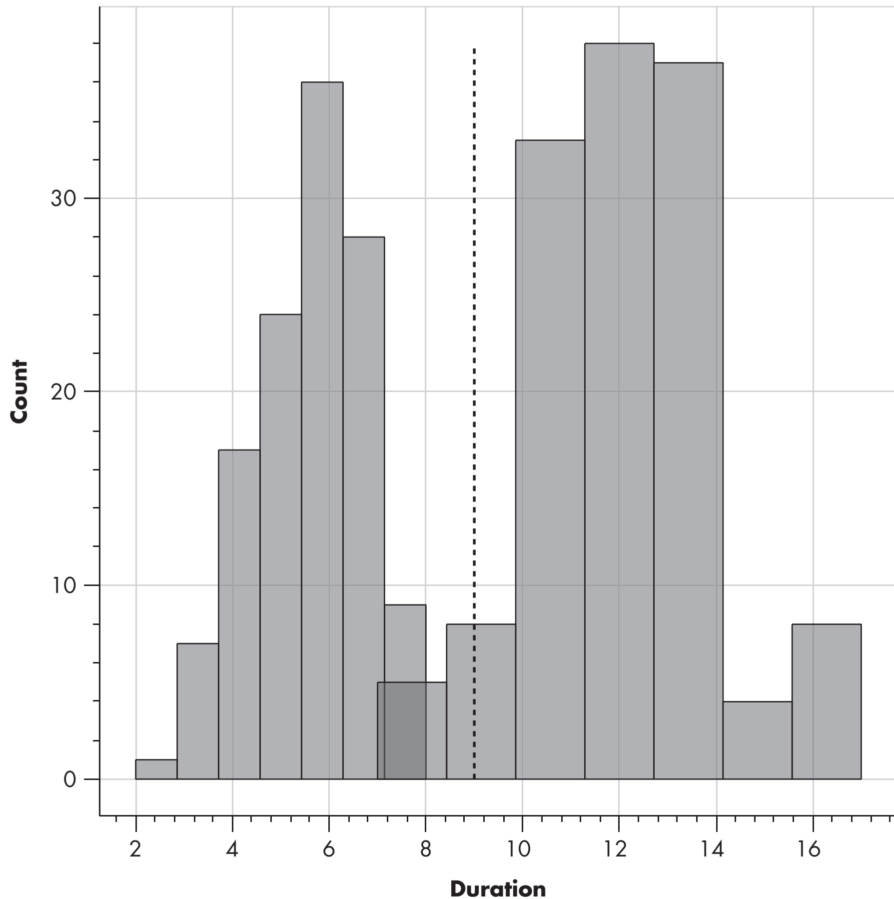

Figure 8-10: The distribution of the durations for a double-only (left) and a double-and-add (right) overlap, disallowing the duration to be a perfect predictor.

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: 0b1111111100011001010111100001101011000111000000110011110100110011
1101000100001011011011001001100100110000001110100011011101010101101000111001
1000010001100000010100001101111010000000010010010000110110111100001101001111101
0110001000110011101000010010100101101
Actual k:  0b1111111100011001010111100001101011000111000010110011110100110011
1101000100001011011011001001111100110000001110100011011101010101101000111001
1000010001100000010100001101111010000000010010010000111110111100001101001111101
0110001000110011101000010010100101101
Bit errors: 4
Bruteforcing bits: [241  60 209 160 161 212  34  21]
No key for you.


Attempt 17
Guessed k: 0b1111101110111000100101000010000110101100000010011100000101101001
1010010000110110000110010010011111000110110111011100110001110101010110000000
1001100011111010001100100011010011000111011010101110001101111100111010010111110
01010001110110001110001101100010
Actual k:  0b1111101110111000100101000010000110101100000011011100000101101001
1010010000110110000110010110011111000110110111011101110001110101010110000000
1001100111111010001110100011010011000111011010101110001101111100111010010111110
01010001110110101110001101100010
Bit errors: 6
Bruteforcing bits: [103 185 135 205  18 161  90  98]
Yeash! Key found:0b110101001000000000010001100011000010100101101011100001101000
1100010111011101111000011100111101101000010100000111001001111110010111100000101
0001001010010111100110100100000001001110001010111100100000100101010010101110100
100111011010001001110000000110010111
```

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.