Hardware Security

6.1600 Course Staff

Fall 2023

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

So far while discussing platform security, we have considered only our software. However, software must run on some hardware, and the security of this hardware is similarly vital—if an attacker can undermine the security of our hardware, it does not matter how strong our software security is. Luckily, hardware is typically much more difficult to attack than software, but powerful attacks are still possible.

As a running example for this chapter, consider the example of a certificate authority that signs certificates given that some policy is met.

An equivalent example is a cryptocurrency wallet—transactions in the cryptocurrency are authorized by a signature over a transaction message.

This server will accept requests and respond with a signature over that request if some policy is met. For example, an MIT CA might enforce a policy like "I will sign only certificates for `*.mit.edu` domains". For this CA server, we can achieve some nice security properties:

- We can prove the security of the signature scheme used under concrete assumptions such as the hardness of discrete log.
- We can verify that the cryptography implementation faithfully implements the signature algorithm on some ideal hardware model.

In order to actually be used, however, we must first buy a computer, load the code onto it, and run it (likely on a machine running other software). This process is outside of the nice security properties with achieved about our CA program. In cryptography and in software, we are able to develop clean characterizations of the power of the attacker. In cryptography, we assume that our adversaries are bounded by probabilistic polynomial time. In software, we can model our attacker as choosing arbitrary inputs to our software and accurately capture an attacker's power. Once we consider the hardware, however, it becomes much more difficult to meaningfully define the attacker's power.

When we consider the real hardware that the system is running on, there are many attacks that we must consider.

1 Hardware Bug

Much of what we have discussed so far has been cryptography schemes that rely on trusted parties knowing some randomness that

the adversary does not know. We have assumed that we have some source of "true" randomness to use, for example, as a seed for a PRF. When actually implementing cryptographic algorithms in a computer, we need to actually materialize this "true" randomness. However, it is not clear where this randomness should come from: computers are designed specifically to behave as they are instructed by the programs they run. Common solutions are to measure statistics about the environment that should be hard for an adversary to predict. For example, devices may use combinations of:

- Keypress timings
- Packet timings
- Clock
- Temperature sensor

All of these require "accumulating" randomness from the environment. A common hardware bug on embedded devices is to generate keys when the randomness source has not accumulated enough random measurements from the environment. For example, if a network card generates a cryptographic key right after boot, this key may be predictable if an attacker is able to accurately guess at the values of the randomness source.

To help with this, many devices include specialized hardware that uses some special circuit to generate randomness by measuring randomness inherent to the universe.. Of course, developers must then use this randomness—a common error is to use insufficient randomness, such as the time, instead of this hardware randomness.

On Intel CPUs, this takes the form of the `RDRAND` instruction

2 *Attacks without Physical Access*

Perhaps the most concerning attacks are those that do not require physical access to a machine.

2.1 Cache Timing Attacks

One major goal of operating systems is to provide isolation between processes. Even if an attacker is able to run some software on the same machine as our signing process, we would like to guarantee that an attacker can not read, for example, the signing key used by our signing process. However, the attacker and victim code both run on the same CPU, and the victim may leave traces of secrets in the state of the CPU.

For example, consider that our signing process runs and, depending on some secret value, either loads the value at memory address A or does not. If the victim loads this address,the CPU will copy the value into the cache to speed up future accesses to the value. An

Because of attacks like this and others, it is important to write secure code such that it does not branch on secret values.

attacker process that runs next can try to access this same memory address and measure how long it takes to read the value. If the victim read that value, the access will be fast since it comes from the cache, but if not, the data will have to come from the much slower main memory. From the difference in this timing, the attacker can learn about the victim's access pattern, which may reveal data about the victim's secret.

This may seem like an unlikely attack since both the victim and attacker must have access to the same memory location, which process-level isolation should prevent. However, operating systems perform something called deduplication that can be cleverly taken advantage of to achieve this: if the victim process uses OpenSSL, the attacker process can also use OpenSSL. The operating system will see that both processes are linking the same library, and may map a section of virtual memory for each process to the same physical memory.

2.2 *Rowhammer*

Data in a computers memory is stored in what is effectively a grid of capacitors. These capacitors do not store values indefinitely, and so their values must be refreshed every so often (commonly every 64 milliseconds). Reading a chunk of memory drains the corresponding capacitors a bit, and they must then be rewritten. Memory is read one row f this grid at a time. Reading a row drains the corresponding capacitors, requiring them to be rewritten. This rewriting involves voltage fluctuations, and since modern memory is so dense, these voltage fluctuations can cause neighboring rows to discharge more quickly that the refresh interval is equipped to handle. Surprisingly, reading a single row repeatedly can cause bits in an adjacent row to flip.

An attacker could take advantage of this by "hammering" a memory location, causing a bit to flip in memory that belongs to another process or to the operating system. In some cases, this was enough to allow the attacker to learn a secret or bypass isolation, etc.

3 *Physical Attacks*

If an attacker has physical access to a device, an entirely new class of attacks becomes possible. They can measure the device, introduce faults to the device, and more.

3.1 *Probing Attacks*

An attacker with physical access to a device can measure many things about the device's behavior that a remote attacker could not. For example:

- Place probes on the pins of a chip
- Measure power consumption of a chip and watch for patterns
- Measure optical emissions of a chip with an electron microscope
- Measure RF emissions from a chip

- Monitor the blinking light on an internet router

The information that an attacker can learn from attacks like this may be limited, but even very slow information leakage can be enough to leak a key in a relatively short amount of time. Protecting against these types of attacks is difficult since the attacker can do such a broad range of things. However, if we make certain assumptions about the attacker's power, we can achieve principled solutions.

Defense against Probing Attacks. One assumption that may be reasonable to make is that an attacker can probe at most t wires of a circuit. By using techniques like secure multiparty communication, it is possible to build a circuit that implements something like a signature scheme, but that does so without leaking anything about the secret given this assumption.

3.2 *Fault Attacks*

An attacker can also introduce faults that the system was not designed to handle. For example, they can point a heat gun or a laser at the chip, hoping to cause some bit flips. If they are successful, these bit flips may leak a secret key or corrupt a kernel data structure, allowing the attacker to take over the system.

Designers of satellite systems have to think about similar attacks, but in their case the attacker is the sun! Cosmic rays carry enough energy to flip bits of CPU registers or memory.

3.3 *Supply Chain Attacks*

When we buy an device, we assume that the hardware inside is not working against us. However, there is a long chain of steps that happens before the device gets to us—it is built in the factory, packaged, mailed across the world to a retailer, stored in a warehouse, packaged and mailed again, and so on. An attacker that has control over any of these steps could intercept the device on its way to you and modify it somehow. For example, they could:

- Modify the randomness source to something predictable
- Preload keys that the attacker knows
- Add extra input/output interfaces
- Add or enable management interfaces

There are no great solutions to defend against this—inspecting the chips is impossible since they are so small, building a device yourself is much too hard, and so on. One solution that can improve the protection is to build a system out of n identical devices and use a strategy like secure multiparty computation to protect against cases

where at most $n - 1$ of these devices is compromised by a supply chain attacker.