Chapter 13

Run-time Assumptions under Attack

For a piece of software to operate as expected, the hardware and lower-level software that define its run-time environment must operate as had been assumed. Assumptions are potential vulnerabilities, and assumptions about a system's run-time environment are no exception. This chapter discusses some of those assumptions, along with attacks to invalidate these assumptions. Where viable defenses are know, we discuss those too. As always, whether a given vulnerability should be concerning will depend on the threat. Not all threats have the motivation, equipment, access, and/or expertise needed for a specific attack.

13.1 Unexpected Communications Channels

A *channel* exists whenenver some medium is modulated by a sender and monitored by a receiver. The modulation might or might not be intentional, and it might or might not be expected by users or even by programmers of a system. Our concern here is with modulation arising from program execution. Electrical voltage, RF, light, sound, power consumption, state (disk, memory, registers, or cache), or event timing each could be modulated as execution proceeds.

When the receiver monitors a channel by executing a program, then it can be useful to distinguish between two ways that modulation could be presented:

- *Storage Channel.* The monitor is affected by state that the sender varies.
- *Timing Channel.* The monitor detects event orderings, occurence times, or intervals that the sender varies.

Shared memory, shared files, and message-passing implement storage channels; timeouts to signal a lost message would be a timing channel.

Knowledge of all channels going from a system to its environment is necessary for enforcing the confidentiality of secrets that system stores. Some of those channels will be obvious. Hardware communicates by using network adapters (including radios for WIFI), displays, and external storage devices. Software communicates by invoking operations, including hardware instructions and procedures exported by lower-level software interfaces. But other channels are likely to be present, too. To ignore an unexpected channels is equivalent to making an assumption: that unexpected channel cannot be used to leak secrets.

Whether making this assumption creates a vulnerability will depend on the unexpected channel's bandwidth, what information the channel conveys, and whether the channel can be monitored by the threats of concern. A high-bandwidth channel would be needed for rapidly leaking images, video, sound, or databases; a low bandwidth channel suffices for transmitting a cryptographic key or password. To focus only on higher-bandwidth channels can be a flawed strategy, because obtaining a key or password can allow an attacker to impersonate a trusted principal.

We might hope to discover unexpected communications channels by manually or automatically analyzing a system's source code or other descriptions. However, certain channels are unlikely to be discovered using this approach.

- Source code and other system descriptions deliberately omit details about the lower layers, such as the run-time environment, the computer hardware, the underlying physics of the hardware, and/or the properties of materials. Yet, as we shall see, all of these elements can create channels.
- Some channels are created by combining functionality from multiple layers of the system. These channels cannot be discovered by consulting separate, independent descriptions for the various different system layers.

Therefore, human analysts who have an in-depth knowledge of the literature are indispensable. These analysts would study documentation of a system, its run-time environment, and the underlying hardware. The analysts also would perform experiments to validate what the documentation says and to learn about aspects of system operation that are not mentioned in the documentation.

13.1.1 Covert Channels

A *covert channel* is created when an attacker repurposes functionality and causes information disclosures that violate the system's security policy. A key characteristic is that a covert channel causes information transmission outside the scope of system authorization mechanisms.[1] The attacker would monitor the covert channel as well as provide a program that gets executed to modulate the covert channel. Many covert channels have low capacity. Some covert channels are also noisy, but an attacker can compensate for noise by using error correcting codes (with some reduction in channel capacity) when modulating the covert channel.

[1] This characteristic is consistent with the meaning "not openly acknowledged or displayed" that "covert" has in non-technical usage. But in contrast to its non-technical usage, the definition of a covert channel used in computer security does not require hiding what is being communicated and does not require hiding whether communication is occurring.

Modulation enabled by time multiplexing. If time multiplexing is used to give each of multiple principals the appearance of exclusive access to some resource then an operation requested by one principal can be delayed because operations were requested by other principals. The length of such a delay can be determined in many ways, and this length can be modulated by varying the requests that are made. So time multiplexing can create covert timing channels.

A classic example involves attacker-controlled principals T (for transmitter) and R (for receiver) that are sharing a time-multiplexed processor where, to achieve higher utilization, the system ends the current time slice whenever the executing principal invokes an operation to await completion of an I/O operation. T leaks the value of a secret bit b to R, as follows. T either runs a compute-intensive task if $b = 1$ holds or T runs an input/output intensive task if $b = 0$ holds. And R measures the time that elapses for the execution of a fixed sequence of instructions requiring multiple time slices. If a longer elapsed time is measured by R then T has executed for full time slices, which implies T is not executing the input/output intensive task and, thus, $b = 1$ holds.

We bound the bandwidth of a covert timing channel that is created by time multiplexing if we limit how well principals can control and/or sense variation in delays associated with access to resources.

Bandwidth Bounds on Timing Channels.

(i) Adhere to a schedule that is unaffected by access requests made by principals. Examples include:
   - a schedule that specifies in advance the disjoint intervals when each principal may access the resource,
   - a schedule where the disjoint intervals when each principal may access the resource start at random times and have unpredictable durations.

(ii) Prevent a principal from measuring the actual starting and ending times of a periods when it may access the resource. □

Defense (i) prevents modulation; defense (ii) prevents monitoring. One suffices.

Defense (ii) can be implemented by intercepting operations that can be used to measure timing. Those interceptions are easily achieved for user-mode code running on processors where system-mode instructions provide the only access to clocks and provide the only way to instigate activity (e.g., an input/output operation) that delivers an interrupt at some fixed, later time.

- System-mode software would provide the only system calls that principals can invoke to obtain timing information. These system calls could return
  - *fuzzy time*, which is timing information that has degraded resolution and/or has been randomly perturbed, or
  - *virtual time*, where the value returned to a principal P is determined only by the number of instructions P has executed.

- Unpredictable delays would be added to the delivery of interrupts and system services to user-mode software.

The two defenses described above in Bandwidth Bounds on Timing Channels have limitations, though. First, defense (i) can result in lower resource utilization. This is because these schedules cause principals requesting operations to be unnecessarily delayed while the resource is allocated to a principal not needing it. Second, attackers can defeat the random variations used in defense (ii) by running multiple experiments and computing an average over those.

Modulation enabled by state. Any part of the system state could support a covert channel if this part of the state can be modulated and monitored by actions that attackers instigate. Often, modulation and monitoring will be side effects of operations intended for other purposes. Here are some examples.

- *create/delete named instances of objects.* Information can be transmitted through the choice of name that an attacker gives to an object instance. Monitoring can be performed if operations are available to indicate whether an object having a given name exists.

- *allocate/deallocate from resource pools.* Information can be transmitted by the quantity being allocated or deallocated to a process the attacker controls. Monitoring can be performed if operations are available to report currently allocated or available capacity.

- *acquire/release locks.* Information can be transmitted by the choice of which exclusive locks are held by a process the attacker controls. Monitoring can be performed by an attacker-controlled process that attempts to acquire a given lock, thereby learning whether some other process already holds that lock.

- *append information to a log.* Information can be transmitted by using a log that records indications of actions undertaken by an attacker-controlled process. Monitoring can be performed if operations are available to retrieve the log contents or to initiate execution that is affected by the log contents.

- *accesses to memory.* Information can be transmitted by the attacker's choice of which address to access. With virtual memory, completion of an access ensures that some page frame will contain the contents of the page containing that address; with a main-memory cache, completion of the access ensures that the cache block for that address will be present in a main-memory cache. In both cases, monitoring can be performed by measuring access delays for a subsequent memory access to that address.

Some covert channels transmit information only between principals executing on the same computer; other covert channels can be used to reach more distant principals. For example, an operation to create files in a local file system creates a covert channel between principals running on the same computer; with a network file server, file creation would support communication between principals running on any client of that file server; and an operation to add a new DNS name to the Internet's Domain Name Server can function as a covert channel between any principals on any computer connected to the Internet.

State and messages that principals are using to communicate with each other can be exploited to create covert channels. Here are some examples:

- State and messages are considered equivalent by a system if they differ only in the values of "unused" fields. Therefore, a covert channel can be created if an unused field is modulated and read by by an attacker.

- Most systems ignore formatting and spellings in documents they store, send, or print. So, an attacker can transmit information through the choice of formatting and/or spelling used in documents.

- Audio files containing samples that differs only in their least-significant bits will sound the same to human listeners. An attacker could change those low-order bits to represent information for transmission.[2] Monitors would be able to recover this transmitted information by inspecting the file contents. The same covert channel construction works for image files.

Modulation from Speculative Execution. A processors instruction set architecture (ISA) is a document that describes (i) the instructions that processor can execute, (ii) the state—called the processor's *architectural state*—that those instructions read and write, and (iii) the changes to architectural state caused by executing sequences of instructions. Implementations of an ISA also might include additional *microarchitectural state* in order to facilitate improved performance. The program counter, general-purpose registers, and main memory are architectural state; a main-memory cache, if present, is microarchitectural state.

An ISA typically will instruct programmers of a processor[3] to assume that instructions are executed sequentially, indivisibly, and at an unknown rate. However, to avoid idle periods due to high memory-access times, ISA implementations often employ *speculative execution.* Speculative execution predicts what values will be fetched from memory, with the effects of this and subsequent execution then reversed if the memory fetches provide different values than what was predicted. A conditional branch, for example, might be taken because that branch almost always has been taken, thus avoiding delays to retrieve from memory values appearing in the branch condition. If predictions are correct often enough then reversal of execution will be infrequent and speculative execution leads to higher throughput.

[2]Such schemes are used in *steganography*, which is the art and science of concealing secret information within non secret data.

[3]We are using the term "processor" to indicate a single-core uniprocessor.

Speculative Execution. Execution of an instruction ι is started before completing execution of all instructions ι' that will write the values needed for executing ι. Early execution of ι is made possible by having the processor predict the values that each ι' will write. If any of the ι' subsequently writes a different value than was predicted, then all writes by ι to architectural state are undone, and execution of ι is repeated (using the correct values). In addition, writes are reordered, if necessary, so that all of the writes by the ι' reach memory before the writes by ι. □

Implementation of speculative execution requires mechanisms for predicting the effects of instruction execution and requires mechanisms for undoing updates to architectural state. The mechanisms to predict branch outcomes and targets, values and addresses to be loaded, and return addresses, use information about past program behavior; that information is kept in microarchitectural state. The mechanisms for undoing updates by an instruction ι use other microarchitectural state to store a copy of the old value for any architectural state that execution of ι updated.

An execution of instruction ι is deemed to have been *transient* if the writes it performed to architectural state had to be undone because values used in the execution of ι were based on a misprediction. The goal with speculative execution of a program P is to produce the same architectural state as a strictly sequential execution of P would produce. That goal is trivially satisfied if the insertion of transient instruction executions into a strictly sequential execution of P does not affect the architectural state that P produces. Such programs are the expected workload for a processor that employs speculative execution.

Programs that are sensitive to transient instruction executions remain possible, though. With these programs, variation in microarchitectural state is being translated into variation in architectural state. This allows covert channels that compromise address space isolation to be created by attackers:

- To transmit the value stored in location L in some address space $\mathcal{A}$, the attacker instigates execution of a modulator $M_{\mathcal{A}}$. $M_{\mathcal{A}}$ changes the microarchitectural state according to the value stored by L.
- To receive a value, the attacker executes a detector D that is sensitive to changes in microarchitectural state caused by executions of $M_{\mathcal{A}}$. D might be executed within address space $\mathcal{A}$ or within some other address space.

To create a modulator within an address space $\mathcal{A}$, the attacker finds a set of bit strings, where

(i) each bit string would be interpreted as an instruction sequence,

(ii) executing these instruction sequences in the correct order modulates the microarchitectural state according to the value stored in location L of address space $\mathcal{A}$, and

(iii) the attacker can cause the instruction sequences to execute in that correct order.

Speculative execution facilitates the creation of such modulators. First, transient execution is not limited to code contained in $\mathcal{A}$. The instructions could be bit strings in any address space[4] and that span existing instructions, appear as parts of existing instructions, or are within the value a variables is storing. Second, transient execution of an instruction can occur in a state that would not arise during normal execution of the code contained in $\mathcal{A}$.

To make this concrete, we show how an attacker might create a modulator using a segment of code written in a programming language (like C) where successive array elements are stored at successive addresses[5] and explicit bounds-checks on array references must be provided by the programmer. If variable `a1Size` stores the number of elements in array `a1` then condition `x < a1Size` in

if `x < a1Size` **then** `y := a2[a1[x] * 4096]`

ensures that the assignment to `y` is performed only in states where expression `a1[x]` refers to an element of `a1`. However, as we show below, speculative execution allows an attacker to learn the value of any variable stored in memory at an address after `a1[0]`.

First, assume speculative execution does not occur. The above assignment to `y` terminates with `x`, `a1[x]`, and `a2[a1[x] * 4096]` residing in the main-memory cache. Therefore, an attacker can derive possible values that variable `a1[x]` stores by detecting[6] whether a memory location corresponding to `a2[a1[x] * 4096]` has become present in the cache. So an attacker who controls the value in `x` and instigates execution of the above **if** statement learns information about possible values stored in `a1[x]` for $0 \leq$ `x` $<$ `a1Size`.

In systems that implement speculative execution, an attacker can learn even more. The attacker would begin by repeatedly instigating evaluation of the condition `x < a1Size` in states where that condition holds. This execution trains the branch predictor to start executing the assignment to `y` before completing evaluation of the condition `x < a1Size` in the **if**. The assignment to `y` would later be reversed if `x < a1Size` is then found to be *false*, but changes to the main-memory cache by the transient execution to evaluate `a2[a1[x] * 4096]` would not be reversed and could be detected by the attacker. Therefore, if (as before) the value in `x` is attacker controlled then speculative execution of the assignment to `y` can now reveal possible values of `a1[x]` for any value of `x`—not just for values that satisfy $0 \leq$ `x` $<$ `a1Size`. That means speculative execution has enabled the attacker to learn values of any variable stored in memory at an address that appears in memory after[7] the address of `a1[0]`. So the covert channel allows memory isolation to be violated.

[4]Permissions may not be checked before a speculative execution.

[5]For a single-dimension array `a` having n elements, if the address of `a[i]` and the value of `i` are known to an attacker then the attacker can calculate the address of `a[j]` from the value of `j` as well as calculating the value of `j` from the address of `a[j]`.

[6]The attacker measures execution times for an instruction that loads `a2[i * 4096]` for $0 \leq$ `i` ≤ 255—those timings suffice to guess the value of `a1[x]` that brought `a2[i * 4096]` into the cache. See page 357 for a detailed discussion of how to perform such an attack.

[7]On some processors, `add` will wrap-around in response to an overflow. Those processors would allow the attacker to learn any value in memory.

Note, a main-memory cache is not the only microarchitectural state that can be used to implement a covert channel. To use some other part of the micro-architectural state, an attacker finds a way to cause variations in that state (modulation) and to expose those variations as variations in the architectural state (monitoring). For example, a processor's microarchitectural state often maintains measurements of temperature, power consumption, and other physical properties that are affected by computing load and that cause changes to execution speed in order to extend battery life or avoid running chips in high temperatures. So modulation can be performed by varying the computation load and monitoring can be performed by measuring execution speed.

Moreover, the unprogrammed transfers of control that speculative execution can cause often suffice as an implementation of monitoring. With return-oriented programming (ROP), for example, a code segment serving as a *gadget* transfers control to the next gadget by using a `return` instruction to load the program counter with a value on the run-time stack. Speculative execution gives attackers an additional vehicle for implementing a transfer of control between gadgets. By training a predictor for return addresses, transient execution can be leveraged to invoke the next gadget even if the address of that gadget does not appear on run-time stack. However, changes to architectural state do not persist when those changes are made by gadgets invoked in this way; changes to microarchitectural state do persist, though.

13.1.2 Side Channels

A *side channel* is modulated by normal operation of a system, where that modulation discloses unexpected information about the current system state, actions currently being performed, past systems states, and/or actions previously performed.

- With *physical side-channels*, receivers monitor physical phenomena that hardware exhibits while operating.
- With *internal side-channels*, receivers run programs to monitor artifacts produced by the execution of a local or remote system.

A typical computer's power consumption, as well as emissions of RF, light, or sound can create physical side-channels; its main-memory cache, translation lookaside buffer (TLB), branch predictor, instruction cache (I-cache), and contention for other shared resources can create internal side-channels.

A side channel need not transmit a value in order to leak that value. Information about instructions executed or memory accessed can suffice, if that information is correlated with the value being leaked. We see this with an encryption or decryption routine that (like many) proceeds in rounds. Round i inspects bit b_i of a secret key $b_1 b_2 \ldots b_n$ and, depending on that bit's value, accesses a different cell in some table or executes a different sequence of instructions. Thus, an attacker can reconstruct the value of the key by monitoring a side channel that conveys the sequence of table cells accessed or that conveys the sequence of instructions executed over the n rounds. And, as we shall see, such side channels are not unusual.

On some processors, some side channels can be eliminated with *constant-time programming*, which is a set of restrictions to ensure that variation in secret values does not cause variation in memory addresses accessed, instructions executed, or execution times.

Constant-time Programming Restrictions.

(i) Memory addresses read and/or stored during execution may not depend on secret values, so variations in secrets do not cause variations in cache contents.

(ii) Evaluation of any expression controlling a conditional statement or a loop may not depend on the values of secrets, so variations in secret values do not affect what statements are executed.

(iii) Variable-latency instructions (e.g., integer division) may not have secret values as operands, so execution times of statements are unaffected by variations in secret values. □

Not all functions of secret inputs can be programmed in a way that satisfies these restrictions, but most of the important cryptographic algorithms have been (sometimes incurring a performance penalty over implementations that do not satisfy the restrictions). Also, program analyzers exist for certifying that all executions by some source code will comply with the restrictions.

Constant-time Programming Restrictions prevents leaks over a given side channel only if certain assumptions are satisfied by the underlying processor. The assumptions would depend on the processor and the side channel—they surface in proving that Constant-time Programming Restrictions attenuates the side channel of concern. In general, the assumptions would rule out processors whose operation exhibits certain forms of variation, where the problematic variation arises from implementation details not typically discussed in an ISA. One example of such an assumption is that memory must not be compressed or use schemes to eliminate duplicate values—otherwise, the size of what is stored reveals information about what is being stored. A second example is that the processor not skip instructions that, because of the values being manipulated, would have no effect—otherwise, execution time reveals information about inputs to certain instructions. Finally, there might be assumptions related to the side channels of concern. For example, if power consumption can be monitored by attackers then we might have to require that the inputs to an instruction have no effect on the power used during execution of that instruction.

13.1.2.1 Physical Side-Channels

A change to the voltage levels at the inputs and outputs to components in a digital circuit will cause detectable and distinctive changes to that circuit's power consumption. If the change is made abruptly then a distinctive RF signal will be produced, too. Since a digital circuit represents binary values 0 and 1 by different voltage levels, program execution that changes a value also causes an abrupt voltage change. So a processor modulates both its power consumption and its RF emissions in ways that are correlated with the instructions it executes and the values it manipulates. Information about program execution is thus conveyed over these physical side channels. Moreover, even though existing circuit simulators do model this, it is impractical for designers to anticipate and eliminate these side channels because it is difficult to determine when useable information is being revealed about program execution.

Different physical processes are involved, but CRT and flat-panel output devices emit both light and RF that is modulated according to what is being displayed on the screen. Such an RF side channel has been monitored at distances over 1 kilometer by using specialized receivers. And the optical side-channel for a CRT does not require direct observation of the screen: the sequence of pixels being illuminated during each raster scan can be recovered by noting the timings for changes in overall luminosity reflected from walls. Finally, LED indicators that monitor the operation of data communications equipment will often indicate the sequence of values being transferred to/from the device. If those changes in luminosity can be monitored then that optical side channel allows recovery of transmitted and received data.

Acoustic side channels are created too during system operation—not only by the mechanical devices used for input and output but also by vibrations of digital electronic components. Different keys on a mechanical keyboard each will make slightly different sounds when pressed, and inter-keystroke timing for human typists depends (in part) on where the keys are located on the keyboard. So keyboard use modulates an acoustic side-channel that conveys what is typed. Dot matrix printers and impact printers emit differing sounds according to what character is being printed, creating an acoustic side-channel that reveals what is being printed. Capacitors and coils in a regulated power supply vibrate at frequencies that depend on the level of activity in the digital circuit being powered. With some hardware, this relatively low-bandwidth acoustic side channel can be sufficient to allow recovery of an RSA cryptographic key from the acoustic emanations produced by decrypting some adaptively chosen ciphertexts.

Learning Secrets from Physical Side-Channels. To exploit any physical side-channel, the attacker must have a way to receive the signal being modulated. Specialized equipment—e.g., a radio receiver, a power monitor, a microphone, or a photosensor—often is required. Radiated signals that are stronger are less likely to be confused with other signals in the environment. So attackers benefit from closer proximity to the system performing the modulation. Attackers also benefit when the receiver they use to monitor a side channel exhibits higher sensitivity and higher selectivity. Finally, by collecting the signals produced by many runs of a given operation (e.g., many encryptions with a given key), an attacker can use averaging to eliminate various forms of noise.

Virtually all physical side-channel signals combine the *indications* that provide the information an attacker is seeking with other things. An attacker must be able to extract those indications from the rest of side-channel signal. To perform that extraction, an attacker might compare side-channel signals generated during multiple executions differing in controlled ways, or the attacker might train a machine learning system to identify events of interest. Both approaches require side-channel signals for specific inputs. Attackers sometimes can obtain those needed signals directly from a targeted system that is connected to a public network. When such access is not available, then an attacker might build or procure a similar system and perform the experiments using this second system.

Mitigations for Physical Side-Channels. The monitor an attacker is using for a physical side-channel would be controlled by that attacker and likely inaccessible to those in charge of defending the system. So mitigations for physical side-channels must focus on preventing useful information from reaching a monitor.

Attenuation. One way to prevent information conveyed by a physical side-channel signal from reaching an attacker is with attenuation, so monitors will find the signal indistinguishable from noise. RF, sound, and light are forms of electromagnetic radiation. Therefore, the propagation of these signals follows the laws of physics, which offer two ways to cause attenuation.

- *Shielding.* Surround the modulator with shielding. A metal enclosure (solid or fine-gauge screen) that is grounded will attenuate RF, an enclosure made of soft material will absorb sound, and an opaque enclosure will block light. Complete attenuation is, however, difficult to achieve in practice. Enclosures leak some signal due to holes, seams, and construction imperfections.

- *Proximity.* Require monitors to be distant from the modulator. The strength of a radiated signal follows an inverse-square law, so a signal with strength S at the modulator has strength S/d^2 at a monitor located distance d away. However, by collecting and averaging signals produced by many runs of the same operation, attackers often can recover content from extremely weak signals.

So, for example, to use shielding in order to attenuate an RF side-channel signal generated by a digital electronics, we (i) enclose its circuitry in a grounded metal case and (ii) wrap a grounded metal foil or wire braid around each cable connected to the system. And to use proximity, we might locate the system in the middle of a large campus but force attackers to remain outside that campus (using gates and guards to prevent campus entry by untrusted individuals).

With some devices—keyboards and displays, for example—direct human access is essential. To avoid obstructing that access, yet still benefit from shielding, we might incorporate shielding into the walls, windows, and doors of the room or building in which the device is located.[8] Shielding that encloses a room,

[8]In the United States, highly-classified information is suppose to be viewed and discussed

however, will not prevent monitoring by devices that are located within that room but connected outside using a network. Access controls can be used to defend against those attacks. Locks (a physical access control) on the doors can help ensure that only trusted individuals enter the room. In addition, devices in the room that could perform monitoring (e.g., because like most laptops today they include a radio, microphone, or video camera) should run access control software that blocks network connections. Devices inside the room now cannot be used for monitoring, either by attackers inside the room or by attackers at remote locations who receive signals relayed over networks.

Jamming. Corrupting the signal being carried by physical side-channel offers yet another way to thwart attacks. With some physical side-channels, monitors detect only the strongest signal. A defender here could transmit strong signals that convey noise. With other physical side-channels, monitors deliver a single, combined signal that sums all signals having certain characteristics. Again, transmitting additional jamming signals can defeat attackers, although some sophistication may be required in order to prevent attackers from using averaging or more advanced signal-processing techniques in order to identify and remove the jamming.

13.1.2.2 Internal Side-Channels

Invoking an operation may change the system's state, return values, and/or instigate system actions. If any of these effects has been influenced by a prior invocation then information is flowing from one operation invocation (and its invoker) to a subsequent operation invocation (and its invoker). That information flow is revealing parts or properties of an earlier system state.

An implementation of an interface satisfies our definition of a side channel if operation invocations result in information flows that are unexpected. Since the specification for an interface describes the effects that clients should expect, an unexpected information flow occurs if there are effects from invoking an operation that

(i) are not described by the interface specification,

(ii) are detectable to the client performing the invocation, and

(iii) vary in ways that reveal information about past invocations.

Notice, weaker specifications—preferred by implementors, because fewer constraints are imposed on the effects of an operation—offer greater opportunities for effects that satisfy (i) – (iii). Therefore, weaker specifications offer greater opportunities for unexpected information flows.

Most interface specifications do not constrain execution times for operations (condition (i)). That flexibility allows an implementation to reduce execution

only within a SCIF (sensitive compartmented information facility), which is a windowless room or building with sound-proofed doors and with walls that include grounded metal shielding to suppress RF emissions.

times for future operation invocations by storing and reusing results from past operation invocations. Reuse of prior results, however, can create information flows if the execution time for an operation invocation is detectable (condition (ii)) and execution time depends on previous invocations of operations (condition (iii)) because it depends on what prior results are available for reuse. So the three conditions for an unexpected information flow are satisfied.

As an example, a processor's ISA is the specification for an interface whose operations are that processor's instructions. An ISA typically imposes no constraints on execution times for instructions so that hardware designers can hide memory latency by incorporating various mechanisms into the processor's microarchitecture: a main-memory cache, a translation lookaside buffer, an instruction cache, and branch predictors. Each of these mechanisms stores information for possible reuse, and reuse of that information reduces the time to execute an instruction. But that means the execution time for an instruction reveals information about previously executed instructions. So an internal side-channel has been created. Moreover, usual implementations of isolation for virtual machines, processes, and containers do not virtualize the processor microarchitecture. With memory-latency hiding mechanisms in the microarchitecture being shared, an internal side-channel conveys information about execution by each virtual machine to all the others, by each process to all the others, and by each container to all others.

The absence of constraints on execution times for operation invocations also allows interfaces to be implemented with fewer instances of limited-capacity resources. For an ISA realization, hardware-assisted multithreading and multiprocessing can be supported with less chip real estate; for the designer of a higher-level interface, a copy of each resource need not be maintained for each client. However, with resource sharing comes execution delays whenever an attempt is made to access a resource while it is in use. The result is an internal side-channel, since an increased execution time for one thread of execution reveals information about the execution of another thread.

Main-Memory Caches as Internal Side-Channels. Mitigations to eliminate internal side-channels often depend on specifics of an interface or its implementation. Below, we explore one example: main-memory caches. These internal side-channels are particularly important because they have been successfully exploited to leak cryptographic keys. Moreover, defenses that work here often can be used for internal side-channels that arise with other kinds of caches—whether the caches are in hardware (e.g., for address translation) or in software (e.g., storing file blocks to anticipate reads).

Main-Memory Cache Exploitation. A main-memory cache typically comprises a set of *cache lines*. Each cache line stores the contents and starting address of a *cache block*. Cache blocks are small and fixed size (e.g., 64 bytes) main-memory regions, with each b-byte cache block starting at a b-byte boundary. When a main memory address m is sent to the cache, the cache returns the

Prime+Probe. Memory references that cause cache misses for the attacker in step (iii) identify cache blocks that victim P accessed while executing during step (ii).

(i) Attacker accesses a sequence m_1, m_2, ..., m_N of memory addresses that fills the entire cache with attacker's cache blocks.

(ii) Attacker suspends while victim P executes the routine of interest.

(iii) Attacker again accesses m_1, m_2, ..., m_N, and notes cache misses. □

Evict+Time. If victim P's execution time is not increased in step (iii) over that measured for step (i) then the attacker learns that P did not reference specified memory address m.

(i) Attacker measures execution time for some short routine by victim P after the cache has filled by P's previous execution.

(ii) Attacker accesses cache block(s) that would occupy the same cache line(s) as the cache block containing victim P's memory m.

(iii) Attacker measures execution time of the routine by victim P in order to determine if an additional cache miss has occurred. □

Figure 13.1: Cache Exploitation to Spy on Principal P

value at address m in main memory if the cache block containing that address is currently present in some cache line. This is called a *cache hit*. If that cache block is not currently being stored in some cache line—a *cache miss*—then the appropriate cache block is fetched from main memory, copied into some cache line, and the value at address m is returned to the requestor. Because caches have finite size, a cache miss can require evicting the current contents of some cache line in order to make space for the new cache block that is being loaded.

Different cache designs impose different restrictions on which cache lines may store the cache block with a given starting address. Typically, the size of a cache will be much smaller than the size of main memory, the memory a given principal can access is stored in the same cache lines that hold memory other principals (and attackers) access, and attackers know the algorithm for assigning cache blocks to cache lines. So by accessing main memory, an attacker can use execution timings to learn something about other principals' recent main-memory accesses. Notice, the attacker is learning about accesses to main-memory regions that the attacker might not itself be authorized or able to access.

Figure 13.1 sketches two kinds of attacks for transforming a main-memory cache into an internal side-channel.[9] To perform either of these attacks re-

[9]The following characterization is sometimes used in connection with information flows from caches. A *trace driven attack* learns from individual cache hits/misses; a *time driven attack* learns from the effects of cache hits/misses on the aggregate execution time of some code. So Prime+Probe is an example of a trace driven attack, and Evict+Time is an example

quires refining the sketch given in the figure, and that refinement will depend on system specifics. For example, some understanding of the scheduling algorithm that dispatches and suspends execution would be required to implement the synchronization implied for starting step (ii) of Prime+Probe and for step (ii) of Evict+Time. Those details depend on whether there is hardware multi-threading versus multiple cores accessing a single cache in parallel versus a single core where the attacker and its target execute in alternation. Various mechanisms could be used to detect the cache misses required for step (iii) of Prime+Probe: a high-resolution real-time clock, performance counters that count cache misses, and memory that is being repeatedly incremented by some process. Access to a high-resolution real-time clock is useful to implement the run-time measurements in steps (i) and (iii) of Evict+Time.

Prevention of Main-Memory Cache Exploitation. One way to prevent a main-memory cache from leaking secret values is to prevent variations in those secret values from causing variations in the cache blocks present in the main-memory cache.

> **Suppressing Variation in Cache Contents.** On processors where there is a static and fixed mapping from memory addresses to cache blocks, variation in the values of secrets will not cause variation in the sequence of cache blocks accessed during execution if a program satisfies the following restrictions.
>
> (i) Which cache blocks are read and/or stored during execution of each instruction does not depend on secret values.
>
> (ii) The expressions controlling a conditional statement or a loop do not depend on the values of secrets. □

Note the connection to Constant-time Programming Restrictions (page 353) and the explicit assumption about how the cache is implemented.

Differences in what cache blocks are present in a main-memory cache cannot leak secret values if one principal's effects on the cache cannot affect the execution of other principals. Various schemes could be used to create that isolation.

> **Isolation of Cache Contents for Separate Principals.** Memory references made by one principal will have no effect on the cache lines that are visible to any other principal provided:
>
> – *Cache Reset.* All cache lines are reset to a fixed, known value as part of any context switch that changes which principal is executing.
>
> – *Name Mapping.* Each principal uses a disjoint subset of the cache lines. Accesses made by a principal load and use only those cache lines.

of a time driven attack.

  – *Time Multiplexing.* At each context switch, a copy is made of the current cache contents; when execution of a principal is restarted, the cache is restored from that copy. □

To implement Cache Reset, most processors provide a `flush` instruction that clears all main-memory cache lines. Executing `flush`, however, does cause higher latency for main-memory accesses until the cache lines have been refilled, resulting in degraded system performance. Name Mapping and Time Multiplexing are likely to have an even higher performance cost, though. With Name Mapping, only a fraction of the cache is available for the principal that is executing; with Time Multiplexing, the cost of a context switch becomes high. Due to these performance costs, hardware support for Name Mapping and Time Multiplexing is rarely present on modern processors.

The final set of defenses we discuss are designed to prevent monitoring. Two tasks must be performed by an attacker in order to infer secret values from a main-memory cache.

- *Synchronization.* Execute code soon after some target principal P has executed.
- *Cache Probing.* Ascertain whether a specific address that some target principal P can access is currently being stored in a cache line.

Therefore, mechanisms that prevent an attacker from performing one or the other of these tasks would prevent cache exploitation attacks.

An attacker's actions cannot alter which principal will run next if the processor's scheduler chooses nondeterministically from a large set of principals. This defense does have a cost, though. Running the wrong principal next can degrade system performance by causing input/output devices to remain idle and/or by disrupting the temporal locality required for cache effectiveness. System designers are reluctant to sacrifice performance for security, so they tend to favor other defenses.

Cache Probing is feasible for attackers because the following properties are expected to hold for any main-memory cache.

(i) Longer memory-access latencies are exhibited for addresses not present in the cache.

(ii) Any address that a principal P can access will be stored in the same cache line as some set of addresses that the attacker knows and can access.

Property (i), in conjunction with a way to compare elapsed times, allows an attacker to detect whether an address it accesses resides in some cache line. That means an attacker can learn about cache contents by making memory accesses. Due to property (ii), an attacker can access one location in order to learn whether the cache is storing some other location. Therefore, an attacker can ascertain whether some target principal P has not accessed an address α by measuring the response time to access an address α' known to be assigned to the same cache line as the cache block containing α.

But if principals do not have sources of timing information then property (i) cannot be used for determining whether a memory access causes a cache hit or a cache miss. Moreover, as discussed for defense (ii) of Bandwidth Bounds on Timing Channels (page 347), blocking access to timing information on some processors is easily achieved for user-mode code.

Turning now to property (ii) above, observe that attackers benefit when each cache block only ever occupies a unique predetermined cache line. Even here, though, detection of a cache miss by the attacker cannot establish whether a given cache block has been recently referenced by some target principal P, because more than one cache block that P might access would each occupy the same cache line. Use of an *n-way set-associative* cache would raise yet further doubts, because a given cache block now might be stored by any of a given set of n different cache lines. That suggests attackers have more difficulty if the main-memory cache is n-way set-associative.

A second way to interfere with property (ii) is to keep attackers ignorant of what addresses they should probe for learning about memory accesses made by some target principal. We can achieve that effect with the system software responsible for compiling and loading each principal's variables and code. It suffices if, each time the system is restarted, this system software creates a new, random, mapping of instruction sequences and variables to the various cache blocks within a memory region. To ascertain which addresses to access for cache probing, attackers must now run a set of experiments after each system restart.

13.2 Hardware Integrity

Physical access to a computer's internals permits attacks that can circumvent authorization checks implemented by software or by hardware. The obvious defense is to prevent attackers from having that physical access. Some defenses, in addition, generate ineradicable indications of attempted attacks. Such indications are useful, because confidentiality compromises are not always detectable.

13.2.1 Leveraging Location

Walls and locked doors are one way to ensure that a computer is inaccessible to attackers. You might place the computer in a locked machine room, in a (locked when unoccupied) office, or at somebody's home. For portable devices, carrying the device in your pocket or keeping it in your briefcase impedes access by depending on social norms about personal distance and who can access personal property.

When physical access to a computer cannot be blocked, we can deter attackers if they know that evidence is being created to attribute accesses they make. Surveillance could be implemented by video cameras, or surveillance could be performed in person by employees or law enforcement. Surveillance is particularly effective for deterring insiders, where walls and locked doors would not otherwise impede an attacker's actions.

Walls together with video surveillance have been used to create a *private cloud* within a cloud data center. The private cloud comprises computer racks enclosed by a cage, where the cage door remains locked while the enclosed computers are running and for an additional period after those computers have been powered-off. Video surveillance of the metal cage deters people from unlocking and entering a cage while the processors or memories it encloses might hold (unencrypted) confidential data. After power-off, the delay period prior to allowing entry ensures that remnants of confidential data in volatile memory will decay before that memory can be read by somebody who has entered the cage.

Large-scale clouds also can hinder attackers from getting physical access to the computers serving a given customer by keeping secret which computers (from the large number in a data center) are running that customer's computations at any time. Here, secrecy is replacing the walls and locked doors as the impediment to access. By periodically migrating a customer's computation from one set of hardware processors to another, a moving-target defense would also be created.

13.2.2 Enclosure and Construction

The location of a device is not always under our control, and surveillance is not always feasible. A set-top cable box, e-book reader, gaming console, or other consumer electronics for providing access to proprietary digital content usually will be physically accessible to its users, and some of those users could be attackers. Credit-card sized artifacts carried in wallets and used to control access to funds or locks on doors are other examples of devices that could come into the possession of attackers. For these situations, a device's construction is the first line of defense against attacks involving physical access.

Packaging. Packaging can protect a device by making it difficult for an attacker to study the internals or to operate the system while monitoring and/or injecting signals. The starting point for implementing such *tamper resistance* often will be a physical enclosure that is difficult to breach without a physical key or special tool. Since theft of information is not visible, an enclosure might also be designed to make evident that an attack has been attempted. One way to create hardware that is *tamper evident* is by using a frangible or highly finished (e.g., polished or crazed) material for the outer shell of the enclosure, so that attempting a penetration causes irreversible and visible changes to the enclosure's appearance. Unforgeable seals, made with multi-layer paints or tapes, are another way to ensure there will be evidence that an attack has been attempted. Finally, a packaging might be *tamper responding* and activate circuits that erase memory (deleting cryptographic keys or other secrets) or that disable the device (perhaps even detonating a small explosive charge). But if physical access to internals could be needed for maintenance, then some means of access must be available for trusted individuals. The device, however, now becomes vulnerable to abuse by an untrustworthy insider who exploits that access.

Sensors. Sensors are the obvious starting point for creating a packaging that is tamper-responding. Penetration of an enclosure can be detected by putting photocells inside to sense the increased level of light from the outside. Another way to detect penetrations of an enclosure is to line it with a membrane that has been printed with a pattern of conductive ink, so the electrical properties of the membrane change when the membrane is punctured. Radiation sensors and temperature sensors will detect attacks aimed at increasing memory remanence (see §13.3.1) in order to facilitate theft of secrets after a system is powered down. Voltage sensors are useful for detecting attacks that could corrupt operation of the electronic circuitry. And attempts to transport the device elsewhere can be detected by having motion sensors. Sensors do bring challenges, however. First, they require a source of power. Second, there is a risk of false-triggering, so a system must be designed to recover from that.

Potting. An attacker will have a harder time finding and physically accessing specific electronic components and interconnecting wires if that circuitry has been embedded in a block of opaque epoxy potting. Moreover, an attacker seeking to operate a system after modifying its components or connections will be hampered if the required physical access to those components or wires is possible only by first destroying other components that (by design) were positioned to be in the way.

A net of fine wires or other conductive material that surrounds the electronic circuitry before the epoxy potting is added can serve as a further barrier to attackers, if breaking, shorting, or altering any of those paths triggers circuits to erase secrets or otherwise disable the device. In addition, attackers who attempt access to interior components by using chemicals or a laser to dissolve portions of the epoxy potting will be detected if the chemical composition of the epoxy potting is more resistant to solvents and if it expands faster when heated than the material used for the embedded wires.

Means of Physical Attack. Whether a device's construction will succeed as a defense depends on the attacker's access, capabilities, and goals. Unsupervised access enables attackers to operate, disassemble, and/or alter a device. This is not necessarily changed by requiring that access be supervised by guards—guards are unlikely to intercede when an attacker is dressed to resemble a *bona fide* maintenance technician. When access is supervised, though, the length of time available for performing an attack could constrain an attacker.

If an attacker can move a device to a remote site then specialized tools can be employed.[10]

- *Machining.* Access by the attacker to a device's internals is enabled by

[10]These tools are developed for the semiconductor industry to use in analyzing chips. The tools typically are quite expensive to purchase when new, but often can be rented on an hourly basis with no questions asked. In addition, the improved capabilities required for each new generation of chips results in decreased demand for older tools. The older tools then become affordable by attackers.

cutting through a shell or by removing potting material that permeates the insides. The cutting might be performed with (fixed or moving) blades, abrasives, high-velocity streams of water, lasers, sandblasting, or shaped (low power) explosive charges. Chemicals also might be used to dissolve potting material.

- *Probing.* Probes provide a means to inject and/or monitor signals being carried within in the device. A probe might be implemented by a narrow gauge tungsten wire, an ion beam, an electron beam, or a laser. Ion beams, in addition, can reconnect fuse links or make other modifications to a chip's circuitry. Electron beams from a conventional scanning electron microscope can read/write bits in EPROM, EEPROM and RAM memory chips if the chip's surface has been exposed (say, by chemical machining). Because silicon is transparent at infrared frequencies (IR), it is not necessary to expose the chip's surface for an IR laser to read/write that storage.

The goal of a physical attack is an important factor when developing a defense. Some physical attacks are undertaken to extract secrets that a device is storing. A defense here could be to incorporate sensors and logic that causes stored secrets to be erased when the start of an attack is detected. The goal of other physical attacks is reverse-engineering—for cloning a system or for discovering its vulnerabilities. As discussed above, packaging plays the critical role in defending against such attacks.

For some devices, an attack would be deemed a failure if it leaves a system inoperable. An attack that that renders a nuclear weapon inoperable will have failed if the attacker's goal was to cause detonation but will have succeeded if the goal was to prevent detonation. For devices used to control access by consumers to proprietary content, an attack is often considered successful if it extracts secrets being stored—even if the device is destroyed by the attack—because the stolen secrets then can be used to provide unlimited access by using some other device.

13.2.3 *Physical Unclonable Functions

Tamper-resistant packaging would not be needed to protect a circuit that unpredictably altered its state and/or operation in response to any physical accesses by attackers. Providing such functionality for circuitry that stores and/or computes functions of secret values is driving research into the development of *physical unclonable functions* (PUFs). They are not yet ready for general use—and some experts argue that simpler alternatives will always be a more sensible choice. Nevertheless PUFs are an intriguing idea to contemplate.

A PUF is a circuit instance C that translates from some fixed, unmeasurable, and unclonable features of its realization on a specific chip to a function $\mathcal{F}_C(\cdot)$ satisfying the following properties.

- Evaluation of $\mathcal{F}_C(\cdot)$ is *repeatable*—the same value is produced every time $\mathcal{F}_C(x)$ is evaluated with a given input x from its domain.

- The value produced by evaluating $\mathcal{F}_C(x)$ cannot be predicted from invasive or non-invasive measurements of the chip that contains C.
- The value produced by evaluating $\mathcal{F}_C(x)$ changes unpredictably if the chip that contains C is modified or probes are attached.

Thus, $\mathcal{F}_C(\cdot)$ is individual, inherent, and unclonable.

The domain of function $\mathcal{F}_C(\cdot)$ depends on the PUF design. Some designs implement a function that takes no inputs and has a fixed (but unpredictable) output, causing the PUF to behave like a small read-only memory. Other PUF designs implement functions that do take inputs. With a *weak PUF*, the number of possible input values is linearly related to the number of components in the circuit used to realize the PUF; with a *strong PUF*, the number of possible input values is exponential in the number of circuit components. Because it is infeasible to collect the values $\mathcal{F}_C(x)$ for the exponential number of possible inputs x to a strong PUF, a strong PUF can be used to implement challenge-response protocols, where each challenge is used at most once.

Examples of PUF Designs. PUF designs typically employ circuits whose output is determined by differences in signal propagation delays, where the differences in delays arise from uncontrollable aspects of chip fabrication. The output of a PUF thus depends, in part, on where the circuitry is located on some specific chip.

SRAM PUF. A 1-bit SRAM PUF implements a function having as its output the unpredictable, but (apparently) repeatable, value at power-up for a specific (uninitialized) SRAM volatile memory cell. With m of these, we obtain an SRAM PUF that produces an unpredictable but repeatable, instance-specific m-bit output. To obtain a PUF that maps an n-bit input to an m-bit output, it suffices to have (i) a set containing 2^n of these m-bit SRAM PUFs and (ii) a decoder circuit that uses the value of an n-bit input to select an associated PUF from the set.

Arbiter-based PUF. A 1-bit arbiter-based PUF outputs a 0 or 1 according to the faster of a selected pair of signal paths, where an n-bit input defines the segments used to form the two signal paths of the pair. Figure 13.2 gives a design. The output of that PUF is the output of the flip-flop labeled arbiter. That output is determined by the relative arrival times of the signal reaching the flip-flop's clock input (labeled >) versus its D input. These arrival times depend on the sequence of MUXes and interconnects that are traversed. That sequence is determined by the input bit to each MUX—input bit i determines for the MUXes in the i^{th} column whether the input port labeled 1 or the input port labeled 0 is the input that the MUX outputs. So an n-bit input defines one pair of the 2^n possible n-segment signal paths.

Because each of the segments has an unpredictable but fixed delay, each of the n-segment signal paths will have an unpredictable but fixed delay. An n-bit

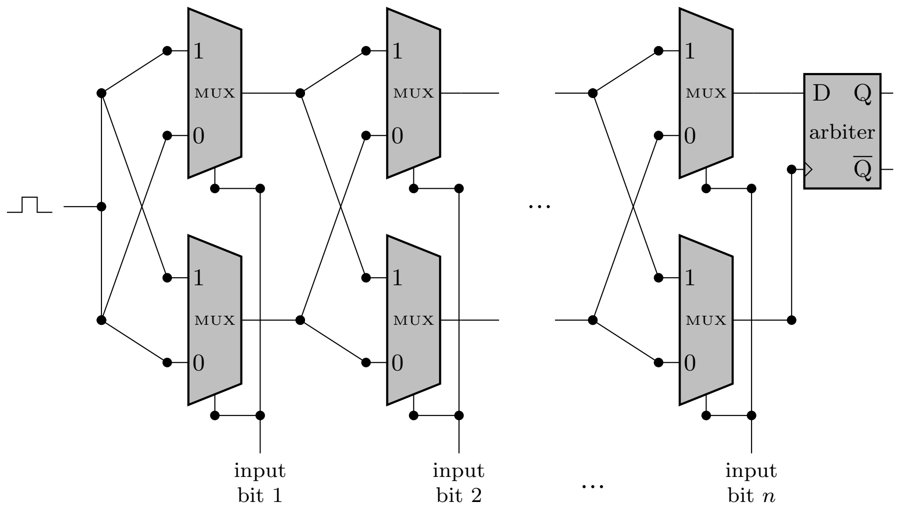

Figure 13.2: Arbiter-based PUF

input selects a specific pair and, thus, always outputs the same unpredictable value. To build a PUF that produces an m-bit output, we use m of these 1-bit arbiter-based PUFs; the i^{th} 1-bit arbiter-based PUF produces bit i of the output.

Ring-Oscillator PUF. The frequency of a ring oscillator is determined by signal propagation delays in a feedback loop, so different instances of a ring-oscillator circuit are likely to have different frequencies. A 1-bit ring-oscillator PUF is built using a set of ring oscillators and some control circuitry. Figure 13.3 illustrates. Each ring oscillator involves a loop that comprises an NAND-gate followed by an even number of inverters. An n-bit input controls a pair of MUXes, causing a pair of ring oscillators to be selected and connected to counters. The frequency of each ring oscillator determines the speed that counter is incremented, so a 0 or 1 will be output by the PUF according to which ring oscillator in the pair has higher frequency. To build a PUF that produces an m-bit output, it suffices to combine m of these 1-bit ring-oscillator PUFs.

PUF Repeatability and Unpredictability. Signal propagation delays in integrated circuits can be affected by operating temperature, power supply voltage, electrical noise, and other aspects of the environment. So a straightforward realization of the above PUF designs might, for a given input x, produce different values for $\mathcal{F}_C(x)$ depending on the current conditions. One way that a PUF circuit can compensate for environmental variation is to have its output depend on delay ratios (which tend to be more stable) rather than absolute delays. Repeatability also can be improved by incorporating error correcting

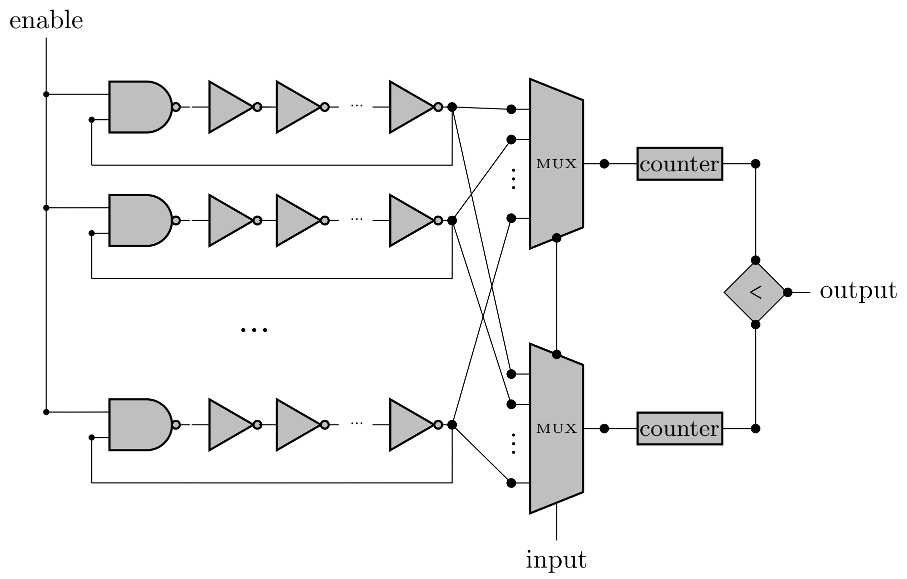

Figure 13.3: Ring-Oscillator PUF

codes into the output of a PUF. Finally, clients that submit an input and then check for a specific ouput can be designed to accept values that differ from the expected response by a small number of bits.

A second key requirement for a PUF realization is to have *unpredictability* of function $\mathcal{F}_C(\cdot)$:

PUF Unpredictability.

- An attacker who learns some set of input-output pairs $\langle x, \mathcal{F}_C(x)\rangle$ must not be able to predict the outputs for other inputs.
- Given a value y that has not yet been output by the PUF, an attacker must not be able to construct an input x satisfying $y = \mathcal{F}_C(x)$. □

The SRAM PUF above satisfies these properties provided the value of each 1-bit SRAM PUF does. This is because each 1-bit SRAM PUF is used in producing the output associated with only one input, so outputs that an attacker has already observed give no information about unseen outputs that the SRAM PUF will produce.

But unpredictability is harder to achieve when a small set of signal propagation delays are being combined in multiple (different) ways, as in the above arbiter-based PUF and ring-oscillator PUF. With an arbiter-based PUF, for example, submitting two inputs that only differ in their i^{th} bit enables an attacker to learn which alternative for the i^{th} segment is faster; $2n$ inputs thus suffice to reveal the faster signal path for all inputs. Under modest assumptions about possible differences in signal-path segment delays, output $\mathcal{F}_C(x)$ now can be used to predict output $\mathcal{F}_C(x')$ for other inputs x' that differ from x at a small number of bit positions.[11] However, unpredictability still can be obtained with this kind of PUF. One solution is to incorporate a cryptographic hash function at the input and/or output of the PUF. Another solution is to restrict the input domain D_C for $\mathcal{F}_C(\cdot)$ to be the subset of inputs x for which $\mathcal{F}_C(x)$ cannot be predicted from $\mathcal{F}_C(x')$ for other inputs x' in D_C; that PUF implementation also would incorporate circuitry that rejects inputs not in set D_C.

PUF Applications. An unpredictable bit string of any desired length can be formed by concatenating the outputs of one or more inputs to a single PUF or to different PUFs. Such a longer bit string could be used, for example, as the unique identifier for a chip instance. Alternatively, if that bit string is kept secret then it can be used to generate a chip-specific symmetric key, a chip-specific public/private key pair, or a chip-specific seed for a random number generator. However, some conditioning of the PUF outputs might be necessary to obtain distributions of values suitable for those applications. The necessary conditioning can be implemented by incorporating a hash function into the final stage of a PUF.

Having a PUF on a chip P generate a chip-specific symmetric key K_P enables secret values used by P to be stored off-chip in a secure way. K_P would be generated whenever a key is needed to encrypt the secret values[12] for storage off-chip or to decrypt content being retrieved. By storing K_P in P's volatile memory only while K_P is needed for performing an encryption or decryption operation, K_P is vulnerable to theft for only short periods. Moreover, an attacker who removes P and probes the PUF used to generate K_P would (i) not be able to learn K_P and (ii) could well cause unpredictable changes to the value being generated for K_P.

Another use for PUF-generated cryptographic keys is to enable authentication of a chip P by its clients. One approach is to provision each client A with a separate symmetric key K_A generated by a weak PUF on P. K_A is then used in a standard shared key authentication protocol: chip P proves to A knowledge of K_A by performing encryption and/or decryption with that key. Notice, if the value of K_A was obtained by A directly from the chip's manufacturer, then this chip authentication protocol even defends against supply-chain attacks that alter P or substitute a different chip for P.

An alternative to using cryptographic functions for authenticating a chip is to leverage the unpredictability of a PUF C located on the chip. Each client A is provisioned with a disjoint set CR_A of challenge/response pairs $\langle c, \mathcal{F}_C(c)\rangle$. To authenticate the chip, a client A removes from CR_A some pair $\langle c, r\rangle$, submits challenge c to the chip, and deems the chip authenticated if the response *resp* that A receives from the chip satisfies $resp = r$. Replay attacks are prevented if

[11] A similar attack is possible for a ring-oscillator PUF. That attack would reduce the number of possibilities by leveraging the transitivity of $<$ used in comparisons of oscillator frequency.

[12] A timestamp should be included in the encrypted information to defend against a rollback attack that replaces the current version of off-chip storage by an older copy.

the client never repeats a challenge. But that means the CR_A sets must be periodically refreshed with another set of fresh challenge/response pairs generated using $\mathcal{F}_C(\cdot)$.

One option for refreshing the CR_A sets, which defends against chip substitution in the supply chain, is for the chip fabricator or system builder to have generated and saved a large set of such pairs produced with the PUF before the chip is put into operation. Another option, which does not defend against chip substitution in the supply chain, is to use the PUF *in situ* for producing these sets of pairs and then securely transfer the new set to a client. This second option requires the capability to transfer CR_A sets off-chip in manner that is confidential and can be authenticated.

13.3 Expectations about Memory and Storage

Programmers make assumptions about the behavior of random-access main memory (RAM) and disks. Those assumptions and attacks to falsify them are the subject of this section. One class of attacks exploits *data remanence*—evidence that reveals information about values that had previously been stored. If those values were not encrypted, then confidentiality can be compromised by an attacker with access to data remanence.[13] A second class of attacks involves writes to one object that also alter information being stored at another object, thereby compromising integrity. Defenses and other mitigations, where they exist, are also discussed in this section.

Assumptions about RAM and Disk. Interfaces to RAM and disks typically provide operations for reading and writing fixed-size, disjoint, addressable objects. Different technologies then lead to storage implementations that differ in cost, performance, as well as other attributes. Programmers, however, will expect `read` and `write` operations to satisfy certain axioms, independent of the technology.

A1: Execution of `read`(x) reveals the value currently stored by x—not a past value of x or the value of some other object.

A2: Execution of `write`(x, val) only changes the value stored by x. The value of no other object changes.

A3: Values stored in *volatile* memory are erased when power is removed; values stored in *stable* storage persist, even after power is removed.

Thus, to erase the value stored at a location x, these axioms imply that a programmer can (i) write a new value to x and depend on A2, or (ii) if x is in volatile memory, then remove power and depend on A3. Also note that axioms A1 and A2 together imply that the value `read`(x) returns cannot be changed by executing `write`(y, val) where x and y identify different objects.

[13]For this reason, newer processors encrypt values written to memory or to other storage.

13.3.1 Attacks on RAM

RAM implemented on an integrated circuit is typically structured as an array of cells, where each cell stores 1 bit. A DRAM (Dynamic Random Access Memory) cell represents that bit by the amount of electrical charge a capacitor stores; an SRAM (Static Random Access Memory) cell represents the bit by carrying current in one of two electrical feedback loops. DRAM cells require less chip area, have higher power consumption, tend to be slower, but are cheaper per bit than SRAM cells. DRAM is typically used for main memory; SRAM is used for CPU registers and cache.

RAM Imprinting. Semiconductor RAM stores information by harnessing certain physical phenomena. Other physical phenomena exist, however, that can be exploited by attackers to cause *RAM imprinting*, whereby information being stored in semiconductor RAM persists long after it should have decayed:

- Low temperatures impede the flow of charge in semiconductors. So by cooling chips that are implementing a volatile memory, an attacker can imprint the contents of that memory for inspection after power has been removed.
- X-ray band irradiation of CMOS RAM transforms the semiconductor in ways that reflect the distribution of charge, are permanent, and are measurable. A memory implemented with these chips may no longer function as expected, but the chips will have recorded—for later inspection—a snapshot of memory.

The obvious defense against attacks that manipulate a memory's physical environment is to thwart physical access by attackers. A tamperproof enclosure is one such defense.[14] Alternatively, a computer could be situated someplace that is inaccessible to attackers. Also, we frustrate an attacker's efforts to remove and read imprinted RAM chips if the chips are permanently glued to the motherboard, so removal destroys them. An operating system can help defend against RAM imprinting caused by low temperatures if the system startup and shutdown code always overwrites all regions of memory that could have been storing secrets. This overwriting forces an attacker (i) to avoid a normal shutdown and (ii) to boot custom code for accessing the RAM chips.

Cold Boot Attacks. Cryptographic keys are often stored in a computer's main memory. Programmers expect this memory to be volatile and, therefore, they assume cryptographic keys stored there will no longer be available after the computer has been powered down. This is not an unreasonable assumption. Main memory invariably is implemented by DRAM, and at standard operating temperatures (25℃–50℃), a powered-off DRAM chip will retain its values for at

[14] X-ray radiation shielding can be unwieldy. Fortunately, X-ray band irradiation attacks can be launched only by well-resourced threats, so X-ray shielding is rarely needed.

most a few seconds before those values decay into random noise. However, when cooled[15] to −50℃, values in a DRAM chip will remain uncorrupted for a minute or more, and when that DRAM chip is submerged in liquid nitrogen (−196℃), data corruption is extremely low, even after an hour. These observations are the basis for so-called *cold boot* attacks, which were developed to recover disk encryption keys from a computer's DRAM main memory after a shutdown or hibernation operation that did not overwrite that memory.

> **Cold Boot Attacks.** Secrets stored in DRAM chips on a running computer can be recovered if an attacker cools those chips and then
> – restarts the computer, booting a kernel that requires only a small memory footprint and that gives the attacker access to the rest of memory, or
> – removes those DRAM chips and inserts them on a computer that gives the attacker access to this memory. □

Other RAM Remanence. If a value V is stored for an extended period at some location in a semiconductor RAM, then electromigration, hot carriers, ionic contamination, and other physical phenomena can change the cell in ways correlated with V. Moreover, these changes remain detectable—even after that location has been overwritten and after the computer is powered-off. So data remanence has been created. To measure some of the changes requires specialized equipment and requires removing the affected RAM chip from the motherboard; only some threats will have those capabilities. Other changes to RAM, though, are directly visible to a running program. For example, a program might be able to deduce what value a given location had stored for a long time simply by reading that location's uninitialized value at power-on.

RAM remanence caused by storing a value for an extended period is more than a theoretical curiosity. It is potentially a significant vulnerability for secure coprocessors, which have separate key memories and are likely to store long-term cryptographic keys and other secrets in the same locations for extended periods. It also is potentially a vulnerability for ordinary operating systems, which typically occupy the same (low) memory region on a given computer and, therefore, use the same fixed memory locations for storing their long-term cryptographic keys and other secrets.

For those of us who do not control the design and fabrication of a system's semiconductor RAM chips, the obvious way to avoid remanence arising from long periods of storing the same value at a given location is to arrange that no memory location holds one of these values for very long. A few minutes is a safe upper bound for storing a value. Two implementations of this defense are:

- Every few minutes, copy the value to a different memory location and then write random values into the memory locations from which the value was just copied.

[15]Cooling to −50℃ can be achieved through evaporation by spraying a DRAM chip with one of the commercially available compressed-air duster products sold to clean equipment.

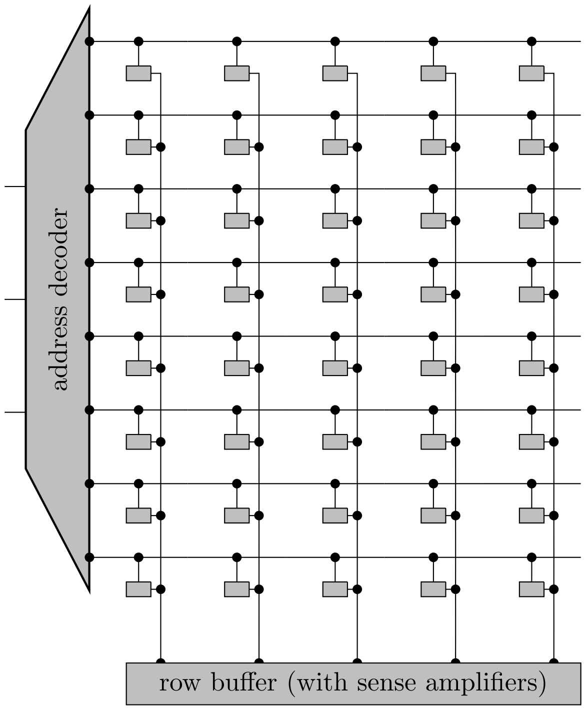

Figure 13.4: DRAM Organization

- Every few minutes, complement the memory locations that are storing the value and update a corresponding *representation indication* that records whether the value or its complement is currently stored. Modify read operations to check the representation indication and, when appropriate, return a complement of the value retrieved from memory.

Row-Hammer Attacks. Increases to the density of DRAM lead to an increase in *disturbance errors*, which are changes to the value being stored in one cell that are caused by accesses to another cell. Disturbance errors violate the axiom (A2, page 369) that writing to a memory location is the only way to change the value this location is storing. Disturbance errors become vulnerabilities if attackers can instigate them to change values being stored by targeted cells. The vulnerability has been present in many of the DRAM chips produced since 2010. With these DRAM chips, cells are internally organized as an array, and making a series of accesses to cells in one row can alter the values being stored by cells in adjacent rows. Such a series of accesses is known as a *row-hammer* attack.

DRAM Internals. To undestand how row-hammer attacks work requires an understanding of DRAM circuitry. As depicted in Figure 13.4, a DRAM

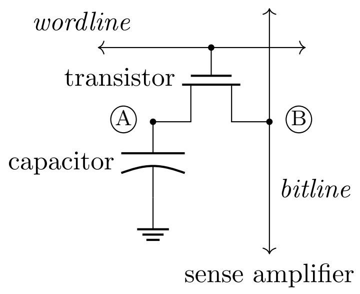

Figure 13.5: Circuit for DRAM Cell

consists of an array of cells (depicted by small rectangles) connected to a *row-buffer* that, for each column in the array, contains a cell and a sense amplifier. DRAM read and write operations access cells in the row-buffer; refresh for a row is done by reloading the entire row from the row-buffer. So the row-buffer at any time will store the same values as the cells in some *selected* row.

All DRAM cells in a given row of the array are connected to a *wordline* for that row, and all DRAM cells in a given column of the array are connected to a *bitline* for that column. Wordlines are driven by an address decoder; it selects a single row for transfer to/from the row-buffer by elevating the voltage on the corresponding wordline and having low voltage on all other wordlines. Each bitline is terminated by a separate *sense amplifier* in the row-buffer.

Figure 13.5 shows a circuit that implements a single DRAM cell. The wordline voltage causes the transistor to control current flow between Ⓐ and Ⓑ and, thus, controls current flow between the capacitor and the bitline. During periods when the wordline voltage is low, current flow between Ⓐ and Ⓑ is blocked, so the capacitor is electrically isolated and retains its charge (except for leakage). An elevated wordline voltage allows current flow between Ⓐ and Ⓑ, enabling the capacitor's charge to be measured and/or changed by the sense amplifier that terminates the bitline.

In order to service a read, write, or refresh operation for the contents of some row r (say), the voltage is elevated on the wordline for r and then the sense amplifier terminating each bitline b performs a sequence of 2 steps.

(i) The sense amplifier measures the voltage on bitline b and compares that value to a threshold. The outcome of that comparison indicates the value of the bit being stored by the capacitor $C_{r,b}$ of the row r cell that is connected to bitline b. A DRAM *true-cell* is storing 1 if the measured voltage was found to be above a threshold and it is storing 0 if the measured voltage was below; in a DRAM *anti-cell*, this representation is inverted.[16]

(ii) After making that voltage measurement, the sense amplifier sets the voltage on bitline b to a value that will cause the charge in $C_{r,b}$ to be set

[16]A DRAM chip might include a mixture of true-cells and anti-cells.

```
L: load  r1,x     access row containing x
   load  r2,y     access row containing y
   flush x        evict x from cache
   flush y        evict y from cache
   mfence         flush pipelines
   jmp L          iterate
```

Figure 13.6: Row-hammer Attack

according to whether the corresponding cell in the row-buffer is storing a 0 or 1.

- For a read operation or a refresh operation, the charge in $C_{r,b}$ is restored, thereby compensating for past charge leakage and for charge loss from performing the measurement in step (i).
- For a write operation, the charge in $C_{r,b}$ is set according to the value of the new bit to be stored.

Physics of Row-hammer Attacks. The laws of physics imply that a change to the current flow on a wordline induces a current flow in all physically parallel wordlines. The induced current is higher in wordlines that are physically closer. So the effect is more pronounced in a higher-density DRAM chips, and it means that a change to the current flow in the wordline for a row r induces the largest flow current in the wordlines for rows $r+1$ and $r-1$. That induced current can cause transistors in row $r+1$ and $r-1$ cells (but possibly cells of other rows too) to allow a short period of modest current flow between the capacitors they control and bitlines. Those current flows are leaking charge from the capacitors.

If enough charge gets leaked from a capacitor $C_{r,b}$ before the next refresh operation is performed for row r, then $C_{r,b}$ would transition from storing an above-threshold charge to storing a below-threshold charge—a disturbance error. For a true-cell, a 1 changes into a 0; for an anti-cell, a 0 changes into a 1. DRAM designers have an incentive to have a large interval between refresh operations, because performing a refresh, read, or write operation requires exclusive use of the row-buffer and, therefore, read and write cannot be overlapped with refresh. So the interval between refresh operations for a given row on a modern DRAM typically is chosen to be just small enough to remediate ordinary charge leakage. That interval does not forestall disturbance errors caused by charge leaks resulting from repeated access to adjacent rows.

Executing a Row-hammer Attack. Disturbance errors that corrupt the values stored by cells in *victim* rows $r-1$ and $r+1$ of a DRAM can be caused when a program repeatedly performs memory accesses that elevate and drop the voltage on the wordline for *aggressor* row r. □

Code that performs a row-hammer attack is given in Figure 13.6. Variable x should be stored in a DRAM row that is adjacent to the victim row; variable `y` should be stored in any other row.[17] The `flush` operations evict `x` and `y` from the cache to ensure that the `load` instructions in each loop iteration elevate the intended wordline voltages by fetching the values from the DRAM. Execution of `mfence` at the end of each iteration drains the pipeline, thereby preventing pipeline logic from suppressing[18] the `load` operations for `r1` and `r2`. By accessing `x` in alternation with `y`, the loop repeatedly applies the voltage for selecting the wordline for the row storing `x`, even with a DRAM implementation where consecutive read accesses to the same row are serviced by the row-buffer without repeatedly selecting and copying that row to the row-buffer.

Defending Against Row-hammer Attacks. Incorporating an error-correcting code (ECC) into each DRAM row might seem like an obvious defense against row-hammer attacks. The number of ECC bits required per row, however, is proportional to the maximum number of cells in a row that could need to be corrected. Since a row-hammer attack can corrupt many cells in a victim row, and any DRAM row could be a victim row, we would require many ECC bits. That storage overhead makes using ECC an impractical defense.

Frequent refresh operations is the other obvious defense. But additional refresh operations consume power and reduce DRAM bandwidth by leaving less time for servicing read and write requests. Moreover, to defend against a row-hammer attack, frequent refresh for all rows is not necessary. Additional refresh is needed only for potential victim rows—any row r adjacent to one or more rows that, in aggregate, were frequently accessed since r was last refreshed.

Various schemes have been suggested for deployment in a DRAM chip or memory controller in order to identify potential victim rows and initiate refresh just for those. Two of the more influential schemes are TRR (Target Row Refresh) and PARA (Probabilistic Adjacent Row Activation). TRR requires additional state; PARA does not require additional state but only gives a probabilistic guarantee. Both schemes have been implemented in some DRAM chips and in memory controllers. TRR support, going by the name refresh management (RFM), appears in recent generations of JDEC (Joint Electron Device Engineering Council) DRAM memory standards.

TRR: A row r' is deemed a potential victim row and refreshed if the number of accesses to any adjacent row has reached a chip-specific threshold MAC within a chip-specific period of length t_{MAW}. Variations include:

- Use the aggregate number of accesses to a set of adjacent rows as the basis for deciding whether a row should be refreshed.

[17]If `x` and `y` are each stored in different rows that are adjacent to the victim row, then the attack is known as *double-sided hammering.* With *many-sided hammering*, there are more than two aggressor rows; it is effective when the geometry of a specific DRAM realization causes additional inductive couplings between wordlines.

[18]Pipeline logic often will skip performing an update to memory or registers if that update will be overwritten before being read. In the loop of Figure 13.6, registers `r1` and `r2` are not read before being loaded again.

- Use sampling on the stream of DRAM accesses to approximate the number of accesses to each row.
- By using a fixed-depth stack, maintain access counts for only some fixed, small number of rows that have received the largest number of accesses within the last t_{MAW} period.

PARA: Whenever a row is activated then, with probability p, refresh one of its two adjacent rows. Thus, the probability that an access to a neighboring row does not cause r to be refreshed is $1-p/2$, and the probability that row r is not refreshed during a row-hammer attack involving N total accesses to rows neighboring r is $(1-p/2)^N$. The table below uses this formula to give the probability that a row-hammer attack involving N accesses would succeed when refresh is expected to be performed only once for every 1000 accesses ($p = .001$). Probability of success for a row-hammer attack is given both for a single refresh period (64 ms) and for a year.

Duration	$N = 50$K	$N = 100$K	$N = 200$K
64 ms	1.4×10^{-11}	1.9×10^{-22}	3.6×10^{-44}
1 year	6.8×10^{-3}	9.4×10^{-14}	1.8×10^{-35}

13.3.2 Attacks on Magnetic Storage

Hysteresis makes magnetization well suited for implementing non-volatile storage. We create a *magnetic storage medium* by applying a thin film of ferromagnetic material to a disk platter or a tape. The ferromagnetic film enables physically disjoint regions—called *domains*—on the surface of that storage medium to assume either of two magnetic *polarities*. A bit string is then represented by the sequence of magnetic polarity changes encountered by a *read/write head* while traveling above the series of domains that constitute a *track* on the storage medium. On magnetic tapes, tracks run parallel to the length; on magnetic disks, tracks are concentric circles.

A storage device is realized by using (i) some magnetic storage medium, (ii) read/write heads to sense and to set the magnetic polarity for domains in a track segment[19] passing underneath, (iii) mechanisms to select which track segments pass underneath the read/write heads, and (iv) electronics for translating a sequence of magnetic polarity changes to/from the string of bits being represented.[20] On a tape drive, one read/write head typically will cover all of the tracks; a motor spools the tape forward or backward, so some selected track segment passes underneath this read/write head. A disk drive typically has a read/write head for each platter surface; all of the platters are rotating in tandem, and there is a mechanism for positioning read/write heads over a selected track on all platters.

[19]Depending on the device, a track segment might be known as a *record*, a *sector*, or a *block*.

[20]Modern magnetic storage devices use run-length limited translation to ensure that frequent changes in the direction of magnetization occur for all bit strings—even bit strings containing long sequences of the same bit.

A write operation is performed by activating the read/write head to set the magnetic polarities for the sequence of domains that are passing underneath. Subsequent read operations recover that bit string by measuring the average magnetization in each domain as it passes underneath the read/write head; a sequence of polarity transitions is constructed from those averages. Because the average magnetization that a read measures for a domain will indicate the polarity of the last magnetization from a write to that domain, axioms A1 and A2 (page 369) that we expect to hold for a storage device should indeed hold.

The averaging performed in a read operation when a domain passes underneath provides a way to compensate for two effects:

- *Positioning Errors.* Positioning a read/write head over a moving track is a mechanical operation. So a slightly different region of the storage medium passes underneath the read/write head each time a given domain is read or written.

- *Partial Magnetizations.* For performing write operations, an electromagnet in the read/write head is used to set the magnetic polarity of domains passing underneath. An electromagnet that is too strong would set the magnetic polarity for a large surrounding region (perhaps including other domains). So a weaker electromagnet is used. But due to natural variations in magnetic susceptibility of materials, using the weaker electromagnet means that portions of a domain passing underneath the read/write head might not get magnetized with a new polarity.

To overcome these effects and have read and write work as expected, (i) the threshold used in deciding a region's magnetic polarity is lowered, and (ii) read/write head positioning is engineered to be accurate enough to ensure significant (but not complete) overlap in the area that passes underneath each time a given domain is being accessed. These implementation tolerances, however, can cause remanence.

Remanence in Magnetic Storage. When attackers can move a magnetic storage medium into a laboratory, analysis with magnetic force microscopy (MFM) becomes possible. MFM creates an image of the medium's surface. Each point in that image depicts the magnetization (strength and polarity) for a small area of that medium's surface—an area far smaller than a domain. Not surprisingly, such an image can be used to recover the values being stored. We average the polarities of points in the image that are located within each domain forming a track, detect transitions from those averages, and use that sequence of transitions to reconstruct the values being stored on the medium.

An image produced by MFM also will contain two kinds of points that are forms of remanence.

- *Magnetized points located outside of any domain.* These points are most likely to be near a track edge or at the boundary between domains within a track. They are caused either by positioning errors or by regions of the storage medium that have high magnetic susceptibility and are near a domain.

- *Disparate points within a domain.* Points arise that are within a domain and have a different magnetic polarity than the average for that domain. This occurs whenever the last write to that domain did not change some small region's magnetic polarity, because that region has low magnetic susceptibility.

What can this remanence reveal? A region containing many disparate points probably is indicating the value being stored prior to the last write.

Magnetic Disk Sanitization. Servers, desktop computers, and personal devices will be decommissioned and replaced from time to time, then repurposed, donated, or discarded. Part of the decommissioning should be to erase information that the computer's disk was storing, since erasure protects the confidentiality of information without requiring assumptions about access controls that computer will enforce in the future.

The procedure to erase the contents of a magnetic disk is called *disk sanitization*. Three approaches[21] are: overwriting, degaussing, and shredding. Which approach is the most appropriate for a given setting will depend on the capabilities of the threat and on whether the disk must still be usable after sanitization has been performed.

Overwriting. With a correctly operating magnetic disk, writing a new value prevents a later read operation from recovering the overwritten value. But a write does not necessarily prevent MFM from recovering the overwritten value and perhaps earlier values, depending on the encoding that was used to represent bit strings.

- With the data encodings used for disks in mid 1990's and earlier, overwriting multiple times with different and unpredictable values is likely to erase a value and any associated remanence.

- With the encodings that achieve high density in modern magnetic disks, recovering old values from remanence is virtually impossible, so overwriting a value once suffices to prevent recovering that value by using MFM.

However, if a given domain has the same polarity for long periods of time and/or the magnetic media is stored at elevated temperatures, then a bias for that polarity is created because the threshold for setting the magnetization to that polarity will be permanently lowered. Later writes will not alter this bias. So,

[21]These approaches also work for sanitizing magnetic tape or other magnetic storage media. They do not work for SSD's (solid-state drives), because the command to delete or update the contents of a block on an SSD leaves the contents of that block intact and, instead, re-maps that block's address. Specialized sanitization commands are therefore typically provided by SSD hardware.

in a laboratory, the original value stored using that domain would thereafter be detectable.

Although remanence is not a problem with modern magnetic disks, they do have features that complicate the use of overwriting for performing disk sanitization.

- Some disk controllers buffer values rather than immediately updating the magnetic storage medium. With such a controller, overwriting (to erase a value) is not guaranteed to perform write operations on the magnetic storage medium, and a sequence of overwrites made to the controller might not translate into the same sequence of writes to the magnetic storage medium.

- Some disks forestall data loss by detecting when a sector or track is becoming marginal and, in response, copy its contents to an alternate region of the magnetic storage medium. Thereafter, accesses to the marginal region are redirected to the new region by the disk controller. So the contents of the sector or track at the time it was deemed marginal cannot be erased by overwriting.

Degaussing. A *degauser* employs an electromagnet to create a strong magnetic field. When the degauser is brought into close proximity to a disk, this strong magnetic field changes the magnetic polarities for all regions of the magnetic storage media. Values represented by magnetic polarities in domains and in remanence are thus destroyed. Use of a degauser, however, also can leave a disk inoperable by (i) corrupting formatting information that had been magnetically encoded on the storage media, and (ii) altering the magnets used in the motors that rotate the disk and that position the read/write heads.

Shredding. The magnetic storage medium is cut into small pieces that cannot be reassembled. To be completely effective, none of these pieces should be large enough for MFM to recover useful information—with modern high-density drives, the pieces must be made quite small. Needless to say, shredding leaves a disk unusable, though some drives do have replaceable magnetic storage media and, therefore, some of the mechanism might be reused with new media.

13.3.3 Remanence Software Generates

Software can generate remanence beyond that exhibited by RAM and magnetic storage. Such software-generated remanence is the subject of this section. Examples will illustrate specific operations that make remanence inevitable. We also will see why the implementation of an interface (along with the implementations of interfaces directly or indirectly invoked) cannot be ignored when trying to avoid remanence. Software engineers favor interfaces that hide implementation details, but security engineers cannot afford to ignore those implementation details when seeking to avoid remanence—unless (as we discuss below) the state is encrypted.

A trivial cause of remanence is operations that we might have expected would provide sanitization but don't. File systems offer good examples. Invoking a file system's `delete` or `write` operations would seem an obvious way for a program to obliterate a file's contents.

- In some file systems, invoking `delete` on a file has no effect on the file contents being stored on disk—it merely removes the file's name from its directory, after copying that file name to the `trash` directory. So `delete` does not implement sanitization, because it does not prevent subsequent access to the file's contents through the `trash` directory.
- In other file systems, a log records the old and new values for each `write`, thereby allowing earlier versions of a file to be recreated when desired. So overwriting a file does not implement sanitization, because it does not prevent subsequent disclosure of file contents that had been overwritten.

You might expect that every file system interface would provide an operation to perform sanitization, even if `delete` or `write` do not have that effect. The prevailing view, however, is to favor user convenience over security. So contemporary file systems make it easy for a user to reverse a `delete` or `write` operation and make it difficult to obliterate information irreversibly. Therefore, sanitization operations are not provided.

A second way that a program might generate remanence is by (i) being assigned access to some stateful resource[22] R, (ii) writing and reading R, and finally (iii) relinquishing access to R. If R is not sanitized before being assigned to some other program P (say), then R will contain remanence that P can read. Note, sanitization of R not only requires overwriting its state; state also would have to be flushed from caches and buffer pools (which might be hidden in lower levels). The operating system would seem a natural place to perform this sanitization, since an operating system allocates resources and has access to memory, buffer pools, and caches. Operating system designers, however, cite performance degradation as the reason for not doing sanitization by default—overwriting state consumes processing time, and flushing caches and buffer pools cause degraded performance when execution restarts. Remanence should thus be expected in a stateful resource R unless each individual program to which R is allocated sanitizes R before relinquishing that access.

A third cause of remanence arises when an implementation maintains copies of state in order to satisfy cost or performance goals.

- A large virtual memory fits into a small real memory because every page is stored in a paging file on disk. Notice, CPU mechanisms that control access to copies of virtual memory pages residing in main memory do not control access to the pages stored on disk.

[22]The resource might be a register, region of real or virtual memory, disk block, or a software abstraction that directly or indirectly uses these hardware storage mechanisms to maintain state.

- A file system is able to deliver faster access by buffering copies of a file's disk blocks in main memory. Notice, access to blocks on disk is controlled by a different mechanism than controls access to the main memory buffers containing disk blocks.

Clients have no direct way to delete or overwrite these state copies. Moreover, because the state copies are invisible to clients, an implementation need not sanitize the state copies when an associated abstraction is sanitized. So the state copies are a form of remanence. Whether that remanence can be accessed by an attacker will depend on whether access to those state copies is being controlled. As illustrated in the virtual memory and file system examples above, the state copies are often protected by a different access control mechanism than used to protect the original.

Finally, remanence—whether created by software or exhibited by RAM or magnetic storage—is harmless if it derives from encrypted state. Moreover, encrypted state can easily be sanitized by deleting or overwriting the key. The costs for encrypting and decrypting state, however, can be significant if done by software. To lower those costs, newer I/O devices and CPUs provide hardware support. A disk controller might, for example, include hardware to generate a symmetric key, thereafter using that key to encrypt blocks as they are written to the disk and to decrypt blocks as they are read. Modern CPU designs (see §2.3) increasingly will generate, store, and use per-principal (sometimes called *enclave*) keys to encrypt, decrypt, and/or digitally sign information to/from the CPU chip, whether that information is en route to a cache, to a page frame in the main memory, to a page file on a disk, or to a network adapter.[23]

Notes and Reading

This chapter discusses a sampling of the assumptions that programmers make and that have proved to be exploitable vulnerabilities; these notes about readings are limited to discussing references that first called out those assumptions. We thus ignore a considerable body of work that reports vulnerabilities in specific systems, gives attacks that exploit these vulnerabilities, and proposes defenses to prevent those attacks.

Covert Channels. The term "covert channel" was first used by Lampson [43] in describing the *confinement problem*—the requirement that client-provided data not be leaked by a service. Lampson illustrates three classes of channels that an attacker might use to perform such a leak: *storage channels* are written by the service but can be read by others, *legitimate channels* are intended to convey information from the service, and *covert channels* are not intended for transferring information but can be repurposed to do so. The meanings of these

[23]Arithmetic calculations and determining transfers of control require plaintext. So the CPU internally uses plaintext, which forces its registers and other on-chip memory to store plaintext.

terms subsequently evolved, and a decade later the Orange Book [17] was stating security requirements in terms of bandwidth limitations for timing channels and storage channels which, its readers are told, constitute the two types of covert channels. That formulation of confinement remains in use today, even though Wray [69] subsequently had showed that some covert channels could be portrayed as being both a timing channel and a storage channel.

Initially, solutions to the confinement problem focused on limiting the mechanisms that system builders could use. As part of an effort at UCLA to build a secure operating system, Popek and Kline [55] suggests the use of virtual time in order to eliminate timing channels. However, as Lipner [46] explains, virtual time can be defeated in settings where end-users can measure response times. Fuzzy time avoids that problem; it is proposed in Hu [30] as a means to reduce the bandwidth of covert timing channels in a secure virtual machine manager kernel being developed for the Digital Equipment Corporation VAX architecture [32, 45]. For blocking storage channels, Lipner [46] suggests enforcing the authorization policy of Bell and LaPadula [11, 10] on all objects named in a formal model of the system. This approach, however, can be unnecessarily restrictive since it does not account for the semantics of operations.

Analysis methods offer system builders the flexibility to eschew restrictive mechanisms where they are not needed. Perhaps the best known of these is the shared resource matrix methodology (SRMM) developed by Kemmerer [34]. To use it, an analyst constructs a table from the (formal or informal) specification for the system. Each row in the table is associated with some attribute of shared state, and each column is associated with a system operation. Entries in each cell indicate whether executing the operation of that column can directly or indirectly read or modify the attribute associated with that row. Certain table configurations, if present, indicate the possibility of a covert storage channel; other configurations indicate the possibility of a covert timing channel.

Wray [69] gives a different table-based analysis method for identifying possible covert timing channels. For this, any generator of detectable events is considered a clock. Each row in the table Wray [69] constructs is associated with a clock that a sender could modulate to transmit a value, and each column in the table is associated with a clock that a receiver uses to detect modulation. Every cell in the table thus corresponds to a potential timing channel.

Unfortunately, any analysis method that depends on people to provide a system description risks being inaccurate or incomplete—there is no guarantee that the input will be accurate and complete account of the system to be analyzed. These difficulties would seem to be remedied by using system source code as the input to an analysis method. But automated methods that use system source code as the input risk being conservative (hence incomplete), because program analysis to deduce whether specific information flows occur is an undecidable question.

Covert channels are often surprising, since most people think about intended uses of given functionality rather than thinking about ways that functionality might be repurposed. The chapter describes only a few possible covert channels. One of them—abuse of speculative execution, first proposed in Kocher et al [37]— at first might seem quite complicated. (The example on page 351 is Spectre Variant 1 from Kocher et al [37].) Concern about speculative execution attacks not misplaced, because little can be done in software to effect a defense. This is because speculative execution skips explicit tests that a programmer might add, and instructions used in an attack need not even appear in the code for a system.

Side Channels. NSA's declassified history [51] of TEMPEST (Telecommunications Electronics Material Protected from Emanating Spurious Transmissions) recounts how Bell Labs engineers in 1943 had discovered that plaintext could be recovered from RF signals being emitted by 131-B2 encryption hardware. The NSA document goes on to say that those side-channel attacks were forgotten after the war ended, to be rediscovered by the CIA[24] in 1951, leading to classified standards for shielding and distancing of devices being used to communicate classified information. Elements of U.S. and NATO standards for what is now called EMSEC (Emissions Security) remain classified, probably to avoid revealing information about current capabilities for exploiting emissions.

As long as information about EMSEC attacks remained classified, few would be aware that such attacks were possible or how to perform them. A 1985 (unclassified) paper by Wim van Eck [65], working at the Netherlands PTT, changed that. It described a low-cost way that RF emissions could be exploited to reconstruct the text appearing on a CRT display, making EMSEC attacks available to any adversary. Van Eck's paper not only suggested the obvious defenses (shielding to attenuate the signal and adding noise to obscure it) but also suggested a novel defense: instead of rendering the scan lines in the usual order, use a secret to determine a permutation on the order in which the scan lines are rendered. Additional defenses were subsequently described in Kuhn's 2003 Ph.D. dissertation [42] at University of Cambridge: for a CRT display, RF emissions could be reduced by altering the shapes of the characters being displayed; for a flat-panel display, adding random, low-order bits to the color combinations used for displaying text could frustrate attempts to reconstruct text from RF emissions.

Exploits involving optical emissions are first reported in the open literature by Loughry and Umphress in a paper [47] describing how to recover transmitted data by monitoring LED status indicators on modems or other data communications equipment. Independently, Kuhn [41] explores optical eavesdropping on CRT displays by attackers who do not have a direct line of sight to the screen. Kuhn's attacks recover the contents of a CRT screen by measuring the sequence of changes to overall (perhaps reflected) luminosity, since that sequence of changes reveals which pixels are being excited in each scan line.

Within the computer security research community, early studies of acoustic

[24]At some point, the Soviet Union also became aware that emissions were a vulnerability. The standards for suppression of radio frequency interference they published in 1954 were mysteriously more stringent for communications equipment than other equipment. And in the mid-1960's, evidence was uncovered that the Soviet Union was monitoring RF and acoustic emissions from devices inside the U.S. Embassy in Moscow.

side channels focused on keyboard emissions. Asonov and Agrawal [4] trained a neural network to recover keypresses from the sounds generated by an IBM PC keyboard.[25] Once trained, this neural network worked for all typists and for all instances of a given keyboard make and model, but retraining was required for different keyboard models. Follow-on work by others focused on improvements to training. For example, having training data be labeled (which is required in [4]) is shown to be unnecessary in Zhuang, Zhou and Tygar [72], and the use of short sequences of keypresses (instead of individual keypresses) for training is investigated in Berger, Wool, and Yaedor [12]. Much work followed; space limitations preclude giving a survey here.

Keyboards are not the only source of acoustic emissions in a computing system. Briol [15] is the first to observe that printing different characters on a dot matrix printer produces acoustic emissions having different waveforms.[26] That paper, however, does not give attacks to recover what is being printed from those "compromising sonsorous [*sic*] vibrations" [15]. Subsequently, Backes et al. [5] does create attacks by leveraging the intervening two decades of developments in machine learning, feature extraction in music and speech, and speech recognition. But mechanical devices are not the only source of problematic acoustic emissions in a computing system. Genken, Shamir and Tromer [23] shows how a 4096-bit RSA key can be recovered by recording and analyzing hum caused by the capacitors and coils in the regulated power supply for a CPU.

Physical side-channels begin to have commercial significance with the deployment of smartcards that controlled access to value by using secret keys and cryptographic operations.[27] To asses the risk of incurring losses required understanding what side-channel attacks would be feasible for threats having physical access to the smartcard. With that goal in mind, Kocher [38] shows how to perform timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other cryptosystems. That paper also suggests some defenses: making all cryptographic operations take the same amount of time, depriving attackers access to an accurate time source, or performing cryptographic operations on data that has been blinded. Constant-time cryptography seemed the most promising of those defenses, so researchers undertook developing constant-time implementations of various cryptographic operations (e.g., Bernstien et al. [14]) as well as methods for analyzing a program to determine if its executions are constant time (e.g., Barth et al. [7]).

Attacks that exploit other side-channels were also explored. Kocher et al. [39] leads the way with DES attacks based on monitoring power-consumption.

[25]An attack previously published in Song, Wagner and Tian [62] had exploited differences in times between key presses (which varied according to the placement of those keys on a keyboard) to reduce the search space for recovering a password typed over an SSH connection. When using an SSH connection, each character typed would be encrypted and transmitted in a separate packet, and the time between transmission of those packets was a good estimate for the time between the key presses that generated those packets.

[26]Briol [15] also showed that printing different characters produced different wave-forms for power consumption and for RF emissions.

[27]Télécerte, launched in 1983 for payment in French pay phones, was the first large-scale use of smartcards cards.

Quisquater and Samyde [57, 58] subsequently discusses how those attacks could be transformed into attacks that use electromagnetic emissions instead of power consumption; actual attacks to retrieve key material being employed by smartcard implementations of DES and RSA are described by Gandolfi, Mourtel, and Francis [18]. Agrawal et al. [3] gives a systematic account of side-channel attacks based on electromagnetic emissions from semiconductor devices.

Attacks that exploit the specifics of a cryptosystem's implementation are not limited to smartcards or to exploiting physical side-channels. Kelsey et al. [33], which generalizes Kocher's timing attacks to implementations of product ciphers, suggests that the information needed by an attacker could come from measuring a cache-hit ratio during an execution. (Hu [31] had already shown how a shared main-memory cache could become a covert channel on a mainframe computer.) Side-channel attacks that used main-memory caches begin with attacks on DES by Page [54] and Tsunoo et al. [64]. The formulation of such cache-based timing attacks in terms of Evict+Time and Prime+Probe is introduced in Osvik, Shamir and Tromer [53] in connection with attacks on AES implementations that they give.

Internal side-channels also can be created using parts of a processor's microarchitecture that are shared with a program executing cryptographic operations. A 2007 attack in Aciiçmez, Koç, and Seifert [2] recovers keys from executions of RSA by using the branch predictor as a side channel; an attack in Acıiçmez [1] uses the instruction cache (I-cache) as a side channel to attack OpenSSL. Gras et al. [24] uses a translation lookaside buffer (TLB) to leak keys RSA and EdDSA secret keys. The survey by Ge et al. [22] discusses these, many other attacks, and the various defenses that have been proposed.

Attacks on cryptosystem implementations by using internal side-channels typically infer details about an execution of some cryptographic operation from measurements of execution timings. Brumley and Boneh [16] is the first to demonstrate that the timing measurement can be done remotely—an attacker learns the private key of an SSL server by remotely measuring the time that server takes to respond to decryption queries. Subsequently, Bernstein [13] devises a cache-timing attack, where an attacker located elsewhere in the network detects the timing variations needed to recover an AES key that is being used.

Some attacks involving internal side-channels require the attacker to execute a program on the processor executing some cryptographic operation being attacked. Clouds, which typically do not give users control over processor assignments, would therefore seem to offer a safe hosting environment. Ristenpart et al. [59] shows that they don't—with high probability, an attacker can cause a program being run in such a cloud to get assigned to the processor executing some target of attack. Stronger isolation of virtual machines, processes, or compartments could eliminate internal side-channels that depend on the sender and receiver being co-resident. There have been numerous proposals in support, but thus far they have not been embraced by hardware and system software producers.

Tamperproof Processors. Programmers assume that computer hardware will function as expected. But even mainframe computers in locked machine rooms are vulnerable to tampering during regularly-scheduled maintenance periods. Molho [49] was among the first to write about this, discussing how a maintenance technician, by changing just a few wires in an IBM 360/50 mainframe, could disable that processor's circuits for restricting execution of privileged instructions and for protecting parts of memory.

With the advent of small and low cost microprocessors, computers no longer had to be housed in machine rooms. Cash machines, smartcards, and personal computers were now feasible.[28] Because legitimate users required physical access in order to operate this equipment, attackers had physical access, too. Price [56] seems the first to describe many of the now standard approaches for building tamperproof packagings: potting to obstruct access to components and connections, fine wires in that potting to detect penetration attempts, and different positioning for those wires in each chip instance so that attackers who deconstruct one instance of a chip do not learn information that helps in compromising another instance.

IBM researchers were among the first to build and write about a system that used these methods. The prevailing wisdom was that sales of personal computers would be fueled by a rich market for application software, but developers would be reluctant to invest in building such software absent barriers to prevent illicit copying. Software alone could not solve that piracy problem, since its execution could be subverted by tampering with the hardware. Tamperproof hardware was required. So a group at the Yorktown research laboratory developed the tamperproof μABYSS coprocessor [66] to support their ABYSS ($\underline{\text{A}}$ $\underline{\text{B}}$asic $\underline{\text{Y}}$orktown $\underline{\text{S}}$ecurity $\underline{\text{S}}$ystem) architecture [68] for preventing illicit distribution of personal computer software. Although μABYSS never became a product, it was a precursor to a series of tamperproof crypto-coprocessor products from IBM that enjoyed considerable success in the market.

Tamperproof packaging interferes with only certain means for getting physical access to monitor or change the operation of a circuit. Physical access to a circuit becomes a less significant vulnerability, however, if a circuit's realization is itself tamperproof. Physically unclonable functions (PUFs) are a class of intrinsically tamperproof circuit realizations. Gassend et al. [21] introduced the term PUF and proposed incorporating a PUF into an integrated circuit. Those authors, originally seeking a way to authenticate silicon chips, also discuss in [20] how to use a PUF in solving other security problems. For more details about PUFS, see the primer by Bauer and Hamlet [9] or the tutorial by Herder et al. [29]. Figures 13.2 and 13.3 are based on Suh and Devadas [63].

The idea of capturing unique physical properties of an inanimate object in a digital signature had been brought to the cryptography community a decade earlier by Simmons [60], who described two schemes developed in the 1980's at Sandia by a colleague Don Bauder. One scheme facilitated detection of counter-

[28]The classified literature doubtless also discusses uses for microprocessors in controlling weapons systems and secure communication.

feit paper money [8]; the other—a *reflective particle tag* (RPT)—enabled inventorying nuclear weapons in support of the Intermediate-range Nuclear Forces (INF) treaty. Oliver and Fritz Kömmerling took the next step in a December 2000 patent filing [40] that showed how properties in the packaging or substate for an integrated circuit could be used for a tamperproof approach to generating a cryptographic key—in effect, describing a special-purpose PUF.

Attacks on Memory. The effects of cooling on RAM remanence was reported by Link and May in 1979 [44], and those effects were reconfirmed for circa 1988 commercially available DRAM in Wyns et al. [71, 70] and for circa 2002 commercially available SRAM in Skorobogatov [61]. Weingart [66] in 1987 suggests that freezing the RAM chips implementing a volatile memory would allow an attacker to recover secrets that had been stored before the computer was powered down. Actual attacks to recover encryption keys from DRAM after a computer had been powered down were demonstrated by a group at Princeton [28]; the term "cold boot attack" was introduced in that paper.

Cooling is not the only physical effect that attackers can use to prolong remanence for RAM chips. Weingart [67] notes that X-ray band irradiation of a RAM chip will imprint the chip's contents for later inspection. He also suggests that short duration high-voltage spikes might have the same effect. See Gutmann [27] for explanations of how various physical phenomena cause data remanence in semiconductor memory devices.

Kim et al. [35], describes why row-hammer attacks ought to be possible, gave code (the basis for Figure 13.6) to cause these targeted disturbance errors, and analyzed possible defenses. Probabilistic Adjacent Row Activation (PARA) was introduced in that paper as a lower-cost alternative to row-hammer defenses that use row-access counts (or approximations) for instigating additional refresh operations of likely victims. DRAM disturbance errors, however, had been observed starting with the first commercially available DRAM, the Intel 1103 introduced in October 1970. By 1999, Van de Goor and de Neef [52] were considering a "hammer test" in experiments to assess ways to evaluate DRAM chip reliability; the hammer test would write each cell 1000 times and then verify that nearby cells were not disturbed.[29] The goal of avoiding disturbance errors for all workloads—especially given expectations of the higher-density chips to come—resulted in Intel engineers developing schemes that used row-access counts to instigate additional row refreshes. These schemes are described in patent applications [6, 25] that were filed in 2012 (becoming public only some months after Kim et al. [35] had been submitted for publication). The Intel work doubtless is the basis for Target Row Refresh (TRR) found in the various DRAM standards from JDEC (Joint Electron Device Engineering Council).

Since virtually all systems included DRAM, the revelations in Kim et al. [35] prompted the security community to engage. A 2020 retrospective by Mutlu and

[29]The term "hammer test" had further evolved by August 2013, where we see a slide deck [48] for a MemCon talk that is using the term "row hammer" for this source of DRAM disturbance errors.

Kim [50] surveys that work, including how attacks to flip a bit can be leveraged for taking control of a system, how software might be modified to resist row-hammer attacks, and various proposals for hardware defenses. Various TRR versions are being implemented today by DRAM manufacturers—probably because TRR is part of JDEC DRAM standards and involves no changes to other hardware or to software. Frigo et al. [50] measure the effectiveness of these TRR implementations in defending against row-hammer attacks, finding that the defenses circa 2020 were not completely effective.

Attacks on Magnetic Storage. Guttmann [26] discusses the relevant physics foundations and engineering challenges for implementing magnetic storage (circa 1996), how remanance is being produced, recovery of values using magnetic force microscopy and other laboratory instrumentation, and protocols for erasing values. Although some of that material does not apply to newer storage technologies, that paper remains an important resource, and Guttmann has been providing updates on his web site. Generally accepted guidance for sanitization of magnetic media is given by NIST [36]. This guidance is not followed often enough, though, as a study by Garfinkel and Shelat [19] showed. In that study, the authors collected a large number of decommissioned computers and, because disk sanitization had been performed poorly or not at all, were able to recover confidential personal from the disks.

Bibliography

[1] Onur Aciiçme. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242, Berlin, Heidelberg, 2007. Springer.

[3] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side—Channel(s). In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45, Berlin, Heidelberg, 2003. Springer.

[4] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 3–11. IEEE Computer Society, May 2004.

[5] Michael Backes, Markus Dürmuth, Gerling Sebastian, Manfred Pinkal, and Caroline Sporleder. Acoustic side-channel attacks on printers. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, USA, August 2010. USENIX Association.

[6] Kuljit S. Bains, John B. Halbert, Christopher P. Mozak, Theordore Z. Schoenborn, and Zvika Greenfield. Row hammer refresh command. US Patent Application Publication US 2014/0006703 A1. Filed June 30 2012, publication date January 2, 2014.

[7] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279. Association for Computing Machinery, 2014.

[8] D. W. Bauder. An anti-counterfeiting concept for currency systems. Technical Report PTK–11990, Sandia National Labs, Albuquerque, NM, 1983.

[9] Todd Bauer and Jason Hamlet. Physical unclonable functions: A primer. *IEEE Security and Privacy*, 12(6):97–1015, November/December 2014.

[10] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: A mathematical model. Technical Report ESD-TR-73-278, Volume II, Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, November 1973.

[11] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Volume I, Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, November 1973.

[12] Yigael Berger, Avishai Wool, and Arie Yeredor. Dictionary attacks using keyboard acoustic emanations. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 245–254. Association for Computing Machinery, 2006.

[13] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[14] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastian Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer-Verlag, 2013.

[15] Roland Briol. Emanation: How to keep your data confidential. In *Proceedings Symposium on Electromagnetic Security for Information Protection*, SEPI '91, pages 225–234, November 1991.

[16] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium (USENIX Security 03)*, pages 1–14, Washington, D.C., August 2003. USENIX Association.

[17] National Computer Security Center. Trusted computer system evaluation criteria. Technical Report CSC-STD-001-83, Department of Defense, August 1983.

[18] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261, Berlin, Heidelberg, May 2001. Springer.

[19] Simson L. Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17–27, January 2003.

[20] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Proceedings of* 18^{th} *Annual Computer Security Applications Conference*, CSAC '02, pages 149–160. IEEE Computer Society Press, December 2002.

[21] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160. Association for Computing Machinery, November 2002.

[22] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[23] Daniel Genkin, Adi Shamir, and Tromer Eran. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 444–461. Springer, August 2014.

[24] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association.

[25] Zvika Greenfield, John B. Halbert, and Kuljit S. Bains. Method, apparatus and system for determining a count of accesses to a row of memory. US Patent Application Publication US 2014/0085995 A1. Filed September 25 2012, publication date March 27, 2014.

[26] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th Conference on USENIX Security Symposium*, USA, July 1996. USENIX Association.

[27] Peter Gutmann. Data remanence in semiconductor devices. In *10th USENIX Security Symposium (USENIX Security 01)*, Washington, D.C., August 2001. USENIX Association.

[28] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA, July 2008. USENIX Association.

[29] Charles Herder, Meng-Day (Mandel) Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[30] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 8–20. IEEE Computer Society Press, May 1991.

[31] Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, SP '92, pages 52–61. IEEE Computer Society Press, 1992.

[32] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19. IEEE Computer Society, May 1990.

[33] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security — ESORICS 98*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110, Berlin, Heidelberg, 1998. Springer.

[34] Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[35] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372. IEEE Press, 2014.

[36] Richard Kissel, Andrew Regenscheid, Matthew Scholl, and Kevin Stine. Guidelines for media sanitization. Technical Report NIST Special Publication 800–88, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, Gaithersburg, Maryland, December 2004.

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, June 2020.

[38] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference (CRYPTO '96)*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[39] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[40] Oliver Kömmerling and Fritz Kömmerling. Anti tamper encapsulation for an integrated circuit. U.S. Patent 7,005,733 B2. Filed December 26, 2000, issued February 28, 2006.

[41] Markus G. Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 3–18. IEEE Computer Society, May 2002.

[42] Markus G. Kuhn. *Compromising emanations: Eavesdropping risks of computer displays.* PhD thesis, University of Cambridge, Computer Laboratory, December 2003. Technical report UCAM-CL-TR-577.

[43] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[44] W. Link and H. May. Eigenschaften von MOS-ein-transistorspeicherzellen bei tiefen temperaturen. *Archiv für Etektronik und Übertragungstechnik*, 33, June 1979.

[45] Steve Lipner, Trent Jaeger, and Mary Ellen Zurko. Lessons from VAX/SVS for high-assurance VM systems. *IEEE Security and Privacy*, 10(6):26–35, 2012.

[46] Steven B. Lipner. A comment on the confinement problem. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 192–196, New York, NY, USA, 1975. Association for Computing Machinery.

[47] Joe Loughry and David A. Umphress. Information leakage from optical emanations. *ACM Transactions on Information Systems Security*, 5(3):262–289, August 2002.

[48] Mike Micheletti. Tuning DDR4 for power and performance. Slides from presentation at MemCon, August 2013. http://cdn.teledynelecroy.com/files/whitepapers/tuningddr4_for_power_performance.pdf.

[49] Lee M. Molho. Hardware aspects of secure computing. In Harry L. Cooke, editor, *Proceedings of the 1970 Spring Joint Computer Conference*, volume 36 of *AFIPS Conference Proceedings*, pages 135–141. AFIPS Press, May 1970.

[50] Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, August 2020.

[51] National Security Agency. TEMPEST: A signal problem. *NSA Cryptologic Spectrum*, 2(3):26–30, Summer 1972. https://cryptome.org/nsa-tempest.pdf.

[52] J. de Neef and A van de Goor. Industrial evaluation of DRAM tests. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 623–630. IEEE Computer Society, March 1999.

[53] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20, Berlin, Heidelberg, 2006. Springer.

[54] Daniel Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Computer Science Department, University of Bristol, 2002.

[55] Gerald J. Popek and Charles S. Kline. Verifiable secure operating system software. In *Proceedings of the National Computer Conference and Exposition*, AFIPS '74, pages 145–151, New York, NY, USA, May 1974. Association for Computing Machinery.

[56] W. L. Price. Physical security of transaction devices. Technical Memo DITC 4/86, National Physical Laboratory, January 1986.

[57] Jean-Jacques Quisquater and David Samyde. A new tool for non-intrusive analysis of smart cards based on electro-magnetic emissions. The SEMA and DEMA methods. Presented at Eurocrypt 2000 Rump Session, May 2000. Burgge, Belgium.

[58] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security,*

Proceedings of International Conference on Research in Smart Cards, E-smart 2001, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210, Berlin, Heidelberg, September 2001. Springer.

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. Association for Computing Machinery.

[60] G. J. Simmons. Identification of data, devices, documents and individuals. In *Proceedings 25th Annual 1991 IEEE International Carnahan Conference on Security Technology*, pages 197–218, 1991.

[61] S. Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, June 2002. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf.

[62] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proceedings of the 10th USENIX Security Symposium*, USENIX Security '01, Washington, D.C., August 2001. USENIX Association.

[63] G. Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 9–14, New York, NY, USA, June 2007. Association for Computing Machinery.

[64] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76, Berlin, Heidelberg, 2003. Springer.

[65] Wim van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269–286, December 1985.

[66] S. H. Weingart. Physical security for the μABYSS system. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 52–52. IEEE Computer Society Press, April 1987.

[67] Steve H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems—CHES 2000*, pages 302–317, Heidelberg, 2000. Springer.

[68] S. R. White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 38–38. IEEE Computer Society Press, April 1987.

[69] John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, May 1991.

[70] P. Wyns and R. L. Anderson amd W. F. DesJardins. Temperature dependence of required refresh time in dynamic random access memories. In *Proceedings Symposium Low Temperature Electronics and High Temperature Superconductors*, volume 88–9, pages 234–239. The Electrochemical Society, 1988.

[71] P. Wyns and R. L. Anderson. Low-temperature operation of silicon dynamic random-access memories. *IEEE Transactions on Electron Devices*, 36(8):1423–1428, 1989.

[72] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 373–382, New York, NY, USA, 2005. Association for Computing Machinery.