Chapter 13

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 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 can be modulated as execution proceeds.

When the receiver executes a program in order to monitor a channel, there are two ways modulation could be sensed:

- *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 connecting a system with its environment is necessary for analyzing the confidentiality of secrets that system stores. Some of those channels will be expected. 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, unexpected, channels are likely to be present, too. To ignore an unexpected channel is equivalent to assuming that the channel cannot be used to leak secrets.

Whether making this assumption creates a vulnerability will depend on the unexpected channel's capacity, what information the channel conveys, and whether the channel can be monitored by the threats of concern. A high-capacity channel would be needed for rapidly leaking images, video, sound, or databases; a low-capacity channel suffices for transmitting a cryptographic key or password. To focus only on high-capacity 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 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 run-time environment, the computer hardware, the underlying physics of the hardware, and/or the properties of materials. Yet, as we shall see, 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 good intuitions and 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 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 for the operations effecting the information transmission to be outside the scope of system authorization mechanisms.[1] The attacker would monitor the covert channel, provide the program that modulates the covert channel, and contrive to have that program be executed. Many covert channels have low capacities. Some covert channels are noisy, but an attacker can compensate for noise by using error-correcting codes (with some reduction in channel capacity) when modulating the covert channel.

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 are being performed to satisfy requests from other principals. The length of such delays usually can be measured, and this length usually can be modulated by varying the requests that are made. So time multiplexing can be abused to create covert timing channels.

[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.

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. If $b = 1$ holds then T runs a compute intensive task; if $b = 0$ holds then T runs an input/output intensive task. And R measures the time that elapses for the execution of a fixed sequence of instructions that requires 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 can bound the capacity 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.

Capacity Bounds on Timing Channels.

(i) Adhere to a schedule that is unaffected by access requests made by principals. Example schedules 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 starting and ending times of the 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 calls that principals can invoke to obtain timing information 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 could be added to the delivery of interrupts and responses by system services.

The defenses described above in Capacity 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 be used to support a covert channel if this part of the state can be modulated and monitored by attackers. Often, the 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 locks are acquired and 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 an attacker's choice of which memory 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 of these 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 can be used to create 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 use to communicate with each other can be exploited to serve as covert channels, in addition. 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 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 differ 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 processor's 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.

> **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 reversed, and execution of ι is repeated (using the correct values). In addition, writes are reordered, as necessary, so that 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 reversing updates to architectural state. The mechanisms to predict branch outcomes and targets, values and addresses to be loaded, and return addresses, use information about

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

[3]Here we use the term "processor" to indicate a single-core uniprocessor.

past program behavior; that information is kept in microarchitectural state. The mechanisms for reversing 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 reversed 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 leads to variation in architectural state. The connection from microarchitectural state to architectural state allows attackers to create covert channels that compromise address space isolation.

- 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 has an interpretation 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 suc-

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

cessive 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

$$\textbf{if}\ \ \texttt{x} < \texttt{a1Size}\ \ \textbf{then}\ \ \texttt{y} := \texttt{a2[a1[x]} * \texttt{4096]}$$

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 $0 \le$ `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 $0 \le$ `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.

Note, 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

[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 $0 \le$ `i` ≤ 255—those timings suffice to guess the value of `a1[x]` that brought `a2[i * 4096]` into the cache. See page 417 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.

changes are made by gadgets invoked in this way; changes to microarchitectural state do persist, though.

Other Abuses of Microarchitectural State. A main-memory cache is not the only microarchitectural state that can be abused to implement a covert channel. To use other parts of the microarchitectural 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.

13.2 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 changes to state or timing.

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 the 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. An attacker, therefore, 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.

Some side channels can be eliminated with *constant-time programming*, which is a (misleadingly named) 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. □

Program analyzers exist for certifying that all executions by some source code will comply with the restrictions. Not all functions of secret inputs can be programmed in a way that complies with the restrictions, although implementations that comply have been developed for most of the important cryptographic algorithms. These implementations invariably are slower (albeit only moderately so) than implementations that do not satisfy the restrictions.

Constant-time Programming Restrictions prevents leaks over a given side channel only if certain properties are satisfied by the underlying processor. Those properties rule out processors whose operation exhibits variation that arises from implementation details not typically discussed in an ISA. One example of such a property is that memory must not be compressed and must not use schemes to eliminate duplicate values—otherwise, the size of what is stored reveals information about what is being stored. A second example of such a property 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 properties 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.

13.2.1 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 will modulate 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 oscillation 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 oscillate at frequencies that depend on the level of activity in the digital circuit being powered. With some hardware, this relatively low-capacity 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 often use averaging to eliminate various forms of noise.

Virtually all physical side-channel signals combine *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 experiments using this second system.

Mitigations for Physical Side-Channels. The monitor that an attacker is using for a physical side-channel would be controlled by that attacker and likely inaccessible to those 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.* 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 typically leak some signal due to holes, seams, and construction imperfections.

- *Proximity.* 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 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, 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.

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

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.

13.2.2 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 operation 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 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 interface 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 interface specifications offer greater opportunities for unexpected information flows.

Most interface specifications do not constrain execution times for operations (condition (i)). That flexibility allows an implementator to reduce execution times for future operation invocations by storing and reusing results from past operation invocations. Reuse of prior results, however, can create information flows since 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 implementations that use 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 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. To make space in the cache for this block, the current contents of some cache line is evicted.

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

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

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

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

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

Evict+Time. If a principal 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 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 P's memory m.

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

Figure 13.1: Cache Exploitation to Spy on Principal P

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 one of these attacks requires refining the sketch, 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.

[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 of a time driven attack.

Prevention of Main-Memory Cache Exploitation. One way to prevent a main-memory cache from leaking secret values is to prevent variation in those secret values from causing variation 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) 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 413) 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.
>
> – *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, leads to a period of 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.

A 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 target principal P has executed.
- *Cache Probing.* Ascertain whether a specific address that 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 monitoring.

An attacker's actions cannot alter which principal will run next if the processor 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 works 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 Capacity Bounds on Timing Channels (page 407), 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, 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 with a main-memory cache that 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.

Notes and Reading

Covert Channels. The term "covert channel" was introduced by Lampson [29] for describing the *confinement problem*—the requirement that client-provided data not be leaked by a service. Lampson identified and gave names to 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 names subsequently evolved, and a decade later the Orange Book [14] was stating security requirements in terms of capacity limitations for what it called timing channels and storage channels which, readers are told, constitute the two types of covert channels. That formulation of confinement is still used, even though Wray [43] subsequently 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 the mechanisms that operating systems provided. As part of an effort at UCLA to build a secure operating system, Popek and Kline [36] suggests the use of virtual time in order to eliminate timing channels. However, as Lipner [31] explains, virtual time can be defeated in settings where end-users can measure response times. Fuzzy time avoids that problem; it was first proposed in Hu [19] as a means to reduce the capacity of covert timing channels in a secure virtual machine manager kernel being developed for the Digital Equipment Corporation VAX architecture [21, 30]. For blocking storage channels, Lipner [31] suggests enforcing the authorization policy of Bell and LaPadula [8, 7] on all objects named in a formal model of the system. This approach, however, can be unnecessarily restrictive, because 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 [23]. 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 [43] gives a different table-based analysis method for identifying possible covert timing channels. For this analysis, any generator of detectable events is considered a clock. Each row in the table 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 could be 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. However, automated methods that use system source code as the input must be conservative (making them incomplete), because program analysis to deduce whether specific information flows occur is an undecidable question.

Covert channels often surprise developers, because most people think about intended uses of given functionality rather than thinking about ways that functionality might be repurposed. The chapter described only a few of many possible covert channels. One of them—abuse of speculative execution, first proposed in Kocher et al [24]— at first might seem quite complicated and, thus, difficult to perform. (The example on page 411 is Spectre Variant 1 from Kocher et al [24].) Concern about speculative execution attacks is not misplaced, because little can be done in software to effect a defense, since 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 [33] 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[10] 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 [42], 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

[10]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.

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 [28] 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 a paper [32] by Loughry and Umphress describing how to recover transmitted data by monitoring LED status indicators on modems or other data communications equipment. Independently, Kuhn [27] 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 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.[11] 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 [44], and the use of short sequences of keypresses (instead of individual keypresses) for training is investigated in Berger, Wool, and Yaedor [9]. Much work followed; space limitations preclude giving a survey here.

Keyboards are not the only source of acoustic emissions in a computing system. Briol [12] is the first to observe that the printing of different characters on a dot matrix printer produces acoustic emissions having different waveforms.[12] But that paper does not give attacks to recover what is being printed from those "compromising sonsorous [*sic*] vibrations" [12]. Subsequently, Backes et al. [5] formulated attacks by leveraging the intervening two decades of developments in machine learning, feature extraction in music and speech, and speech recognition. However, mechanical devices are not the only source of problematic acoustic emissions in a computing system. Genken, Shamir and Tromer [17] shows how a 4096-bit RSA key can be recovered by recording and analyzing hum caused by capacitors and coils in the regulated power supply for a CPU.

[11]An attack previously published in Song, Wagner and Tian [40] 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.

[12]Briol [12] also showed that printing different characters produced different wave-forms for power consumption and for RF emissions.

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.[13] To assess 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 [25] 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. [11]) as well as methods for analyzing a program to determine if its executions are constant time (e.g., Barth et al. [6]).

Attacks that exploit other side-channels were also explored. Kocher et al. [26] leads the way with DES attacks based on monitoring power-consumption. Quisquater and Samyde [37, 38] 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 [15]. 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. [22], 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 [20] 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 [35] and Tsunoo et al. [41]. The formulation of such cache-based timing attacks in terms of Evict+Time and Prime+Probe is introduced in Osvik, Shamir and Tromer [34] 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] learns 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. [18] uses a translation lookaside buffer (TLB) to leak keys RSA and EdDSA secret keys. The survey by Ge et al. [16] 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

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

measurements of execution timings. Brumley and Boneh [13] 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 [10] 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. [39] 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 for supporting stronger isolation, but thus far they have not been embraced by hardware and system software producers.

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] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.