Side-Channel Attacks and Spectre

6.1600 Course Staff

Fall 2026

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

Up to this point, we have mostly reasoned about the security of a system by looking at its *intended* interfaces: the inputs that an attacker can send to a program and the outputs that the program returns. Real systems also leak information through channels that their designers never intended to be outputs at all: how long an operation takes, how much power it draws, which parts of the processor's cache it touches, the sounds it makes, and so on. These unintended outputs are called *side channels*.

We already saw a few side channels in the chapter on isolation, when we discussed the limits of isolation.

Historically, the side channel that people worried about most was electromagnetic emanations: electronic equipment radiates signals that an eavesdropper with an antenna can sometimes pick up and decode from a distance.[1] The broader lesson is that a system may need to worry about many unexpected ways in which information can be revealed.

[1] National Security Agency. "TEMPEST: A Signal Problem". In: *Cryptologic Spectrum* 2.3 (1972). Declassified 2007. URL: http://cryptome.org/nsa-tempest.pdf.
The U.S. government's program to study and shield against these emanations went by the code name TEMPEST.

What can leak. Researchers have studied many side channels as a means to extract cryptographic keys, including:

- how long it takes to decrypt a message,
- how decryption affects shared hardware resources (the cache, the TLB, the branch predictor), and
- physical emissions from the CPU itself (radio-frequency signals, sound, power consumption, etc.).

Side channels beyond cryptography. Many side-channel attacks have nothing to do with cryptography:

- The time a password check takes may reveal *which* character of a password guess was incorrect (see the Tenex attack in Section 1).
- The time a social network takes to answer a query may reveal how many friends you have in common with some other user.
- The time a browser takes to load a page reveals whether the page was already in the browser's cache—and thus whether the user visited that page recently.
- The sound of a dot-matrix printer is enough to recover the text it prints.[2]
- The sizes and timing of network packets can reveal a lot, even when the traffic is encrypted.

[2] Michael Backes et al. "Acoustic Side-Channel Attacks on Printers". In: *USENIX Security Symposium*. 2010.

The last example deserves a closer look, since encryption is often treated as a cure-all. Encryption hides the contents of a message, but the ciphertext is roughly the same size as the plaintext, and an

eavesdropper still sees when each message is sent. Researchers have shown that one can infer a great deal about TLS and VPN traffic from sizes and timing alone. The CRIME attack,[3] which we discussed earlier in the term, is a related problem.

[3] Juliano Rizzo and Thai Duong. *The CRIME Attack*. Presented at the ekoparty Security Conference. 2012.

In CRIME, compressing data before encrypting it makes the ciphertext length depend on the plaintext contents.

Example: Inferring stock holdings. At one point, Fidelity let customers manage their stocks through a TLS-protected website. The site displayed a pie-chart image for each stock in the user's portfolio, so the user's browser fetched one image per stock. An attacker can download every possible pie-chart image ahead of time and record its size. Then, by watching the sizes of the (encrypted) transfers between the user's browser and the server, the attacker can tell which stocks the user holds.

Even with all of these examples, attacks that recover passwords or cryptographic keys are usually the most damaging, so we focus on them for the rest of this chapter.

1 *Timing Attack on Password Checking: Tenex*

A famous early timing attack targeted password checking in the Tenex operating system. Suppose that the kernel provides a system call that checks a user's password. The kernel compares the guess against the stored password one byte at a time and returns an error as soon as it finds a mismatch:

Tenex stored passwords in plaintext, which is a problem in its own right, as we saw when we discussed password storage.

```
check_password(guess):
    for i in 0 .. len(password)-1:
        if guess[i] != password[i]:
            return ERROR
    return OK
```

The attacker exploits the fact that the kernel reads the guess out of the attacker's own memory, and that the attacker controls where that memory lives:

1. The attacker places the guess so that its first byte is the last byte of one memory page and the rest of the guess is on the next page.
2. The attacker arranges for the second page to be swapped out to disk. (Or the attacker simply unmaps the second page entirely, using the equivalent of `munmap`.)
3. The attacker calls the password-checking system call and measures how long it takes to return an error.

If the call takes a long time, the kernel had to read the second page in from disk. (If the page was unmapped and the call crashes, the kernel tried to read the second page.) Either way, the kernel only

touches the second page if the first byte of the guess was correct! The attacker then shifts the guess one byte to the left and repeats the process to learn the second byte, and so on.

With this attack, an attacker can guess an N-character password in $256 \cdot N$ tries, rather than 256^N.

The fix is to compare passwords in a way whose running time and memory accesses do not depend on where the first mismatch occurs—or, better, to store and compare password hashes.

Source code for a password-checking routine similar to the original is preserved in the Xerox Alto archives at the Computer History Museum: the `chkpsw` routine calls `stcmp` to compare strings byte by byte.

2 *Timing Attacks on Cryptography*

The same basic idea—learn a secret a little bit at a time by measuring how long an operation takes—applies to cryptographic implementations. The main difference is that the signal is much noisier, so attacks require many more measurements and more careful statistics.

2.1 Remote timing attack on RSA

Brumley and Boneh showed that timing attacks on RSA are practical even across a network.[4] In their setting, the victim is an Apache HTTPS web server using OpenSSL, which holds an RSA private key in memory. The attacker's goal is to extract that private key.

The attacker controls a client machine on the same campus network as the server (Stanford's, in the original experiment). The attacker sends the server specially constructed ciphertexts as part of the TLS handshake. The server decrypts each ciphertext, finds garbage padding, and returns an error. The client measures how long it takes to receive the error message, and uses these response times to guess the bits of q, one of the two secret primes p and q that make up the RSA secret key.

The overall response time is on the order of 5 milliseconds, while the time difference that the attack exploits can be around 10 microseconds. About one million queries were enough to recover the 512-bit primes p and q of a 1024-bit RSA key. In fact, the attacker only needs to learn the top 256 bits of q; a separate algorithm (due to Coppersmith) can then factor the modulus.

Countermeasures are needed, such as *RSA blinding*: the server multiplies the ciphertext by a random value r^e before decrypting and divides out r afterwards, so that the time the decryption takes is uncorrelated with the ciphertext that the attacker chose.

[4] David Brumley and Dan Boneh. "Remote Timing Attacks Are Practical". In: *USENIX Security Symposium*. 2003. URL: `https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf`.

The timing variations come from optimizations that make RSA efficient: OpenSSL switches between Karatsuba and ordinary multiplication depending on the sizes of the operands, and Montgomery multiplication occasionally needs an "extra reduction" step. Both depend on the relationship between the ciphertext and the secret prime.

How worried should we be about RSA timing attacks?

- Developing an exploit is relatively tricky, but this is a one-time cost: once someone writes it, anyone can run it.

- A server operator could notice an attack, since it requires a large number of connection requests. This might not be so easy on a busy web-server cluster, though.
- The attacker has to be close by in terms of the network. This is not much of an obstacle: the attacker can average over more queries, co-locate a machine nearby (e.g., in the same cloud data center), run the attack from a nearby bot or browser, and so on.
- The attacker may need to know the exact version of OpenSSL, its optimization flags, and so on. Is it a good idea to rely on such a defense? How big of an impediment is it really?
- If an attacker does mount the attack, the effects are quite bad: the server's private key leaks.

2.2 *Cache timing attacks*

A processor's cache is shared by all of the processes that run on it. When one process accesses memory, it brings that memory into the cache, which necessarily evicts something else from the cache. An attacker who can run code on the same machine can observe these evictions and learn about the victim's memory-access pattern.

For example, a fast implementation of modular exponentiation (for RSA or Diffie-Hellman) uses a *sliding window*: it precomputes a table of small multiples and, as it walks through the bits of the secret exponent d, it looks up different table entries depending on the bits of d. Accessing one of these entries brings it into the cache. A malicious process can fill the cache with a large array of its own, let the victim run, and then check which parts of its array were evicted by timing accesses to them. The offsets that were evicted reveal which table entries the victim used, and thus parts of the secret exponent.

Similar attacks work against table-based implementations of AES. Tromer, Osvik, and Shamir used them to recover AES keys from `dm-crypt`, the Linux disk-encryption subsystem, by running an unprivileged process on the same machine.[5]

Cache attacks are particularly worrisome in the presence of *mobile code*: WebAssembly modules, JavaScript, etc. that run on your laptop or phone. Mobile code is untrusted and may come from an attacker. It runs inside a sandbox, but the sandbox may still allow it to measure side channels.

[5] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (2010), pp. 37–71.

3 *Spectre and Meltdown*

In 2018, researchers disclosed a family of attacks known as Spectre and Meltdown.[6] Cache timing attacks had been known for years, so

[6] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *IEEE Symposium on Security and Privacy*. 2019; Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018; Mark D. Hill et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

why did these attacks cause such an uproar?

- **They changed the threat model.** Before these attacks, most system designers assumed side channels away. Spectre and Meltdown showed that micro-architectural side channels are a real, practical threat.
- **They exploit speculative execution**, a standard processor feature that is essential for high performance.
- **The attacker has some control over speculative execution.** This is a big difference from earlier cache timing attacks, in which the attacker could only passively observe the victim's (legitimate) memory accesses. Here the attacker can steer the victim into accessing memory that the victim's program never accesses in a correct execution.
- **Speculative execution has observable side effects.** In particular, it brings values into the cache, even when the speculation turns out to be wrong.
- **They break process isolation.** The attacker probes the cache to read out the secrets, and this side channel has high bandwidth.
- **They are architectural.** The underlying problem is a consequence of how modern processors are designed, so it is not easy to fix. There are several workarounds, each with costs.
- **They can be exploited remotely**, through JavaScript, WebAssembly, and other mobile code.

On Linux, you can see which of these attacks your processor is vulnerable to, and which mitigations are enabled, by running:
`grep . /sys/devices/system/cpu/vulnerabilities/*`

3.1 Processor features exploited

Both of the features that these attacks exploit exist to make processors fast.

Speculative execution. Consider code with a branch:

```
if (cond) {
    ...
}
```

Computing `cond` may take a long time—for example, if it depends on a value that must be loaded from main memory. Rather than wait, the processor *guesses* which way the branch will go (using a *branch predictor* that learns from the branch's past behavior) and speculatively executes the instructions on the predicted path. On today's processors, the processor may speculatively execute hundreds of instructions before it learns whether its guess was correct. If the guess

was wrong, the processor cleans up its architectural state—registers, etc.—and proceeds down the correct path, as if the speculatively executed instructions never ran.

Caching. Processors have several levels of caches. The last-level cache is shared between all of the cores on the chip. On failed speculation, the processor does *not* undo the effects of the speculative instructions on the cache: data that those instructions loaded stays in the cache.

3.2 *Attack variants*

There are many variants of these attacks, including:

1. Bounds-check bypass (Spectre, variant 1).
2. Branch-target injection (Spectre, variant 2).
3. Data access that bypasses protection checks (Meltdown, variant 3).
4. . . . and any other micro-architectural side channel.

The list of known attacks is surprisingly long.[7] We will start with variant 1, then discuss variants 2 and 3.

[7] Jonathan Behrens et al. "Performance Evolution of Mitigating Transient Execution Attacks". In: *European Conference on Computer Systems (EuroSys)*. 2022. DOI: 10.1145/3492321.3519559.

3.3 *Spectre variant 1: Bounds-check bypass*

Outline of the exploit. Suppose that the kernel holds a secret byte—for example, the random offset that it uses for address-space layout randomization. Suppose also that the kernel contains a code fragment (a *gadget*) like the following, where `x` is a value that the attacker supplies, say as an argument to a system call:

```
if (x < array1_size) {
    v = array1[x];
    y = array2[v * 4096];
}
```

We multiply `v` by 4096 so that each possible value of `v` maps to a different page, and thus a different cache line. The exact constant does not matter much, as long as different values land in different cache lines; the proof of concept below uses 512.

The bounds check means that, architecturally, this code never reads outside `array1`. But the attacker can make it do so speculatively:

1. **Train the branch predictor.** The attacker calls the gadget many times with in-bounds values of `x`, so that the branch predictor learns that the `if` branch is usually taken.
2. **Set up the cache.** The attacker arranges for `array1_size` to *not* be in the cache, so that the bounds check takes a long time to resolve and the processor speculates. The attacker arranges for `array1` (and the secret) to be in the cache, so that the read `array1[x]` completes quickly and there is time to start the next instruction before the speculation is squashed. Finally, the attacker makes sure that no part of `array2` is in the cache.

3. **Trigger the gadget.** The attacker calls the gadget with an out-of-bounds `x` such that `array1 + x` is the address of the secret byte. The processor speculatively reads the secret byte into `v` and then speculatively reads `array2[v * 4096]`, which brings that one line of `array2` into the cache. The attacker cannot read `v` directly. When the bounds check finally resolves, the processor realizes that it mispredicted and squashes its internal state. But it does not remove `array2[v * 4096]` from the cache, which lives outside the processor core.
4. **Probe the cache.** The attacker measures the time to read `array2[0 * 4096]`, `array2[1 * 4096]`, ..., `array2[255 * 4096]`. If one of them is fast, the attacker knows the value of `v`.

3.4 A concrete example

Appendix A of the Spectre paper[8] contains a short C program that demonstrates variant 1. It is illustrative, not a real attack: the victim code and attacker code live in the same address space. The program has the following parts:

[8] Kocher et al., "Spectre Attacks: Exploiting Speculative Execution".

Victim code. The victim has a gadget like the one above, a secret string, and the arrays `array1` and `array2`.

The victim's global variables include some unused padding (`unused1` and `unused2`). The padding forces `array1_size` to live in a different cache line from `array1`, so that the attacker can flush `array1_size` from the cache while `array1` stays cached.

Main function. The main function tries to read the secret one byte at a time by calling `readMemoryByte` with `malicious_x`, the out-of-bounds offset from `array1` to the secret byte (the attacker-supplied `x` from Section 3.3). `readMemoryByte` uses cache measurements to determine the byte.

The `readMemoryByte` function. This function does the following, repeatedly:

1. **Flush `array2` from the cache**, using the `clflush` instruction on each of the 256 lines of `array2` that the gadget might touch—one for each of the 256 possible values of a byte. Consecutive values are 512 bytes apart in `array2`, so that each lands in its own cache line and the processor's hardware prefetcher is less likely to pull in neighboring values.
2. **Train the branch predictor and invoke the gadget.** The function calls the victim's gadget several times with an in-bounds `training_x`, then once with `malicious_x` (flushing `array1_size` before each call).
3. **Time every possible value.** The function times a read of each of the 256 candidate lines of `array2`, probing them in a scrambled order (again to defeat the prefetcher), and counts a

The code computes `x` with a convoluted sequence of bit operations rather than an `if` statement. This avoids introducing a branch into the attacker's own loop, which could confuse the branch predictor that the attacker is trying to train.

cache hit for any line that loads quickly.

Any single trial is noisy: other activity on the machine may evict lines or bring them into the cache. So `readMemoryByte` repeats this procedure up to 1,000 times and picks the value with the most hits. It stops early once one value clearly has more hits than any other.

Bandwidth. This side channel leaks on the order of 10 KB/s.

3.5 Making the attack real

To mount the attack against a real victim, rather than against code in its own address space, the attacker must:

- find a suitable gadget in the victim process (or kernel),
- be able to access `array2`, so that it can time reads to it,
- evict `array1_size` from the cache in the victim process,
- evict `array2` from the cache in the victim process, and
- train the branch predictor in the victim process.

Stealing a secret from a victim process on Windows. The Spectre paper's Windows attack exploits dynamically loaded libraries (DLLs). Common DLLs are mapped into every process's address space, backed by the same physical memory. So any process can read the DLL's memory and flush it from the cache with `clflush`—and the attacker can even write to it (copy-on-write gives the attacker its own private copy, but this is enough to train the branch predictor). The shared DLL memory plays the role of `array2`.

Stealing a secret from the Linux kernel. Google Project Zero's attack against the Linux kernel[9] exploits eBPF, a mechanism that lets user-space programs load small programs (e.g., packet filters) into the kernel. The kernel checks eBPF programs to make sure that they cannot access memory out of bounds, and then runs them. The attack uses two eBPF filters:

1. one to train the branch predictor and to find the kernel address of an array that serves as `array2`, and
2. one that contains the gadget and leaks secret data.

The attacker then measures the cache state of `array2` from user space. This is possible because the kernel's address space includes a mapping of all physical memory, including the user's pages: an array in user memory also has a kernel address, so the gadget in the kernel and the attacker's timing code in user space can touch the same cache lines.

[9] Jann Horn. *Reading Privileged Memory with a Side-Channel*. Google Project Zero blog. 2018. URL: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

3.6 Discussion

- The attack breaks process isolation, which is the foundation that most of the system defenses in this course build on. It is a serious attack!
- The attacker can be remote: the attack can be carried out from JavaScript running in a web page.
- Any piece of micro-architectural state may be a side channel: the branch predictor, the write buffer, the cache-coherence directory, and so on.

3.7 Spectre variant 2: Branch-target injection

Variant 2 exploits speculation on *indirect* branches—jumps and calls through a function pointer. Consider kernel code like:

```
if (x < array1_size) {
    v = array1[x];
    (*f)();   // f reads memory at an address that depends on
         v
}
```

When the processor reaches an indirect call, it may not yet know the value of `f`, so it predicts the call's target and speculatively executes the code there. Fast indirect calls are important for object-oriented code, which makes heavy use of virtual method calls.

The attacker chooses a convenient `f`—a fragment of existing kernel code that reads memory depending on `v`—and trains the branch predictor so that the processor speculatively jumps to it. This is much like a return-to-libc attack: the attacker does not inject code, but redirects (speculative) execution to code that already exists. The attack requires reverse-engineering the processor's branch predictor, in order to know how to train it to predict the chosen target.

3.8 Meltdown (variant 3): Bypassing protection checks

Meltdown lets user-space code read kernel memory directly.[10] The core of the attack is a few user-space instructions:

[10] Lipp et al., "Meltdown: Reading Kernel Memory from User Space".

```
mov  al, byte [some_kernel_address]
shl  rax, 12
mov  rbx, qword [some_user_address + rax]
```

The `shl` (shift-left by 12) multiplies the secret byte by 4096, so that each possible value touches a different page of the attacker's user-space array.

Architecturally, the first instruction faults: user code may not read kernel memory. But the first instruction is implemented as several micro-operations, including a permission check and a load. The

processor executes instructions out of order, and some processors perform the load before the check has completed. The loaded value is forwarded to the following instructions, which speculatively use it to compute an address and load from it. When the check fails, the processor raises the fault and discards the results—the loaded value is never committed to a register—but the user-space address that depended on the secret is in the cache. The attacker then uses a Flush+Reload probe[11] to determine which address is in the cache, and thus the secret byte.

This works because the kernel and the user process share one address space. The kernel's memory is present in the user process's page table, marked as not accessible to user code.

A fairly realistic attack using a Meltdown-style technique, called Foreshadow, stole secret keys from Intel SGX enclaves.[12]

[11] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

Mapping the kernel into every process's address space makes system calls faster, since the processor does not have to switch page tables on each entry into and exit from the kernel.

[12] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

4 Mitigations

Variant 1.

- **Linux kernel:** Kernel developers use code analysis to find potentially vulnerable gadgets, and modify the code to constrain the index in the gadget even under speculation. The kernel provides helpers in `include/linux/nospec.h` for this purpose:

```
if (index < ARRAY_SIZE) {
    // array_index_nospec() blocks speculation
    index = array_index_nospec(index, ARRAY_SIZE);
    return array[index];
}
```

`array_index_nospec()` computes a mask from the comparison using arithmetic, rather than a branch, so that a mispredicted branch cannot make the processor use an out-of-bounds index.

- **Browsers:** JavaScript engines avoid generating vulnerable code sequences when they compile JavaScript and WebAssembly to machine code.

Variant 2. The defenses are microcode updates and kernel patches, such as *retpoline* ("return trampoline"). A retpoline replaces each indirect jump with a sequence built around a `ret` instruction. The sequence contains an infinite loop that is never executed architecturally; any speculation lands in this harmless loop, which prevents the CPU from speculating on the target of the indirect jump. Retpolines slow down every indirect jump. Another approach ("optpoline") resolves indirect jumps at runtime and replaces them with direct jumps where possible.

Variant 3. The defense is *kernel page-table isolation* (KPTI): the kernel uses two page tables, one for the kernel and one for user space. The user-space page table maps almost none of the kernel's memory, so there is nothing for a Meltdown attack to read.

KPTI was developed (under the name KAISER) before Meltdown was known, to protect kernel address-space layout randomization against other side channels.

Hardening browsers. Since many attacks need a precise timer, browsers stopped giving web pages precise timing information.

Performance. All of these mitigations have a performance cost, which varies across processor generations as vendors add hardware support for them.[13]

[13] Behrens et al., "Performance Evolution of Mitigating Transient Execution Attacks".

Other side effects of speculative execution. The cache is not the only observable side effect of speculative execution. Speculatively executed instructions also affect power consumption, bus traffic, and more.

5 How to Stop Side Channels?

There is no single answer, but there are several broad strategies:

- **Remove side channels.** For example, one could redesign processors so that speculative execution does not change the cache state until it is committed. But every shared resource could be a side channel, so it is hard to be sure that one has found them all.
- **Reduce shared resources when processing secret data.** For example, process confidential data on a separate core, with its own cache, etc. This requires refactoring software, which is not easy.
- **Measure channels and make them low bandwidth.** A channel that leaks a bit per hour is much harder to exploit than one that leaks 10 KB per second.

Further reading. Percival[14] and Yarom and Falkner[15] describe early cache attacks that work across processes; Zhang et al.[16] extend them across virtual machines in the cloud. The Ed25519 signature scheme[17] is an example of a cryptosystem designed from the start to run in constant time.

[14] Colin Percival. *Cache Missing for Fun and Profit*. BSDCan. 2005.

[15] Yarom and Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack".

[16] Yinqian Zhang et al. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *ACM Conference on Computer and Communications Security (CCS)*. 2012.

[17] Daniel J. Bernstein et al. "High-Speed High-Security Signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

References

Backes, Michael et al. "Acoustic Side-Channel Attacks on Printers". In: *USENIX Security Symposium*. 2010.

Behrens, Jonathan et al. "Performance Evolution of Mitigating Transient Execution Attacks". In: *European Conference on Computer Systems (EuroSys)*. 2022. DOI: 10.1145/3492321.3519559.

Bernstein, Daniel J. et al. "High-Speed High-Security Signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

Brumley, David and Dan Boneh. "Remote Timing Attacks Are Practical". In: *USENIX Security Symposium*. 2003. URL: https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf.

Fogh, Anders. *Negative Result: Reading Kernel Memory from User Mode*. cyber.wtf blog. 2017. URL: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.

Genkin, Daniel, Itamar Pipman, and Eran Tromer. "Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2014.

Hill, Mark D. et al. "On the Spectre and Meltdown Processor Security Vulnerabilities". In: *IEEE Micro* 39.2 (2019), pp. 9–19.

Horn, Jann. *Reading Privileged Memory with a Side-Channel*. Google Project Zero blog. 2018. URL: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

Kocher, Paul et al. "Spectre Attacks: Exploiting Speculative Execution". In: *IEEE Symposium on Security and Privacy*. 2019.

Lipp, Moritz et al. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018.

National Security Agency. "TEMPEST: A Signal Problem". In: *Cryptologic Spectrum* 2.3 (1972). Declassified 2007. URL: http://cryptome.org/nsa-tempest.pdf.

Percival, Colin. *Cache Missing for Fun and Profit*. BSDCan. 2005.

Rizzo, Juliano and Thai Duong. *The CRIME Attack*. Presented at the ekoparty Security Conference. 2012.

Tromer, Eran, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (2010), pp. 37–71.

Van Bulck, Jo et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security Symposium*. 2018.

Yarom, Yuval and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

Zhang, Yinqian et al. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *ACM Conference on Computer and Communications Security (CCS)*. 2012.