COMP 435 Quiz 2 Study Guide

Computer Security Concepts · UNC Fall 2026 · Unit 1: Trusting the Machine · Covers L05–L14 + the Rowhammer guest lecture

0. Quiz logistics and how to use this guide

When. Friday, September 25, 2026, in class (10:10–11:00). Written quiz.

Format. Short answer on paper: write properties, trace attacks, read a timing table, order the steps, true/false-and-correct.

Scope. Unit 1 — Trusting the Machine. L05 trust through L13 side channels, with L14 OS material as background.

0.1 The six topics Kaki listed on the L14 slide

These are the announced quiz topics, verbatim from the **Quiz Topics** slide in Lecture 14. Everything in this guide is organised around them.

Topic	What the slide says you must know	Guide section
Hardware as the root of trust	trusted vs. trustworthy, TCB	§1, §2
Hardware bugs & errata	specifications, implementations, security consequences	§4
Security properties	writing simple properties, counterexamples	§3
Information flow	explicit vs. implicit flow; which security primitives unintended/indirect flows violate	§5
Caches & timing side channels	the mechanisms a timing side channel requires; how timing observations infer secrets	§6
Speculative execution & Spectre	the steps of the attack and *why each step is necessary*	§7

Extra context beyond the slides: Rowhammer (L12) and the L13 side-channel survey are not on the topic list by name, but L12/L13 are inside the covered window and the official practice set *does* ask about DRAM bit flips (Q8.2). The practice set also asks nothing about L14 OS material, so treat §8 as insurance, not a priority.

0.2 Lecture-by-lecture scope

Lecture	Date	Title	Core ideas
L05	8/28	Defining Trust	Saltzer & Schroeder recap; trusted vs. trustworthy; TCB; honest-but-curious
L06	8/31	Hardware as the Root of Trust	the trust stack; security boundaries; trusted-systems process; Orange Book / Common Criteria
L07	9/2	Hardware Security & Security Properties	ISA vs. microarchitecture vs. RTL; policy → property; counterexamples; testing vs. model checking; hardware CWEs
L08	9/4	Security in Practice	in-class incident discussion — no new examinable definitions
L09	9/9	Hardware Bugs, Errata and CWEs	errata; wrong/imprecise specs; Pentium FDIV; testing vs. formal methods; CWE vs. CVE
L10	9/11	Information Flow & Intro to Side Channels	explicit vs. implicit flow; IF properties ( ); cache background; the first timing side channel
L11	9/14	Putting it all together — Spectre	branch prediction; the six steps; flush & reload; the architectural/microarchitectural gap
L12	9/16	Rowhammer (guest: Noah Brown)	DRAM banks/rows/row buffer; charge leakage; double-sided hammering; TRR and TRRespass
L13	9/18	Hardware Side Channels in Practice	Meltdown, Foreshadow, GoFetch, ZombieLoad, Hertzbleed; the pattern across all of them
L14	9/23	Operating Systems Security	user vs. kernel mode; features of a trusted system; four styles of separation; fence / base–bounds / segments; race conditions

0.3 How to use this guide

Each section ends with a **Check yourself** set whose questions are modelled on the actual in-class practice sheets for that lecture. §10 is a full mock quiz built to mirror the official Quiz 2 practice set — nine questions, same shapes, different numbers and scenarios — and §11 walks the official practice set itself with the released answers plus commentary on *why* each answer is what it is. §12 is a one-screen cram sheet.

EXAM TIP. This quiz is a *written* quiz, not multiple choice. Three answer shapes repeat across the whole practice set, and you should drill all three until they are automatic:

- **Write a property.** Always `IF <condition> THEN <forbidden thing> must not happen`, or for information flow, `secret  observable`.
- **Give a counterexample.** A concrete *sequence of events* reaching a state where the property's antecedent holds but its consequent fails.
- **Explain why an indirect path still leaks.** Name the chain: secret → computation → microarchitectural state → timing → attacker.

L05 1. Defining trust

The whole of Unit 1 hangs on one distinction, and the quiz topic list names it first. Get this pair exactly right and several questions answer themselves.

1.1 Trusted vs. trustworthy

Trust. Relied upon to meet its security specifications. Trust is about one system *depending* on another system.

Trustworthy. Deserving of trust; *meets* its security specifications. Trustworthiness is about a system's security posture.

So "trusted" is a statement about **you** — about what your security happens to depend on. "Trustworthy" is a statement about **the component** — about whether it actually behaves. A component can be trusted and not trustworthy; that combination is exactly what a vulnerability is.

Watch out. The two words are not synonyms and the quiz will punish you for treating them as such. Practice Q1.2 hands you a processor with a bug and asks for "evidence that would make us question whether it is trustworthy." The bug does not make the processor *untrusted* — the system still depends on it either way. It makes it *un-trustworthy*.

The slides list what trustworthy means concretely. A trustworthy component is secure against:

- intentional, malicious actions,
- accidental actions, and
- third-party actions.

1.2 The Trusted Computing Base

Trusted Computing Base (TCB). Those components upon which the security of the system relies.

Kaki pairs the definition with a quotation that is worth memorising because it says the same thing from the other direction:

Goguen and Meseguer. "Trusted system: a subsystem that is permitted to violate some global security policy."

That second framing is the useful one for exam answers. A component is in the TCB precisely because it *can* break the policy — the OS kernel can read any process's memory, the processor can ignore a privilege check. Being in the TCB is not an endorsement; it is an admission of exposure.

Extra context beyond the slides: the two definitions match because "security relies on it" and "it could violate the policy if it misbehaved" describe the same set of components. If a component could not possibly violate the policy, nothing about your security depends on it.

1.3 The postcard example (know this one — it is the recurring illustration)

Kaki runs the same scenario three times, shrinking the TCB each round. "Mail this for me, please":

Round	What you hand over	Who you trust	Trusted for what
1	A postcard with the secret written on it	the TA, the postal service, the roommates of your friend	confidentiality, integrity and availability — everyone in the chain can read it, change it, or lose it
2	A sealed envelope	the same entities	the same list, but the *amount* of trust required drops: accidental reading is no longer a concern
3	Encrypted and signed contents, in an envelope	the same entities, plus the crypto	availability only. Confidentiality and integrity no longer depend on the couriers — but **some crypto is now part of the TCB**.

EXAM TIP. The punchline of round 3 is the examinable bit: you never eliminated trust, you *relocated* it. The couriers left the confidentiality TCB and the cryptographic implementation entered it. Any answer that says "encryption removes the need for trust" is wrong.

1.4 Should the TCB be big or small?

This is a literal in-class practice question (L05 practice Q1) and it is the most likely short-answer opener on the quiz.

ANSWER. Small. Every component in the TCB must be proven trustworthy, and the effort of doing that grows with size and complexity. A small, well-defined TCB means less code and hardware to verify, a smaller attack surface, and fewer places a single failure can break the whole policy.

Note how this connects backwards to Saltzer and Schroeder: a small TCB *is* economy of mechanism plus least privilege applied at the architecture level. L14 restates it as the first "feature of a trusted system": the TCB should be **small and well defined**.

1.5 Honest, but curious

HONEST, BUT CURIOUS. An adversary that tries to glean information while adhering to the protocol.

In the postcard story, the honest-but-curious TA does not open the sealed envelope — they follow the rules — but they will happily read anything visible without breaking a rule, i.e. the postcard.

The in-class practice sheet (L05 Q2) is multiple choice, and the correct option is **"It can reveal information leaks in protocols."** The reasoning: precisely because the adversary stays inside the protocol, anything they learn is leaked *by the protocol's own design*, not by an attack on it. It is not the most powerful threat model — a malicious adversary who deviates is strictly stronger — which is what makes it an interesting lower bound.

WATCH OUT. "It has the most powerful threat model" is the tempting wrong answer. Honest-but-curious is a *weak* adversary. That is the point: if a protocol leaks even to an adversary this constrained, the leak is structural.

1.6 Check yourself

1. A cloud provider runs your VM. Is the hypervisor trusted, trustworthy, both, or neither? [L05]

It is **trusted** — your workload's security relies on it to isolate you from co-tenants, so it is in your TCB. Whether it is **trustworthy** is a separate, empirical question about whether it actually meets its security specification; you cannot conclude that from the fact that you depend on it. (Evidence such as a published escape vulnerability would be reason to doubt trustworthiness while leaving it just as trusted.)

2. You switch from sending a postcard to sending a signed, encrypted message. Did the TCB shrink? [L05]

For confidentiality and integrity, yes — the TA, postal service and roommates drop out of the TCB for those properties. But the cryptographic implementation joins the TCB, and everyone in the delivery chain remains trusted for **availability** (they can still throw the envelope away). The TCB was reshaped more than it was shrunk.

3. Give one reason a large TCB is dangerous, phrased in terms of a principle from L04/L05. [L04 · L05]

Economy of mechanism: a large TCB is a large amount of code and hardware that must be shown correct, and the assurance effort scales worse than linearly with complexity. Equivalently, least privilege — components in the TCB are permitted to violate the global policy, so putting components there that do not need to be there grants unnecessary authority.

4. Why is an honest-but-curious model "instructive" even though it is not the strongest adversary? [L05]

Because it isolates leaks that are inherent to the protocol. An adversary that follows the protocol exactly and still learns something has learned it from information the protocol itself exposes — so any such finding is a design flaw, not an implementation or deployment failure.

L06 2. Hardware as the root of trust

Guiding question from the deck: *what must we trust at the lowest level in a computing system?*

2.1 The stack we are trusting

Every software security mechanism rests on assumptions about the layers below it. The deck draws exactly this stack, twice (L06 and again in L11):

```
Applications
    ↑ assumes
Operating System
    ↑ assumes
ISA / Processor
    ↑ assumes
Hardware
```

Read it upward and it is a chain of assumptions. The application assumes the OS isolates processes; the OS assumes the ISA enforces privilege levels; the ISA assumes the hardware implements it faithfully. Hardware sits at the bottom with nothing beneath it to appeal to — which is precisely what "root of trust" means.

HARDWARE IN THE TCB. Our security relies on it, so yes, hardware is part of the TCB — and therefore it has to be proven trustworthy. That is hard to do for a full piece of hardware (or for a complex piece of code like an operating system).

2.2 Why hardware is the security foundation

Four bullets from the slide, and they are the four things a "why is the processor part of the TCB?" answer should draw on:

- The processor decides which instructions actually execute.
- Hardware mediates access to memory and devices.
- Software relies on hardware checks to enforce security boundaries.
- If hardware allows software to bypass a boundary, higher-level protections may fail.

EXAM TIP. Official practice Q1.1 is exactly "Why is the processor part of the system's TCB?" The released answer is: *because system security depends on it correctly enforcing protection and privilege boundaries.* Note the shape — "security depends on it" (the TCB definition) plus "enforcing privilege boundaries" (what specifically it is depended upon for). Give both halves.

2.3 Security boundaries

> **Security boundary.** A separation between parts of a system that have different levels of trust, privilege, or allowed access.

The examples given on the slide:

- one process vs. another process,
- user code vs. kernel code,
- one user's memory vs. another user's memory,
- software vs. privileged hardware operations.

And the rule attached to it: **crossing a security boundary should require a controlled check** → that is complete mediation, straight out of Saltzer and Schroeder.

2.4 Hardware-enforced privilege

The canonical mechanism. Not all code should have the same authority:

```
User applications          Controlled transition          Privileged system code
(less privilege)     ──(syscall / interrupt / trap)──      (more privilege)
```

1. The CPU tracks the current privilege level.
2. Some instructions and resources are only accessible in privileged mode.
3. If a user requests a privileged operation, the CPU traps it to the OS.

All three steps are hardware doing the work. This is why practice Q1.3 — "what assumption made by the operating system has failed?" — has the answer *the OS assumed the hardware would prevent user-mode code from modifying protected state.* The OS has no independent way to enforce that; it delegated the check to step 2 above.

2.5 Thought experiment: no hardware protection

Suppose every program could read any memory, modify any memory, execute any instruction, directly control devices, and interrupt or stop other programs. This is the L06 in-class practice sheet, and it is a good compact way to rehearse the C.I.A. mapping.

Capability	Example harm	Property primarily at risk
read any memory	one process reads another's private key or password buffer	Confidentiality
modify any memory	a program rewrites the OS's permission tables or another process's data	Integrity (and then Authorization, since the tables *are* the access control)
execute any instruction	user code runs privileged instructions and reconfigures the machine	Integrity / Authorization

directly control devices	a program drives the disk or network card behind the OS's back	Integrity, Confidentiality
interrupt or stop other programs	one process halts every other process	Availability

The third practice question — *are there guarantees the OS could still reliably provide?* — has an answer worth rehearsing, because it is a "root of trust" argument in miniature:

ANSWER SKETCH. Essentially none that are *reliable against a malicious program*. The OS is itself just code and data in memory; if any program can modify any memory, it can modify the OS's own access-control structures, so every guarantee the OS offers becomes advisory. The OS can still provide *convenience and protection against accidents* — conventions, APIs, bookkeeping for well-behaved programs — but it cannot enforce anything against an adversary. Enforcement requires a layer the adversary cannot reach, and that layer is hardware.

Extra context beyond the slides: authentication survives only in the weak sense that the OS can still *ask* for a password — it cannot protect the stored credentials or the "is authenticated" flag from being overwritten.

2.6 Trusted systems: standards and process

Two standards named on the slides. Know the contrast between them, not the details.

	Orange Book	Common Criteria
Official name	Trusted Computer System Evaluation Criteria	Common Criteria for Information Technology Security Evaluation
Who / when	US DoD, 1970s	a group of international governmental agencies, 1990s
Structure	6 rankings; **couples** security features with assurance features	7 assurance levels; **decouples** security features from assurance requirements

EXAM TIP. The one-word difference is **coupled vs. decoupled**. Decoupling matters because it lets you say "this product has these features, and they were evaluated to this depth" as two independent claims — a simple product can be highly assured, and a feature-rich product can be barely evaluated.

The trusted-systems **process** has three stages, and the deck expands each:

Stage	What it is
specification	defines the desired security policy and functionality
design process	1. model → 2. design → 3. implement

evaluation	testing · red-team / blue-team exercises · formal verification

2.7 What guarantees do we expect from hardware?

Correct execution.
Instructions behave according to the architecture

Restricted operations.
Sensitive actions cannot be performed by ordinary programs

Controlled transitions.
Crossing into privileged code happens through defined mechanisms

Memory checks.
A program cannot freely access memory

WATCH OUT. Notice what is *not* on this list: anything about timing, power, or cache state. That omission is the entire second half of the unit. The final L06 slide says it outright — "correct architectural behavior is not the whole story"; attacks exploit side effects of implementation details, and timing, caches, power and speculation can leak information. **Hardware is both a security mechanism and a potential source of vulnerabilities.**

2.8 Check yourself

1. In one sentence each, say why the processor is in the TCB and what it is trusted to do. [L06]

It is in the TCB because the system's security relies on it. It is trusted to correctly enforce protection and privilege boundaries: to track the privilege level, restrict privileged instructions and resources to privileged mode, trap user attempts at privileged operations, and mediate access to memory and devices.

2. Name the security principle that "crossing a security boundary should require a controlled check" restates. [L04 · L06]

Complete mediation — the system checks authorization on *every* access to a protected resource, not just the first one.

3. A vendor advertises "EAL4 certified." Under the Orange Book, could you make an equivalent claim without also specifying the feature set? [L06]

No. The Orange Book couples security features with assurance features, so a single ranking bundles both — you cannot state an assurance depth independently of the features. Common Criteria decouples them, which is why an assurance level (EAL) is quotable on its own alongside a separate statement of what was evaluated.

4. Which of the four expected hardware guarantees does a Spectre attack violate?
[L06 · L11]

Arguably none of them — and that is the point. Architecturally, instructions still behave per the architecture, privileged operations are still restricted, transitions are still controlled, and the out-of-bounds read is rolled back so the program never gets the value. Spectre leaks through microarchitectural state, which these four guarantees say nothing about. Functional correctness does not imply secure information flow.

L07 3. Security properties and verification

Guiding question: *how can we determine or test if hardware deserves our trust?* This section is the most mechanical part of the quiz — there is a fill-in-the-blank template and you should leave the room able to produce it under pressure.

3.1 Hardware background (311 is not a pre-req)

CPU / processor.
Hardware that executes program instructions

SoC.
A processor plus other hardware components in one system

ISA.
The behavior the processor exposes to software — e.g. MIPS, RISC-V, x86

RTL.
A code-like description of hardware behavior and state — e.g. Verilog, VHDL

Signal.
A value moving through hardware

Register.
Stores a value across clock cycles

Clock cycle.
One step of hardware execution

Module.
A hardware component with inputs, outputs, and state

The example on the slide is a four-bit counter in SystemVerilog. You will not be asked to write RTL, but you should be able to read this much:

```
module counter (
    input  logic clk,
    input  logic reset,
    output logic [3:0] count
);

always_ff @(posedge clk) begin
    if (reset)
        count <= 0;
    else
        count <= count + 1;
end

endmodule
```

The slide's own annotation is the reason it is there: **security properties are often specified as "this state should never change in this situation."** Hardware has state that persists across clock cycles, so "never changes" is a meaningful, checkable claim about it.

3.2 Architecture vs. implementation — the three levels

Level	What it is
ISA / Architecture	what software sees
Microarchitecture	how the processor actually does it
RTL	hardware description

THE THESIS OF THE WHOLE UNIT. A design can be functionally correct at one level and still create security problems at another. Verification asks whether an implementation satisfies stated properties. Side channels often exploit behavior *below* the architectural abstraction — microarchitectural timing, cache state.

This slide reappears verbatim in L10, which tells you how central it is. Official practice Q9 — "can the processor be functionally correct and still create a security problem?" — is this slide restated as a question.

3.3 From security policy to security property

Security policy.
A statement of what a system should and should not allow in order to protect its assets

Security property.
A precise rule describing behavior that a secure design should always satisfy

Counterexample.
A sequence of events showing the property can be violated

The worked pair from the slide:

Security policy	Security property
Ordinary programs should not perform privileged operations.	`IF privilege = USER THEN privileged_write = FALSE`

3.4 The property pattern — memorise this

THE TEMPLATE. `IF ____________ THEN ____________`

The three examples the deck gives, which are also the three policies on the L07 practice sheet:

- IF **privilege = user**, THEN **privileged register cannot change**
- IF **debug interface is locked**, THEN **debug write cannot occur**
- IF **authentication has not succeeded**, THEN **protected state cannot be modified**

A "state cannot change" consequent is written formally as a claim about the next cycle's value:

```
lock = 1     config_next = config_current
```

That is the released answer to official practice Q2.1, and it is the form to reach for whenever the policy says "should not change." Writing `config_next = config_current` rather than English earns the point cleanly, but the slides are explicit that English or pseudocode is acceptable — "just use pseudocode + math notation, doesn't have to be perfect."

EXAM TIP. The L07 practice sheet asks for **four** things per policy, and the quiz may too. Rehearse all four: **asset → forbidden behavior → property → counterexample**. Worked out for the three policies:

	A: user code must not modify a privileged config register	**B: once the lock is enabled, the protected register must not change**	**C: debug must not write protected state before authentication**
Asset	the privileged configuration register	the protected/locked register	the protected state reachable from the debug interface
Forbidden behavior	a write to that register originating from user-mode code	any change to the register's value while lock is set	a debug-interface write while authentication has not succeeded
Property	`IF privilege = USER THEN config_next = config_current`	`IF lock = 1 THEN protected_next = protected_current`	`IF authenticated = FALSE THEN debug_write = FALSE`
Counterexample	1. set privilege = USER · 2. execute the buggy instruction · 3. config changes	1. set lock = 1 · 2. send a debug command · 3. the register changes	1. leave authenticated = FALSE · 2. issue a debug write · 3. protected state changes

Notice the shape of every counterexample: **establish the antecedent, perform an action, observe the consequent fail.** Three lines. Nothing more is required — and the slide notes that "a counterexample is not always the full real-world attack, but it can reveal a feasible violation path."

3.5 How do we check a property?

Simulation-based testing

Choose inputs → run the design → look for failure. Finds bugs on the executions you try.

Model checking

Design + property → tool → proof or counterexample. Checks *all* modeled executions within the stated assumptions.

Each has a named limitation on its own slide, and the quiz will want the word:

Technique	Limitation	Why
Simulation-based testing	**Coverage**	The slide's example is `if (x == 13): ERROR` buried among branches that random inputs hit constantly. A rare input is a rare test. All tests passing shows nothing; one test failing potentially shows a bug.
Model checking / formal methods	**Scalability**	It reasons over a *formal model*, and building a model that is both faithful and small enough to check is hard. A full proof provides assurance; a counterexample potentially shows a bug.

L09 draws the trade-off as a single axis: testing wins on **scalability**, formal methods win on **completeness**. Testing "often takes random sequences in as inputs"; formal methods "often take a formal security property as input."

THE SENTENCE TO WRITE ON THE QUIZ. Testing checks only the executions that were actually tested; passing many tests does not prove that no other execution can violate the property. Formal verification attempts to reason about all executions represented by its model and assumptions.

WATCH OUT. Official practice Q2.3 asks whether 10,000 passing tests prove the property. The released answer is "No" — but note the released phrasing carefully: formal verification reasons about all executions *represented by its model and assumptions*. That qualifier is the scalability limitation sneaking back in. Do not write "formal verification proves the property holds absolutely"; write that it covers all executions *of the model*.

3.6 Hardware CWEs

Why CWEs? They help researchers and security professionals **compare bugs and properties across designs** instead of treating every flaw as totally unique.

CWE VS. CVE. A **CWE** (Common Weakness Enumeration) is a *class* of weakness — a reusable description of the kind of mistake, independent of any product. A **CVE** (Common Vulnerabilities and Exposures) is one *specific instance* — a particular vulnerability in a particular product and version. Many CVEs map to one CWE.

Extra context beyond the slides: the slide asks "Different from CVE! How?" without answering on the slide itself, so this is exactly the kind of thing that gets asked. One line: CWE = the weakness type, CVE = the identified vulnerability in a shipped thing.

The five example hardware CWEs listed (in both L07 and L09):

- Improper access control for protected resources
- Improper isolation of shared resources

- Incorrect privilege management
- Debug/test interface exposed or insufficiently protected
- Side-channel exposure through timing, power, or shared state

The last one is worth pausing on: side channels are not an exotic curiosity, they are a catalogued weakness class. And the second one — improper isolation of shared resources — is precisely what a shared cache is.

3.7 The debug interface example

The recurring concrete asset across the practice problems. Desired behavior: imagine a boolean `debug_mode`; only when it is TRUE should the processor be able to inspect the contents of the protected registers. The slide notes that most ISAs only expose behavior and state at clock-cycle boundaries or after the end of an instruction — which is why debug interfaces, which see *internal* state, are so dangerous when left unlocked.

Property: `IF debug_mode = FALSE THEN protected_register  debug_output`, or in the simpler if-then form, `IF debug_mode = FALSE THEN debug_read = FALSE`.

3.8 Check yourself

1. Policy: "A device's firmware update slot must not accept a new image once the secure-boot lock fuse is blown." Give the asset, the forbidden behavior, a property, and a counterexample. [L07]

> **Asset:** the firmware image / update slot. **Forbidden behavior:** writing a new firmware image while the fuse is blown. **Property:** `IF fuse_blown = 1 THEN firmware_next = firmware_current` (equivalently `IF fuse_blown = 1 THEN update_write = FALSE`). **Counterexample:** 1. blow the fuse (`fuse_blown = 1`) · 2. issue an update command over the manufacturing/test interface · 3. the firmware image changes.

2. Why is a counterexample more useful than a failing test log? [L07]

> A counterexample is tied to a stated property: it does not just say "something went wrong," it exhibits a concrete execution in which a specific security claim is false. It shows a feasible violation path, which tells you what to fix and what to re-verify. It need not be the full real-world attack to be actionable.

3. A team model-checks a processor and the tool returns "proof." Name one reason the shipped chip could still be insecure. [L07 · L09]

The proof covers the *model* under its stated assumptions. If the model abstracts away the piece where the bug lives — most obviously the microarchitecture, caches and timing — or if the property written down does not capture the real policy (or the specification itself is wrong or imprecise), the chip can satisfy the proof and still leak. Scalability pressure is exactly what pushes teams to abstract those parts away.

4. State the difference between a CWE and a CVE in one sentence, and give a hardware example of each kind of thing. [L07 · L09]

A CWE is a reusable class of weakness ("debug/test interface exposed or insufficiently protected"); a CVE is one specific vulnerability in one specific product ("the Spectre variant-1 entry for a particular processor family"). Many CVEs are instances of the same CWE, which is what makes cross-design comparison possible.

L09 4. Hardware bugs, errata and the FDIV case study

Guiding question: *where do hardware bugs come from?* The quiz topic line is "specifications, implementations, security consequences" — three words that are also the three origins of a hardware bug.

4.1 Errata and the three origins of a bug

HARDWARE ERRATA. Design defects or errors. Errata may cause the processor's behavior to deviate from published specifications. Companies and organizations document errata and changes to designs to help hardware (and software) engineers.

But a mismatch between spec and implementation is only one of three cases. The slide adds two more:

Case	What went wrong	Example
Errata — implementation ≠ spec	The spec is right; the silicon does not match it	Pentium FDIV: the spec for floating-point division was fine, the lookup table was missing entries
Incorrect specification	A bug in the spec leads to a bug in the hardware. The implementation is faithful — to the wrong thing	A spec that permits a debug write while locked; the chip that implements it is "correct" and insecure
Imprecise specification	Vagueness in the spec leads to buggy or unwanted behavior; implementers fill the gap differently	A spec that never says what happens to microarchitectural state on a rollback

EXAM TIP. If asked "where do hardware bugs come from," give all three. The second and third are the ones students forget, and they are the interesting ones — a chip can pass every conformance test against its specification and still be insecure, because the specification never said the right thing.

Two structural reasons this keeps happening, both on their own slides:

- **Hardware bugs are hard to patch post-deployment.** You cannot push an update to silicon. Mitigations end up in microcode, firmware, compilers or the OS, and they usually cost performance.
- **Increasing design complexity.** The transistor-count-over-time chart appears three separate times across the decks. More transistors → a larger attack surface.

4.2 Case study: the Pentium FDIV bug (1994)

The one named historical hardware bug in the course. Know the story, the cost, and the moral.

Element	Detail

The spec	Floating point as `(1 + f) × 2^e`, with `0 ≤ f < 1`, `e` . The problem with implementing this in digital logic: we only have a finite number of bits to work with.
The implementation	The divider repeatedly chooses the next quotient digit; that choice is driven by a small lookup table. A few entries were missing → rare but incorrect divisions.
Discovery	Thomas Nicely, a math professor computing reciprocals of large primes in 1994, started seeing errors. He tested other processors and could not reproduce it; other users then could.
The famous number	4195835 ÷ 3145727. Correct ≈ 1.333820449…; buggy Pentium ≈ 1.333739068…
Cost	Much bad press. Intel first replaced chips only if you could prove it affected you, then switched to no-questions-asked replacement: **$475M** in hardware replacement costs in 1994.
The moral	A turning point for formal methods: a growth in attention, funding and research, because the field realised that having experts stare at the code was not enough.

Extra context beyond the slides: FDIV is a *correctness* bug rather than a security bug — nobody's secret leaked — which makes it the perfect contrast case for the unit's central claim. FDIV is the case where the architecture was wrong and the harm was obvious. Spectre is the case where the architecture was right and the harm was invisible to every architectural test. Both motivate formal methods, for opposite reasons.

4.3 Bug-finding, revisited

L09 restates the L07 comparison and adds the axis explicitly:

	Testing	Formal methods
Typical input	random sequences	a formal security property
Strong on	**scalability**	**completeness**
What a pass means	all tests passing shows nothing	a full proof provides assurance
What a fail means	one test failing potentially shows a bug	a counterexample potentially shows a bug

> WATCH OUT. "All tests passing shows nothing" is deliberately blunt and it is the phrase to echo. It is the asymmetry of testing: tests can only ever demonstrate the presence of bugs, never their absence.

The third leg the deck adds: **security experts identifying common weaknesses and documenting vulnerabilities**, and **the writing of good security properties and specifications** — which is what lets the community as a whole vet and check their designs for the same bugs and flaws. That is the CWE programme (§3.6).

4.4 Check yourself

1. A chip behaves exactly as its datasheet describes, but the datasheet permits the debug port to read key material when unlocked. Is this an erratum? [L09]

> No. Errata are deviations from the published specification; here the implementation matches the spec. This is the second case — an **incorrect specification**. The bug is in the spec and the hardware faithfully implements it, which is why conformance testing would never catch it and why writing good security properties matters independently of verifying conformance.

2. Why did FDIV increase funding for formal methods rather than for testing? [L09]

> Because the bug was in a small number of lookup-table entries reached only by rare operand combinations — exactly the coverage blind spot of simulation-based testing. Random or directed tests can run for a very long time without hitting the failing inputs, while a formal proof over the divider reasons about all of them. The $475M price tag also made the cost of the alternative look small.

3. Give one security consequence (not a correctness consequence) that could follow from an arithmetic erratum. [L09]

> Cryptographic code depends on exact arithmetic; a rare wrong result inside a signature or key-agreement computation can produce a faulty output that leaks information about the private key, or can make two parties disagree about a computed value in a way an attacker can exploit. More generally, any bounds check or access-control decision computed arithmetically can be made to come out wrong. This is the same shape as a fault attack: an unintended *change*, threatening integrity.

4. Why is "hardware bugs are hard to patch post-deployment" a security argument and not just an economic one? [L09]

> Because the window of exposure is effectively permanent for deployed devices — the vulnerability stays exploitable for the life of the hardware. Mitigations must be pushed to a higher layer (microcode, firmware, compiler, OS), which means they are partial, they cost performance, and they enlarge the set of components that must now be trusted to do the mitigating. The TCB grows as a result of the fix.

L10 5. Information flow

Guiding question: *how can an attacker learn secret information without directly accessing it?* Information flow is the vocabulary that makes the rest of the unit sayable, and the quiz topic line asks for two things: explicit vs. implicit flow, and **which security primitives unintended or indirect flows violate**.

5.1 Explicit vs. implicit flow

The slide's example, in Verilog-ish pseudocode with line numbers, because the line numbers are how the distinction is taught:

```
1  input A
2  output E
3  reg B, C, D;
4  assign E = D;

5  if (A)
6      D <= B;
7  else
8      D <= C;
```

Flow	Where	Why
Explicit: B → D	line 6	The value of B is *assigned into* D. The data itself moves.
Implicit: A → D	lines 5–6	A is never assigned to D. But A decides *which* assignment happens, so D's value depends on A. The information moves through the control flow.

THE ONE-LINE TEST. An **explicit** flow is an assignment: the value is copied. An **implicit** flow is a dependency created by control flow: the secret chose the branch, and the branch chose the value.

Apply it to the official practice Q3 code and you get the released answers directly:

```
if (secret)
    x = public_a;
else
    x = public_b;

output = x;
```

- **Explicit flow:** `x → output` (last line — a plain assignment).
- **Implicit flow:** `secret → x` (the value of `secret` determines which branch assigns `x`).

- **Why observing `output` reveals something about `secret`:** by observing the value of `output`, an observer may be able to determine which branch executed, and therefore learn information about `secret`.

> **Watch out.** The two flows compose. Neither one alone is the leak: `secret → x` is implicit, `x → output` is explicit, and chaining them gives `secret → output`. When a question says "even though secret is never directly assigned to output," it is asking you to describe that chain. Also note this only leaks if `public_a ≠ public_b` — if the two branches assign the same value, the implicit flow carries no information.

5.2 Writing information-flow properties

The deck introduces a notation for "must not flow to," drawn as a squiggly arrow. The AES example on the slide: a module takes `key`, `data` and `rst` as inputs and produces `rdy` and `ciphr`.

```
φ:  key    rdy
```

Read as: information must not flow from `key` to `rdy`. (The slide writes the flow relation; the property being verified is that this flow does *not* exist. Write it as `key  rdy`, or just say in words "no information flows from key to rdy" — the slides accept English.)

The L10 in-class practice sheet is four of these. Worked:

Policy	Property
The secret key should not influence the AES ready signal.	`key  rdy` — no information flows from `key` to `rdy` under any execution.
Privileged information should not flow to user-visible output.	`kernel_secret  user_output`
When the user is not authenticated, key information should not flow to the debug output.	`IF authenticated = FALSE THEN key  debug_output` — note the conditional: the flow is only forbidden in the unauthenticated state.
The `if (secret) access(array[0]) else access(array[1000])` timing case.	`secret  memory_access_time`, or more precisely `secret  attacker-observable timing` — the secret must not influence the observable execution time or the resulting cache state.

> **Exam tip.** Two property notations, two jobs. Use `IF … THEN …` when the policy is about *state changing* ("must not change while locked"). Use `source  sink` when the policy is about *information reaching somewhere* ("must not be revealed to"). Some policies want both — a conditional information-flow property, as in the debug row above.

Extra context beyond the slides: an information-flow property is strictly stronger than a "do not assign" property, and that is the reason the course introduces it. "The key is never written to the debug register" is a statement about explicit flows only. "No information flows from the key to the debug output" also forbids the implicit paths — including timing, if timing is in your set of observables.

5.3 Which security primitives do unintended flows violate?

The quiz topic line names this explicitly, so have the mapping ready.

Kind of unintended flow	Direction	Primitive violated	Example
Secret information leaks out to an observer	secret → attacker	**Confidentiality**	cache timing side channel; Spectre; the AES key influencing `rdy`
Attacker-controlled data reaches protected state	attacker → protected	**Integrity**	Rowhammer flipping a bit in another process's row; a debug write while locked
Flow crosses a privilege boundary at all	either	**Authorization / isolation**	user code influencing or observing kernel state; one process observing another's cache footprint

THE RULE OF THUMB. Ask which way the arrow points. Information flowing *out* of the protected region is a confidentiality violation. Influence flowing *into* the protected region is an integrity violation. This is exactly what official practice Q8.3 wants: cache timing threatens **confidentiality**, a DRAM bit flip threatens **integrity**.

5.4 Check yourself

1. Classify each flow in this snippet:

```
t = secret & 1;
if (t)
    log = "odd";
else
    log = "even";
count = count + t;
```

[L10]

`secret → t` is **explicit** (an assignment, via a computation). `t → log` is **implicit** — `t` is never assigned to `log`, it picks the branch. `t → count` is **explicit**. Composed, `secret` reaches both `log` and `count`, so if either is observable the low bit of the secret leaks.

2. Write an information-flow property for: "a co-tenant VM must not be able to learn anything about another tenant's memory contents through shared hardware." [L10]

> `tenant_A_memory` ⇝ `tenant_B_observables`, where the observables include not only tenant B's architectural reads but also its measurable execution timing and cache state. The qualifier is the whole point — restricting the property to architectural reads would permit exactly the side channels the course is about.

3. A designer says "the key is never assigned to any output wire, so we're fine." What is wrong with this reasoning? [L10 · L07]

> They have ruled out explicit flows only. The key can still influence outputs implicitly — by determining which branch of the control logic runs, how many cycles an operation takes, which memory locations are touched, or when `rdy` asserts. Each of those is an information flow even though no assignment from the key to an output exists.

4. Is every implicit flow a security problem? [L10]

> No. An implicit flow is only a problem when the source is sensitive and the sink is observable by someone who should not learn about the source. A branch on public data that assigns to a public variable is an implicit flow and entirely harmless. The property, not the mechanism, decides — which is why you have to write down the property.

L10 6. Caches and timing side channels

The quiz topic line asks for two things: **the mechanisms required for a timing side channel**, and **how timing observations are used to infer secret information**. Both are conceptual — you will not be asked to compute a hit rate — but the cache background is what makes the mechanism explicable.

6.1 Why caches exist

The observation that drives the whole memory hierarchy: **as capacity goes up and cost goes down, latency gets worse.**

Technology	Capacity	Latency	Cost
Register	1000s of bits	10 ps	$$$$
SRAM	1–8 MB	0.5–1 ns	$2K/GB
DRAM	1–8 GB	50 ns	$20/GB
Hard disk (non-volatile)	100s of GB	10 ms	20¢/GB
What we want	*huge*	*low*	*cheap*

You cannot have all three from one technology, so you stack them:

```
CPU — SRAM "CACHE" — DRAM "MAIN MEMORY" — DISK "VIRTUAL MEMORY" / "SWAP SPACE"
```

PRINCIPLE OF LOCALITY. If a location is accessed in memory, the same location or its neighbors will likely be accessed soon. **Temporal locality:** if an item is referenced, it will tend to be referenced again soon. **Spatial locality:** if an item is referenced, nearby items will tend to be referenced soon.

The cache holds temporary copies of selected main-memory locations, and the average access time is

```
t_ave = α·t_c + (1 − α)·(t_c + t_m) = t_c + (1 − α)·t_m

α    = hit ratio, the fraction of references found in cache
1−α  = miss ratio
t_c  = time to access cache
t_m  = time to access main memory
```

WATCH OUT. You are unlikely to be asked to evaluate this formula, but read what it *says*: access time is a function of whether the data was in the cache. That is the side channel. The performance equation and the leak are the same equation.

6.2 Caches are a microarchitectural optimisation

Architectural level	Microarchitectural level
Load `x` → returns `x`. That is all the ISA promises.	Was `x` cached? Which cache line changed? How long did the access take?

THE SLIDE'S OWN SUMMARY. These differences don't matter much for functionality — but they do for security. **Side-channel attacks turn implementation details into information.**

The thought experiment on the next slide makes it concrete. Given

```
array[0];               // warm up
time(array[0]);   vs.   time(array[1]);
```

the left is faster, because `array[0]` was just accessed and is now cached. Conclusion: **memory access time reveals something about cache state.**

6.3 The mechanisms a timing side channel requires

This is the topic line "conceptually: the mechanisms required for a timing side channel," so here is the checklist. All four must hold:

---

1. SECRET-DEPENDENT BEHAVIOR. The victim's execution must depend on the secret — which address it touches, which branch it takes, how long it runs.

2. SHARED MICROARCHITECTURAL STATE. Attacker and victim must share a resource that records that behavior. Usually a cache; also the branch predictor, the DRAM row buffer, or a power rail.

3. A MEASURABLE DIFFERENCE. The state difference must show up in something the attacker can observe — here, a hit/miss timing gap large enough to distinguish.

4. ATTACKER ABILITY TO MEASURE. The attacker needs a clock (or a proxy) and a way to put the shared state into a known starting condition, so the difference is attributable.

Remove any one and the channel closes. That framing is also how mitigations are classified: partitioning the cache kills #2, constant-time code kills #1, adding timer noise attacks #3 and #4.

6.4 The simple timing side channel

The victim program from the slide:

```
if (secret)
    access(array[0]);
else
    access(array[1000]);
```

The attacker cannot read `secret`. But they can measure timing, and they see `array[0]` → FAST, `array[1000]` → SLOW. Slide conclusion: **the secret was probably true.**

The debrief slide draws the path, and this diagram is worth being able to reproduce from memory because official practice Q5 asks you to fill in its boxes:

```
        Victim                                  Attacker
Secret ──────────── Cache State ──────────── Observation
       computation              timing
```

TIMING SIDE CHANNEL. Secret-dependent behavior changes execution time in a way an attacker can observe. **The attacker never directly reads the secret.**

The fully-labelled four-stage chain, which is the sentence to write whenever a question says "why is this an inference rather than a read":

```
secret → secret-dependent memory access → cache state → timing → attacker
```

EXAM TIP. Official practice Q5.1 and Q5.2 give you `Secret → Secret-dependent memory access → ___ → ___ → Attacker measures`. The released answers are **cache state changes** and **memory access timing**, in that order. Getting the order right matters: the access changes the cache, and the cache changes the timing — not the reverse.

6.5 What is leaking?

The second victim on the slide:

```
if (password[0] == 'A')
    access(table[0]);
else
    access(table[1]);
```

Question	Answer
What is the secret?	The password — specifically, whether its first character is `'A'`.
What is observable by the attacker?	The access times of `table[0]` and `table[1]`; equivalently, which of the two became cached.

What can the attacker infer?	If `table[0]` is fast, `password[0] == 'A'`. One character of the password, per run. Repeat across characters and the whole password falls.

Extra context beyond the slides: note the attack is cheap because it decomposes. A password guessed all-at-once is exponentially hard; a password leaked one character at a time is linear. Side channels frequently convert an exponential search into a linear one, which is why "it only leaks one bit" is rarely reassuring.

6.6 Reading a timing trace

Official practice Q4 hands you a table and a threshold. The skill is mechanical — do not overthink it.

Location	Access time	Hit or miss? (threshold: < 40 cycles = hit)
A	84 cycles	miss — not touched by the victim
B	79 cycles	miss
C	**18 cycles**	**HIT — the victim accessed this one**
D	81 cycles	miss

The reasoning, in the order to write it: the attacker **flushed all four locations first**, so every one started slow. After the victim ran, exactly one is fast. Something must have brought C back into the cache, and the only thing that ran in between was the victim. If A, B, C, D correspond to secret values 0, 1, 2, 3, then `secret ≈ 2`.

WATCH OUT. Write ≈, not =, and be ready to say why (practice Q4.3). The attacker never reads the secret variable. They observe *timing*, use timing to infer *cache state*, and use cache state to infer something about the *victim's behavior* — and from that, the secret. Each arrow is an inference that could in principle be wrong: noise, prefetching, an unrelated access by another process, or cache line sharing can all produce a fast probe that the secret did not cause. That is why real attacks repeat the measurement many times.

6.7 Check yourself

1. Why must the attacker flush the probe array *before* the victim runs? [L10 · L11]

To establish a known baseline. If some probe locations were already cached for unrelated reasons, a fast probe afterwards would not be attributable to the victim — the attacker could not tell "the victim touched this" from "this happened to be resident." Flushing makes every location slow, so any location that is fast afterwards must have been touched in the interval.

2. An attacker measures these probe times with a 50-cycle threshold: P0 = 120, P1 = 31, P2 = 118, P3 = 29, P4 = 125. What can they conclude? [L10]

Two hits, P1 and P3. That is ambiguous: the victim may have accessed both (e.g. the secret-dependent access plus an unrelated one), or one of them is noise or a prefetch of a neighbouring line. The attacker cannot infer a single secret value from this trace and should repeat the experiment — the reading that survives across many trials is the signal.

3. Give the four mechanisms a timing side channel requires, and name the one that a cache-partitioning defense removes. [L10]

(1) secret-dependent victim behavior, (2) microarchitectural state shared between victim and attacker, (3) a difference in that state that is measurable, (4) the attacker's ability to measure it and to set a known baseline. Cache partitioning removes (2) — if the victim and attacker never share cache lines, the victim's footprint is invisible to the attacker's probes.

4. Is a cache a violation of a secure design principle? [L04 · L10]

Yes — the L14 practice sheet asks this directly. A shared processor cache violates **least common mechanism**: it is a mechanism shared by more than one user or process, and shared mechanisms are paths for unintended communication. (It is also the hardware CWE "improper isolation of shared resources.") The point is not that caches are a mistake but that the sharing is what creates the channel.

L11 7. Speculative execution and Spectre

Guiding question: *how can cache timing + speculative execution create a complete attack?* The topic line is unusually specific — "the steps of the attack **and why each step is necessary**" — so this section is organised around justifying every step, not just listing them.

7.1 Why processors speculate

When the CPU encounters

```
if (condition) {
    do_A();
} else {
    do_B();
}
```

it does not wait to learn the result of `condition`. It predicts which path will execute and starts the work early. If the prediction is correct → faster. If it is wrong → it must discard the results.

BRANCH PREDICTOR. Hardware in a CPU that guesses the outcome of a conditional jump. Whether the prediction was correct is not known until the conditional jump has cleared the execution stage of the pipeline. If wrong, the speculatively (or partially) executed instructions are discarded and we start over. A wrong prediction costs modern processors between **10 and 20 cycles**.

The slide poses the question that opens the door: **if the CPU discards the result, does it undo every effect of that execution?** Answer: no. Some microarchitectural side effects can remain — such as changes to the cache.

WATCH OUT. "Speculation is a performance feature, not a bug you can simply delete." Do not write "the fix is to turn off speculative execution." The deck's mitigation list is: add checks, fences, isolation, or compiler transformations — and **many defenses trade security against performance and compatibility**.

7.2 Spectre, simply put

SPECTRE. Spectre tricks a processor into temporarily executing code that should not have run, then uses cache timing to infer what that temporary execution touched.

Three key points from the slide:

- The attacker **does not receive the secret as a normal program output**.
- The attack creates an indirect path — an **implicit flow** — from `secret → cache state → timing`.

- The architectural mistake is rolled back; **the cache footprint may remain**.

The four pieces the attack needs:

#	Piece	Role
1	A bounds check	Code meant to stop invalid accesses
2	A trained prediction	CPU expects the check to pass
3	A secret-dependent access	Secret chooses which cache line changes
4	A timing probe	Attacker finds which line is fast

7.3 The victim code

Every slide in this part of the deck refers back to two lines:

```
if (x < array1_size):
    y = array2[array1[x] * 4096]
```

Three things to notice before the attack starts:

- The bounds check `x < array1_size` is *correct*. The code is not buggy.
- `array1[x]` is the value that becomes the secret when `x` is out of bounds.
- The multiplier **4096** spreads consecutive secret values one page apart, so each possible byte value maps to a *different cache line*. Without it, several secret values would land in the same line and the probe could not distinguish them.

7.4 The steps, and why each is necessary

Step	What happens	Why it is necessary
0. Flush	Flush every `array2[i * 4096]` from the cache using an ISA primitive, making all probe locations slow.	Establishes a known baseline. Without it, a later fast probe cannot be attributed to the victim — the attacker could not distinguish "the victim touched this" from "this was already resident."
1. Train	Call `victim_function(valid_x)` many times with in-bounds values. The CPU learns that the bounds check usually passes.	Speculation follows the *prediction*, not the truth. If the predictor expected the check to fail, the CPU would never speculatively run the body, and there would be nothing to leak. Key idea from the slide: **past behavior influences future predictions**.

2. Mispredict	Call the victim with a malicious, out-of-bounds `x` — chosen as `(address of the secret) - (array1 base)`.	This is what turns `array1[x]` into a read of the secret. If the CPU waited for the check to resolve, the load would not happen — but the trained predictor guesses "in bounds" and speculation proceeds before the check resolves.
3. Speculatively read the secret	During speculative execution the secret value is used *transiently* to compute the address of another memory access.	The secret must influence something before the rollback. The program never gets the secret as a committed result — but the secret can still influence what happens next. This is the moment the architectural boundary is bypassed.
4. Encode into the cache	If the secret byte is 83, then `array2[83 * 4096]` is touched and becomes fast to access later.	The transient value has to be moved into state that *survives* the rollback. The cache is that state. This step converts a register value that is about to be discarded into a persistent, measurable footprint — it is the covert channel's transmit side.
5. Rollback happens	The CPU realizes the prediction was wrong. Architecturally, wrong-path results are discarded; at the ISA level the instructions are redone and the program is not allowed to read the secret.	Not a step the attacker performs — a step the attacker **survives**. Rollback is incomplete: microarchitecturally, the cache may still remember which `array2` location was touched. **Spectre lives in the gap between these two levels.**
6. Measure	Time an access to each `array2[i * 4096]`. Exactly one is unusually fast; if `array2[83 * 4096]` is fast, infer `secret ≈ 83`.	The receive side of the covert channel. The slide's own reassurance: "it is not magic — the attacker is just timing accesses and looking for the unusually fast one."

7.5 The canonical ordering question

Official practice Q6 shuffles six lettered steps. The released order:

B → E → C → A → D → F.

- **B.** The attacker trains the branch predictor.
- **E.** The processor speculatively follows the wrong path.
- **C.** Cache state is changed based on the secret.
- **A.** The CPU realizes its prediction was wrong.
- **D.** The attacker measures cache timing.
- **F.** The attacker infers the secret.

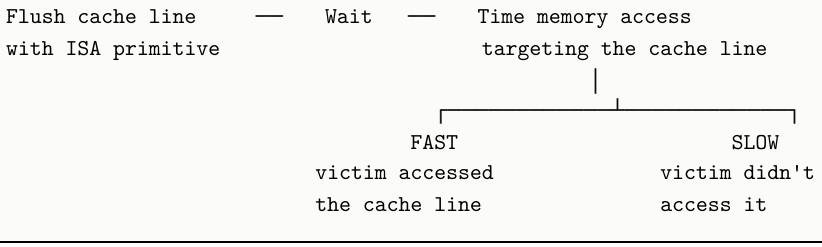
And the follow-up, Q6.1 — **why does step A not eliminate the attack?** The released answer: the processor eventually realizes that the speculative path was incorrect and discards the architectural results; however, microarchitectural effects such as changes to cache state may remain, and the attacker

can measure those remaining effects.

EXAM TIP. The trap in the ordering is putting **A before C**. The cache change has to happen *during* speculation — before the CPU notices its mistake — or there would be nothing left to measure. Remember the sequence as: train, speculate, encode, *then* get caught, then read.

7.6 Flush & Reload

The measurement primitive, drawn on its own slide:




```
flush(array2[0   * 4096]);
flush(array2[1   * 4096]);
...
flush(array2[255 * 4096]);
// ... victim runs ...
time(array2[0   * 4096]);
time(array2[1   * 4096]);
...
time(array2[83  * 4096]);   // FAST
...
time(array2[255 * 4096]);
```

256 probe locations because a byte has 256 possible values. One fast index → one secret byte.

7.7 The gap — the sentence the whole unit is building toward

SUMMARY SLIDE, VERBATIM. Spectre shows that "the program was not allowed to read the secret" is not the same as "the secret could not influence anything observable."

Architectural state	Microarchitectural state
Wrong-path results are discarded. At the ISA level we redo the instructions, and the program is not allowed to read the secret.	Some effects may remain. The cache may still remember which `array2` location was touched.

The full attack path, with the information flow written underneath — reproduce this diagram and you have answered half a dozen possible questions:

```
Train predictor ──  Mispredict bounds check ──  Speculatively read secret
                                                    + determine cache location
                                                                  |
Infer secret  ── Measure cache timing  ── CPU rolls back architectural state

Information flow:
secret → speculative access → cache state → timing → attacker
```

7.8 Check yourself

1. True or false, and correct it if false: "Spectre requires the attacker to directly read the protected secret." [L11]

> **False.** The attacker does not need to receive the secret as a normal program result. The secret can affect microarchitectural state, which the attacker then observes indirectly through timing.

2. True or false, and correct it if false: "For Spectre to work, the speculative instruction must eventually become part of the program's committed architectural execution." [L11]

> **False** — and this is the single most important false statement in the set. The speculative execution may later be discarded; Spectre relies on microarchitectural effects that remain even when the architectural result is rolled back. If the speculation *did* commit, it would not be an attack, it would just be the program running.

3. Why is training the branch predictor necessary? What would happen without it? [L11]

> Speculation follows the predictor's guess. Without training, the predictor has no reason to expect the bounds check to pass, so on the malicious call the CPU would either not speculate down the body at all or would resolve the check before the secret-dependent load issued. No speculative read of the secret means no cache encoding means nothing to measure. Training is what converts a correct bounds check into a bypassable one.

4. Why multiply by 4096 rather than, say, by 1? [L11]

> To give each possible secret byte its own distinct cache line (4096 bytes is a page, comfortably larger than a cache line). With a multiplier of 1, all 256 candidate addresses would fall within a handful of cache lines, and spatial locality — a single line fill bringing in neighbours — would make the probe unable to distinguish which value was used. The stride converts a value into a uniquely identifiable cache footprint.

5. The victim code contains a correct bounds check. Whose bug is Spectre, then? [L11 · L07]

Not the software's — the check is correct and the architecture honours it. It is a gap between the architectural specification (which says the out-of-bounds read does not happen) and the microarchitectural implementation (which performs it transiently and leaves a trace). In L09 terms this is an **imprecise specification**: the ISA never specified what happens to microarchitectural state on a rollback, so implementers were free to leave the cache changed.

L12 · L13 8. Rowhammer and the side-channel landscape

L12 was a guest lecture by Noah Brown; L13 was the group research session. Neither appears by name on the quiz topic list, but official practice Q8.2 asks about DRAM bit flips and Q8.3 asks which primitive they threaten — so the distinction between **observing a signal** and **causing a change** is examinable.

8.1 Side channel vs. fault attack — the distinction that is on the quiz

SIDE CHANNEL. The attacker **observes an unintended signal**. They learn something they should not know. Primarily threatens **confidentiality**. Example: secret-dependent execution changes cache state and the attacker measures the resulting timing difference.

DISTURBANCE / FAULT ATTACK. The attacker **causes an unintended change**. They alter something they should not be able to touch. Primarily threatens **integrity**. Example: repeated accesses to one DRAM row cause a bit in a neighboring row to flip.

EXAM TIP. One-word test: **read or write?** If the attacker ends the attack knowing something new, it is a side channel and the victim's confidentiality lost. If the attacker ends it having changed something, it is a fault attack and the victim's integrity lost. Rowhammer is the fault attack in this course; cache timing is the side channel.

Extra context beyond the slides: Rowhammer is a nice case because it uses a side channel *as a tool* (timing, to reverse-engineer which addresses share a bank — see §8.4) while being a fault attack in its *effect*. If a question asks what Rowhammer "primarily" is, the answer is the effect: an unintended change.

8.2 DRAM organisation

Enough structure to explain the attack:

```
Bank
 ├── Row 0
 ├── Row 1
 ├── …
 └── Row Buffer   ← one per bank
```

How a read works:

1. A read request is sent to a row.
2. The **entire row** is copied into the row buffer.
3. Data in the row buffer is sent to the CPU.

4. The row buffer **acts as a cache**: until a new row is activated, requests are fulfilled directly from the row buffer.

> WATCH OUT. The row buffer is a second layer of caching, below the CPU caches, and it is the reason the naive attack does not work. Remember there are two things standing between your code and a real DRAM row activation: the CPU cache, and the row buffer.

8.3 What Rowhammer is

> ROWHAMMER. A phenomenon that occurs due to electrical interference at the hardware level. It causes the values of bits to flip (0→1 or 1→0). It is highly reproducible once found, meaning attackers can exploit it. The underlying interference is a byproduct of how DRAM works; it cannot be avoided.

The physics, in the four beats the guest deck uses:

1. The value of a cell (bit) is stored as charge or no charge in the cell.
2. Reading a row causes charge to leak out of adjacent cells.
3. If enough charge leaks, a bit can flip (Kim et al., SIGARCH '14).
4. Therefore an attacker controlling two rows can write to the row *between* them — **even if someone else's memory is there**.

That last line is the security statement: the OS ensures memory stays isolated, and Rowhammer breaks that isolation without ever issuing a write to the victim's memory.

Why the periodic refresh does not save you: charge leakage was known before Rowhammer, so DRAM issues a refresh to all rows roughly every 64 ms, restoring each cell's charge to its full value (or lack of one). Rowhammer accelerates the leakage *beyond* what those periodic refreshes catch. If you hammer a 1 to a 0 before the next refresh, the bit is permanently flipped — the refresh then faithfully restores the wrong value.

8.4 How to actually hammer (three obstacles)

The deck teaches this as a sequence of naive attempts that each fail for an instructive reason. Goal: flip a bit in Row C.

Attempt	Why it fails
`loop: mov (Row B), %eax; jmp loop`	We'd hit the **CPU cache** over and over. After the first access the line is cached and DRAM is never touched again.
`loop: mov (Row B), %eax; clflush(Row B); jmp loop`	We'd hit the **row buffer** over and over. Flushing the CPU cache forces a memory request, but the row is already in the bank's row buffer, so the row is not re-activated.

`loop: mov (Row B); mov (Row D); clflush(Row B); clflush(Row D); jmp loop`	**This works.** Alternating between two rows in the same bank evicts the row buffer every time, forcing a real activation on each access. This is **double-sided hammering**: B and D sandwich the victim row C.

Remaining obstacle: **we only see physical addresses, and we do not know which rows they map to.** Page tables translate virtual → physical, but physical → DRAM location is an undocumented vendor function. So the attacker reverse-engineers it — with a timing side channel:

Observation	Reason
Alternating accesses to two addresses in **different banks** is **fast**	Each bank has its own row buffer, so both accesses keep hitting their own row buffer.
Alternating accesses to two addresses in the **same bank** is **slow**	The two rows are fighting for the same row buffer, so every access forces a new activation.

Repeat over large amounts of memory, apply some math, and you recover the address-mapping functions. Bottom line from the slide: **by using a timing side channel, we can find rows that map to the same bank — which is what allows us to Rowhammer effectively.**

8.5 What Rowhammer buys an attacker, and the mitigation arms race

- Escape browser sandboxes (Seaborn and Dullien, '15)
- Read private RSA keys (Kwong et al., IEEE S&P '20)
- Change host memory from the GPU (Hu et al., IEEE S&P '26)

The defense, and its break:

	What it does
TRR (Target Row Refresh)	The memory controller keeps a table tracking frequently-activated rows. When a row is activated enough times, it refreshes that row's neighbors — restoring their charge before a flip can happen. Nearly every DDR4+ module today has mitigations. TRR effectively squashed the first class of Rowhammer attacks.
TRRespass (Frigo et al., IEEE S&P '20)	The tracking table is finite. Hammer **many** rows rather than just two, and no single row's activation count crosses the threshold — the table fills with low counts and TRR never fires, while the aggregate hammering still flips bits.

Mitigations were proposed on both the software side (Brasser et al., SEC '17) and the hardware side (Hassan et al., MICRO '21), and later work broke both — software (Cheng et al., TDSC '19) and hardware (Frigo et al., IEEE S&P '20).

Closing remark from the guest lecture. Bugs and vulnerabilities can be patched; Rowhammer cannot — it is a byproduct of how DRAM works. Thus clever mitigations are required.

8.6 L13: the pattern across real side-channel attacks

L13 was a research-and-present session. The attacks assigned were **Meltdown**, **Foreshadow**, **Flush&Reload-style cache attacks**, **GoFetch**, **ZombieLoad** and **Hertzbleed**. You are not expected to know the internals of each; you are expected to know the two patterns Kaki put on the wrap-up slide.

PATTERNS I NOTICED. (1) A hardware optimization or implementation detail creates an observable effect that violates an assumption made by software. (2) Information can leak through hardware behavior even when the attacker cannot directly access the protected data.

The framework for analysing any of them, taken from the assignment instructions — a good scaffold if the quiz hands you an unfamiliar attack:

1. What is the attacker trying to learn or control?
2. What hardware behavior or side channel is exploited?
3. What can the attacker observe?
4. How does the attack turn that observation into useful information?
5. Connect it back to class concepts: cache state, timing, speculation, information flow.

Extra context beyond the slides: the optimisation being exploited is a useful one-word index into each attack — Meltdown and Foreshadow: out-of-order/speculative execution past a permission check; ZombieLoad: internal CPU buffers being forwarded before they are validated; GoFetch: a data-memory-dependent prefetcher treating data as if it were a pointer; Hertzbleed: dynamic frequency scaling, so power behavior becomes timing behavior. In every case an optimisation made execution depend on data it was not supposed to depend on.

8.7 Check yourself

1. An attacker repeatedly accesses two rows adjacent to a victim's page table entry until a bit flips, then gains write access to memory they did not own. Side channel or fault? Which primitive is threatened? [L12]

Fault (disturbance) attack — the attacker causes an unintended change rather than observing an unintended signal. The primitive primarily threatened is **integrity**. (Authorization is threatened downstream once the corrupted page table entry grants access, and confidentiality follows, but the direct violation is integrity.)

2. Why does `clflush` alone not suffice for hammering? [L12]

`clflush` evicts the line from the CPU cache, so the access does reach DRAM — but the bank's **row buffer** still holds that row, so the request is satisfied from the row buffer without re-activating the row. Charge leakage is caused by row *activations*, not by memory requests. You must alternate between two different rows in the same bank to force an activation each time.

3. Rowhammer needs two rows in the same bank. How does the attacker find them without vendor documentation? [L12]

With a timing side channel. Alternate accesses to two candidate physical addresses and time the loop: fast means different banks (each hits its own row buffer), slow means the same bank (the rows contend for one row buffer). Repeat across a large region and solve for the address-mapping functions. A side channel is used as reconnaissance for a fault attack.

4. Why can't Rowhammer be "patched"? [L12]

It is not a defect in a design that could have been made correctly — it is a byproduct of how DRAM physically works, namely that reading a row leaks charge from adjacent cells, and that cells are packed densely enough for that leakage to matter. There is no erratum to fix. Only mitigations are possible (refresh policies like TRR, ECC, allocation policies), and each mitigation has so far been circumvented by attacks that adapt to it — TRR by TRRespass, for instance.

5. State the single sentence that links Spectre, Meltdown, GoFetch and Hertzbleed. [L13]

In each, a hardware optimization creates an observable effect that violates an assumption software was making — so information leaks through hardware behavior even though the attacker never gains direct access to the protected data.

L14 9. Operating systems security (background)

Guiding question: *how does the operating system create isolation and manage access?* L14 was taught two days before the quiz and its material is **not** on the announced topic list — but it was accompanied by a practice sheet, so it is cheap insurance. Read this section once; do not let it displace §7.

9.1 User mode vs. kernel mode

Operating system.
Software that manages hardware and software resources, and provides common services for software programs

System call (syscall).
The mechanism by which a program in user space asks the operating system to do something on its behalf

The CPU can operate in several modes; the two common ones are **user mode** and **supervisor (kernel) mode**. When executing OS code the CPU is in supervisor mode, where more instruction types and more memory ranges are accessible. Two rules from the slide:

- The processor should **never** execute user-provided code with supervisor-level privileges.
- **System calls are the mechanism for mode transition** — the controlled transition from §2.4, seen from the software side.

9.2 The five responsibilities of the OS

Practice sheet Q2 asks for these by name. Memorise the list with its syscall examples:

Responsibility	Syscall examples	Expanded on the slides as
Manage resources	`fork()`, `mmap()`	allocation · sharing · protection
Provide an interface to hardware	`read()`, `write()`	memory · file system · device I/O
Provide user authentication	`getuid()`, `setuid()`	user accounts · roles · passwords
Mediate interprocess communication	`pipe()`	shared memory · message passing
Protect itself	`kill()`, `mprotect()`	data · code · permissions

On the last one the deck is explicit about why it is a responsibility and not vanity: the data structures and code the OS uses to enforce access control, process separation and fair time sharing all reside in memory, so they need to be protected from other code running on the machine.

9.3 Five features of a trusted system

TCB.
should be small and well defined

Trusted path.
authenticates the OS to the user

Secure start-up.
starts in a known good state

Object sanitization.
no object reuse — free your memory!

Auditing.
audit log tracks any changes

Extra context beyond the slides: "trusted path" runs the *opposite* direction from the usual authentication story. Passwords authenticate the user to the system; a trusted path authenticates the system to the user, so that a fake login prompt drawn by a malicious program cannot harvest your credentials.

9.4 Four styles of separation

Style	Slide annotation	Examples
Physical	not efficient, not complex, secure	air gapping; separate machines; a hardcoded memory fence
Temporal	more efficient, more complex, less secure	time sharing; object sanitization — object reuse requires sanitization
Logical (aka algorithmic)	more efficient, more complex, less secure	a configurable fence register; browser tab isolation; process address spaces
Cryptographic	issues of key management	encrypting each record under a different recipient's key

The practice sheet asks you to classify four scenarios. Answers:

Scenario	Type	Why
Memory fence	**physical** (the hardcoded version)	A hardcoded address in the processor marking where OS code starts — a fixed, unconfigurable boundary. Note the slides show the *fence register* version as **logical**, because making it configurable turns a physical boundary into an algorithmic one.
Many programs executing on a single processor	**temporal**	Time sharing — they are separated by *when* they run, not by where they are.
Code and data of one browser tab kept separate from another	**logical**	Same hardware, same time; separated by an algorithmic mechanism (address spaces, process boundaries).

A published grade spreadsheet with each line encrypted under that student's public key	**cryptographic**	The data is literally shared with everyone; only key possession separates who can read what.

EXAM TIP. The one-question test: **what is doing the separating?** Different hardware → physical. Different time slots → temporal. Software/addressing rules → logical. Different keys → cryptographic. The trade-off runs consistently down the table: efficiency up, complexity up, security down.

9.5 Memory protection mechanisms

Mechanism	How it works	Limitation
Fence	A hardcoded address in the processor denoting the starting point of OS code. Everything below is OS, everything above is user.	Protects the OS from users but not users from each other; the hardcoded version cannot adapt to different memory layouts.
Fence register	The fence becomes configurable, so the boundary can move between configurations.	Still one boundary — still only OS-vs-user.
Base–bounds registers	A `base` and a `bound` register delimit the current process's region, preventing code from one process from interfering with another.	One contiguous region per process; no per-region permissions.
Segment addressing	A table of segment descriptors per process: `base addr · length · R W X M F`. Addressing is `<segment #, offset>`. Introduces virtual memory — each process gets an isolated view of memory — and allows separate access permissions (read, write, execute, mode i.e. execute-as-supervisor, fault) per region.	More complex; the descriptor tables themselves become protected state that must be defended.

Practice sheet Q3 asks which statement best describes base–bounds and segmented memory protection. The answer: **"Hardware and the operating system work together to protect the memory of one process from access by another process."** Both halves matter — the OS populates the registers and descriptor tables, the hardware enforces them on every access. Neither could do it alone, which is §2.1 in miniature.

WATCH OUT. The distractor "a process enforces its own isolation mechanism" is wrong for a reason worth stating: self-enforced isolation is not enforcement at all, since a malicious process simply declines to enforce it. Isolation must come from a layer the process cannot reach.

9.6 Race conditions

RACE CONDITION. Concurrent access of a resource is not serializable. It occurs when (1) two or more processes have access to the same resource, and (2) the final state of the resource depends on the relative timing of the two processes.

The ATM example. Balance starts at 900; two withdrawals interleave:

```
   ATM 1                 Bank                  ATM 2
                    balance = 900
w/d $100                                        w/d $50
             get_bal() → 900      get_bal() → 900
  $100 out                                        $50 out
             set_bal(800)         set_bal(850)
                    balance = ??
```

The customer withdrew $150 and the balance ends at 800 or 850 depending on which write lands last — either way the bank has lost money.

THE SLIDE'S FRAMING, WHICH IS THE EXAMINABLE BIT. The bug is *not* "two users accessed the account at once." The bug is that the supposedly single operation `withdraw` was implemented as multiple interleavable operations. Potential solution: use locking or transactions so only one withdrawal updates the balance at a time.

The L14 practice sheet also asks which principle a **TOCTTOU** (time of check to time of use) vulnerability violates. The answer is **complete mediation**: the check and the use are separated in time, so the authorization decision is made once and then relied on at a later moment when it may no longer hold. Complete mediation demands the check happen at every access, not once up front.

9.7 Check yourself

1. Two engineers from separate groups must review any code before commit. Which principle? [L04 · L14]

Separation of privilege — a sensitive action requires more than one condition or party, so no single compromised or malicious individual can perform it alone.

2. A shared memory cache on a processor violates which principle? [L04 · L10 · L14]

Least common mechanism — mechanisms shared between users or processes are potential channels for unintended information flow, which is exactly what a cache timing side channel exploits.

3. Name the five OS responsibilities and give a syscall for each. [L14]

Manage resources (`fork`, `mmap`); provide an interface to hardware (`read`, `write`); provide user authentication (`getuid`, `setuid`); mediate interprocess communication (`pipe`); protect itself (`kill`, `mprotect`).

4. Object sanitization is which style of separation, and what goes wrong without it? [L14]

Temporal. Resources are reused across time, so the separation between an old owner and a new one is purely a matter of when. Without sanitization, the next process to receive a freed page or buffer reads the previous owner's residual data — a confidentiality violation via object reuse. Hence the slide's parenthetical: no object reuse, free your memory.

10. Mock quiz

Nine questions in the same shapes as the official practice set, with different scenarios and numbers. Do it closed-book on paper, with a 40-minute timer, before you read §11.

10.1 Questions

Question 1: Trusting the hardware. A secure enclave is supposed to prevent code outside the enclave from reading enclave memory. A flaw in the memory controller lets a DMA-capable device read those addresses.
1.1. Why is the memory controller part of the system's TCB?
1.2. The memory controller is trusted. What evidence here bears on whether it is *trustworthy*?
1.3. What assumption made by the enclave software has failed? [L05 · L06]

> **1.1.** Because the security of the system relies on it: it mediates access to memory, and the enclave's confidentiality guarantee depends on it correctly enforcing that boundary. In Goguen and Meseguer's phrasing, it is a subsystem permitted to violate the global policy — it can reach any physical address — so it must be counted as trusted.
> **1.2.** The flaw is evidence *against* trustworthiness: it shows the component does not meet its security specification, since it permits an access the specification forbids. Note that this does not change whether it is trusted — the system still depends on it either way.
> **1.3.** The enclave software assumed the hardware would block all non-enclave reads of enclave physical memory, including accesses that do not come from the CPU. It delegated that check downward and had no way to verify it.

Question 2: Security properties. A specification states: *Once* ***`secure_boot_done = 1`****, the value of* ***`boot_key`*** *must not be readable on any external interface.*

2.1. Write a security property expressing this rule.

2.2. The following execution occurs: **`secure_boot_done = 1; jtag_read(boot_key);`** **the key appears on the JTAG output. Why is this a counterexample?**

2.3. The team runs a 72-hour randomized simulation campaign with no violation. Does this prove the property holds? [L07 · L09]

2.1. `IF secure_boot_done = 1 THEN boot_key  external_interface` — no information flows from `boot_key` to any externally visible output once secure boot has completed. An `IF … THEN external_read = FALSE` form is also acceptable; the information-flow form is stronger because it also forbids indirect paths.

2.2. The execution reaches a state where the antecedent holds (`secure_boot_done = 1`) and the consequent nevertheless fails — key information reaches an external interface. It exhibits a concrete, feasible sequence of events violating the stated property, which is exactly what a counterexample is.

2.3. No. Testing checks only the executions that were actually run; a randomized campaign, however long, still covers a vanishing fraction of the state space, and the violating input may be one it never generated. This is the **coverage** limitation. Formal verification attempts to reason about all executions represented by its model and assumptions — subject, in turn, to whether the model is faithful.

Question 3: Information flow. Consider:

```
len = strlen(secret_token);
if (len > 16)
    buf = big_pool;
else
    buf = small_pool;

memcpy(buf, greeting, 8);
send(buf);
```

3.1. Give one explicit flow.

3.2. Give one implicit flow.

3.3. The attacker only ever sees the bytes of **`greeting`****, which are constant. Could they still learn anything about** **`secret_token`****?** [L10]

3.1. `secret_token → len` (an assignment via a computation), or `greeting → buf` (the `memcpy`).

3.2. `len → buf` — the value of `len` selects which branch assigns `buf`, so `buf`'s value depends on `len` without `len` ever being assigned into it. Composed with 3.1, this yields `secret_token → buf`.

3.3. Yes. The sent *contents* are constant, but the choice of buffer is not, and anything that reveals which buffer was used — the destination address, the allocation pattern, the cache lines touched, the timing of the copy — leaks whether the token is longer than 16 characters. The leak rides on the implicit flow even though the explicit output carries no secret data.

Question 4: Reading a timing trace. An attacker flushes five probe locations, lets the victim run, and measures:

Location	Access time
P0	**142 cycles**
P1	**138 cycles**
P2	**147 cycles**
P3	**26 cycles**
P4	**140 cycles**

A hit is any access below 60 cycles.
4.1. Which location did the victim most likely access?
4.2. The victim executes `access(probe[secret])`, with P0–P4 corresponding to secret values 0–4. What does the attacker infer?
4.3. Why did the attacker have to flush first? [L10]

4.1. P3 — at 26 cycles it is the only access below the 60-cycle threshold, so it is the only cache hit.
4.2. `secret ≈ 3`. Write the approximate sign: it is an inference from timing to cache state to victim behavior, not a read of the variable.
4.3. To establish a known baseline in which every probe location is slow. Without the flush, a fast probe afterwards would be unattributable — it could have been resident before the victim ran, for unrelated reasons. Flushing makes "fast" mean "something touched this during the interval," and the victim is the only thing that ran in the interval.

Question 5: Trace the channel. Fill in the boxes:
`Secret → ______ → cache state changes → ______ → attacker infers secret`
5.1. What belongs in the first box?
5.2. What belongs in the second box?
5.3. The victim's function returns `void` and prints nothing. Explain why there is still a security problem. [L10 · L11]

5.1. A secret-dependent memory access (the secret selects which address the victim touches).
5.2. A measurable difference in memory access timing (the attacker times each probe location).
5.3. Because the secret affects an attacker-observable property indirectly: `secret → cache state → timing → attacker`. Confidentiality is a property of what an observer can *learn*, not of what the program *returns*. A function that outputs nothing can still leak, because the leak travels on the implicit path through microarchitectural state.

Question 6: Ordering. Put these in order and justify two adjacencies.
W. The attacker times each of the 256 probe locations.
X. The processor discards the wrong-path architectural results.
Y. The attacker calls the victim repeatedly with in-bounds indices.
Z. The speculatively-read secret byte selects which `array2` line is loaded.
V. The attacker calls the victim with an out-of-bounds index.
U. The attacker flushes the entire probe array. [L11]

U → Y → V → Z → X → W. Flush the probe array to set a baseline; train the predictor with valid indices; invoke with the malicious index so the trained predictor speculates past the bounds check; the secret is read transiently and encoded into a cache line; the CPU then resolves the branch and rolls back the architectural state; finally the attacker probes by timing.
Two adjacencies worth justifying. *Y before V:* training must precede the malicious call, or the predictor has no reason to guess "in bounds" and the body is never speculatively executed. *Z before X:* the cache encoding must happen during speculation; once the CPU has resolved the misprediction, the wrong-path work stops and there is nothing further to encode. (U must also precede Z, or the probe array might already be cached.)

Question 7: Concept check. True or false; correct the false ones.
7.1. Disabling the branch predictor would close the timing side channel in `if (secret) access(a) else access(b)`.
7.2. Spectre's out-of-bounds read is possible because the victim's bounds check is written incorrectly.
7.3. A cache hit is faster than a cache miss, and that difference is what the attacker measures.
7.4. If the CPU never rolled back mispredicted work, Spectre would not exist.
[L10 · L11]

7.1. False. That channel needs no speculation at all. The victim genuinely executes the secret-dependent access, and the cache state it leaves behind is measurable. Speculation is what makes *Spectre* possible; plain cache timing channels predate it and survive without it.
7.2. False. The bounds check is correct, and architecturally it is honoured — the program never commits the out-of-bounds read. The problem lives in the gap between the architectural specification and the microarchitectural implementation, which performs the access transiently and leaves a cache trace.
7.3. True.
7.4. False, and backwards. Rollback is what makes the attack *invisible* architecturally; without it the illegal read would commit and be a far more obvious access-control failure. Spectre exists because rollback is *incomplete* — it restores architectural state but not microarchitectural state.

Question 8: Side channel or fault?
8.1. An attacker measures the power draw of a smartcard and recovers the bits of the key being processed.
8.2. An attacker shines a laser at a chip during a signature computation, causing an incorrect intermediate value that reveals the private key.
8.3. Which primitive is most directly threatened in each? [L12]

8.1. The attacker observes an **unintended signal** — a side channel (power analysis). Nothing about the card's operation is altered.
8.2. The attacker causes an **unintended change** — a fault/disturbance attack, the same category as Rowhammer.
8.3. 8.1 primarily threatens **confidentiality**. 8.2 primarily threatens **integrity** — the computation is corrupted. (Its *consequence* is a confidentiality loss once the key is recovered, but the question asks what the attack directly violates: the distinction is observe-vs-change, and a laser-induced fault is a change.)

Question 9: Synthesis. A hardware team model-checks their new accelerator against a full set of functional properties and every proof passes. The chip ships. Six months later a paper shows the accelerator leaks key bits through completion timing. Did the verification fail? Answer in a short paragraph. [L07 · L09 · L10]

The verification did not fail on its own terms — it proved what it was asked to prove. What failed is the set of properties. Functional correctness says only that the design produces the behavior its architectural specification requires; it says nothing about whether secret data influences timing, cache state, power, or other observable microarchitectural effects. A completion-timing leak is an information-flow violation (`key  completion_time`), and no functional property forbids it. Two lessons follow. First, a design can be correct at one level and insecure at another — Spectre is the same story. Second, in L09's taxonomy this is an **incorrect or imprecise specification**: the bug is in what was written down, not in the silicon's fidelity to it. Adding a timing-independence property to the spec is the fix; the model checker would then have had something to catch.

11. The official practice set, walked

These are the nine questions from `435-fa26-ep2.pdf` with the released solutions, plus commentary on what each one is really testing and where students lose points. If you only have twenty minutes left, read this section.

1. Trusting the hardware. An OS is supposed to prevent user programs from modifying a protected configuration register. A processor bug allows a user-mode instruction, under certain conditions, to modify the register. (1.1) Why is the processor part of the TCB? (1.2) The processor is trusted — what evidence would make us question whether it is trustworthy? (1.3) What assumption made by the OS has failed? [L05 · L06]

Released answers. (a) The processor is part of the TCB because system security depends on it correctly enforcing protection and privilege boundaries. (b) The bug demonstrates that the processor may fail to enforce the security behavior that the system relies on. It is trusted because security depends on it, but this behavior gives us reason to question whether it is trustworthy. (c) The operating system assumed that the hardware would prevent user-mode code from modifying protected state.

What it is testing. Whether you can keep *trusted* and *trustworthy* apart under pressure. The give-away sentence in the released answer for (b) — "it is trusted *because* security depends on it, but this behavior gives us reason to question whether it is trustworthy" — is worth reproducing almost verbatim. Part (c) is the L06 trust-stack slide: the OS has no independent enforcement mechanism, so it assumed the layer below.

2. Security properties. Spec: "Once `lock = 1`, the value of `config` should not change." (2.1) Write a property. (2.2) Why is `lock = 1; debug_write(config, 42); config becomes 42` a counterexample? (2.3) 10,000 tests pass — does that prove the property? [L07]

Released answers. (a) `lock = 1  config_next = config_current`. Equivalent English or pseudocode answers are acceptable. (b) The execution reaches a state where `lock = 1`, but `config` nevertheless changes. It therefore demonstrates an execution that violates the stated property. (c) No. Testing checks only the executions that were actually tested. Passing many tests does not prove that no other execution can violate the property. Formal verification attempts to reason about all executions represented by its model and assumptions.

What it is testing. The mechanical property/counterexample skill from §3.4. Two point-losers: writing `config = config` instead of `config_next = config_current` (the property is about the *transition*, not an equality), and answering (b) with "because a debug write happened" — the counterexample is not the debug write, it is the fact that the antecedent held and the consequent failed. For (c), do not oversell formal verification; the released answer includes the phrase "represented by its model and assumptions" for a reason.

3. Information flow. `if (secret) x = public_a; else x = public_b; output = x;` (3.1) One explicit flow. (3.2) One implicit flow. (3.3) Why might observing `output` reveal information about `secret`? [L10]

Released answers. (a) One explicit flow is `x → output`. (b) One implicit flow is `secret → x`; the value of `secret` determines which branch assigns `x`. (c) By observing the value of `output`, an observer may be able to determine which branch executed and therefore learn information about `secret`.

What it is testing. That you attach the right label to the right arrow. The commonest error is swapping them — calling `secret → x` explicit because `secret` appears on the line above the assignment. It is implicit precisely because `secret` is never assigned into `x`; it only chooses which assignment runs. Part (c) wants the two-hop chain spelled out: branch taken → value of `x` → value of `output`.

4. Reading a timing trace. Four flushed probes: A = 84, B = 79, C = 18, D = 81 cycles; hit threshold 40. (4.1) Which location did the victim access? (4.2) With A–D mapping to secret values 0–3 and the victim running `access(probe[secret])`, what is the inferred secret? (4.3) Why is this an inference rather than a direct read? [L10]

Released answers. (a) C — its access time of 18 cycles is below the 40-cycle threshold. (b) `secret` ≈ 2. (c) The attacker never reads the secret variable itself. Instead, the attacker observes timing, uses timing to infer cache state, and uses cache state to infer something about the victim's behavior.

What it is testing. Pure mechanics for (a) and (b) — read the table, apply the threshold, map the index. Part (c) is the real question and the released answer is a three-link chain: timing → cache state → victim behavior. Write it as a chain, not as "because the attacker used timing." And keep the ≈: each link is an inference that noise, prefetching or an unrelated access could in principle break.

5. Trace the side channel. `Secret → secret-dependent memory access → ___ → ___ → attacker measures`. (5.3) Why is this a security problem even if the victim never prints or returns the secret? [L10]

Released answers. (a) Cache state changes. (b) Memory access timing. (c) The secret affects an attacker-observable property indirectly: `secret → cache state → timing → attacker`. Therefore information about the secret can leak even without a direct program output.

What it is testing. Whether you have the causal order right. The access changes the cache; the cache changes the timing. Reversing them ("timing → cache state") shows the mechanism is not understood. Part (c) is the thesis of the unit in one sentence: confidentiality concerns what an observer can learn, not what the program outputs.

6. Putting Spectre in order. A: the CPU realizes its prediction was wrong · B: the attacker trains the branch predictor · C: cache state is changed based on the secret · D: the attacker measures cache timing · E: the processor speculatively follows the wrong path · F: the attacker infers the secret. (6.1) Why does step A not eliminate the attack? [L11]

Released answers. Order: **B → E → C → A → D → F**. The processor eventually realizes that the speculative path was incorrect and discards the architectural results. However, microarchitectural effects such as changes to cache state may remain. The attacker can measure those remaining effects.

What it is testing. That you know the cache encoding happens *inside* the speculation window — C before A. If you place A third, you have implicitly claimed the CPU finishes rolling back before the secret is encoded, which would mean there is nothing to measure. Note the set omits the flush step; if a version of this question includes it, flush comes first, before training.

7. Spectre concept check. (7.1) Spectre requires the attacker to directly read the protected secret. (7.2) Speculatively executed instructions can affect cache state. (7.3) For Spectre to work, the speculative instruction must eventually become part of the committed architectural execution. (7.4) Timing differences can reveal which cache location was accessed. [L11]

Released answers. (a) **False.** The attacker does not need to receive the secret as a normal program result. The secret can affect microarchitectural state that is observed indirectly. (b) **True.** (c) **False.** The speculative execution may later be discarded. Spectre relies on microarchitectural effects that remain even when the architectural result is rolled back. (d) **True.**

What it is testing. The two falses are the same misconception from two directions: the belief that a leak requires the secret to reach the program's real output. Also note the instruction — "if false, briefly correct the statement." A bare "False" earns partial credit at best; supply the corrected version.

8. Side channel or fault? (8.1) Secret-dependent execution changes cache state and the attacker measures the timing difference. (8.2) Repeated accesses to one DRAM row flip a bit in a neighboring row. (8.3) Which security primitive is most directly threatened by each? [L10 · L12]

Released answers. (a) The attacker primarily **observes an unintended signal** — this is a side channel. (b) The attacker **causes an unintended change** — this is a disturbance/fault attack. (c) The cache-timing example primarily threatens **confidentiality**; the DRAM bit flip primarily threatens **integrity**.

What it is testing. Observe vs. change, and the mapping onto C/I. Do not be pulled into "but Rowhammer can be used to read keys, so it's confidentiality" — the question says *most directly*, and the direct violation of a bit flip in memory you do not own is integrity.

9. Synthesis. A processor produces the correct architectural output for a program, but its execution time varies depending on a secret value. Can the processor be functionally correct and still create a security problem? [L07 · L10 · L11]

Released answer. Yes. Architectural or functional correctness only tells us that the processor produces the behavior required by its architectural specification. If a secret also changes timing, cache state, power use, or another observable microarchitectural effect, information may still leak. Correct functionality therefore does not automatically imply secure information flow.

What it is testing. This is the essay question and it is the whole unit compressed. A full-credit answer has three moves: (1) say what functional correctness does and does not claim; (2) name the gap — architectural specification vs. microarchitectural implementation; (3) name the leak in information-flow terms, `secret` `timing`, and note that no functional property forbids it. Adding a concrete instance (Spectre, or the `if (secret)` cache channel) costs one sentence and makes the answer concrete.

12. One-screen cram sheet

Everything the quiz topic list names, in the order you would want it in the last ten minutes before the room opens.

Trust vs. trustworthy. Trusted = relied upon to meet its security specs (about *your* dependence). Trustworthy = deserving of trust, actually meets them (about the *component*). A vulnerability is the gap.

TCB. Components on which the security of the system relies. Goguen & Meseguer: "a subsystem permitted to violate some global security policy." Should be **small and well defined.** .

Honest but curious. An adversary that gleans information while adhering to the protocol. Useful because anything it learns is leaked by the protocol's design — it reveals information leaks in protocols. Not the strongest model.

Trust stack. Applications → OS → ISA/Processor → Hardware. Each layer assumes the one below. Hardware is the root: nothing beneath it to appeal to.

Security boundary. Separation between parts with different trust/privilege/access. Crossing one should require a controlled check → **complete mediation.** .

HW-enforced privilege. CPU tracks the privilege level · some instructions are privileged-only · user requests for privileged ops are trapped to the OS. Transition via syscall / interrupt / trap.

HW guarantees expected. Correct execution · restricted operations · controlled transitions · memory checks. **Nothing about timing, cache, or power.** — that omission is the whole unit.

Orange Book vs. CC. Orange Book: DoD, 1970s, 6 rankings, **couples.** features + assurance. Common Criteria: international, 1990s, 7 assurance levels, **decouples.** them.

Trusted-systems process. Specification (policy + functionality) → design (model, design, implement) → evaluation (testing, red/blue team, formal verification).

Three levels. ISA/architecture = what software sees. Microarchitecture = how the processor actually does it. RTL = the hardware description. Side channels live below the architectural abstraction.

Policy vs. property. Policy = what the system should/should not allow, to protect assets. Property = a precise rule the design must always satisfy. Counterexample = a sequence of events showing it violated.

Property template. `IF ___ THEN ___`. For "must not change": `lock = 1  config_next = config_current`. For info flow: `secret  observable`. Four parts: asset, forbidden behavior, property, counterexample.

Counterexample recipe. Three lines. 1. Establish the antecedent. 2. Perform an action. 3. Observe the consequent fail. Need not be the full real-world attack.

Testing vs. model checking. Testing: inputs → run → look for failure; limited by coverage. ; wins on scalability; all tests passing shows nothing. Model checking: design + property → proof or counterexample; limited by scalability. ; wins on completeness; covers all executions *of the model*.

CWE vs. CVE. CWE = a class of weakness, reusable across designs. CVE = one specific vulnerability in one product. Many CVEs per CWE. CWEs let the community compare bugs instead of treating each as unique.

HW CWE examples. Improper access control for protected resources · improper isolation of shared resources · incorrect privilege management · debug/test interface exposed · side-channel exposure via timing, power, or shared state.

Three origins of a HW bug. Errata (implementation ≠ spec) · incorrect spec (faithful implementation of the wrong rule) · imprecise spec (vagueness filled in badly). Hardware bugs are hard to patch post-deployment.

Pentium FDIV (1994). Missing lookup-table entries in the divider → rare wrong quotients. 4195835 ÷ 3145727 wrong in the 5th decimal. $475M replacement cost. Turning point for formal methods funding.

Explicit vs. implicit flow. Explicit = an assignment copies the value (`x → output`). Implicit = control flow creates the dependency (`secret → x` via the branch). They compose into `secret → output`.

Which primitive?. Information flowing *out* of the protected region → confidentiality. . Influence flowing *in* → integrity. . Crossing a privilege boundary at all → authorization/isolation. .

Memory hierarchy. Register (10 ps) → SRAM cache (0.5–1 ns, 1–8 MB) → DRAM (50 ns, 1–8 GB) → disk (10 ms). Justified by locality: temporal (same item again) + spatial (neighbours).

Cache: arch vs. microarch. Architecturally, load x returns x. Microarchitecturally: was x cached, which line changed, how long did it take. Side-channel attacks turn implementation details into information.

Timing side channel. Secret-dependent behavior changes execution time in an observable way. **The attacker never directly reads the secret..** Chain: secret → secret-dependent access → cache state → timing → attacker.

Four mechanisms required. (1) secret-dependent victim behavior · (2) microarchitectural state shared with the attacker · (3) a measurable difference · (4) the attacker's ability to measure and to set a baseline. Kill any one and the channel closes.

Reading a trace. Below the threshold = HIT. = the victim touched it. Map the index to the secret value. Write ≈. : timing → cache state → behavior → secret is a chain of inferences, each defeasible by noise.

Why flush first. To establish a baseline where every probe is slow, so a fast probe afterwards is attributable to the victim rather than to prior residency.

Speculation. The CPU predicts the branch and works ahead. Right → faster; wrong → discard. A misprediction costs 10–20 cycles. Rollback restores architectural state, *not* microarchitectural state.

Spectre's four pieces. A bounds check · a trained prediction · a secret-dependent access · a timing probe. Victim: `if (x < array1_size) y = array2[array1[x] * 4096];`.

Spectre steps. Flush → train → mispredict with out-of-bounds x → speculatively read secret → encode into cache line → rollback (incomplete) → time the probes. Ordered practice answer: B → E → C → A → D → F. .

Why each step. Flush = baseline. Train = speculation follows the prediction. Malicious x = turns the load into a secret read. Encode = moves a transient value into surviving state. Probe = reads the channel. Rollback is survived, not performed.

Why × 4096. One page per candidate value, so each of the 256 possible bytes maps to a distinct cache line and spatial locality cannot blur them together.

The gap. "The program was not allowed to read the secret" ≠ "the secret could not influence anything observable." Spectre lives between the architectural and microarchitectural levels.

Side channel vs. fault. Side channel = attacker OBSERVES AN UNINTENDED SIGNAL. → confidentiality. Fault/disturbance = attacker CAUSES AN UNINTENDED CHANGE. → integrity. Test: did they end up knowing something, or having changed something?

Rowhammer. Reading a DRAM row leaks charge from adjacent cells; enough leakage flips a bit in a neighbouring row before the ~64 ms refresh. Byproduct of DRAM physics — cannot be patched, only mitigated.

Hammering correctly. Plain loop → hits the CPU cache. `clflush` loop → hits the row buffer. Must ALTERNATE TWO ROWS IN THE SAME BANK. (double-sided) to force real activations. Find same-bank rows via a timing side channel: same bank = slow, different banks = fast.

TRR / TRRespass. TRR: the memory controller counts activations per row and refreshes neighbours past a threshold. TRRespass: hammer many rows at once so no single count crosses the threshold and the tracking table is overwhelmed.

L13 pattern. A hardware optimization creates an observable effect that violates an assumption software made; information leaks through hardware behavior even when the attacker cannot directly access the protected data.

OS: 5 responsibilities. Manage resources (fork, mmap) · interface to HW (read, write) · user authentication (getuid, setuid) · mediate IPC (pipe) · protect itself (kill, mprotect).

Trusted-system features. TCB (small, well defined) · trusted path (authenticates the OS to the user) · secure start-up (known good state) · object sanitization (no object reuse) · auditing (log tracks changes).

Four separations. Physical (secure, inefficient) · temporal (time sharing; needs sanitization) · logical / algorithmic (address spaces, tabs) · cryptographic (key management). Efficiency ↑, complexity ↑, security ↓ down the list.

Memory protection. Fence (hardcoded OS boundary) → fence register (configurable) → base–bounds (per-process region) → segments (`<seg#, offset>`, per-region R W X M F, virtual memory). HW + OS *together* protect one process from another.

Race condition. Concurrent access to a resource is not serializable: 2+ processes share it, and the final state depends on relative timing. The bug is that one logical operation was implemented as multiple interleavable ones. Fix: locks/transactions. TOCTTOU violates complete mediation.

The synthesis sentence. Functional correctness means the processor meets its architectural specification. If a secret also changes timing, cache state or power, information can still leak. **Correct functionality does not imply secure information flow..**
