

Question 1: Trusting the Hardware A computer's operating system 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 system's **Trusted Computing Base (TCB)**?
- 1.2. In this scenario, the processor is **trusted**. What evidence would make us question whether it is **trustworthy**?
- 1.3. What assumption made by the operating system has failed?

Question 2: Security Properties A hardware specification states:

Once `lock = 1`, the value of `config` should not change.

- 2.1. Write a simple security property expressing this rule. You may use English, pseudocode, or mathematical notation.
- 2.2. Consider the following execution:

```
lock = 1
debug_write(config, 42)
config becomes 42
```

Why is this execution a **counterexample** to the property?
- 2.3. Engineers test the hardware with 10,000 different inputs and never observe a violation. Does this prove the property holds for every possible execution? Briefly explain.

Question 3: Information Flow Consider the following code:

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

output = x;
```

- 3.1. Give one **explicit** information flow in this example.
- 3.2. Give one **implicit** information flow in this example.
- 3.3. Why might observing **output** reveal information about **secret**, even though **secret** is never directly assigned to **output**?

Question 4: Reading a Timing Trace An attacker flushes four probe locations from the cache. After the victim executes, the attacker measures the following access times:

Location	Access Time
A	84 cycles
B	79 cycles
C	18 cycles
D	81 cycles

Assume an access time below 40 cycles indicates a cache hit.

- 4.1. Which location was most likely accessed by the victim?
- 4.2. Suppose the victim executes:


```
access(probe[secret]);
```

 and locations A, B, C, and D correspond to secret values 0, 1, 2, and 3, respectively. What value would the attacker infer for **secret**?
- 4.3. Why is this an **inference** about the secret rather than a direct read of the secret?

Question 5: Trace the Side Channel Fill in the missing steps in the information-flow path below for a cache-based side channel attack.

Secret → Secret-dependent memory access → → Attacker measures

- 5.1. What belongs in the first box?
- 5.2. What belongs in the second box?

5.3. Why is this path a security problem even if the victim never prints or returns the secret?

Question 6: Putting Spectre in Order Put the following steps of a Spectre-style attack in the correct 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.

Correct order: _____

6.1. Why does step A not eliminate the attack?

Question 7: Spectre Concept Check For each statement, write **True** or **False**. If false, briefly correct the statement.

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 program's committed architectural execution.

7.4. Timing differences can reveal which cache location was accessed.

Question 8: Side Channel or Fault? For each scenario, identify whether the attacker primarily **observes an unintended signal** or **causes an unintended change**.

8.1. Secret-dependent execution changes cache state, and an attacker measures the resulting timing difference.

8.2. Repeated accesses to one DRAM row cause a bit in a neighboring row to flip.

8.3. Which security primitive is most directly threatened by the first scenario? Which is most directly threatened by the second?

Question 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? Explain why or why not.

Practice Problem Solutions

1. Trusting the Hardware

- (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.

2. Security Properties

- (a) One possible property is:

$$\text{lock} = 1 \Rightarrow \text{config}_{\text{next}} = \text{config}_{\text{current}}.$$

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.

3. Information Flow

- (a) One explicit flow is:

$$x \rightarrow \text{output}.$$

- (b) One implicit flow is:

$$\text{secret} \rightarrow x.$$

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`.

4. Reading a Timing Trace

- (a) C. Its access time of 18 cycles is below the 40-cycle threshold.
- (b) The attacker would infer:

$$\text{secret} \approx 2.$$

- (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.

5. Trace the Side Channel

- (a) Cache state changes.
- (b) Memory access timing.
- (c) The secret affects an attacker-observable property indirectly:

$$\text{secret} \rightarrow \text{cache state} \rightarrow \text{timing} \rightarrow \text{attacker}.$$

Therefore information about the secret can leak even without a direct program output.

6. Putting Spectre in Order

$$B \rightarrow E \rightarrow C \rightarrow A \rightarrow D \rightarrow F$$

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.

7. Spectre Concept Check

- (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.**

8. Side Channel or Fault?

- (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**.

9. Synthesis

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.