CS161 Summer 2026 — Introduction to Computer Security — Discussion 12

Q1 *Potpourri* **(6 points)**

Q1.1 (1 point) Select all true statements

- ☐ To perform a rowhammer attack, the attacker needs write access
- ☐ Memory deduplication allows the attacker to place any code anywhere
- ■ Rowhammer doesn't require the attacker to have physical access to the machine
- ■ The repeated reads must bypass the cache to work

Solution:
- Attackers only need read access
- Memory deduplication maps the duplicated code to the same address
- Rowhammer can be don't with read instructions
- Repeated reads typically result in data being served from the cache. So this attack must make uncached reads

Q1.2 (1 point) Select all true statements

- ■ A password check that returns at the first wrong character leaks how many leading characters are correct
- ■ Constant-time code should avoid branching on secret values
- ■ Repeated squaring can leak the Hamming weight of a secret exponent through timing
- ☐ Adding a dummy computation prevents all side channel leaks since our code could be padded to run in a fixed time

Solution:
- Early-return checks and repeated squaring both leak via runtime
- Attacker uses timing attack to infer the Hamming weight of a variable. Avoid branching (if-else) on secret values to prevent this. The junk-multiply trick equalizes timing but is still vulnerable to prime-and-probe cache attacks

Q1.3 (1 point) Select all true statements

- [x] Spectre abuses speculative execution to run a forbidden path on secret data
- [ ] When probing, a cache hit indicates the location of the speculated execution
- [x] Separate OS processes mitigate Spectre, but WebAssembly isolation does not
- [x] Probing attacks can be mitigated with shielded rooms (SCIF) and circuit masking

Solution: Spectre speculatively runs the forbidden branch and leaks the secret via cache timing Separate OS provides and SCIF isolation of the software and hardware respectively.

Q1.4 (1 point) Select all true statements

- [x] LLMs are trained primarily for next-token prediction
- [ ] LLMs reliably distinguish the user's instructions from third-party data
- [ ] Inferences are deterministic

Solution: LLMs are optimized for next token prediction: Given some input phrase x, what token should appear next?

During inference, LLMs predict the next token repeatedly to create an output message based on the input prompt. Usually decide which input to pick probabilistically

Q1.5 (1 point) Select all true statements

- [x] A restaurant review saying "ignore all previous instructions" is an example of prompt injection
- [ ] Prompt injection is fundamentally different from SQL injection and buffer overflows
- [x] Jailbreaking aims to make the model do something it shouldn't do for anyone
- [ ] The attacker must get the system-instruction syntax exactly right for injection to work

Solution: Jailbreaking targets the model's own rules; prompt injection smuggles additional instructions through data. This is the same "code as data" confusion as SQL injection and buffer overflows

Q1.6 (1 point) Select all true statements

- ☐ Delimiters / data layers guarantee that data is never treated as instructions
- ■ A quarantined LLM with no tool access limits the damage from an injection
- ☐ LLM defenses prevent the model from relaying misinformation
- ☐ Generative AI can be implemented for defense and never for attack

Solution: Delimiters only give a probabilistic average-case improvement and can be beaten by completion attacks, so they guarantee nothing. Quarantined LLMs and code-generation privilege separation prevent tool misuse

Q2 *Attacks* **(4 points)**

Q2.1 (1 point)

```
1 def login(email_address, password):
2     password_hash = fetch_password_hash(email_address)
3     if password_hash is None:
4         return False
5     return bcrypt.checkpw(password.encode(), password_hash)
```

Which attack is the code above vulnerable to?

- ● Timing/Cache Timing Attack
- ○ Hamming Weight Attack
- ○ RowHammer Attack
- ○ Safe against all above attacks
- ○ Spectre Attack

Q2.2 (1 point) Which line(s) make this program vulnerable to the attack chosen?

- ☐ Line 2
- ■ Line 3
- ■ Line 4
- ☐ Line 5

Why? What information is leaked?

Solution: We exit out of the function early if no password hash is found so an attacker can easily tell if an email is a valid existing account.

We could extrapolate to a case where Mallory only knows Alice and Bob's names, then try a bunch of variations of firstname/lastname and so on until Mallory finds a valid username.

Q2.3 (1 point)

Assume the function below is just one step inside a black boxes block cipher operation that mixes the key with the encrypted data.

- `a` is data that changes per each call.
- `k` is a subkey. It is **the same every call**, and is private
- `r` is a random mask, a new value per each call.

Alice realized that without `r`, an attacker who knows `a`, can simply prime-and-probe to guess each bit of `k` through measuring the power output, since a circuit holding or switching more 1 bits draws more current. By adding a random value `r`, the measurement become unpredictable.

Mallory believes that she can still break it. She can trigger the function as many times as she wants, and can measure power output. However, she never sees the `a`, `r`, or return value.

Is Mallory able to leak information about the subkey `k`?

```
1 def mask(a, k):
2     r = random_32_bits()
3     z = a + r
4     z = z + k
5     z = z - r
6     return z
```

Assume that `z, a, r, k` are all 32-bit values that wrap around in case of overflow.

Which attack is the code above vulnerable to?

- ○ Timing/Cache Timing Attack
- ○ RowHammer Attack
- ○ Spectre Attack
- ● Hamming Weight Attack
- ○ Safe against all above attacks

Q2.4 (1 point) Which line(s) make this program vulnerable to the attack chosen?

- [x] Line 2
- [ ] Line 3
- [x] Line 4
- [ ] Line 5
- [ ] Line 6

Why? What information is leaked?

Solution: We can narrow the range of values we need to guess for k.

We can measure power consumption per run to find the approximate number of 1 bits use in the circuit. Even though r makes this value unpredictable, if we make a huge number of measurements, we can take advantage of the expectation of r and the constant presence of k.

Let |x| = Hamming weight of x (how many of its bits are 1)

measurement $\approx |a + r| + |k| + |a + r + k|$

`r` is a random value so we can set E(|r|) ≈ 16 (since they are 32-bit values)

measurement_avg $\propto 16 + |k| + 16$

After isolating the reading average, what remains is the Hamming weight of k. This decreases the search range that Mallory has to do to guess the value of k.

[Extra]

The leak is actually slightly richer than the weight, there are also "carry" expectation to be accounted to in the readings, which depend on k's bits positionally, which can give more information to distinguish 00001111 from 11110000, further narrowing the guessing range.