UCLA Samueli School of Engineering

Cache Side-Channel Detection and Security

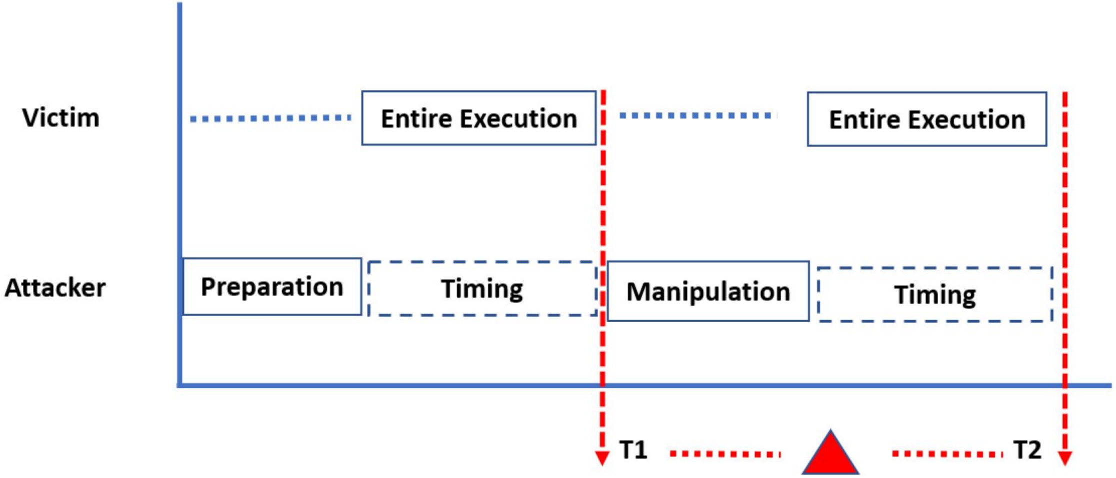
Group 10

Zehao Yuan, HanLuo, Qinghua Gu, Tianqi Chen

Cache Side-Channel Attacks

BACKGROUND

- Cache side-channel attacks exploit the **timing variations** in accessing cache memory to infer sensitive information from a victim process.
- These attacks take advantage of how modern processors manage cache memory, particularly shared caches in multi-core and multi-threaded environments.



Types of Cache Side-Channel Attacks

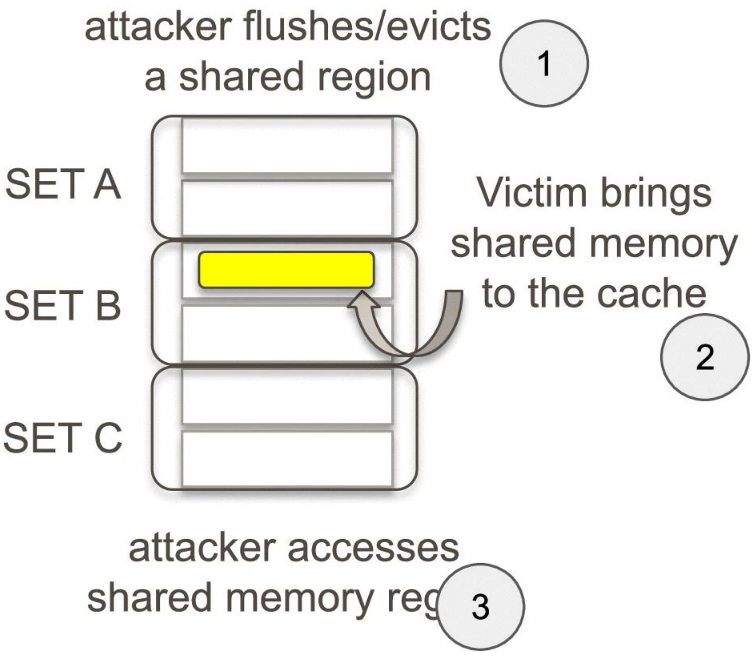
- Flush+Reload
- Prime+Probe
- Evict+Time
- Flush+Flush
- Cache Collision Attacks
- Spectre and Meltdown (Speculative Execution Exploits)
- DRAM Rowhammer (Cache-Related)

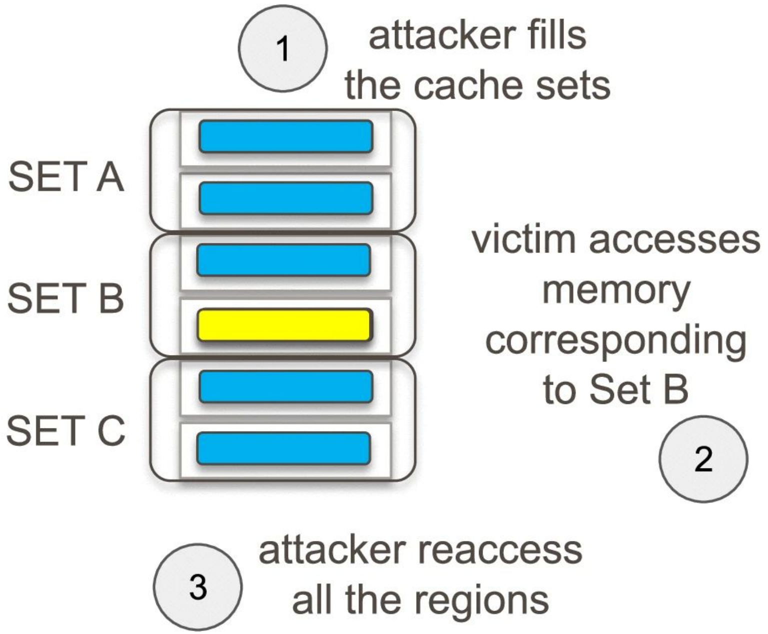
Simulation of attack

Flush+Reload

- **Mechanism**: Attacker flushes a specific cache line using the clflush instruction and then measures the reload time.
- **Goal**: If the victim accesses the same memory address, it gets loaded into the cache, reducing the reload time for the attacker.
- **Scope**: Works across processes, virtual machines, and sometimes even across networks if shared memory is involved.
- **Common Targets**: Cryptographic libraries (e.g., AES, RSA), keystrokes, control flow of programs.

Flush+Reload





Flush+Reload

Example python code setup

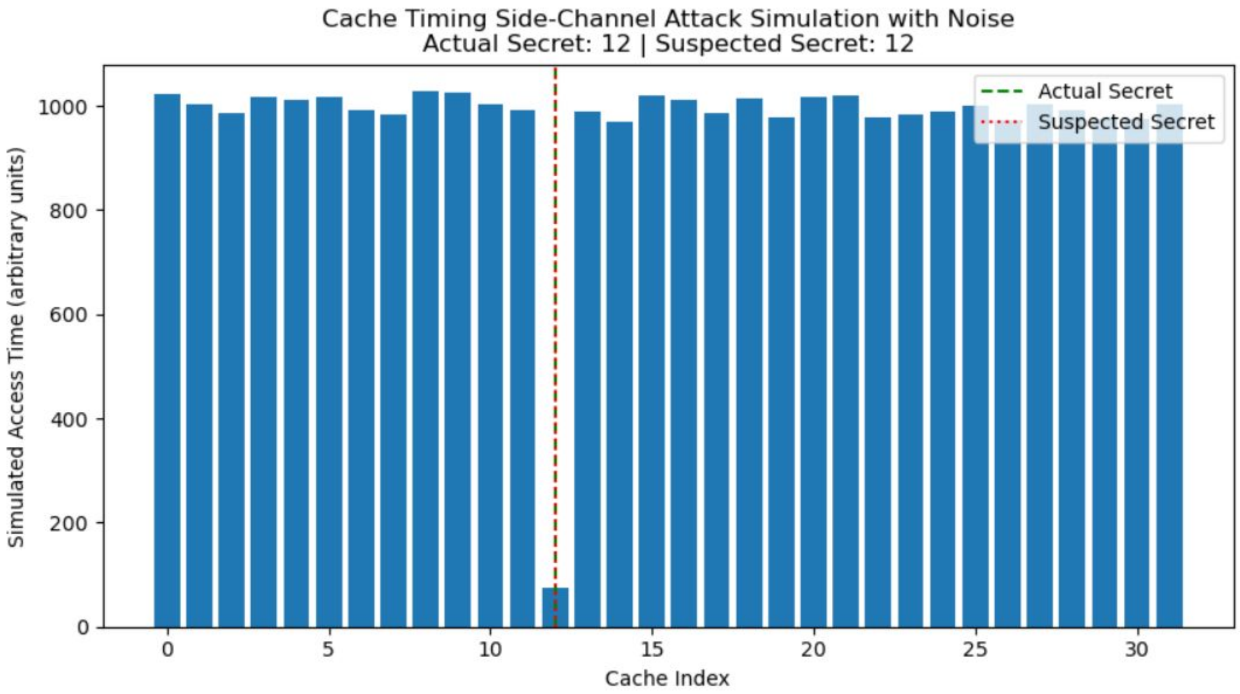
```python
class CacheSimulator:
    def __init__(self, cache_size=32):
        self.cache = [0] * cache_size

    def flush_cache(self):
        self.cache = [0] * len(self.cache)

    def victim_access(self, secret_index):
        # Victim secretly accesses the cache line containing the secret
        self.cache[secret_index] = 1

    def attacker_measure(self, index):
        # Simulate timing differences with added noise
        noise = random.randint(-30, 30)
        if self.cache[index]:
            return 100 + noise  # Fast cache hit with noise
        else:
            return 1000 + noise  # Slow cache miss with noise
```

```python
# Simulation setup
cache_size = 32
secret_value = random.randint(0, cache_size - 1)  # Victim's secret
cache_sim = CacheSimulator(cache_size)

# Flush cache and victim secretly accesses the correct data
cache_sim.flush_cache()
cache_sim.victim_access(secret_value)

# Attacker performs measurements
timings = [cache_sim.attacker_measure(i) for i in range(cache_size)]

# Identify suspected secret based on timing
suspected_secret = timings.index(min(timings))

# Plotting the results
plt.figure(figsize=(10, 5))
plt.bar(range(cache_size), timings)
plt.xlabel('Cache Index')
plt.ylabel('Simulated Access Time (arbitrary units)')
plt.title(f'Cache Timing Side-Channel Attack Simulation with Noise\nActual Secret: {secret_value} | Suspected Secret: {suspected_secret}')
plt.axvline(secret_value, color='green', linestyle='--', label='Actual Secret')
plt.axvline(suspected_secret, color='red', linestyle=':', label='Suspected Secret')
plt.legend()
plt.show()

print(f"Actual Secret Value: {secret_value}")
print(f"Suspected Secret Value: {suspected_secret}")
```

UCLA Samueli School of Engineering

Flush+Reload Example

result




```
Actual Secret Value: 12
Suspected Secret Value: 12
```

UCLA Samueli School of Engineering

Implementation-Flush Reload

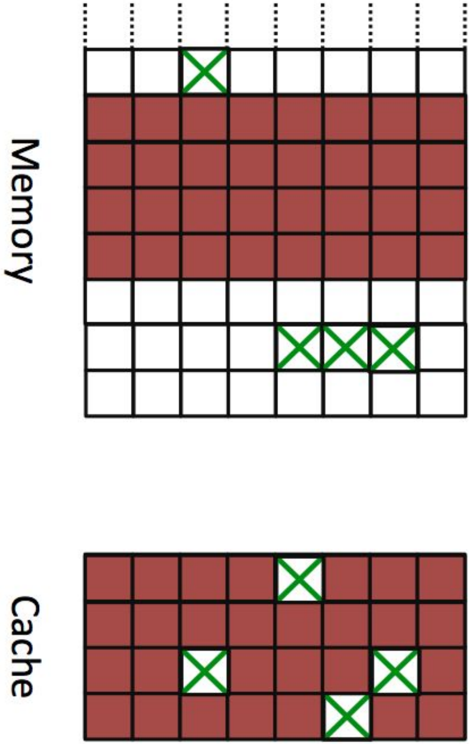
	Mem	Cache
Minimum	402	30
Bottom decile	480	38
Median	502	138
Top decile	794	160
Maximum	65535	65535

- Large time difference between accessing data from cache vs mem
- Time difference is the key vulnerability cause cache-side channel attack

Prime+Probe

- **Mechanism**: Attacker fills a cache set with its own data (prime phase), waits for the victim to execute, then measures the access time (probe phase) to see if any cache lines were evicted.
- **Goal**: Detect which cache sets the victim used and infer accessed memory patterns.
- **Scope**: Does not require shared memory and works across processes, VMs, and even browsers.
- **Common Targets**: Cryptographic algorithms, kernel memory access patterns.

Prime+Probe Example



- Attacker chooses a cache-sized memory buffer
- Attacker accesses all the lines in the buffer, filling the cache with its data
- Victim executes, evicting some of the attackers lines from the cache
- Attacker measures the time to access the buffer
  - Accesses to cached lines is faster than to evicted lines

4

Prime+Probe

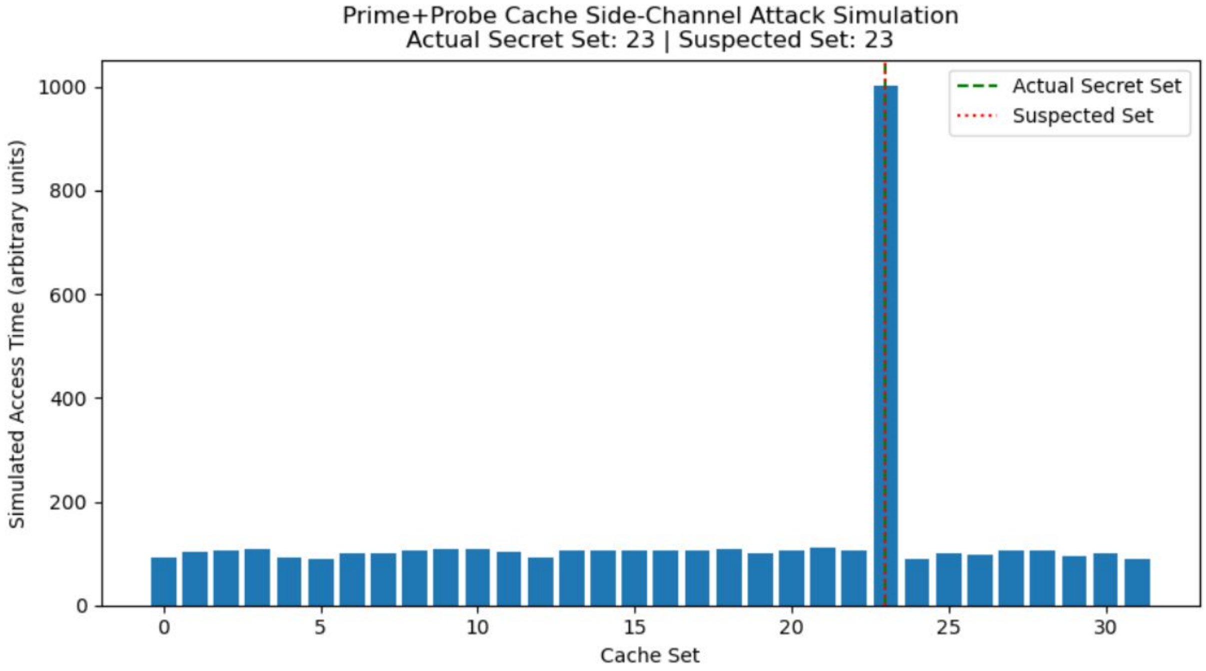
Example python code setup

```python
class PrimeProbeSimulator:
    def __init__(self, cache_sets=32):
        self.cache = [0] * cache_sets

    def prime_cache(self):
        # Attacker fills (primes) all cache sets
        for i in range(len(self.cache)):
            self.cache[i] = 1

    def victim_access(self, secret_set):
        # Victim accesses the cache set secretly
        self.cache[secret_set] = 2  # Indicates victim usage

    def probe_cache(self):
        # Attacker measures which cache sets were evicted by victim
        measurements = []
        for set_value in self.cache:
            if set_value == 1:
                measurements.append(100 + random.randint(-10, 10))  # Fast hit
            else:
                measurements.append(1000 + random.randint(-10, 10))  # Slow miss due to eviction
        return measurements
```

```python
# Simulation setup
cache_sets = 32
secret_set = random.randint(0, cache_sets - 1)
simulator = PrimeProbeSimulator(cache_sets)

# Attacker primes cache
simulator.prime_cache()

# Victim secretly accesses cache
simulator.victim_access(secret_set)

# Attacker probes cache to find which set was evicted
measurements = simulator.probe_cache()
suspected_set = measurements.index(max(measurements))

# Plotting the results
plt.figure(figsize=(10, 5))
plt.bar(range(cache_sets), measurements)
plt.xlabel('Cache Set')
plt.ylabel('Simulated Access Time (arbitrary units)')
plt.title(f'Prime+Probe Cache Side-Channel Attack Simulation\nActual Secret Set: {secret_set} | Suspected Set: {suspected_set}')
plt.axvline(secret_set, color='green', linestyle='--', label='Actual Secret Set')
plt.axvline(suspected_set, color='red', linestyle=':', label='Suspected Set')
plt.legend()
plt.show()

print(f"Actual Secret Cache Set: {secret_set}")
print(f"Suspected Secret Cache Set: {suspected_set}")
```

Prime+Probe Example

result




```
Actual Secret Cache Set: 23
Suspected Secret Cache Set: 23
```

UCLA Samueli School of Engineering

Cache Attack Detection Tool

Detect Flush+Reload

Detection

```python
# Detect Flush+Reload attack patterns
def _detect_flush_reload(self):
    if len(self.cache_access_times) < self.window_size:
        return 0.0

    # Calculate statistics on cache access times
    times = np.array(self.cache_access_times)
    mean = np.mean(times)
    std = np.std(times)

    # Check for bimodal distribution (characteristic of Flush+Reload)
    hist, _ = np.histogram(times, bins=50)
    peaks = np.where(np.diff(np.sign(np.diff(hist))) < 0)[0] + 1

    # Flush+Reload typically shows a clear bimodal distribution
    if len(peaks) >= 2 and std > mean * 0.3:
        # Calculate the valley-to-peak ratio (lower means more suspicious)
        valley = np.min(hist[peaks[0]:peaks[1]])
        peak = np.max(hist[peaks[0]:peaks[1]])
        ratio = valley / peak if peak > 0 else 1.0

        confidence = 1.0 - min(ratio * 2, 1.0)
        return confidence

    return 0.0
```

Looks for bimodal distribution in cache access timing

Analyzes histogram of memory access times for two distinct peaks

Calculates valley-to-peak ratio to determine attack confidence

Detect Prime+Probe

Detection

```python
# Detect Prime+Probe attack patterns
def _detect_prime_probe(self):
    if len(self.cache_access_times) < self.window_size:
        return 0.0

    # Prime+Probe shows periodic spikes in cache miss rates
    times = np.array(self.cache_access_times)

    # Check for periodicity using autocorrelation
    autocorr = np.correlate(times, times, mode='full')
    autocorr = autocorr[len(autocorr)//2:]

    # Normalize
    autocorr = autocorr / autocorr[0]

    # Find peaks in autocorrelation
    peaks = [i for i in range(1, len(autocorr)-1)
             if autocorr[i] > autocorr[i-1] and autocorr[i] > autocorr[i+1]]

    if len(peaks) >= 3:
        # Check if peaks are evenly spaced (characteristic of Prime+Probe)
        intervals = np.diff(peaks[:5] if len(peaks) > 5 else peaks)
        interval_std = np.std(intervals)
        interval_mean = np.mean(intervals)

        # Calculate periodicity score (lower variation means more periodic)
        if interval_mean > 0:
            variation = interval_std / interval_mean
            confidence = 1.0 - min(variation * 2, 1.0)
            return confidence

    return 0.0
```

Identifies periodic patterns in cache miss rates

Uses autocorrelation to find repeating sequences

High confidence for consistent peaks intervals

Specific repeating sequence of operations for attack

Complex to detect as its subtle behavior

CacheMonitor

Performance Counter Integration: Call Linux's `perf` tool to obtain cache miss and cache reference counts.

Access Time Measurement: In the final actual implementation, high-precision timers (`rdtsc` instruction) should be used to measure memory access times. Our method in code now is a simplified simulation that would be replaced with real hardware measurements in actual deployment.

Data Collection and Window Analysis: Uses a fixed-size sliding window to collect cache access times for statistical analysis.

```
2025-03-08 19:13:15,675 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.71
2025-03-08 19:13:15,708 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.73
2025-03-08 19:13:15,749 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.71
2025-03-08 19:13:15,761 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.71
2025-03-08 19:13:15,766 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.71
2025-03-08 19:13:15,777 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.71
2025-03-08 19:13:16,124 - CacheLeaks - WARNING - Potential Prime+Probe attack detected! Confidence: 0.72
```

Implementation

Attack Method: L1 Cache Prime+Probe technique

Target: AES Encryption implementation

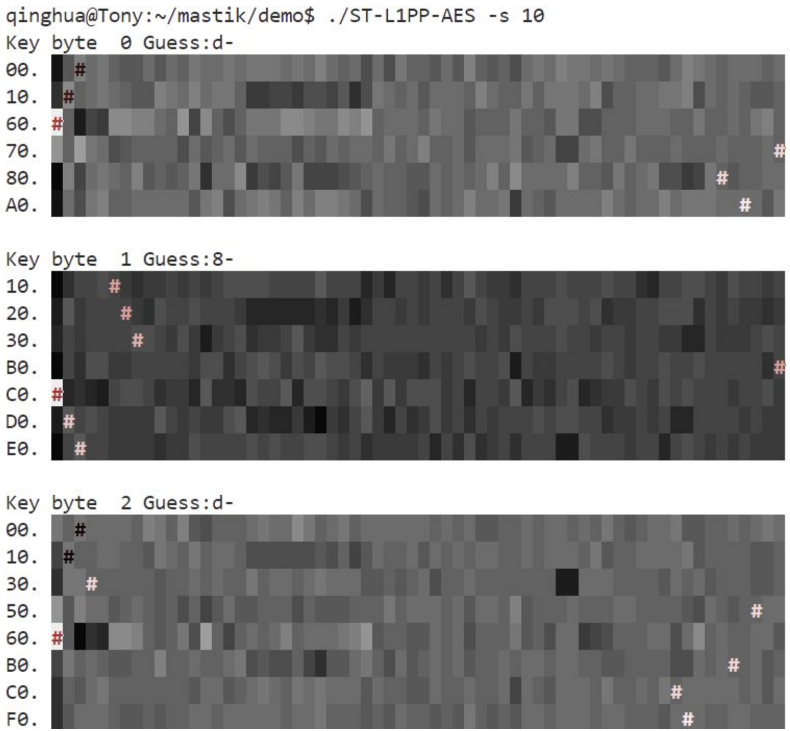
Tool Used: Mastik（A Micro-Architectural Side-Channel Toolkit）framework

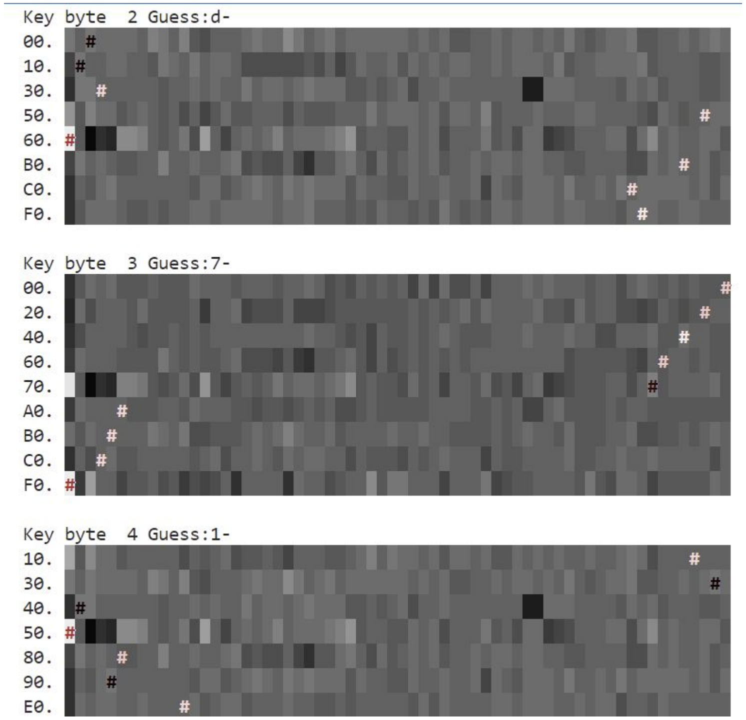
Parameters: Sample size (10) for optimal detection

Attack Workflow

1. **Prime** — Fill specific L1 cache sets with attacker's data
2. **Wait** — Allow victim to perform AES encryption operation
3. **Probe** — Measure access times to identify evicted cache line
4. **Analyze** — Correlate cache access patterns with key values.

Results and Implications





Conclusion and Future work

- Successfully demonstrated cache side-channel vulnerability in AES implementation
- Identified key timing differences that enable information leakage
- Proved practical feasibility of key recovery with minimal resources

- Performance counter-based detection of unusual cache activity
- Implementation of constant-time cryptographic operations
- Randomization strategies to disrupt timing measurements

Q&A

Thank You

Requirements

- The presentation should be based on your understanding of the paper, not just repeating what was mentioned in the paper.
- You will be judged based on your presentation and content.
- **Need good visuals**
- Each presentation should be about 10 minutes long. (no more)
- Skip any motivation and/or introduction (we already know that).
- **Your first slide should be describing the problem.**
- **Your next few slides should provide background on this problem.**
- Suppose you want to teach others this concept. Don't just repeat what the paper said, we already read that!
- Connect what was mentioned in the paper with concepts taught in class.
- **Then, describe the proposed solution/contribution**. Try to provide an intuition. Skip most of the details.
- **Present the evaluation methodology next (very short!).**
- **Only discuss the main results and your insights/interpretation from it. Extremely briefly!!**
- **Finish with the potential future work, discuss if there is any follow up to this work.**
- **Your final slide should be two-three discussion points. (goal is to discuss paper more)**