Covert and Side Channels

Robert Brotzman-Smith

PennState

Covert Channels

- Any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy
  - US DoD 1985
- Basically a covert channel is any unconventional means of communication

Brief History

- Covert channels have existed for thousands of years
- One of the earliest known was recorded~500 BC in Greece
- Messengers would get their heads shaved and a message tattooed on their scalp
- Then let their hair grow back before making the trip to the destination where their head would be shaved to reveal the message
- Microdots were used in WW1 by Germany to conceal communication
  - Microdots were very small black dots that could contain messages when read under magnification
  - Commonly found in dots above the letter i or periods
- Many more clever methods of communication have been developed since

PennState

Types of Covert Channels

- Storage
  - Communicates data by directly of indirectly writing to a storage location and another process directly or indirectly reading that location
  - Ex) Printer Queues, file locks
- Timing
  - Uses the time of an operation to communicate data
  - Ex) CPU cache
- Steganography
  - Hides information inside a typical communication channel

What Makes a Good Covert Channel

- Hard to detect
  - Some channels even when you know where to look are difficult to read the data
  - Ex) Steganography
- High Bandwidth
  - Typically increasing bandwidth increases detectability
- Easy to achieve
  - Should be easy for the sender/receiver to communicate provided they both know how the channel works
- Encryption
  - Even if channel is discovered the data is not revealed

Least Significant Bit

- Common and simple technique to embed data in an image
- Idea is to replace the lest significant bits of each byte of data in an image
  - Color image usually consist of 8 24 or 32 bits per pixel
- Replacing only a couple of the least significant bits only slightly alters the image
- As more data is hidden, the original image becomes more distorted
- A general rule is the hidden data should be ~25% of the total image size
- Demo

PennState

1 Bit Hidden

PennState

2 Bit Hidden

PennState

4 Bits Hidden

PennState

5 Bits Hidden

PennState

6 Bits Hidden

PennState

7 Bits Hidden

PennState

Side Channels

Side Channels

- Side-channel attacks extract information by observing implementations on systems
- These attacks do not rely on code vulnerability
  - Ex) Buffer overflows, SQL injection, etc.
- Do not rely on theoretical weaknesses of algorithms

PennState

Example Side Channel

- Timing
- CPU Cache
- Power Usage
- Electromagnetic field
- Acoustic
- Thermal
- Speculation

Timing Side Channels

Timing Side Channels

- Timing side channels work by observing how long a task takes to complete
- They obtain information when an algorithm takes different amounts of time to execute depending on the inputs
- This is particularly problematic when the execution time depends on secret data
- Can be exceptionally dangerous since they do not require the adversary and victim to necessarily share resources

Real Timing Attack

- Example is from libgcrypt which is a common cryptographic library
  - Implements modular exponentiation
  - Commonly used in RSA and ElGamal
- Essentially the algorithm will square and take the modulus every time
- When the current bit is set it will also multiply by the base and again apply the modulus
  - Note that the key here is the exponent
- Notice that based on the key's value the then branch of the if statement is executed
- Thus every time a bit is set in the key, that loop iteration will take more time to execute
  - This can leak many bits of the key quickly

```
1: void squareNMultiply ()
2: {
                    // details omitted for  brevity
54:             while (c)
55:         {
56:             res = res * res;
57:             res = res % mod;
58:                     if (((1 << 31 ) & key) != 0)
59:                 {
60:                     temp = res * base;
61:                     temp = temp % mod;
62:                     res = temp;
63:                 }
64:
65:                 key <<= 1;
66:                 c--;
67:         }
```

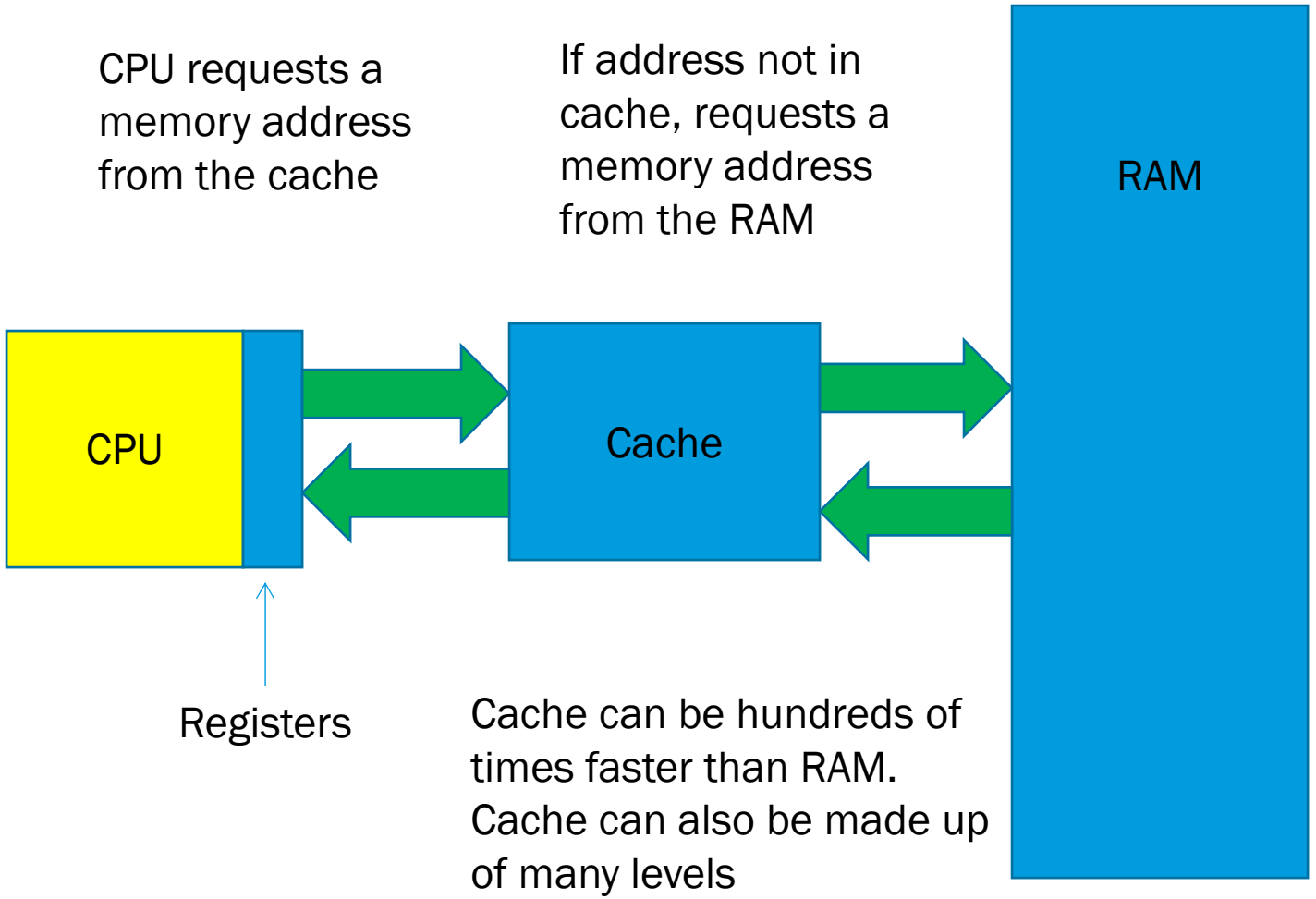
PennState

Other Timing Attacks

- Not all attacks need a secret dependent branch to cause timing differences
- Some instructions will take different amounts of time to execute based on their operands
  - Ex) division
- More timing variation can occur based on where data is located in the memory hierarchy
  - i.e. registers, cache, ram, disk

Cache Side Channels

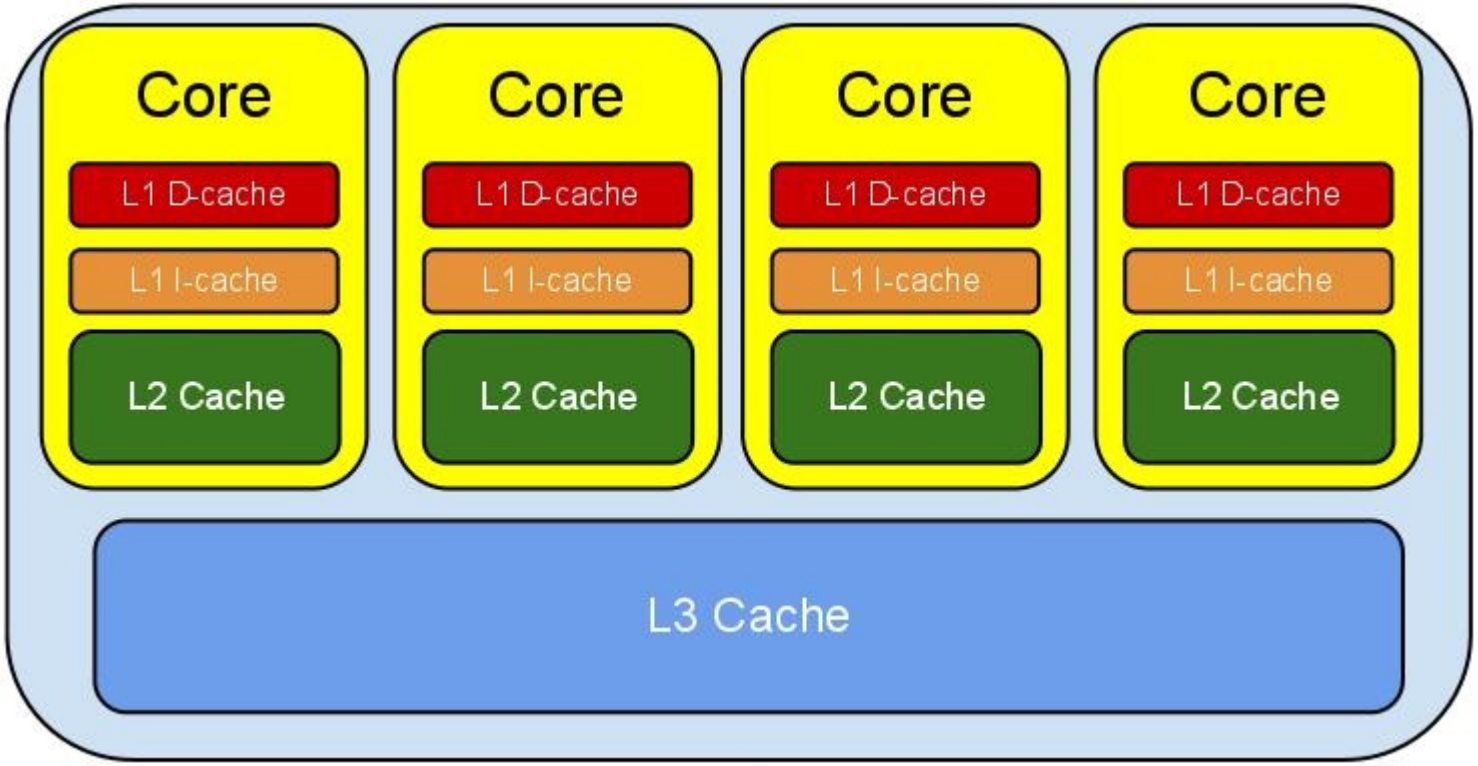
CPU Cache Attacks: Preliminaries



PennState

CPU Cache Attacks: Preliminaries

- CPU caches are also broken into slower and faster memory
  - L1 faster than L2 faster than L3
- CPU caches store both data and instructions



CPU Cache Attacks: Preliminaries

- Modern CPU caches are typically N-way set associative
  - Opposed to being directly mapped or fully associative
- In N-way set associative caches, there are cache size/N sets
  - Ex) 32 kb 8-way cache will have ~4000 cache sets
- Memory can be mapped to one cache set and the processor will apply a replacement policy to each cache set
  - Ex) least recently used, pseudo least recently used, not most recently used
- The replacement policy is typically what allows adversaries to learn information through a side channel
  - This is because most commercial processors use a replacement policy that is related to recent program behavior

PennState

CPU Cache Attacks

- Cache side-side channel attacks leverage the state of the cache to infer sensitive data used during program execution
  - State refers to memory addresses present in the CPU cache
  - The key insight is that as programs execute the state of the cache is constantly being updated
- Since CPU caches are very fast, these side channels can leak large amounts of data quickly
  - 100's of kilobytes/second
- Cache side channel attacks are often categorized into three cases
  - Time
  - Access
  - Trace

PennState

Side Channel Categories

- Time
  - Adversary is able to observe the total execution time of some target piece of code
  - Can be launched remotely
  - Leaks the least amount of information
- Access
  - Adversary is able to determine whether certain memory addresses are cached
  - Requires shared cache with victim
  - Leakage is limited by how long it takes to probe memory addresses
- Trace
  - Adversary knows the order memory addresses are cached
  - Requires shared cache with victim
  - Fine grained traces are difficult to achieve
  - Usually used to analyze countermeasures

PennState

CPU Cache Side Channel Overview

CPU cache improves performance by storing recently used data in fast memory

Information about recent program execution is in the cache state

Cache side channel attacks infer what data is in the cache

PennState

Determining What is Cached

- Many access based attacks to determine the cache state
  - Flush+Reload
  - Flush+Flush
  - Prime+Probe
- The goal of each one is to determine whether or not a set of memory addresses has ben accessed by a victim process
  - Usually the target memory locations will be related to some sensitive data used by the process

Flush+Reload

- This attack requires memory to be physically shared between the victim and adversary to share the same physical memory location
  - This scenario is more common than one would initially think
  - Commonly happens as a result of sharing libraries
  - OS will not duplicate read-only memory
- The adversary targets a region memory that will be accessed based on some sensitive data
- The attack consists of three phases
  - 1) flush cache line(s) from memory using the clflush instruction
    - Clflush takes a virtual address as input and will flush the memory from the entire cache hierarchy
  - 2) wait for victim to access their data
  - 3) reload the memory that was flushed and time how long it takes
- Reloads that are fast mean the victim accessed the data

PennState

Flush+Flush

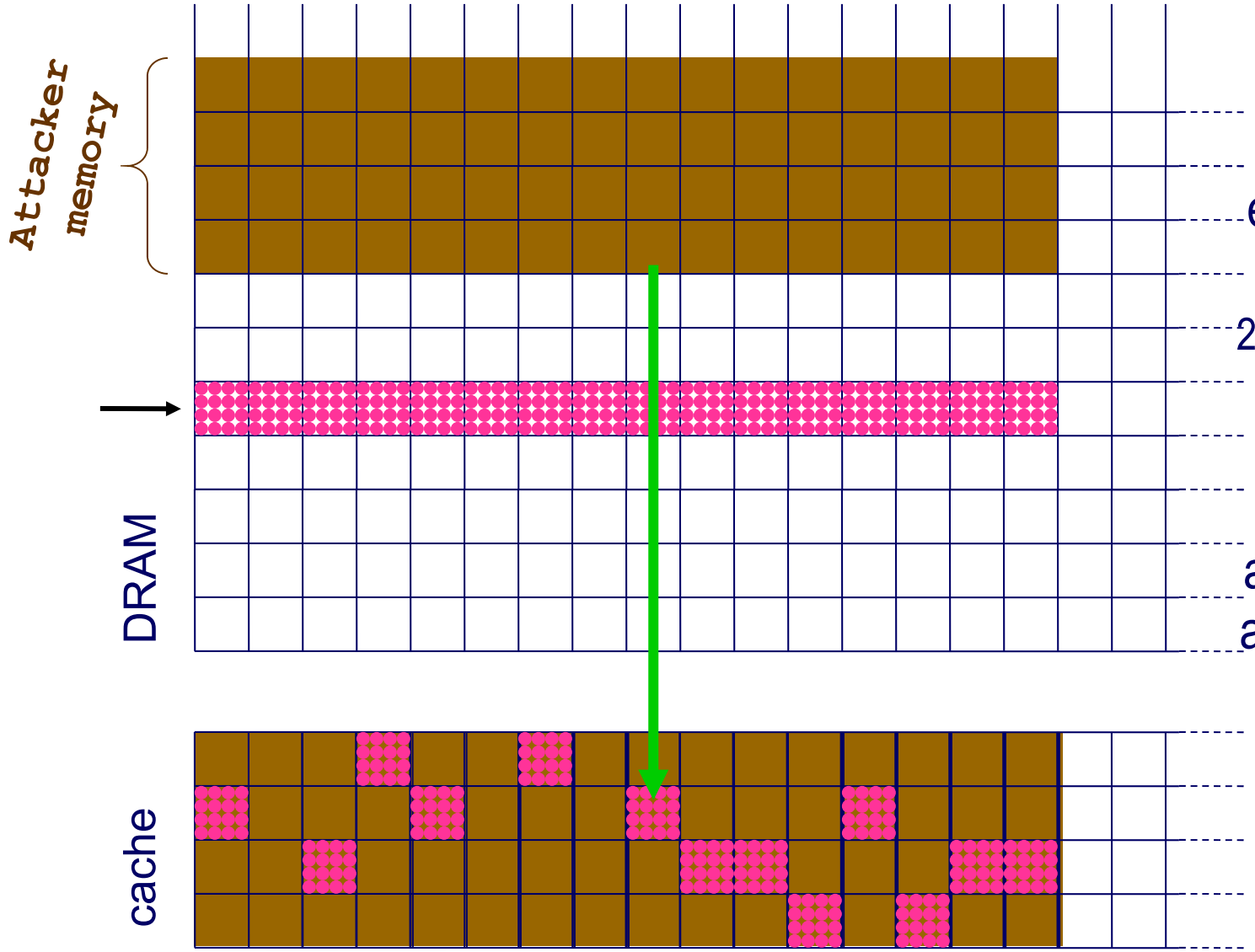
- Similar to Flush+Reload, Flush+Flush also requires physically shared memory
- The insight key insight that allows this attack to work is flushing memory that is not in the cache takes a different amount of time than flushing memory in the cache
- The steps in this attack are similar to Flush+Reload
  - 1) Flush target cache line(s)
  - 2) wait for victim
  - 3) Flush target cache line(s) again
- The second flush is timed
  - Low time means data was cached fast time means data was not cached
- Flush+Flush evades many cache side-channel detection methods
  - Most detection methods use performance counters
  - Look for cache hits/misses
  - Flush+Flush does not make memory accesses thus no misses/hits

PennState

Prime+Probe

- Does not require shared physical memory
- Adversary needs to figure out how to map their data to the same cache sets as victim
  - Very easy when targeting the L1 cache
  - A bit more challenging for the L3 cache
- Attack requires three steps
  - Fill one or more cache sets with data
  - Wait for the victim to execute
  - Probe the data loaded

PennState



1. Completely evict tables from cache (Prime)
2. Trigger a single encryption
3. Access attacker memory again. See which cache sets are slow (Probe)

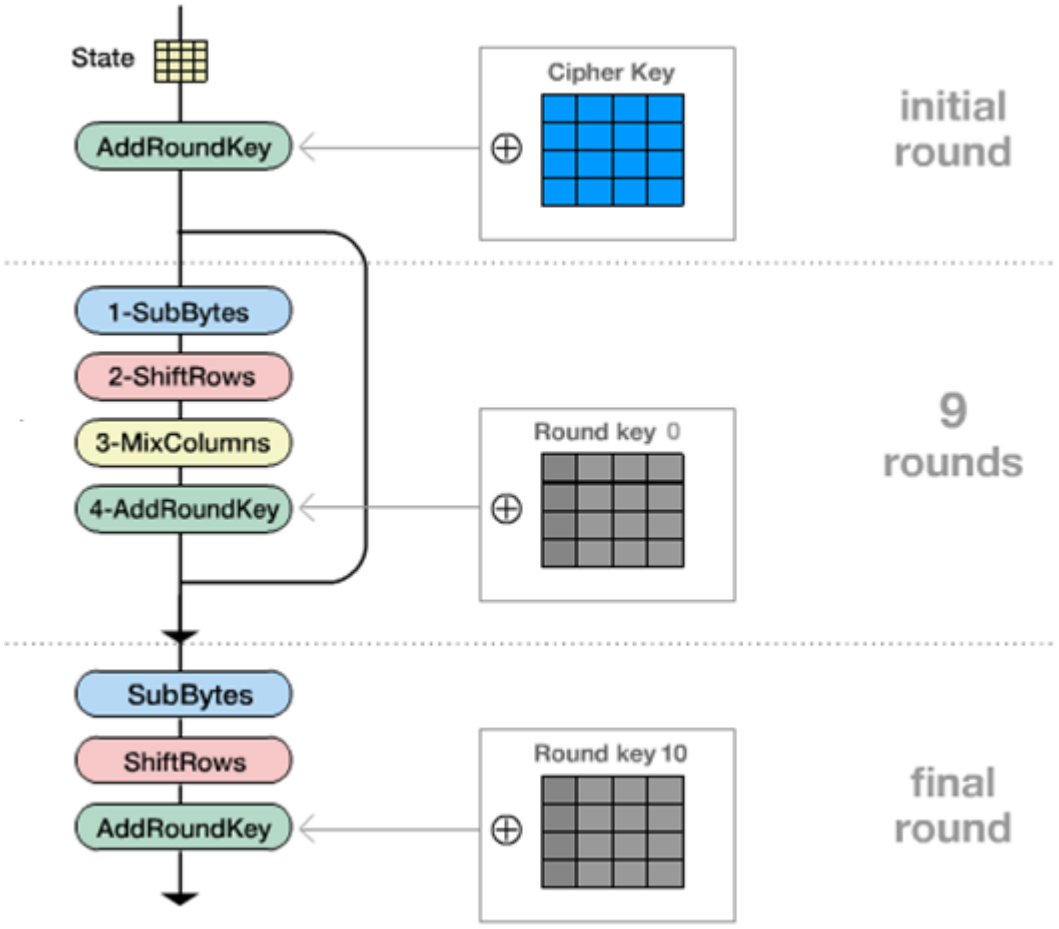
PennState

Mapping data in L1

- The L1 cache is usually physically tagged and virtually indexed
- What this means is the lower bits of our virtual address tells us which set our data will map to
  - Typically the virtually indexed portion of the address corresponds to the page offset
  - Pages are usually 4096 bytes
- All the adversary needs to know is what the page offset will be for some target data
  - This is usually very easy to figure out

PennState

AES Eample: Algorithm



PennState

First Round Attack

- Only the round key is processed
- The computation done in the initial round is: $x_i = p_i \oplus k_i$
- x_i will be used as the index into the sbox
  - Ex) sbox[x_i]
- By using either Prime+Probe, Flush+Reload, etc. on the sbox locations we will learn k_i
  - $k_i = p_i \oplus x_i$
  - Assuming we know the plaintext p_i

Synchronous vs Asynchronous

- Synchronous
  - Adversary can start the victim processes execution
  - More likely the adversary observes the target cache accesses
  - Allows for a higher throughput channel
- Asynchronous
  - Adversary does not interact with the victim process
  - Needs to be able to detect when the victim is running
  - Typically requires more samples
  - Lower throughput than synchronous channels
  - More practical since they do not require interaction with the victim program

PennState

Try these out yourself

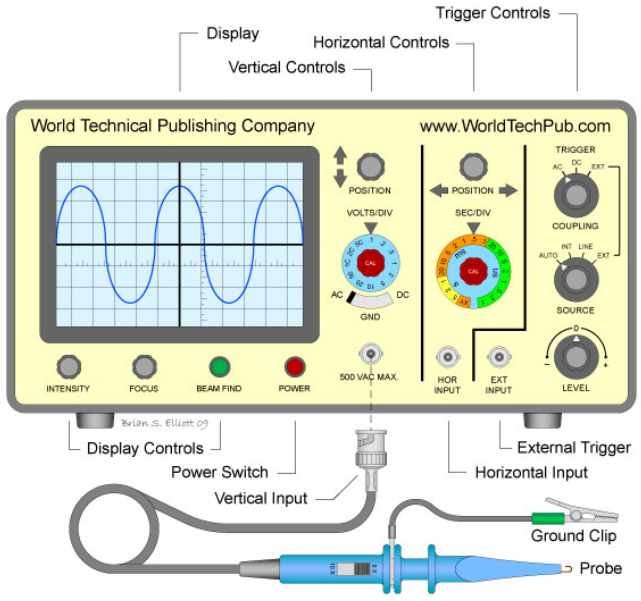
- Open source tool called Mastik
  - https://cs.adelaide.edu.au/~yval/Mastik/
- Has an implementation for the three side channels we discussed
- Has test cases to demonstrate how to use the tool to extract cryptographic keys
  - Disclaimer: These programs are for educational purposes only and should not be installed on university owned machines

PennState

Power Side Channels

Power Side Channel

- Power side channels leverage fluctuations in a machine's power usage when performing different operations
  - Power usage depends on what instruction is being executed
  - Also can depend on the operands of the instructions
- Typically an adversary will require physical access to the machine to measure power usage
  - Uses oscilloscope to measure voltage



PennState

Power Side Channel Attack Model

- Adversary may control the cipher or plaintexts
- Goal is to learn the key used by the device doing the encryption/decryption

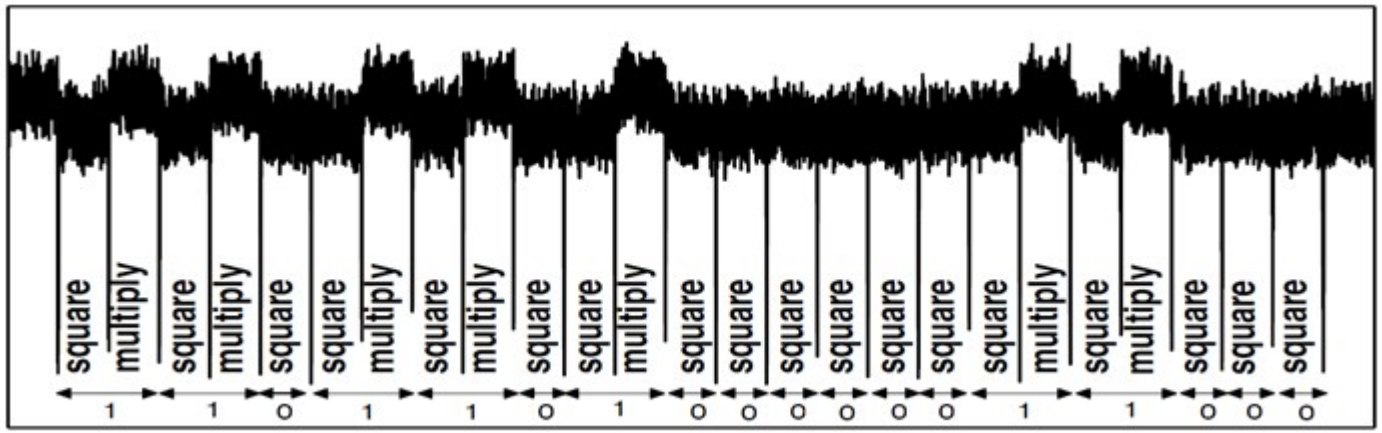
PennState

Power Analysis Types

- It is commonly divided into three categories
  - Simple Power Analysis (SPA)
    - Adversary can learn information by visually looking at power trace
  - Differential Power Analysis (DPA)
    - Adversary applies statistical techniques to learn information
      - Ex) Difference of means, Correlation, error correction, etc
  - High-Order Differential Power analysis (HO-DPA)
    - Considers data from multiple source simultaneously
    - Data must be synchronized by time

Simple Power Analysis

- Example shows square-and-multiply algorithm used to compute modular exponentiation



- We can clearly see the extracted key here is: 11011010000011000
- This approach can be used to learn a key of arbitrary length

PennState

Speculation Side Channels

Out-of-Order & Speculative Execution

- Out-of-order execution happens anytime another instruction is executed before previous instruction(s) are retired
- Speculative execution happens when the result of a branch is unknown and execution proceeds down a guessed branch
- Both of these optimizations help to maximize the CPU resources

Transient Instructions

- Any instruction which can be executed out of order and leaves measurable side effects
  - Ex) memory loads/stores
- These instructions are key to create any kind of side channel or covert channel

Branch Prediction

- Modern processors use heuristics to improve their guess of a branch value before it is known
- The processor will then execute instructions assuming the guess is correct
- If the guess is correct nothing needs to be done
- If the guess is wrong, the processor needs to roll back any changes it made to the program's state
  - This does not include everything such as the cache state
- The branch predictor is often shared between processes
  - Allowing an adversary to train the branch predictor from their process

PennState

Spectre

1

Trains branch predictor to execute a branch not taken by normal execution

2

Speculatively execute instruction(s) which reveal some sensitive data

3

Use side channel to recover sensitive data

- Flush+Reload
- Evict+Reload

PennState

Spectre Variants

Exploit conditional branches

- Influence branch predictor to incorrectly guess result of condition
- CVE-2017-5753

Exploit indirect branches

- Train predictor that an indirect branch will execute an attacker specified gadget
- CVE-2017-5715

PennState

Spectre Exploiting Conditional Branches

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

Listing 1: Conditional Branch Example

- Assumptions:
  - x is chosen by the attacker and can read anything in memory (possibly a secret value)
  - Array1_size is not in the cache
  - The branch predictor has been trained to predict true for the branch condition

PennState

Spectre Exploiting Conditional Branches

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

Listing 1: Conditional Branch Example

- The Attack:
  1) Since array1_size is not cached and branch predictor guesses true, the true branch will begin execution
  2) x was chosen by attacker and reads secret byte k
  3) Array2 will then access its $(k*256)^{th}$ element
  4) Attacker then checks to see what element from array 2 was accessed via a side channel attack from another process

PennState

Meltdown

1

Attacker identifies address in kernel space to read

2

Transient instructions are used to force address into the cache

3

Another process observes the cache line used by the transient instruction(s)

PennState

Meltdown's Core

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Line 4 reads a byte from the kernel address space
- Line 5 improves throughput of covert channel by multiplying kernel data by 4096
  - Ensures each possible value read will be placed in a separate page
- Line 6 keeps performing the attack until something is read
- Line 7 is the transient instruction which modifies the cache
  - It will probably get executed due to out of order execution
- *Note that this instruction sequence can be placed into a transaction avoiding the exception from being raised (but the effect on the cache will persist)

PennState

Side-Channel mitigations

Timing Channel Mitigations

- Randomize control flow
- Insert random noise
- Padding
  - Adds instructions to control flow paths to make them uniform
- Constant time algorithms
  - Requires code rewriting
- Constant time instructions
  - Means all instructions take the same amount of time
  - Requires all instructions to take the same amount of time as the slowest instruction

PennState

Cache Side-Channel Mitigation

- Preload/Pin data to the cache
  - Prevents detectable changes to the cache state
- Write side channel resistant code
  - Requires code rewriting and knowledge regarding how to avoid side channels
- Cache partitioning
  - Each processes gets its own part of the cache
- Random cache accesses
  - Makes it more difficult to distinguish legitimate cache accesses

Power Side-Channel Mitigation

- Make timing synchronization more difficult
  - Modulate cpu frequency
  - Add delays to the program execution
  - Randomize the execution of the program
- Make power usage uniform
- Add noise to power consumption
- Change cryptographic keys often
  - Power analysis often requires many traces

PennState

Side Channel Detection

Side Channel Detection

- Detecting side channel attacks in progress is important
- Allows further defensive action to be taken
  - Such as isolating the malicious process or killing it
- Particularly important in cloud environments
  - Since many users will share the same hardware

How to Detect Cache Side Channels

- Most state-of-the-art tools use performance monitors
- These monitors keep track of various metrics
  - L1 cache hits/misses, l2 cache hits/misses, cycles, etc
- Tools typically look for abnormal cache hit/miss rates
- Recent work has shown that using transactional memory can also be used to detect cache-based side channels

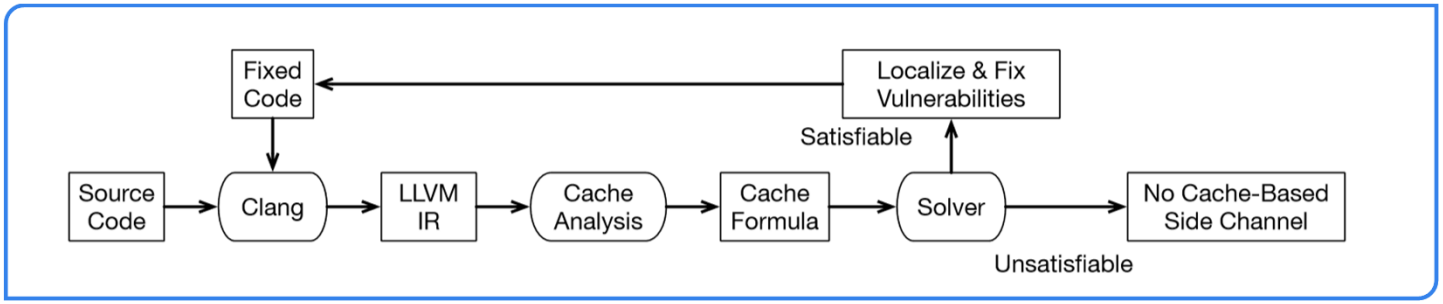
Identifying Side Channels in Programs

- Identifying side channels in software can be a challenging problem
- Particularly because code that looks seemingly innocuous can leak large amounts of data
- Recall the AES example
- Simply making a memory access dependent on a secret key can leak half of an encryption key
- How can we identify code that potentially reveals information ahead of time?

PennState

Cache Aware Symbolic Execution

- Keeps track of program state
  - Creates symbolic values for all program variables
  - Treats data in arrays and structs are arbitrary values
- Uses two abstract cache models
  - Infinite
  - Age
- Combines the program and cache states to check for side channels
- Key idea is to see if at least two unique program executions result in different cache states



PennState

Cache Models

- Infinite
  - Treats cache as an infinite set
  - Once something is accessed and placed in the cache, never gets evicted
- Age
  - Assigns an age to all variables
    - Initialized to infinity
  - Upon access
    - Increment all variables' ages
    - Set accessed variables age to 0

PennState

Improving Performance

- Compositional Reasoning
  - Break program up into multiple chunks
  - Check to see if any chunk has 2 paths with a different cache state
- Loop Transformation
  - Check to see if loop body can ever result in different cache states

$$
\begin{aligned}
&S_1; (\text{WHILE } B \text{ DO } S); S_2 \\
&\quad \Rightarrow \\
&S_1; \text{check } K_0; \text{reset}; \\
&(\text{IF } B \text{ THEN } (S; \text{check } K_1) \text{ ELSE SKIP}); \\
&\text{reset}; \\
&\neg B \rightarrow S_2
\end{aligned}
$$

PennState

Improving precision

- Tainted secret
  - Keep track of which variables are secret at all program points
  - Allows reduced sensitive variable sets after reset
- Tainted arrays
  - Non-tainted arrays are fixed between the two execution traces
  - Removes false positives

PennState

Fixing side channels

- Preloading
  - Preloading a variable ensures that it is in the cache for the infinite cache model
  - Commonly used to fix symmetric key algorithms in practice
- Pinning
  - Pinning a variable makes the variable permanently in the cache
  - Allows removing side channels under the age based model

PennState

Overall

- Cache aware symbolic execution can be used to identify cache-based side channels
- It will indicate the exact line of code causing the side channel and the kind of side channel
  - Ex) either key dependent branch or array access
- Allows users to iteratively remove the side channel and then check again and see if there are more side channels to fix
- Guarantees that if no cache-based side channels are reported that none are possible
  - i.e. the analysis is sound

Final Questions?

Acknowledgements

- [1] http://www.cs.tau.ac.il/~tromer/istvr1516-files/lecture3-power-analysis.pdf
- [2] https://en.wikipedia.org/wiki/Oscilloscope#/media/File:WTPC_Oscilloscope-1.jpg
- [3] Steganography and Covert Channels, K. Reiland, W. Oblitey, S. Ezekiel, J. Wolfe
- [4] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwjJtKvnw8fgAhXMnOAKHU6AC0wQFjAAegQICRAC&url=https%3A%2F%2Fwww.cs.clemson.edu%2Fcourse%2Fcpsc420%2Fpresentations%2FSpring2007%2FCovert%2520Channels.ppt&usg=AOvVaw0FhHuxgRjmuXZmHE5GWNUc
- [5] Topics in Cryptography:Lecture 7, Moni Naor
- https://incoherency.co.uk/image-steganography/