Privacy and Security

Sekar Kulandaivel, Jennifer Xiao - April 21, 2020

Agenda

Understanding Contention-Based Channels and Using Them for Defense

Spectre Attacks: Exploiting Speculative Execution

Understanding Contention-Based Channels and Using Them for Defense (HPCA ‘15)

Distrustful tenants living within a neutral cloud provider

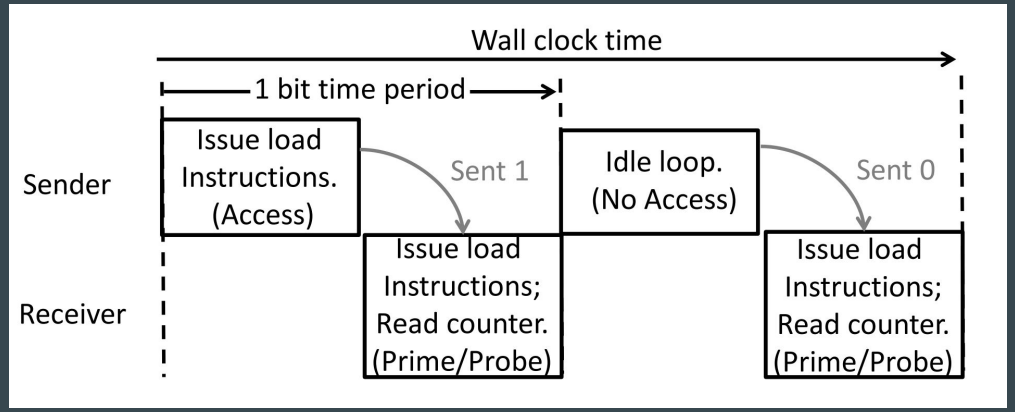
- Shared hardware can be exploited to leak information
  - e.g. CPU usage vs. operation can expose secret key

- Two bodies of solutions:
  - **HW-based:** state-of-the-art is either limited in scope or requires impractical architecture changes
  - **SW-based:** HomeAlone forgoes shared hardware and permits only friendly co-residency, but still vulnerable to an intelligent attacker

Threat model of a co-resident attacker

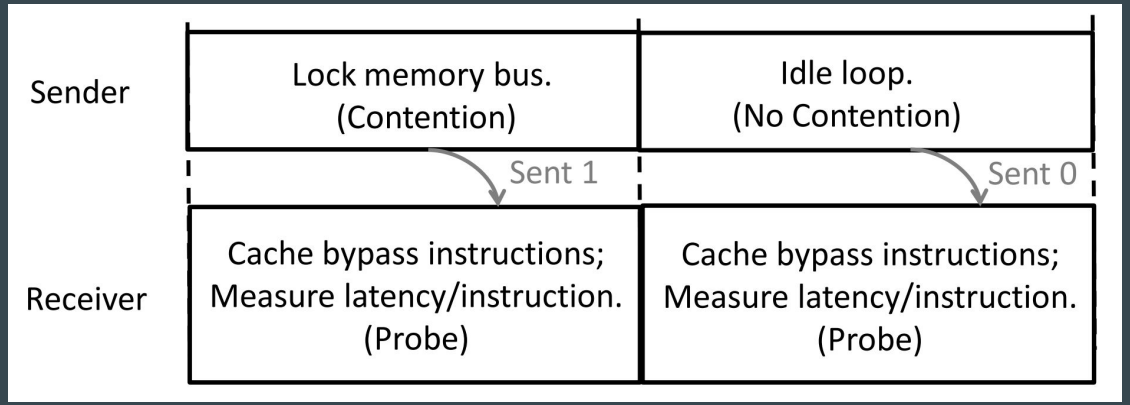
- Distrustful tenants violate confidentiality or compromise availability
- Goal: infer info about victim VM via microarchitectural structures e.g. cache and memory controllers
- **Side-channel**: victim inadvertently (oops!) leaks data inferred by attacker
- **Covert channel**: privileged malicious process on victim deliberately leaks data to attacker

Known side-channels to transmit a ‘0’ or a ‘1’ (alt. exec.)



- Alternative execution attacks
  - **Timing-driven**: measure time to access memory portion
  - **Access-driven**: measure time to access specific cache misses

Known side-channels to transmit a ‘0’ or a ‘1’ (parallel exec.)



- Parallel execution attacks
  - No time sharing required
  - E.g. Receiver monitors latency of memory fetch, sender either issues more instructions or idles

Formal model of covert channels

- Detection failure (undetectable flow) = same rate of false positives and false negatives for **both** legitimate and covert traffic

- Network vs. microarchitectural channels:
  - Network receivers read **silently**
  - Microarch. receivers read **destructively** (overwrites when reading)

- **Main insight**: network channels are provably undetectable whereas microarch. channels *cannot* be undetectable (in a noise-free setting)

Creating *high-capacity* covert channels

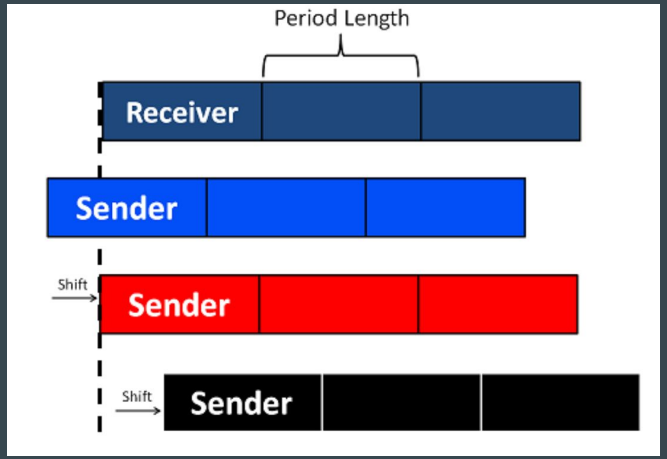
1. **Deconstructing covert channels into three primitives**
    a. Communication through contention; offline analysis to determine channel parameters; precise synchronization
2. **Synchronizing sender-receiver clocks**
    a. Explicit communication; fine-grained alignment; pilot signal
3. **Communication through contention**
    a. Performance counter-based channels; memory bus and AES timing channels; MIMO covert channels

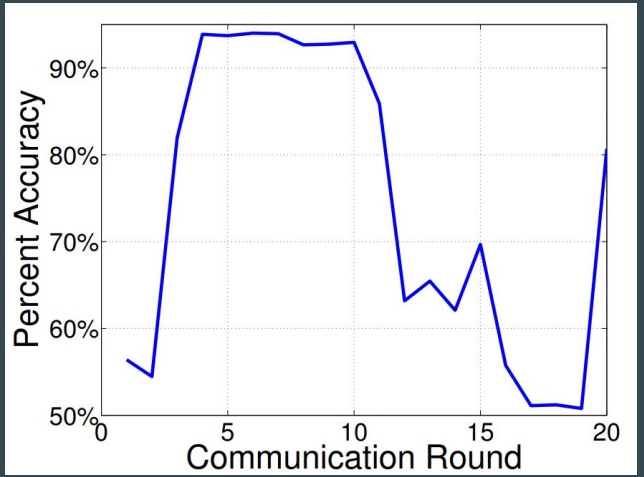
What makes up a covert channel?

1. **Communication through contention**
    a. High vs. low contention = ‘0’ or ‘1’
2. **Offline analysis to determine channel parameters**
    a. Fix time slot duration to prevent bit insertions and deletions
    b. With fixed time slot, receiver only needs to handle bit flip errors
3. **Precise synchronization**
    a. Unaligned time periods reduce channel capacity
    b. Alignment offers more precise contention and lower errors

Synchronizing sender-receiver clocks

1. **Explicit communication**
   a. W/o, offset can be seconds; okay to assume for defense and info. flow control implementations
2. **Fine-grained alignment**
   a. Make minor adjustments until timing lines up
   b. Goal = achieve maximum contention!
3. **Pilot signal**
   a. Use distinguishable pattern





Communication through contention

1. **Performance counters**
    a. Alternative exec.
        i. Cache misses: evict receiver data for a ‘1’; busy waits for a ‘0’
        ii. Branch misprediction: sender forces poor prediction accuracy for a ‘1’; busy waits for a ‘0’
    b. Parallel exec.
        i. Load and store: issue many loads for a ‘1’; issue none for a ‘0’
        ii. Can be used to detect attackers

Other ways to communicate through contention

2. **Memory bus and AES timing channels (parallel exec.)**
    a. Sender issues bus locking instr. or AES instr. for a ‘1’; o/w ‘0’
    b. Key consideration: don’t impact CPU frequency scaling

3. **MIMO channels**
    a. More resilient for both attacker comms. and detector
    b. Put sender and receiver in same process (avoid scheduling issues)

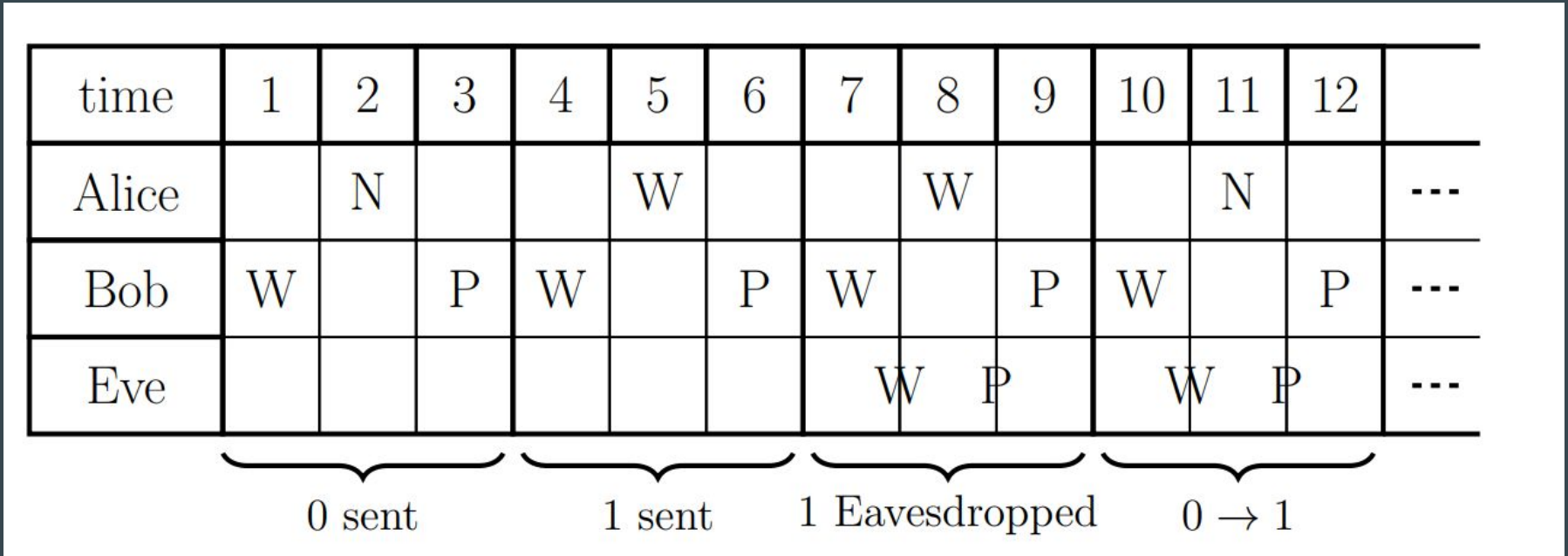
The bucket model

Assumptions:

- No bkgd. noise, OS and bkgd. processes can fill bucket
- Time synch., time slotted, Eve is aware

Fig. 6: Alice, Bob and Eve communicating through a shared resource abstracted by our bucket model. Alice, Bob and Eve have two operations: *Write*: add their water into bucket, and *Probe*: ask if their water is still in the bucket. When somebody adds their water into the bucket, previous content is discarded. Users can only ask if their water is currently in the bucket.

How does Eve eavesdrop and jam?



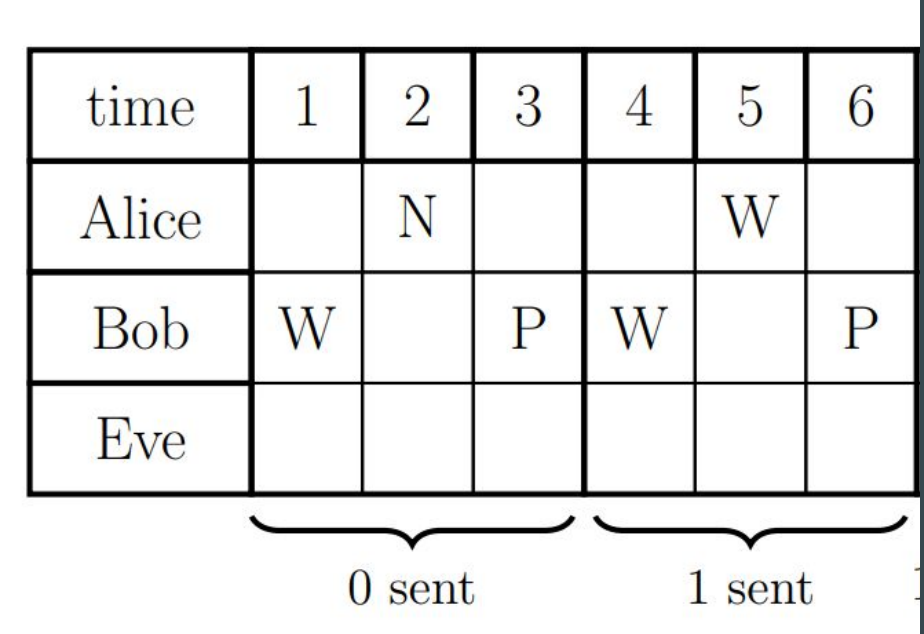
Secrecy and integrity from differential code

$$\mathbf{x} \rightarrow \{\mathbf{x}, (1 - \mathbf{x})\}. \quad (1)$$

This maps a $0 \rightarrow \{0, 1\}$ and $1 \mapsto \{1, 0\}$ and has rate $1/2$ (*i.e.* maps one bit into two).

Theorem 1: *The rate* $1/2$ *Differential Code provides the following guarantees. If Eve tries to eavesdrop, she is detected with probability* $1/2$ *per communicated bit. Further, Eve cannot introduce any bit error without being detected.*

Why use the differential code?



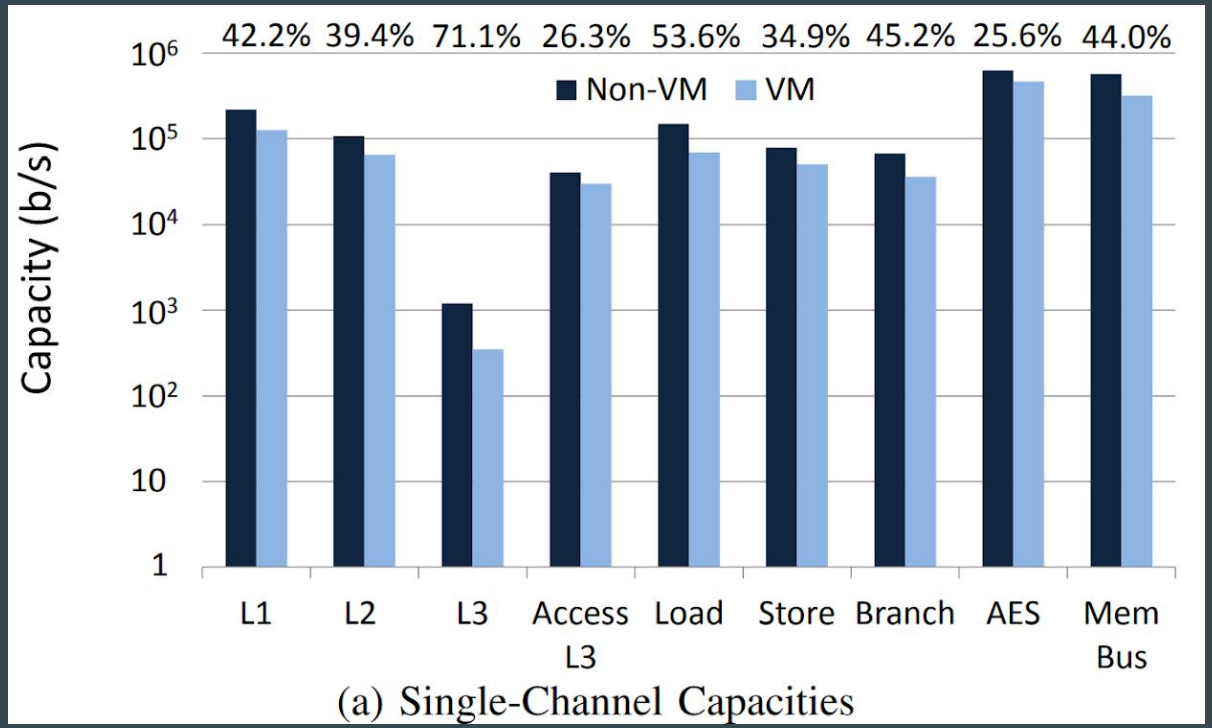
time	1	2	3	4	5	6
Alice		N			W	
Bob	W		P	W		P
Eve						

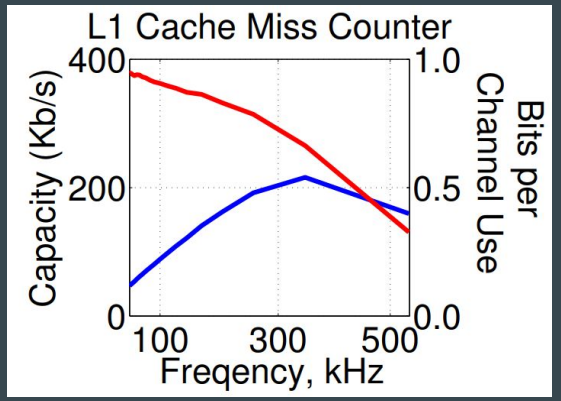
0 sent 1 sent

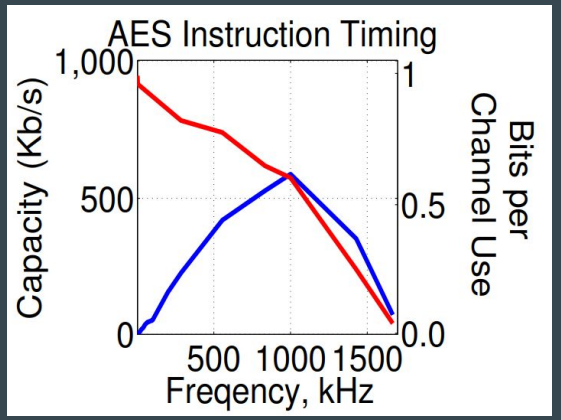
Here, Alice transmits a ‘0’ under this differential code.

If Eve eavesdrops, then Bob MUST read ‘1’. Thus, Bob observes two consecutive ‘1’s and alerts!

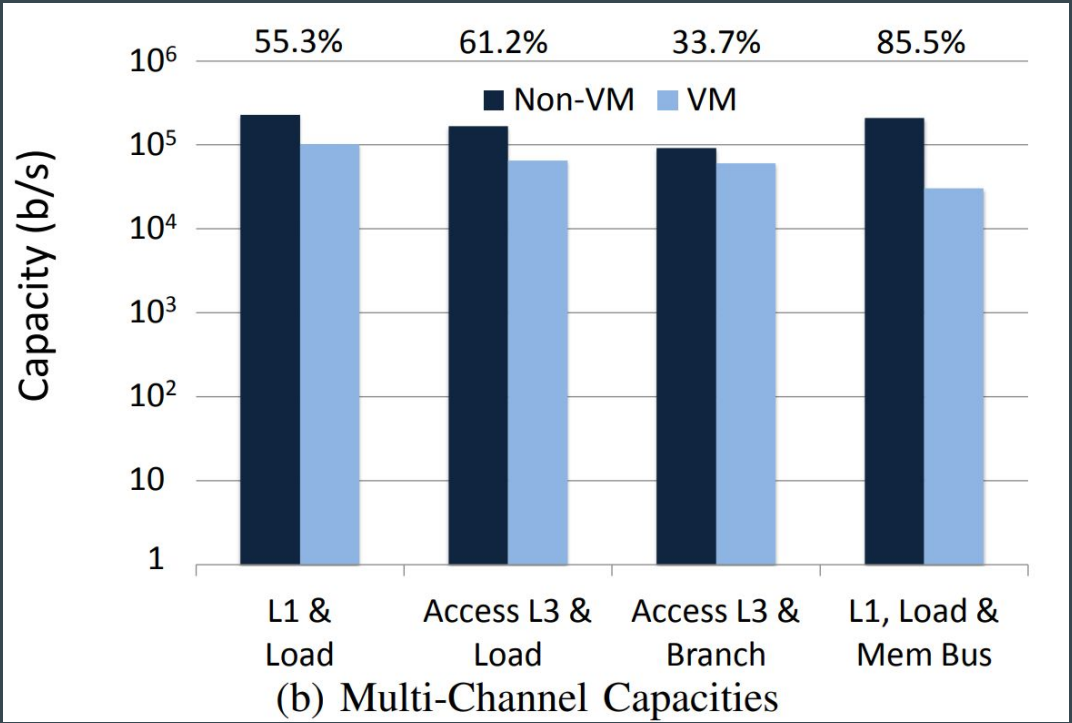
Experimental results for channel capacity







Pros and cons of MIMO comms.



(b) Multi-Channel Capacities

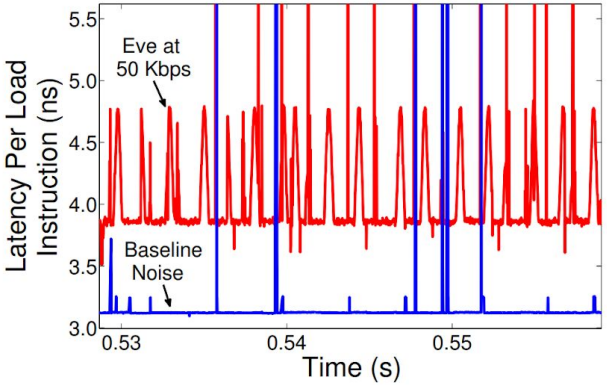
How to detect an Eve with a friendly VM (Claude)

- Optimal detection pattern is to transmit consecutive 0s
- H/e, noise makes it hard for Claude to distinguish btwn Eve and real noise

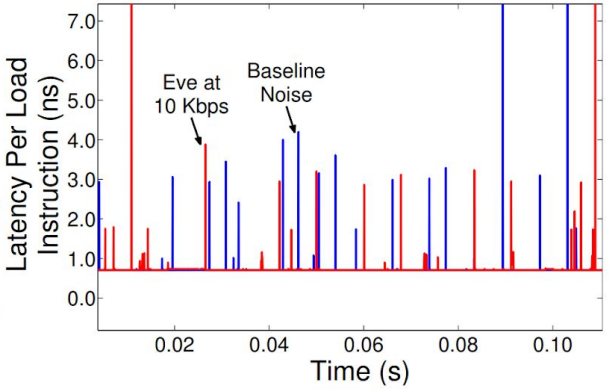
- **Idea**: Claude should mimic the target application so Eve cannot know when Claude is monitoring

Intelligent Principals	Claude	Eve	Result
Neither	Probes @ 50 Kbps	Probes @ 50 Kbps	Eve is easily detected
Eve	Probes @ 50 Kbps	Hides in noise until Claude has finished detection	Claude cannot distinguish between Eve and noise
Both	Intelligent detector mimicking httpd	Attempts to hide from Claude to avoid detection	Eve cannot distinguish between httpd and Claude, and is detected

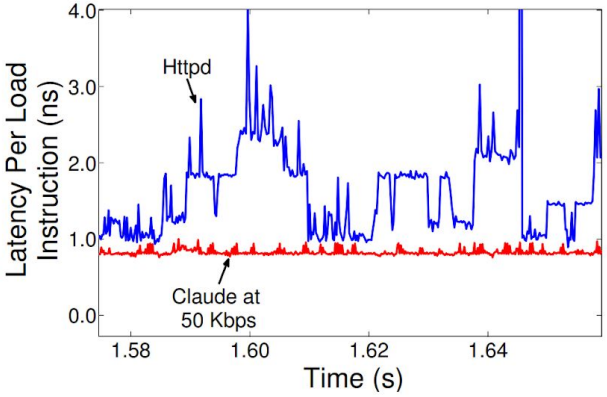
TABLE I: Three stages in the detection game between Claude and Eve.



(a) Claude's view: Eve v. Baseline Noise



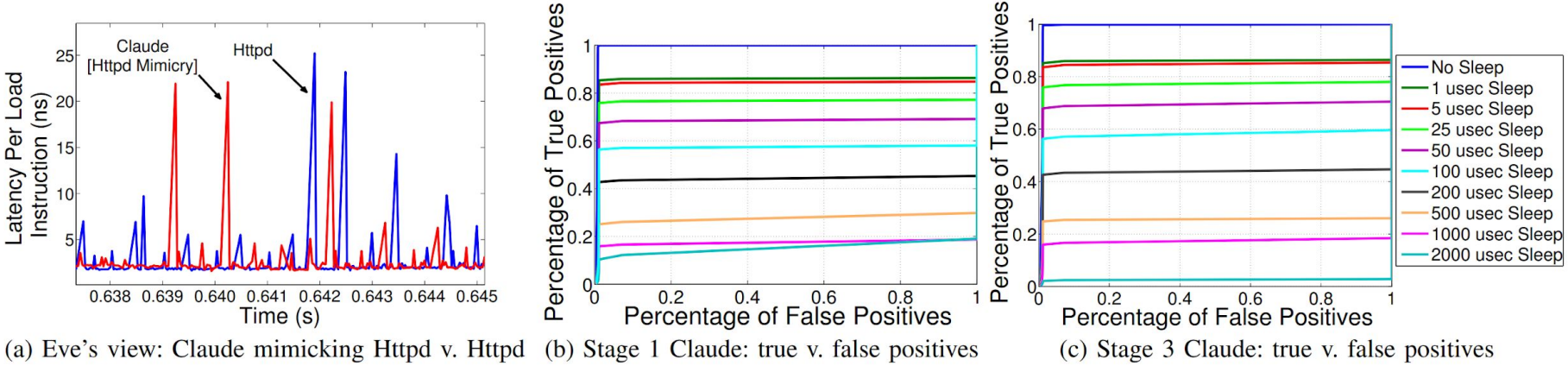
(b) Claude's view: Eve hiding within noise



(c) Eve's view: Claude v. httpd

Fig. 11: (a) Stage 1: Claude wins since noise and Eve at 50 Kbps are markedly different. (b) Stage 2: Eve reduces probe bandwidth to 10 Kbps and Claude cannot reliably tell Eve and Baseline noise apart. (c) Eve, however, can tell Claude apart from the target application (`httpd`).

Claude's detector seems unaffected by his added noise



(a) Eve's view: Claude mimicking Httpd v. Httpd (b) Stage 1 Claude: true v. false positives (c) Stage 3 Claude: true v. false positives

Fig. 12: (a) Claude adds noise into the channel by creating a Markov Model of `httpd` and mimicking the effect that `httpd` has on the load channel. To Eve, Claude looks like `httpd` but Claude's detector can look for additional noise in the channel to identify Eve. (b) Detection v. false positive results for the unintelligent Claude (Stage 1) generated by varying detection threshold (number of 1s before alarm is raised). Eve at 50 Kbps is almost 100% detectable. Forcing Eve to 10 Kbps (100μs sleep) still yields a detection rate around 40%. (c) Intelligent Claude (Stage 3), by adding noise into the channel should hamper its own detector, yet detection rates are close to that of Stage 1 Claude.

Takeaways

- Software leaks and covert channels can abuse this well!
- Alternate solution from CCS '15: **Nomad** focuses directly on co-residency problem and proposes migration-as-a-service as a side-channel agnostic solution

- **Security is necessary even as earlier as design time, waiting until deployment is too late**

Spectre Attacks: Exploiting Speculative Execution

Speculative Execution

- Speculative execution helps speed up performance when the CPU would otherwise stall
- The CPU speculatively executes branches in order to not stall
  - Ex. jump target is not readily available (ex. stored in memory, not available yet)
- As long as the CPU reverts state correctly, there are no correctness issues
  - But what about security?

Spectre

- The Spectre paper focuses on exploiting speculatively executed branches and jumps
- Variant 1: Conditional branches
- Variant 2: Indirect branches / jumps
- Alternative Variants (briefly mentioned, but not discussed)
  - Mistrained returns
  - Timing variations
  - ALU contention

Spectre Variant 1: Conditional Branches

- Train the CPU's branch predictor to predict one way, then gains access to memory through a later mispredict

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

- Here, an attacker would input valid values for x in order to train the branch predictor, then use an invalid value.
- The processor still pulls the reads from array1 and array2 into the cache, which wouldn't get flushed during a revert

Spectre Variant 2: Gadgets

- Based on getting the victim to run “gadgets” in the victim’s address space
  - Gadgets are small snippets of assembly, typically ending in a ret instruction
- Especially hard to mitigate because the victim code can be structurally safe, but still vulnerable

Spectre Variant 2: Indirect Branches

- An attacker trains the BTB to mispredict on an indirect branch (one where the address is stored in memory) to jump to the gadget.
- Attacker choose a gadget in the **victim's** virtual address space
- Trains the BTB while in the **attacker's** address space by performing indirect jumps to the correct address
- When the victim code runs, an indirect jump that aliases to the same BTB entry will result in the gadget running

Exploiting Spectre

- Cache Attacks (Flush+Reload / Prime+Probe / Evict+Time)
- Flush+Reload / Prime+Probe
  - Measures which location in array2 was brought into the cache to reveal the value of array1[x]
- Evict+Time
  - Call target function again with inbounds array1[x], and if the values are equal, it’ll be faster since it’s cached

Exploiting Spectre

- Alternative timing attacks / observable effects
  - Instruction timing
  - Register file contention
  - Bus contention
  - Electromagnetic radiation
  - Power consumption

Mitigations

- Prevent speculative execution / branch prediction
  - Dramatic performance drops
  - Legacy code would *all* have to be updated
- Prevent access to secret data
  - Best for just-in-time compilers / interpreters / language based protections or programs like Chrome where each program can be a separate process with different permissions

Mitigations

- Prevent data from entering covert channels
  - Track if data fetch was a result of a speculative operation
  - Not possible with current hardware
- Prevent data extraction from covert channels
  - Disable or add jitter to timing sources
  - Not a total mitigation - just adds error

Mitigations

- Prevent branch poisoning
  - Hardware
    - BTB flush
    - Privilege level isolation (lower privilege levels cannot affect indirect jumps of higher privilege levels)
    - Restricting branch prediction sharing
  - Software - Retpoline
    - Replaces indirect branches with return instructions

Merits

- Discovered a new variant of attacks on a broad range of processors
- Describes many different ways to exploit the attack
- Created multiple proof of concepts for each variant with high accuracy
- Discussion of mitigation options
- Responsible disclosure of results to CPU vendors

Methodology

- Performed experiments on multiple x86 processors from Intel and AMD as well as ARM processor, in several different OSes on Google Cloud
- Developed proof-of-concepts in C, Javascript, and eBPF for Variant 1
- Developed proof-of-concepts on Windows and a Linux VM for Variant 2

Methodology

- Observed indirect branch poisoning on a variety of x86 and ARM processors
- Reverse engineered branch predictor internals for Haswell processors to develop an “oracle” (a method of determining correctness - in this case, whether or not a bit affected the branch prediction)

Conclusions

- Speculative execution produces side effects
- Visible side effects can be exploited in many, many ways depending on how much energy an attacker wants to put in
- Speculative execution helps with performance, but at the cost of security vulnerabilities

Differences between Spectre and Meltdown

- Spectre is based on speculative execution (branch prediction / branch target prediction)
  - Affects Intel / AMD / ARM processors
- Meltdown is based on out of order execution based on trap instructions
  - Only Intel / ARM processors
- Meltdown mitigations don’t protect against Spectre!