Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races

CS 563
Young Li
10/31/18

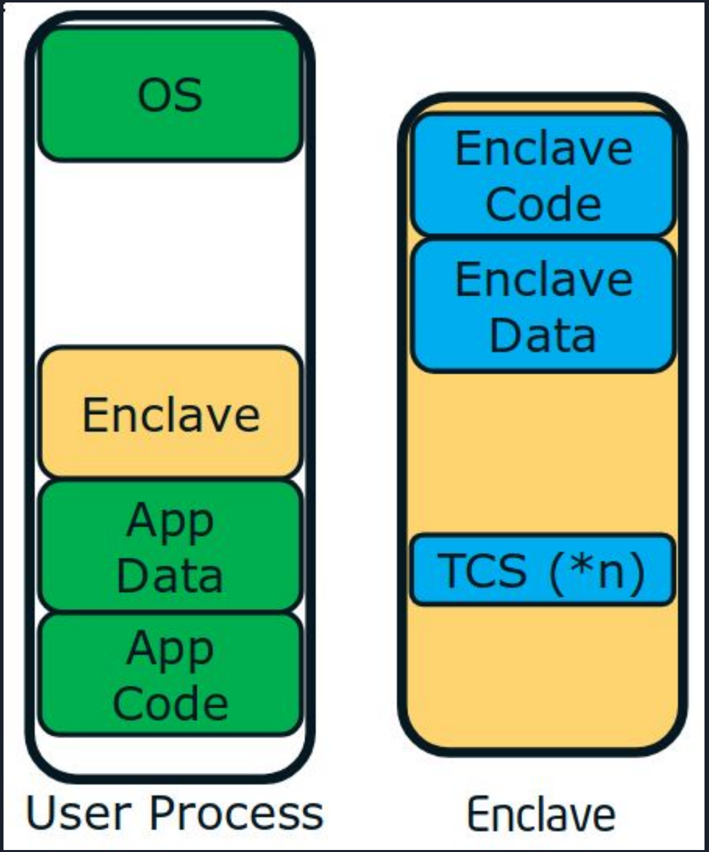
Intel Software Guard eXtensions (SGX) and Hyper-Threading

What is Intel SGX?

- Set of CPU instructions
- Present in Skylake and newer (6th gen and up)

What is Intel SGX?

- Lets programs create *enclaves*
  - Separate code and data
  - Supports multithreading
  - Enclaves have access to the program's memory



What is Intel SGX?

- Hardware provides isolation between enclaves and untrusted world
    - Virtual memory isolation
    - Physical memory isolation
    - Memory encryption for swapped-out enclave pages

What is Hyper-Threading?

- Intel’s proprietary implementation of Simultaneous MultiThreading (SMT)
- Presents two logical cores on each physical CPU core
- Logical cores share *execution units*
    - Caches
    - Translation lookaside buffers (TLBs)
    - Branch prediction units (BPUs)
    - Floating point units (FPUs)
    - etc.

Hyper-Threading Side Channels

An Example: TLBleed

- Attack by Gras et al. from Vrije Universiteit Amsterdam
- The Translation Lookaside Buffer (TLB) caches virtual memory mappings
    - Hyper-Threads share TLBs (L1 Data TLB and L2 TLB)
- Side-channel attack allows an attacker to determine data access patterns of a target program
    - Private key reconstruction
    - Image reconstruction
    - etc.

An Example: TLBleed

- Demonstrated cryptographic key reconstruction
  - libgcrypt EdDSA
  - libgcrypt RSA (less effective due to larger key size)
- Unaffected by mitigations to side-channel cache attacks

Other examples, briefly:

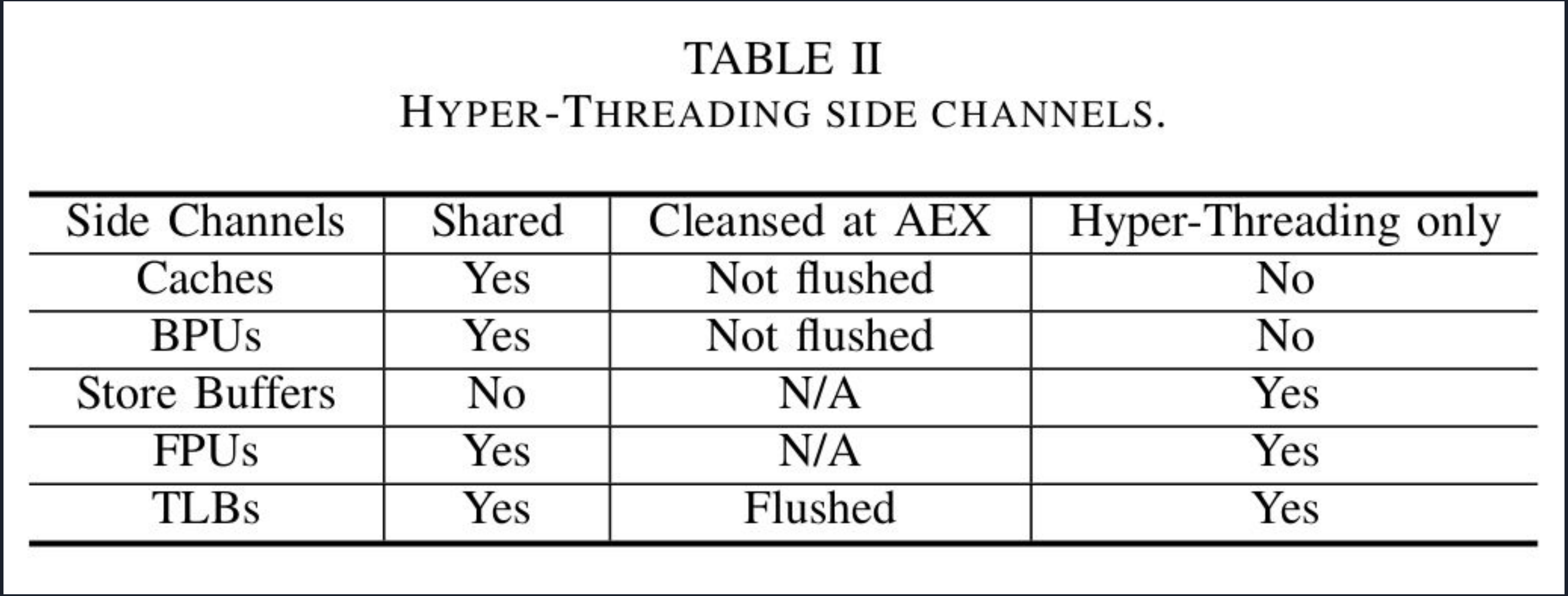


TABLE II
HYPER-THREADING SIDE CHANNELS.

Side Channels	Shared	Cleansed at AEX	Hyper-Threading only
Caches	Yes	Not flushed	No
BPUs	Yes	Not flushed	No
Store Buffers	No	N/A	Yes
FPUs	Yes	N/A	Yes
TLBs	Yes	Flushed	Yes

HyperRace: A software defense against Hyper-Threading side channel attacks

Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races

- Paper by Chen et al.
  - Ohio State University
  - Indiana University Bloomington
  - SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
- Proposed **HyperRace**, a tool to eliminate Hyper-Threading side channel attacks

Preventing Hyper-Threading Side Channels

- An attacker must schedule a thread on the same core as the enclave thread
- If we can prevent this from happening, the attacker would not be able to execute *any* kind of HT side channel attack!

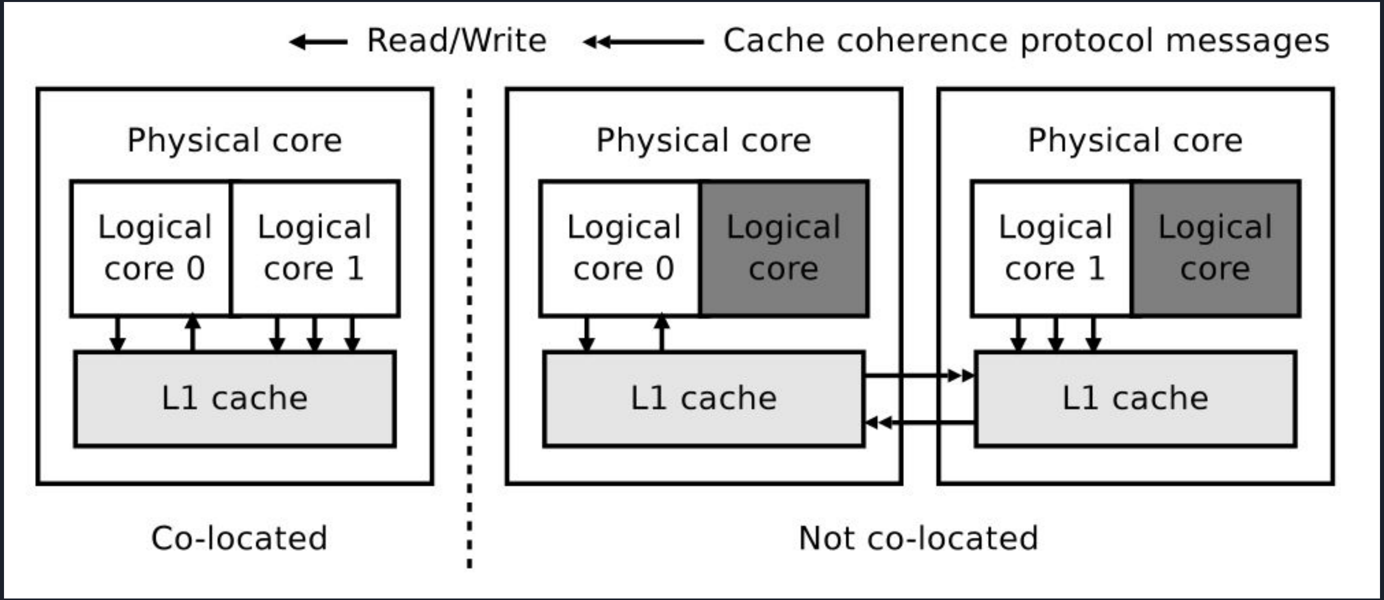
Preventing Hyper-Threading Side Channels

- For each enclave thread, create a *shadow thread*
- Must keep checking whether the enclave thread and shadow thread are co-resident on the same core

Checking co-residency

- Use knowledge of *shared resources* across logical cores
- Chen et al. chose to use L1 cache
  - Each physical core has a private L1 cache
- Measure memory access timings through the cache

Checking co-residency



~5 cycles

~190 cycles (on Intel Skylake)

Co-residency tests using contrived data races

- Intel SGX does not support secure clock instructions
- Chen et al. use *contrived data races* on an integer v

Enclave thread loop:

```
    Write 0 to v

    Wait for 10 cycles

    Read v
```

Shadow enclave thread loop:

```
    Write 1 to v
```

Co-residency tests using contrived data races

- Enclave thread will read 1 with **high** probability if **co-resident**
- Enclave thread will read 1 with **low** probability if **not co-resident**

Putting it another way:

- Co-resident: *communication time* < *execution time*
- Not co-resident: *communication time* > *execution time*

When should co-residency checks be used?

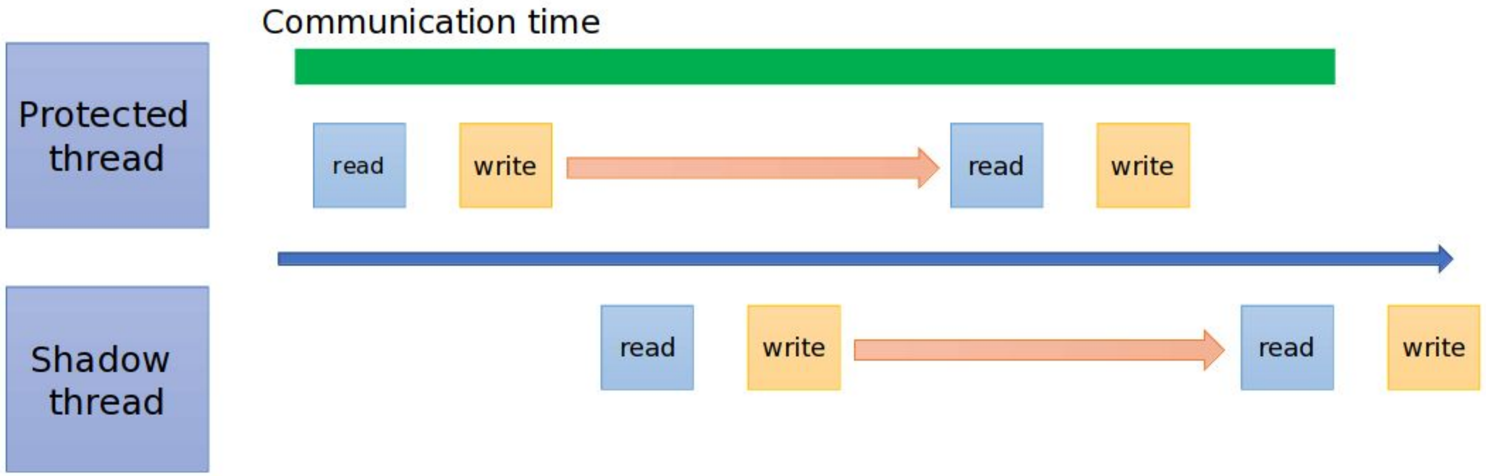
- **AEX**: Asynchronous Enclave eXit
  - Executed when enclave code is interrupted (context switches)
  - Saves registers, flushes TLB, etc.
- Must recheck co-residency after an AEX

Co-residency tests under stronger attacker model

Chen et al. consider an attacker who can:

- Cause cache contention
- Adjust CPU frequency
- Cache primes
- Disabling caching
- Disable caching + adjust CPU frequency
- ...

A refined data-race design



- When **not** co-located, communication time > execution time
- Each thread read the value written by the other thread with very **low** probability.

Co-residency tests under stronger attacker model

New design must satisfy two requirements under new attacker model:

1. Enclave thread (T_0) and shadow thread (T_1) observe data races on the shared variable v with high probabilities when they are co-located
2. When T_0 and T_1 are not co-located, at least one of them observes data races with low probabilities

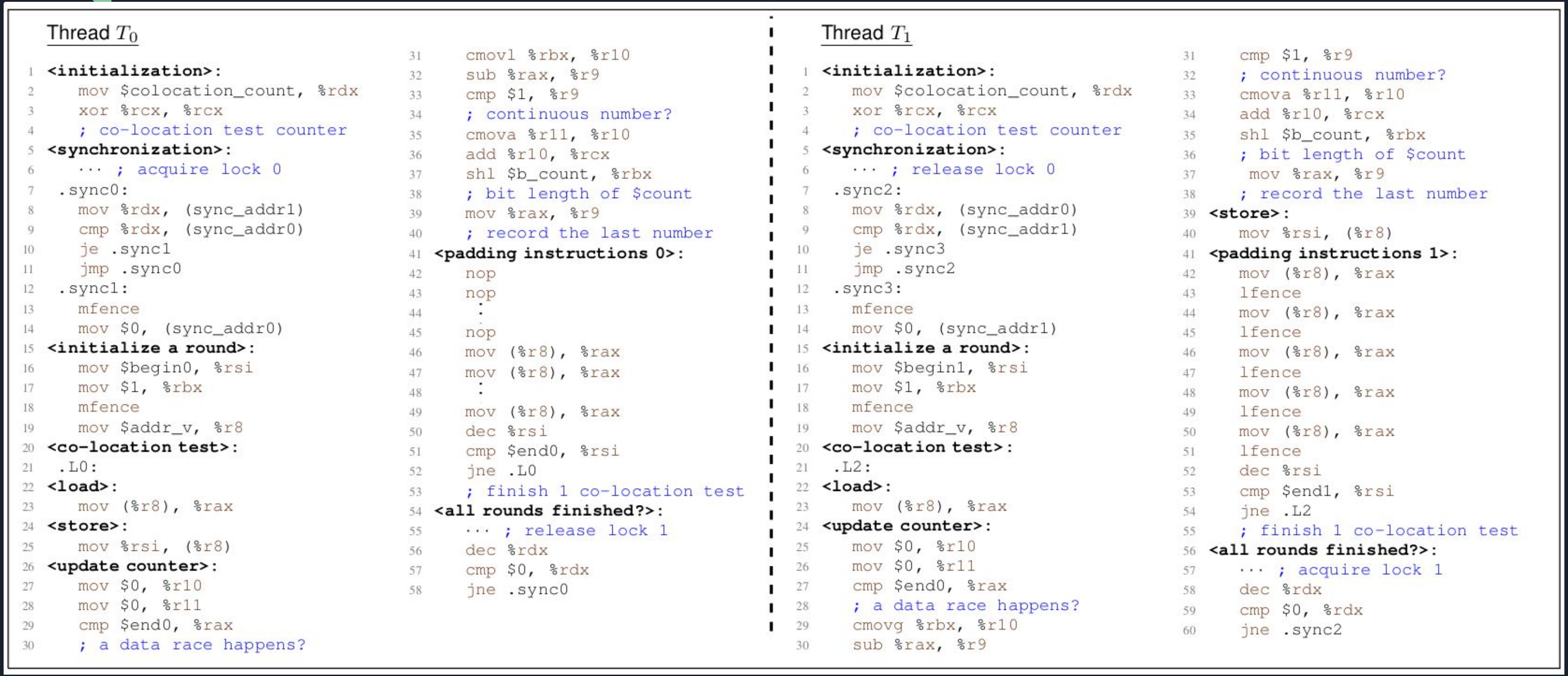
Security Evaluation of Co-residency Test

Attacker cannot meet both security requirements!

Considered:

- Latency of cache eviction
- Latency of cross-core communication
- Effects of CPU frequency change
- Effects of disabling caching

Co-residency tests under stronger attacker model



Thread T_0

```
<initialization>:
  mov $colocation_count, %rdx
  xor %rcx, %rcx
  ; co-location test counter
<synchronization>:
  ··· ; acquire lock 0
 .sync0:
  mov %rdx, (sync_addr1)
  cmp %rdx, (sync_addr0)
  je .sync1
  jmp .sync0
 .sync1:
  mfence
  mov $0, (sync_addr0)
<initialize a round>:
  mov $begin0, %rsi
  mov $1, %rbx
  mfence
  mov $addr_v, %r8
<co-location test>:
 .L0:
<load>:
  mov (%r8), %rax
<store>:
  mov %rsi, (%r8)
<update counter>:
  mov $0, %r10
  mov $0, %r11
  cmp $end0, %rax
  ; a data race happens?
  cmovl %rbx, %r10
  sub %rax, %r9
  cmp $1, %r9
  ; continuous number?
  cmova %r11, %r10
  add %r10, %rcx
  shl $b_count, %rbx
  ; bit length of $count
  mov %rax, %r9
  ; record the last number
<padding instructions 0>:
  nop
  nop
   :
  nop
  mov (%r8), %rax
  mov (%r8), %rax
   :
  mov (%r8), %rax
  dec %rsi
  cmp $end0, %rsi
  jne .L0
  ; finish 1 co-location test
<all rounds finished?>:
  ··· ; release lock 1
  dec %rdx
  cmp $0, %rdx
  jne .sync0
```

Thread T_1

```
<initialization>:
  mov $colocation_count, %rdx
  xor %rcx, %rcx
  ; co-location test counter
<synchronization>:
  ··· ; release lock 0
 .sync2:
  mov %rdx, (sync_addr0)
  cmp %rdx, (sync_addr1)
  je .sync3
  jmp .sync2
 .sync3:
  mfence
  mov $0, (sync_addr1)
<initialize a round>:
  mov $begin1, %rsi
  mov $1, %rbx
  mfence
  mov $addr_v, %r8
<co-location test>:
 .L2:
<load>:
  mov (%r8), %rax
<update counter>:
  mov $0, %r10
  mov $0, %r11
  cmp $end0, %rax
  ; a data race happens?
  cmovg %rbx, %r10
  sub %rax, %r9
  cmp $1, %r9
  ; continuous number?
  cmova %r11, %r10
  add %r10, %rcx
  shl $b_count, %rbx
  ; bit length of $count
   mov %rax, %r9
  ; record the last number
<store>:
  mov %rsi, (%r8)
<padding instructions 1>:
  mov (%r8), %rax
  lfence
  mov (%r8), %rax
  lfence
  mov (%r8), %rax
  lfence
  mov (%r8), %rax
  lfence
  mov (%r8), %rax
  lfence
  dec %rsi
  cmp $end1, %rsi
  jne .L2
  ; finish 1 co-location test
<all rounds finished?>:
  ··· ; acquire lock 1
  dec %rdx
  cmp $0, %rdx
  jne .sync2
```

Determining co-location statistically

- Each trial is a Bernoulli random variable with parameter *p*
  - Co-location with probability *p*
  - No co-location with probability *1-p*
- Each trial is *independent* because the two threads are synchronized every round

Determining co-location statistically

Running hypothesis testing:

- Define q as the observed ratio of passed co-location tests
- Define p as the expected ratio of passed co-location tests

Null hypothesis	$H_0: q \geq p$
Alternative hypothesis	$H_1: q < p$

Determining co-location statistically

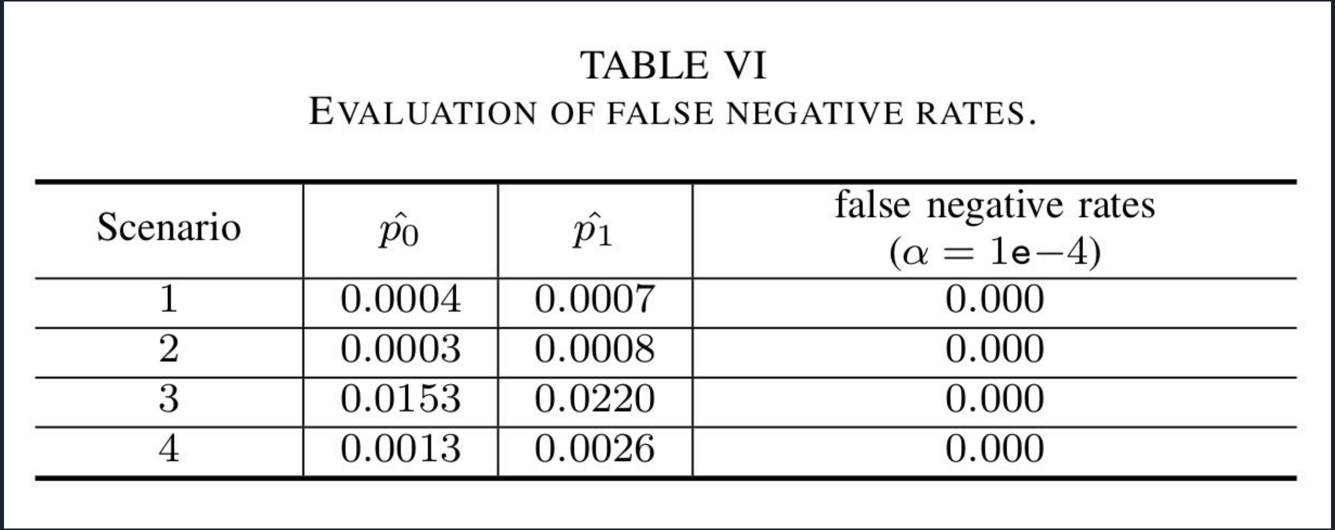


TABLE VI
EVALUATION OF FALSE NEGATIVE RATES.

Scenario	$\hat{p_0}$	$\hat{p_1}$	false negative rates ($\alpha = 1\text{e}{-4}$)
1	0.0004	0.0007	0.000
2	0.0003	0.0008	0.000
3	0.0153	0.0220	0.000
4	0.0013	0.0026	0.000

Implementing HyperRace

- Implemented as LLVM IR optimization pass when compiling enclave code
    - Perform AES detection code every *m* instructions
    - Execute co-location test routines
    - If co-location test fails, can retry or terminate

Performance Evaluation

Evaluation performed on:

- i7 6700 quad core (eight logical cores)
- 32 GB RAM
- p-value = 1e-6
- Ran *nbench* as enclave code and measured overhead of HyperRace

Memory Overhead

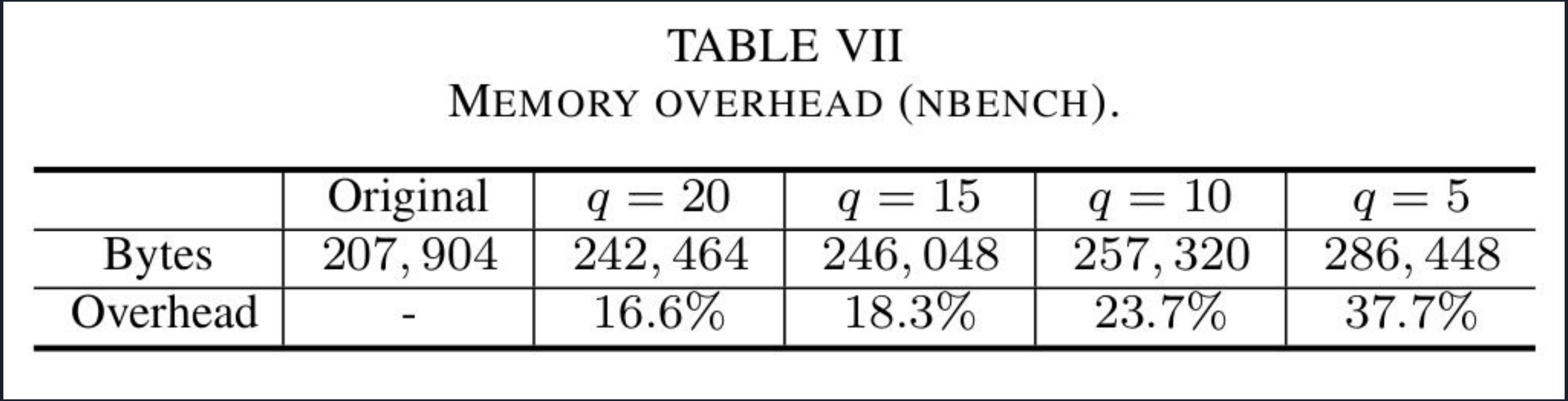


TABLE VII
MEMORY OVERHEAD (NBENCH).

	Original	$q = 20$	$q = 15$	$q = 10$	$q = 5$
Bytes	$207,904$	$242,464$	$246,048$	$257,320$	$286,448$
Overhead	-	16.6%	18.3%	23.7%	37.7%

q represents one AEX detection every *q* instructions in a basic block

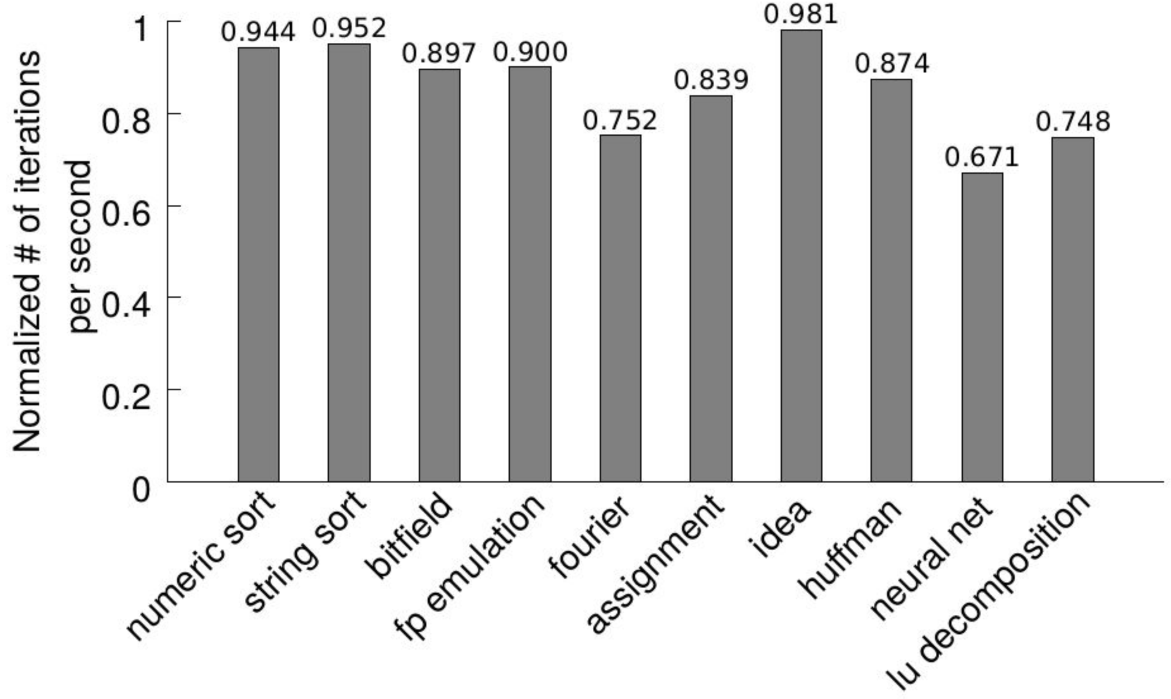


Fig. 8. Normalized number of iterations of *nbench* applications when running with a busy looping program on the co-located logical core.

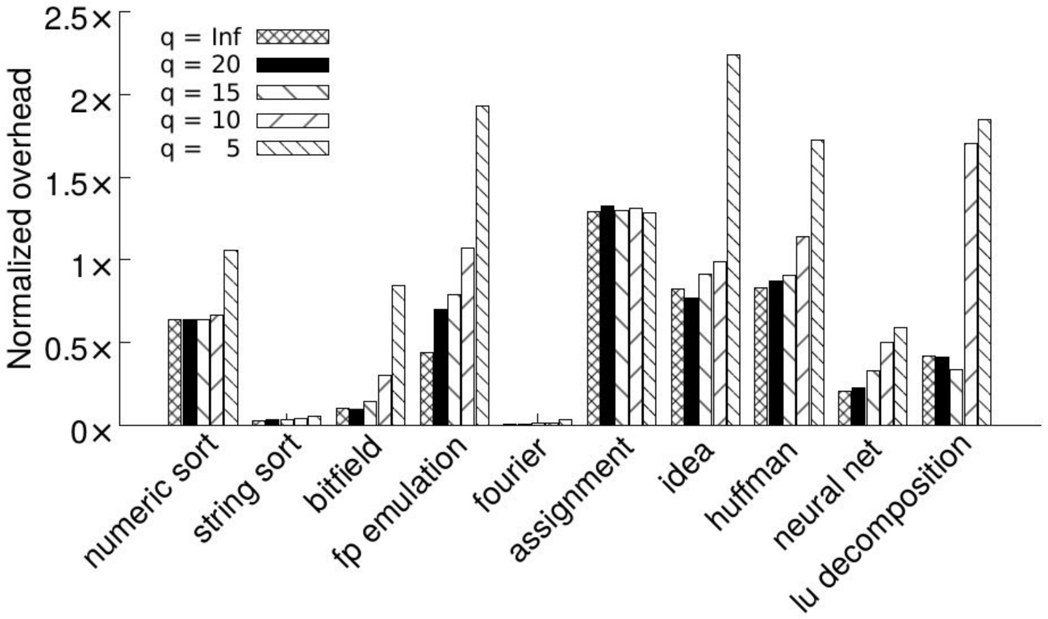


Fig. 9. Runtime overhead due to AEX detection; $q = \texttt{Inf}$ means one AEX detection per basic block; $q = 20/15/10/5$ means one additional AEX detection every q instructions within a basic block.

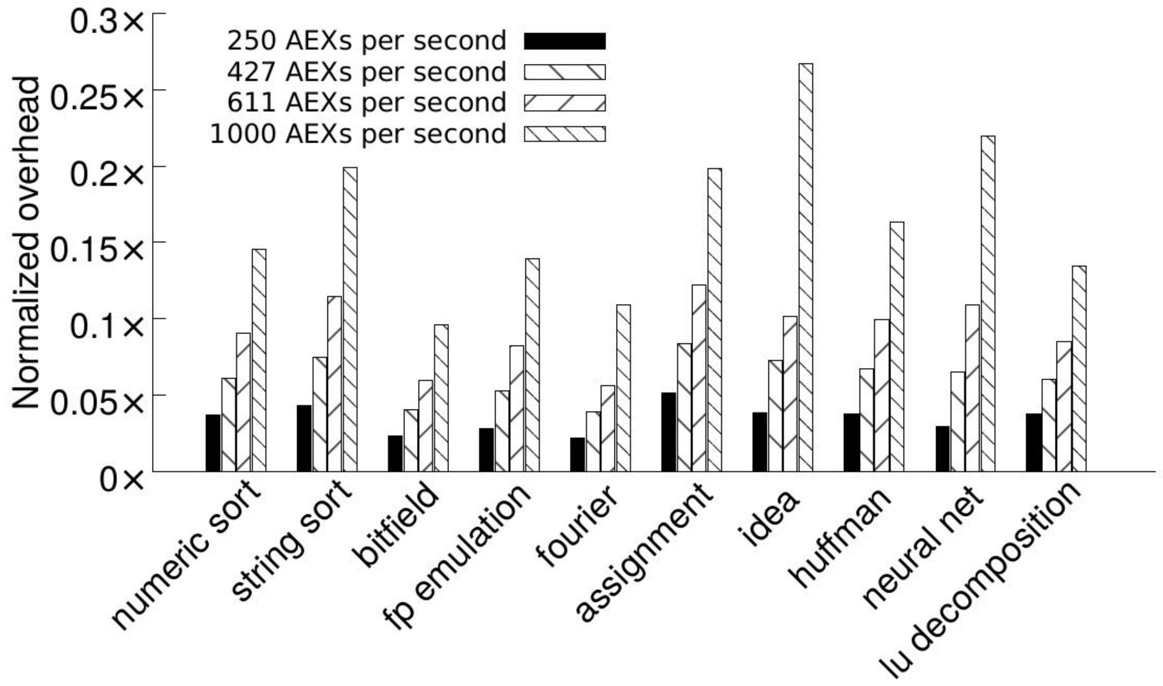


Fig. 10. Runtime overhead of performing co-location tests when $q = 20$.

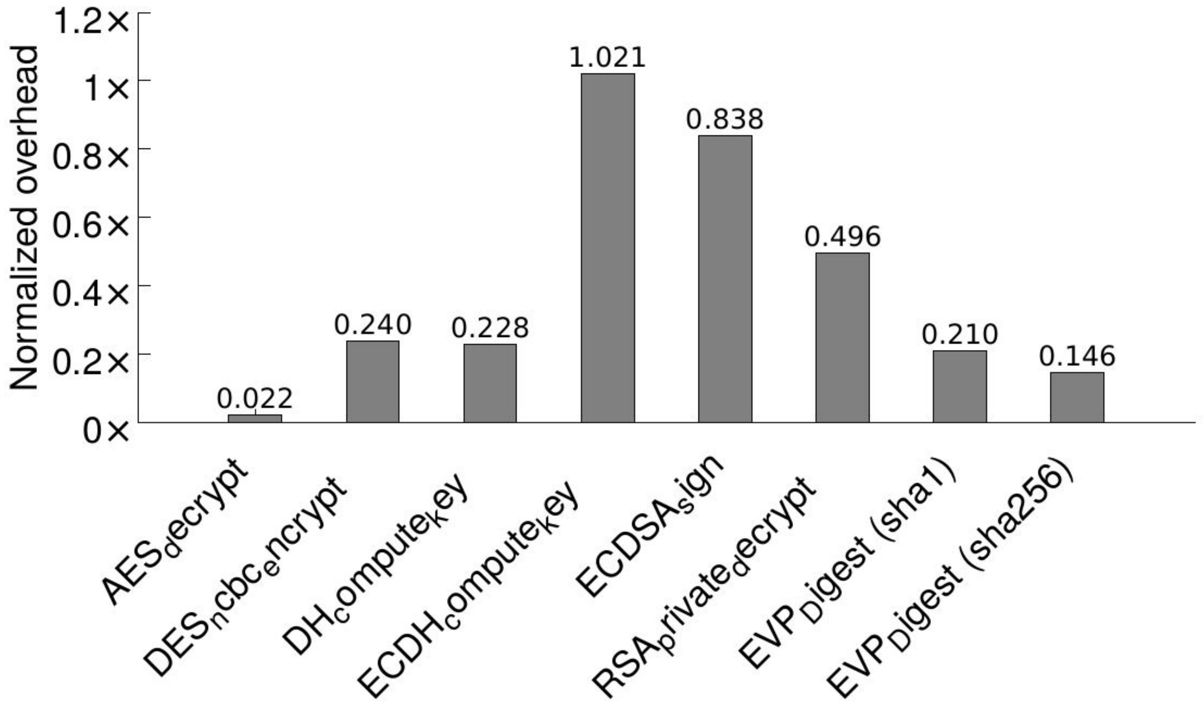


Fig. 11. Overhead of crypto algorithms.

Limitations of HyperRace

- Modest to high performance overhead
    - Depends highly on *q*
- Cost of non co-residency detection of enclave thread and shadow thread is high
    - Enclave thread should terminate itself
    - Attacker can perform denial-of-service
    - Shadow thread is not doing “useful” work

Thank you!

Any questions?

Sources

1. G. Chen et al., "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races," 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, US, , pp. 388-404. doi:10.1109/SP.2018.00024
2. G. Chen et al., “Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races” slides (http://web.cse.ohio-state.edu/~chen.4329/slides/sp18.pptx)
3. B. Gras, K. Razavi, H. Bos, C. Giuffrida, Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks, in: USENIX Security, 2018. https://www.vusec.net/download/?t=papers/tlbleed_sec18.pdf.
4. F. McKeen, Intel SGX slides (https://web.stanford.edu/class/ee380/Abstracts/150415-slides.pdf)
5. "TLBleed." VUSec. Accessed October 22, 2018. https://www.vusec.net/projects/tlbleed/.
6. Wang, Wenhao, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX." In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2421-2434. ACM, 2017.

Backup Slides

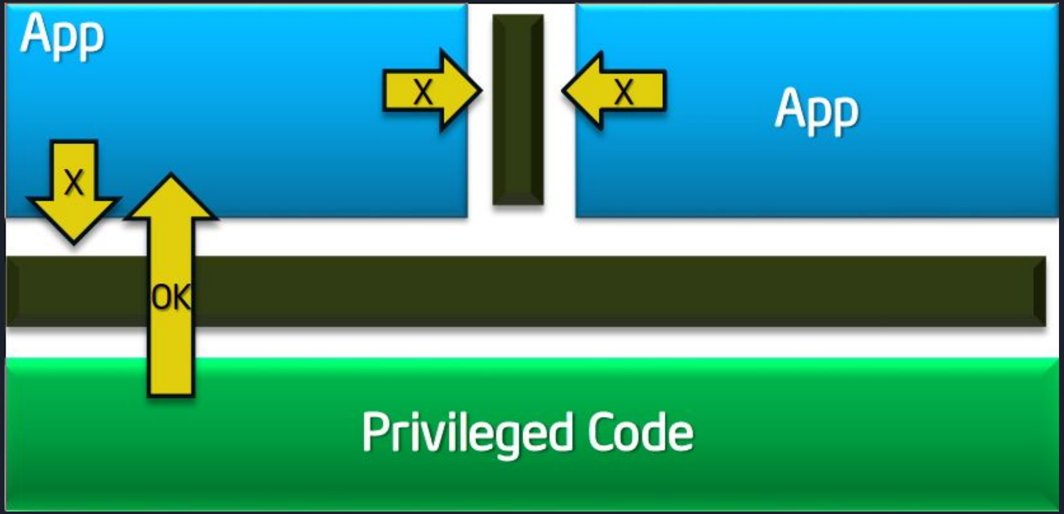
Motivation behind Intel SGX

Motivation: defending against malicious programs

- Preventing malicious user-space apps from doing damage to our app
    - Process isolation
    - Virtual memory
    - Protection rings

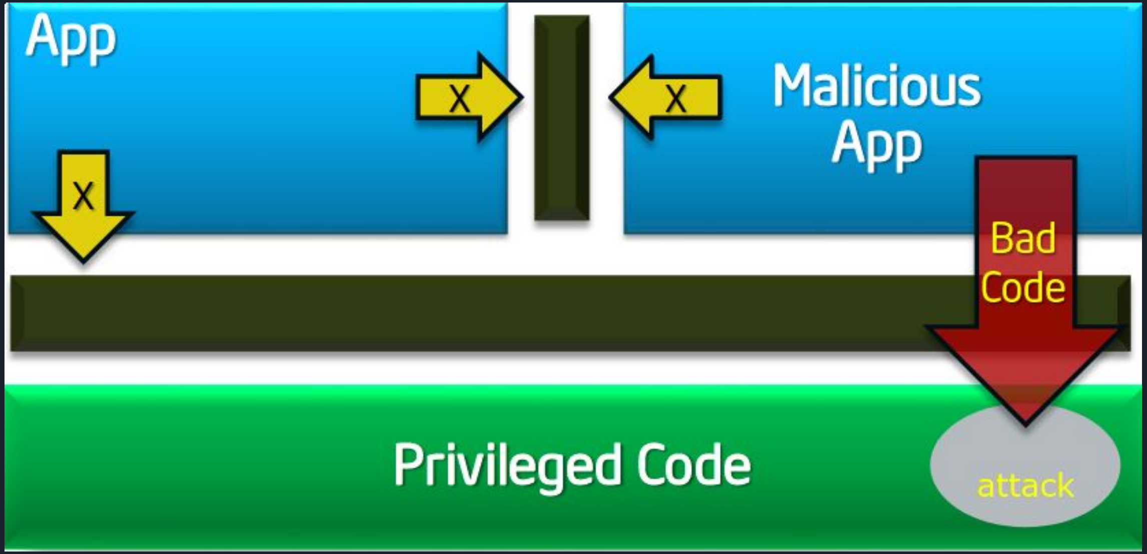
Motivation: defending against malicious programs

- Apps protected from each other
- OS protected from malicious apps



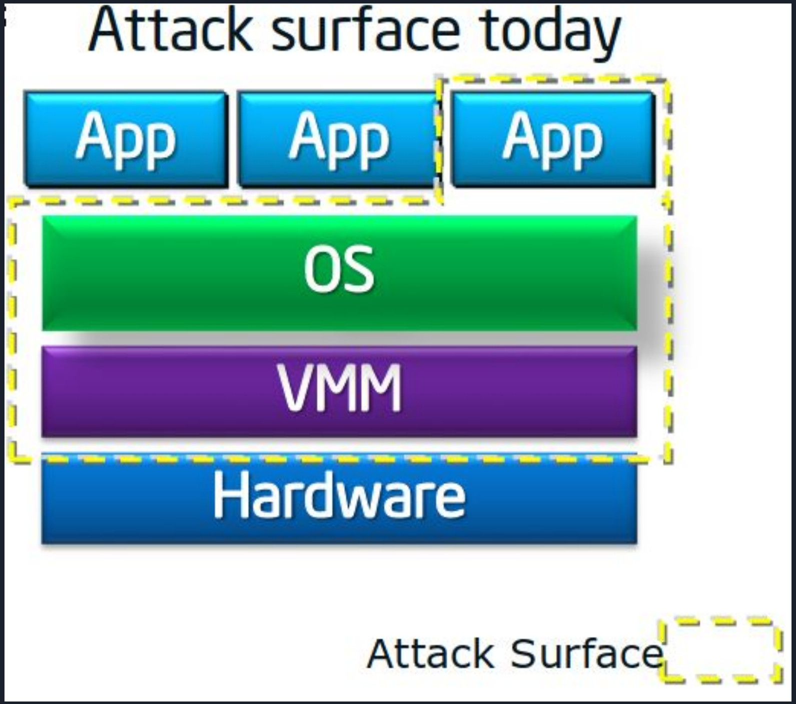
Motivation: use privilege escalation

- Malicious app can attack privileged code, get full privileges
- Privileged code: hypervisor, OS kernel



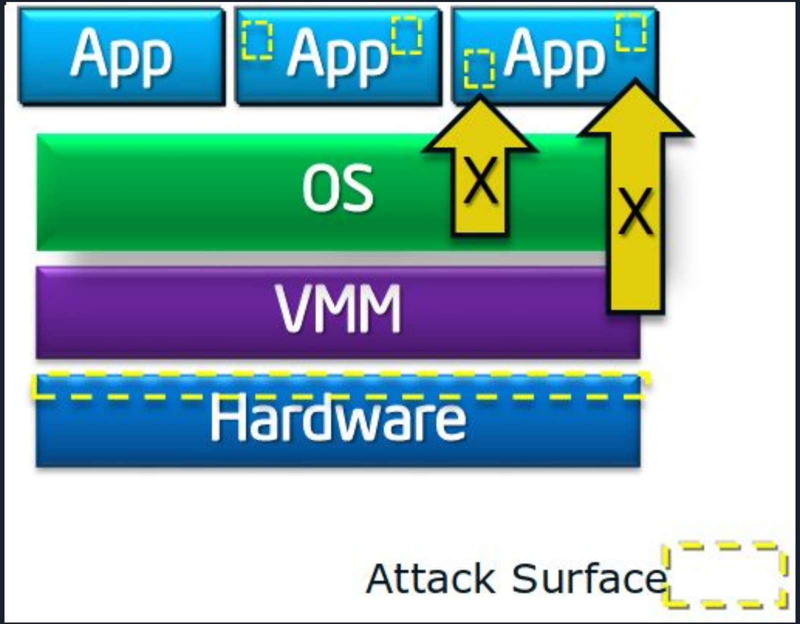
Insight: reduce the attack surface

- Apps can be attacked from multiple angles
  - OS
  - Hypervisor (VMM)
  - Hardware, somewhat?



Insight: reduce the attack surface

- Let's give an app the power to protect itself, using hardware
- Attack surface is *minimized*: only app itself, and hardware (CPU)


App
App
App
OS
X
X
VMM
Hardware
Attack Surface