RSAConference2018
San Francisco | April 16 – 20 | Moscone Center

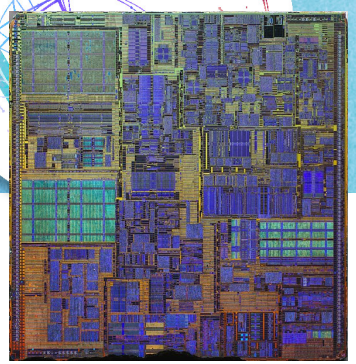
SESSION ID: PDAC-T07

SPECTRE ATTACKS: EXPLOITING SPECULATIVE EXECUTION

Paul Kocher
www.paulkocher.com

	Meltdown	Spectre
Consequence	Exposes physical memory	Exposes memory from kernel, other processes, or sandbox
Requires	Malicious user process	Malicious user process or sandboxed code (JavaScipt…)
Customization required	None	Tailor to victim SW (comparable to buffer overflow)
CPUs impacted	Intel, a few ARM	Virtually all fast CPUs – Intel, ARM, AMD, Power, Sparc…
Mitigation	Complete fix via O/S update	Messy partial fixes, if any x86: Microcode + O/S + drivers + application ARM: varies/none

Squeezing out CPU performance



Single-thread speed gains require getting **more done per clock cycle**

- Memory latency is slow and not improving much
- Clock rates are maxed out: Pentium 4 was 3.8 GHz in 2004

How to do more per clock cycle?

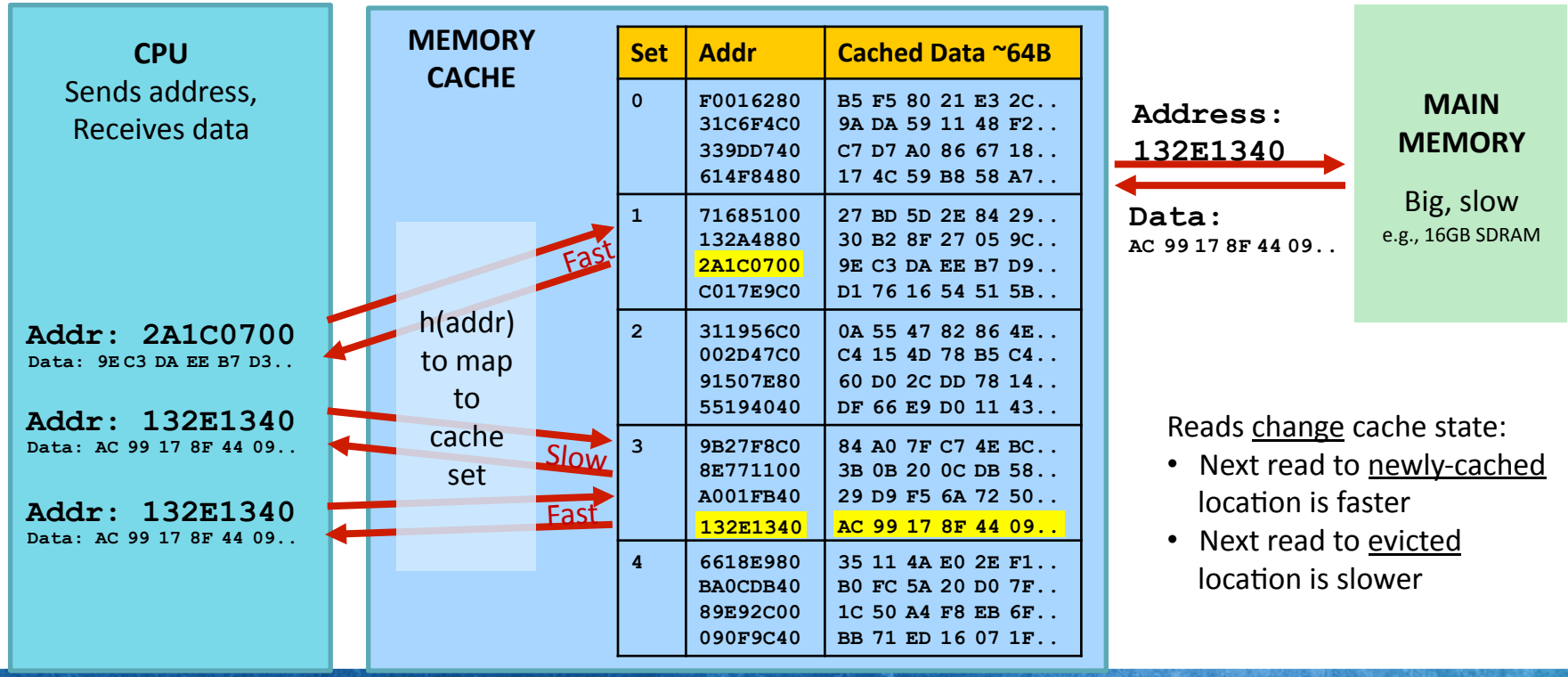
- Reducing memory delays → Caches
- Working during delays → Speculative execution

Spectre leverages interplay of these optimizations

Public domain image of Pentium 4 die by Ritzchens Fritz

Memory caches

- Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



Reads change cache state:

- Next read to newly-cached location is faster
- Next read to evicted location is slower

Speculative execution

CPUs run instructions

Correct result is defined as the result of performing instructions in-order

CPUs may run instructions out-of-order if this doesn't affect result

Example:

```
a ← constant
b ← slow_to_obtain
c ← f(a)     // start before b finishes
```

CPUs can also *guess* likely program path and do speculative execution

Example:

```
if (uncached_value_usually_1 == 1)
    compute_something()
```

- Branch predictor guesses that if() will be 'true' based on prior history (not current operation)
- Starts executing compute_something() speculatively -- but doesn't save changes
- When if() can be evaluated definitively, check if guess was correct:
  - Correct: Save speculative work – performance gain
  - Incorrect: Discard speculative work

Architectural World

Guarantees final register values and memory state

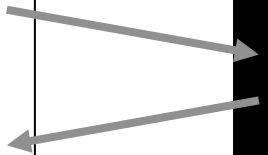
Speculative World

CPU regularly performs incorrect calculations, but deletes its mistakes

Software security assumes CPU correctness.
Do speculation errors violate this assumption?

Set up the conditions so the processor will make a desired mistake

Fetch the sensitive data from the covert channel



Mistake leaks sensitive data into a covert channel (e.g., state of the cache)

Conditional branch (Variant 1) attack


```
if (x < array1_size)
   y = array2[array1[x]*512];
```

Imagine this code was in a kernel API where `x` came from caller (e.g., attacker)

Execution without speculation is safe:

CPU will not evaluate `array2[array1[x]*512]` unless `x < array1_size`

What about speculative execution?

Conditional branch (Variant 1) attack


```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attacker objective

- Read sensitive memory
- Example reads `array1[N+8]`
  … where N+8 > `array1_size`

Attack setup:

- Train branch predictor to expect if() is true
  (e.g., call with `x < array1_size`)
- Evict `array1_size` and `array2[]` from cache

Memory & Cache Status

`array1_size = 00000008`

Memory following `array1` base address:
8 bytes of data (value doesn't matter)
[… N bytes of other memory …]
09 F1 98 CC 90... (something secret)

```
array2[ 0*512]
array2[ 1*512]
array2[ 2*512]
array2[ 3*512]
array2[ 4*512]
array2[ 5*512]
array2[ 6*512]
array2[ 7*512]
array2[ 8*512]
array2[ 9*512]
array2[10*512]
array2[11*512]
  ...
```

Contents don't matter
only care about cache ***status***

Uncached | Cached

Conditional branch (Variant 1) attack


```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attacker calls victim code with x=(N+8)

- Speculative exec while waiting for array1_size
  - Predict that if() is true
  - Read address (array1 base + x) w/ out-of-bounds x
  - Read returns secret byte = 09 [fast – in cache]
  - Request memory at (array2 base + 09*512)

Memory & Cache Status

array1_size = 00000008

Memory following array1 base address:
8 bytes of data (value doesn't matter)
[… N bytes of other memory …]
09 F1 98 CC 90... (something secret)

array2[0*512]
array2[1*512]
array2[2*512]
array2[3*512]
array2[4*512]
array2[5*512]
array2[6*512]
array2[7*512]
array2[8*512]
array2[9*512]
array2[10*512]
array2[11*512]
…

Contents don't matter
only care about cache ***status***

Uncached Cached

Conditional branch (Variant 1) attack


```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attacker calls victim code with `x`=(N+8)

- Speculative exec while waiting for `array1_size`
  - Predict that if() is true
  - Read address (`array1` base + `x`) w/ out-of-bounds `x`
  - Read returns secret byte = **09** [fast – in cache]
  - Request memory at (`array2` base + **09***512)
  - Brings `array2[`**09**`*512]` into the cache
  - Realize if() is false: discard speculative work
- Finish operation & return to caller

Attacker times reads from `array2[i*512]`

- Read for `i`=**09** is fast (cached), revealing secret byte

Memory & Cache Status

`array1_size = 00000008`

Memory following `array1` base address:
8 bytes of data (value doesn't matter)
[… N bytes of other memory …]
09 F1 98 CC 90... (something secret)

```
array2[ 0*512]
array2[ 1*512]
array2[ 2*512]
array2[ 3*512]
array2[ 4*512]
array2[ 5*512]
array2[ 6*512]
array2[ 7*512]
array2[ 8*512]
array2[ 9*512]
array2[10*512]
array2[11*512]
  ...
```

Contents don't matter
only care about cache ***status***

Uncached Cached

Violating JavaScript's sandbox

- JavaScript code runs in a sandbox
  - Not permitted to read arbitrary memory
  - No pointers, array accesses are bounds checked
- Browser runs JavaScript from untrusted websites
  - JavaScript engine can interpret code (slow) or compile it (JIT) to run faster
  - In all cases, engine must enforce sandbox (e.g., apply bounds checks)
- Speculative execution can blast through safety checks…
  - Can we write JavaScript that compiles into machine code that leaks memory contents?

Violating JavaScript's sandbox

index will be in-bounds on training passes, and out-of-bounds on attack passes

JIT knows this check ensures **index** < **length**, so it omits bounds check in next line. Caller evicts **length** for attack passes.

Do the out-of-bounds read on attack passes!

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

Hint to JS that result is an integer

4096 bytes (= page size)

Keeps the JIT from adding unwanted bounds checks on the next line

Use the result so computation isn't optimized away

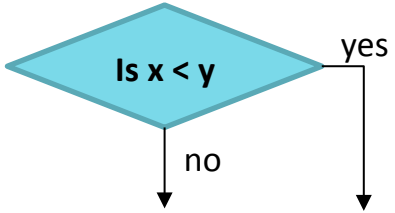
Leak out-of-bounds read result into cache state!

- Branch predictor matches jump history, so bit operations used to load `index` for training vs. attack passes
- No access to **`clflush`**, so eviction done by bulk reads on 4KB intervals
- JavaScript timers degraded, so timing done via a thread ("worker") that decrements a shared memory location in a tight loop

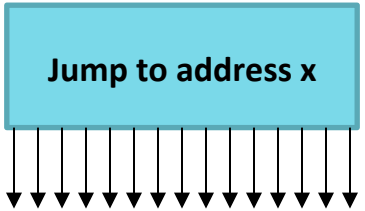
Indirect branches

Conditional branches have 2 possible destinations

- E.g., next instruction if false, jump destination if true



- Indirect branches can go anywhere
  - Examples on x86: `jmp [1234567]` `jmp eax` `ret`
  - Branch target buffer: CPU tracks past jump destinations to make quick guesses
  - If destination is delayed, CPU guesses and proceeds speculatively



- Vastly more freedom for attacker – billions of possible destinations (vs 2)

Poisoning indirect branches

- Pick indirect branch to redirect speculatively
  - e.g., a jump that occurs with attacker-controlled values in some registers
- Pick destination for victim to speculatively execute (the “gadget”)
  - e.g., gadget that leaks attacker-chosen memory address into covert channel
  - Like return oriented programming, but gadget also doesn’t have to return nicely
- Attack
  - **Mistrain** branch prediction/BTB so speculative execution will go to gadget
  - **Evict** or flush destination address from the cache to ensure long duration speculative execution
  - **Execute** victim so it runs gadget speculatively
  - **Detect** change in cache state (e.g., EVICT+RELOAD) to determine memory data
  - **Repeat** for more bytes

Indirect branch attack example

Victim program

- Example code repeatedly hashes a secret key & header of a file
- Happens to calls Sleep() with attacker-controlled data in **ebx** & **edi**
- First instruction a DLL call (incl. Sleep) is an indirect jump: `jmp dword ptr ds:[754F009Ch]`

Gadgets

- Searched a simple program's system DLL code to identify gadgets leveraging in **ebx** & **edi**
- Gadget in ntdll.dll:

```
adc   edi,dword ptr [ebx+edx+13BE13BDh]
adc   dl,byte ptr [edi]
```

  First instruction adds word from attacker-controllable address [edx=3 in victim, carry is clear] onto edi
  Second instruction reads memory location [edi]

- Used an another gadget to break ASLR:

```
sbb   eax,dword ptr [esp+ebx]
```

  - Repeat: Try a value for ebx then check cache state of a chunk of shared readable memory
  - Reveals value of esp

Attack

- **Mistrain** jmp to gadget, **evict** address with jmp destination, **execute** victim, **detect** cache state

Poisoning indirect branches: Notes

- Predictor works on virtual addresses
  - Training using attack process virtual addresses gets applied to victim process
  - Collisions in history tracking allow various simplifications
- Speculative execution only runs from addresses the victim can execute
  - Gadget must be executable in victim's address space
  - Example: gadgets in Windows DLLs
  - Example: gadgets created using eBPF JIT
- Reminder: Spectre leverages victim's permissions
  - Speculative execution occurs in victim context
    [Meltdown is different – privilege escalation]

Mitigations

Mitigation. noun. “The action of reducing the severity, seriousness, or painfulness of something”

Not necessarily a complete solution

Public domain image by Svetlana Miljkovic
Definition from https://en.oxforddictionaries.com/definition/mitigation

Mitigations: Conditional branch variant

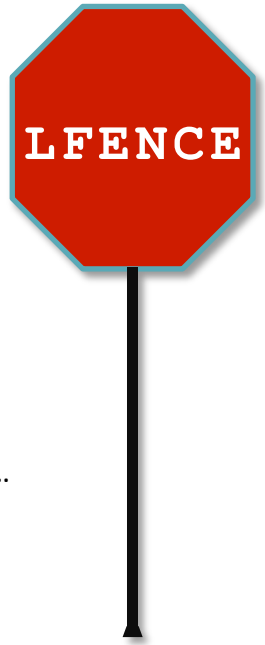
CPU vendors redefining instructions that work on existing CPUs

- `LFENCE` (Intel, some AMD may need microcode update)
- `CSDB` for some ARM processors, others have no mitigation other than using conditional loads

Works in theory… what about in practice?

- Must protect all exploitable code patterns: Miss one and attacker can read entire process memory
- Manual insertion is impractical
  - Many codebases have millions of jumps
  - Code reviews are really hard (speculation runs far -- e.g., 188++ instructions)
- Insert everywhere = simple-ish but big performance hit
  - e.g., SHA-256 test = 59.4% performance reduction
  - Static analysis is hard: Choose performance or security
    - Microsoft Visual C/C++ `/Qspectre` only adds `LFENCE` to a narrow known-bad code pattern
    - 13 of 15 code examples yielded unprotected output https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html
- Lots of software potentially impacted: Kernel, device drivers, web servers, interpreters/JITs, crypto code, databases…
    - Kernels will get careful attention, but lots other software won't (drivers, etc.)

Allows blaming developer post-breach ("should've added an `LFENCE` there…")



Mitigations: Indirect branch variant

x86:

- New MSRs created via microcode
  - IBRS: Restrict speculation of indirect branches - e.g., disable predictor
  - STIBP: Prevents hyperthread from training predictor - e.g., stop training
  - IBPB: Prevent prior learning from influencing predictions - e.g., clear predictor state
  - Messy:
    - Microcode updates for many (but not all) Intel & AMD CPUs. Stability problems + retractions…
    - Available from kernel mode only (use by applications unclear)
    - Substantial performance impacts
- Retpoline proposal from Google
    - Replace indirect jumps with construction that resists indirect branch poisoning
    - Works on some CPUs (e.g., Haswell), but need microcode updates for others (e.g., Skylake)

- ARM: No generic mitigation option
  - Faster CPUs broadly impacted (e.g., Cortex-A9, A15, A17, A57, A72, A73, A75...)
  - On some chips, software may be able to invalidate/disable branch predictor (with “non-trivial performance impact”) https://developer.arm.com/support/security-update/download-the-whitepaper

DOOM with only MOV instructions

Only MOV instructions

- No branch instructions
- One big loop with exception at the end to trigger restart

Sub-optimal performance

- One frame every ~7 hours

A branchless DOOM

This directory provides a branchless, mov-only version of the classic DOOM video game.

DOOM, running with only mov instructions.

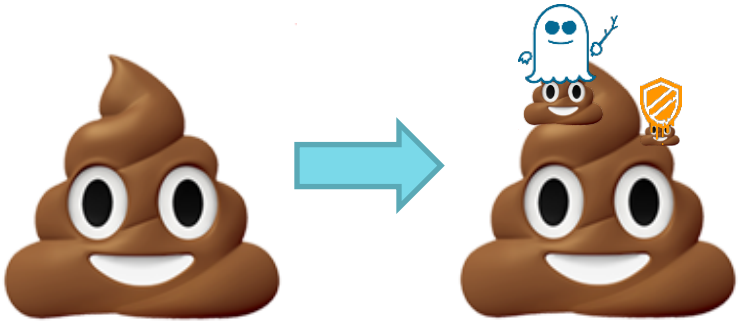
This is thought to be entirely secure against the Meltdown and Spectre CPU vulnerabilities, which require speculative execution on branch instructions.

https://github.com/xoreaxeaxeax/movfuscator/tree/master/validation/doom

Public domain image by Svetlana Miljkovic

Risk in context

Because of software bugs, computer security was in a dire situation



Spectre doesn't change the magnitude of the risk, but adds to the mess

Complexity of fixes -> new risks

Psychology of unfixed vulnerabilities

Is Spectre a bug?

- Everything is working 100% as intended
  - Branch predictor is learning from history, as expected
  - Speculative execution unwinds architectural state correctly
  - Reads are fetching data the victim is allowed to read
  - Caches are allowed to hold state
  - Covert channels known to exist (+ often impractical to eliminate)
- Everything complies with architectural specs from Intel, ARM, etc.

?!

A simple instruction: `add rax,[rbx]`

Architectural Guarantee

Compute the correct value in rax*

* assuming voltage/clock/temp… all in spec

Microarchitectural Properties

- Computation time
- Change in memory cache state
- Timing impact on other threads
  (e.g., due to use of the memory bus utilization, arithmetic unit…)
- Analog effects (power consumption, heat…)

. . .

Architecture ⇔ Software security gap

- CPU architecture guarantees are **insufficient for security**
- Under-specified -> software & HW developers make different assumptions
- No way to know if code is secure on today's chips
- Future chips may be completely different

Hardware challenges

- Better/faster CPU mitigations
  - Trade-offs between narrow patches (faster) vs. comprehensive fixes (safer)
    - Tricky – lots ways to adjust Spectre attacks to work around fixes
    - Unlikely to abandon speculative execution – too important for performance
  - Roadmap challenge
    - Break trend of increasing use of speculative execution for performance gains?
  - Tempting to punt to software developers… who will fail
    - Complex & virtually untestable – software teams are failing on easier problems
    - Developers can't cope with microarchitectural complexity (LFENCE, cache minutae…)
    - Punt responsibility vs. fix
- Spectre will linger
  - Long device lifetimes
  - Initial countermeasures are narrow
  - New attack techniques tend to evolve (DPA, buffer overflows, use-after-free…)
  - Embedded chips with old ARM designs will keep getting made for a very long time

Apply: For users/IT departments

- Install updates + more updates + more updates…
  - O/S fix for Meltdown + kernel hardening against Spectre
  - Iterate as new attacks using Spectre arise
- Improve separation
  - Consider segregating sensitive & untrusted workloads on different servers
  - Be especially cautious about Hyperthreading -- expect new side channel attacks
- No viable alternatives in the market (“buy new hardware”)
  - Intel + AMD + ARM + Power + etc. all impacted – none better than the other
  - Vendors touting immunity largely just too small/obsolete/unoptimized to use speculative execution
- Longer-term: Press vendors to deliver practical high-assurance/low-complexity separation
  - Today’s CPUs + hypervisors + compilers all problematic: prioritize performance > security

Why wasn't Spectre found earlier?

- Attack is obvious… in hindsight:
  - Component behaviors all widely known (e.g., taught in CPU design textbooks)
  - Speculative execution -> queasy feelings even without an exploit
  - Independent discovery seems to be complete coincidence

- Why missed?
  - Dismissal of prior side channel attacks as out of scope (cache, DPA, fault…)
  - Assumption that established architectures are correct
  - Overwhelming complexity narrows focus of separate technical teams
  - Focus on minutiae (bugs)
  - Performance-first culture

Spectre isn't the first – or last – hardware security vulnerability…

"The reality is there are probably other things out there like [Spectre] that have been deemed safe for years. Somebody whose

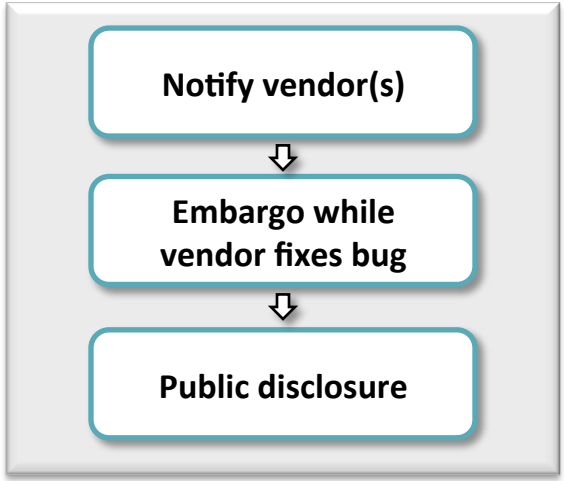
mind is sufficiently warped

toward thinking about security threats may find other ways to exploit systems which had otherwise been considered completely safe."

— Simon Segars (CEO, Arm Holdings)

https://www.cnet.com/news/spectre-meltdown-arm-intel-apple-qualcomm-softbank-simon-segars-pcs-phones/

HW issues & responsible disclosure



Messy but works… for software

Which vendors? Who decides? Who coordinates?

- Vendors: logic cores… chips… products… O/S… apps… cloud services…
- Who else: Governments… big customers… CERT… security experts…

How long? Who decides length?

- Time to 'fix' may be years (or ∞)
- Worst case: Bad guys exploiting & defenders lack info

How?

- Role of marketing, legal (32+ lawsuits), investor relations, press…?
- Spectre/Meltdown coordinated embargo (many vendors + researchers)

"AMD is not susceptible to all three variants. […] there is a near zero risk to AMD processors at this time."

I'm 0-for-2 applying to HW issues: Spectre + Differential Power Analysis (DPA)
(DPA affected virtually all smart cards used for payments, pay TV…)

(# people with need-to-know) >> (# who can keep a secret)

- Both embargos ended abruptly due to leaks
- Most impacted organizations got no advance warning

What is the right ethical process for handing hardware vulnerabilities?

Today: Costs of insecurity >>> Value of performance gains

Technical challenge

Building foundations that are unlikely to harbor vulnerabilities

Cultural challenge

Unlearning 50+ years of obsession with performance

- Current designs are too complex to secure
- Too many constraints: Performance, legacy compatibility…

* Estimates of insecurity costs: Symantec $172B, RAND (depending on assumptions) $275B-$6.6T

A path forward

Today: Same CPU and O/S designs for everything

- Video games, corporate email, wire transfers…
- No tailoring to security vs. performance needs
  - Example: All cores are identical for multi-core x86 CPUs

Need to bifurcate designs into **faster** vs. **safer** flavors

- Can co-exist on the same chip (analogous to ARM's big.LITTLE for power)
- 'Safer' must be much less complex -- not just a different mode like TrustZone/SGX

Urgent need for designs that are unlikely to harbor unknown vulnerabilities

Q&A

If the surgery proves unnecessary, we'll revert your architectural state at no charge.