Side Channels, and Other Gaps Between Theory and Practice

PAUL KOCHER

Korea Crypto Forum
Nov 15, 2019 9:00-10:30

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

All trademarks are the property of their respective owners. This presentation is provided without any guarantee or warranty whatsoever.

Intro

Researcher & entrepreneur

Technical work

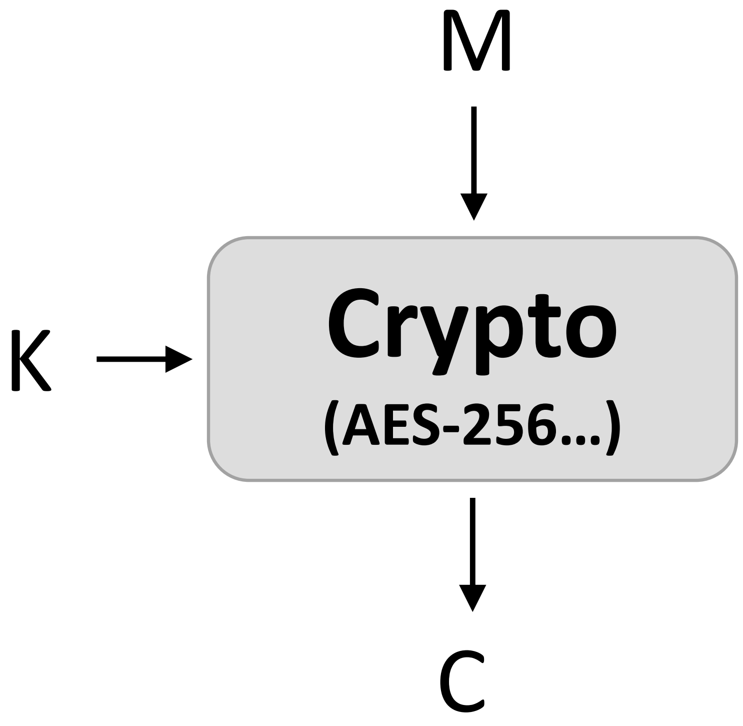
- Protocols (incl. SSL v3.0 / TLS “🔒”)
- Timing attacks/side channels
- Differential power analysis & countermeasures
- Numerous HW/ASIC projects
  - Pay TV, anti-counterfeiting, keysearch
- Renewability & forensics
  - Blu-ray BD+, Vidity/SCSA…
- CryptoManager ASIC & mfg solutions
- Spectre Attacks
  - Co-discovery w/Jann Horn of Google

Non-technical

- Founded and led Cryptography Research (1995-April 2017)
  - Bootstrapped – no external funding
  - Multiple stage evolution:
    Consulting → Licensing → Products → → Solutions
  - Acquired by Rambus ($342.6M)
- Co-founded ValiCert (IPO 2001, acquired 2003)
- Advisor+investor to security start-ups
- Favorite nonprofits include Doctors Without Borders (MSF), Muttville, IACR, Bay Area Discovery Museum

Historical cipher machines: Algorithm limited by mechanical constraints
Security limited by algorithm complexity

Modern age: Algorithm designers have won



More compute favors the cryptographer over the cryptanalyst

- 2-8X CPU power = 2X key length = **square** the effort for cryptanalysis
- Example: DES to triple DES = 3X CPU, keysearch from ~2^{56} -> ~2^{112}

Modern crypto algorithms generally have LARGE safety margins:

Survive very conservative scenarios

e.g., exabytes of adaptively-chosen plaintexts...

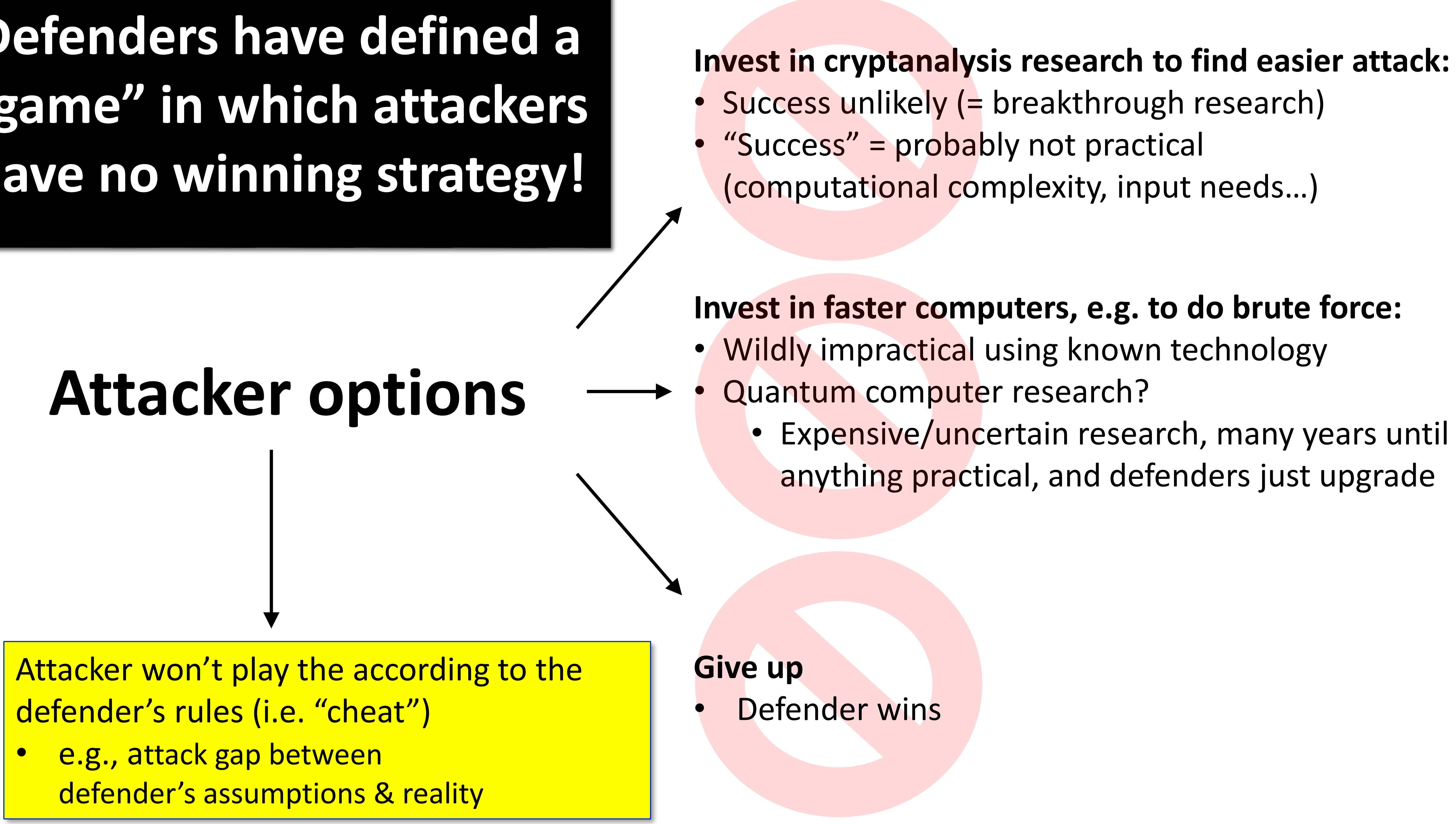
Survive huge adversary compute power

e.g., attacker with more than world's compute power

Global bitcoin mining ~2^{91} hashes/year vs AES brute force ~2^{255}

$2^{255}/2^{91}$ = 23,384,026,197,294,446,691,258,957,323,460,528,314,494,920,687,616 years

Defenders have defined a “game” in which attackers have no winning strategy!



Attacker options

Invest in cryptanalysis research to find easier attack:

- Success unlikely (= breakthrough research)
- “Success” = probably not practical (computational complexity, input needs…)

Invest in faster computers, e.g. to do brute force:

- Wildly impractical using known technology
- Quantum computer research?
  - Expensive/uncertain research, many years until anything practical, and defenders just upgrade

Give up

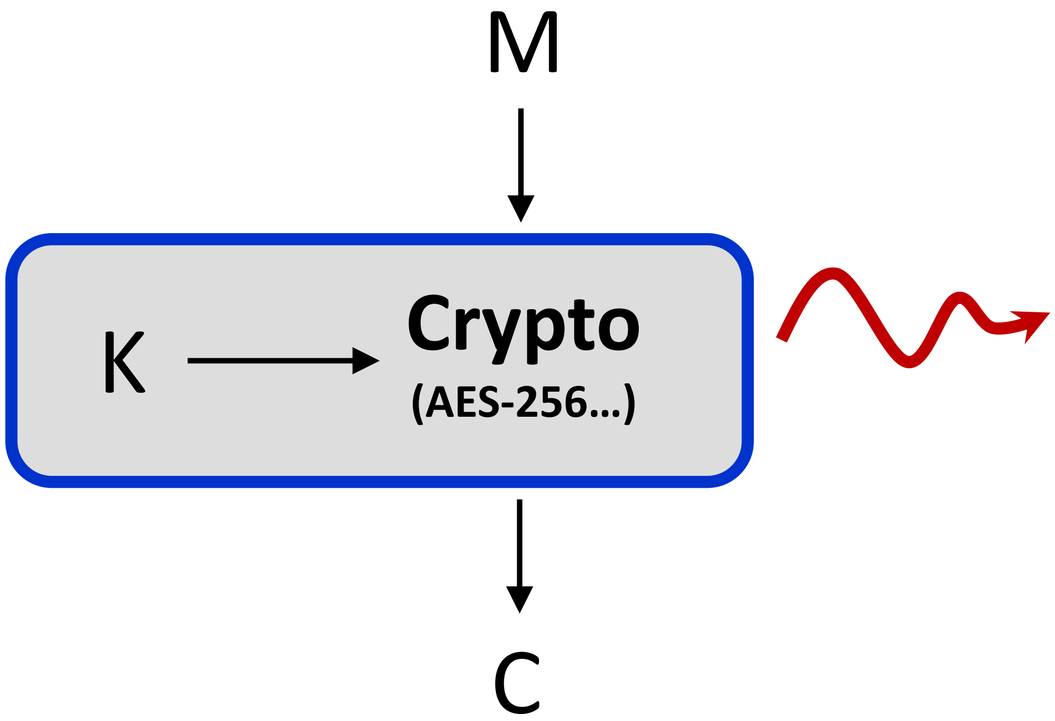
- Defender wins

Attacker won’t play the according to the defender’s rules (i.e. “cheat”)

- e.g., attack gap between defender’s assumptions & reality

Model (algorithm) vs. reality (implementation)

Algorithm assumptions:
Attacker sees many plaintexts and/or ciphertexts, but nothing else



Practice:
Implementation uses semiconductors (and maybe software), not pure math

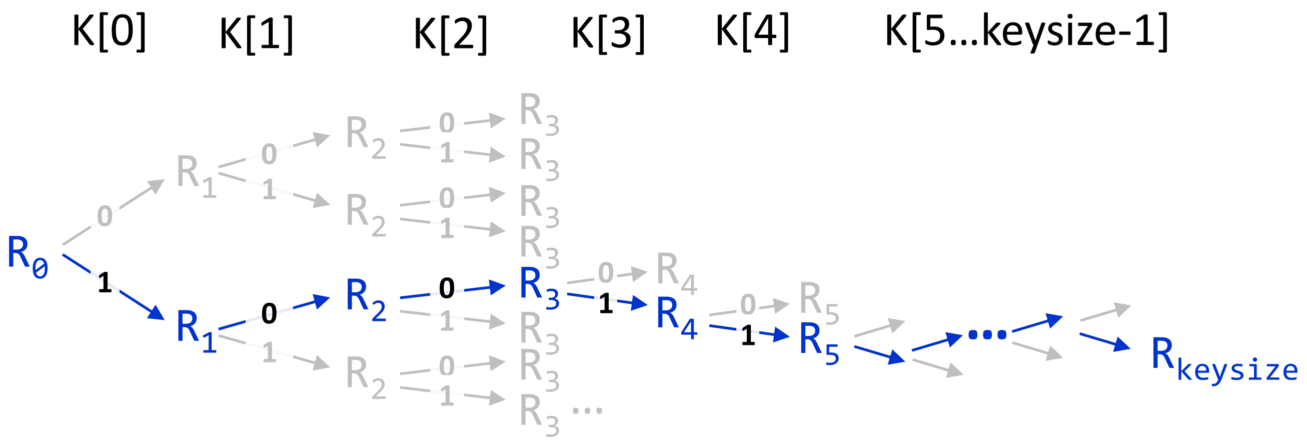
- Timing variations
- Power consumption
- RF emissions
- Heat
- Debug registers
- Microscope images of a chip
- Erroneous outputs
- Error messages

etc

Exploiting tiny gaps between assumptions & reality

Timing

```
Let $R_0$ = input message M
for $i$ in 0..keysize-1:
    Let $R_{i+1}$ = F($R_i$,K[$i$])
Output $R_{keysize}$
```



Suppose:

- The time t_i to compute **F**( ) in iteration i varies slightly depending on its inputs (R_i and bit K[i])
- Adversary can measure the total operation time (all iterations + some measurement error)

Each measured time T reflects:

$$T = \text{constant} + \text{noise} + \sum_{i=0}^{keysize-1} t_i$$

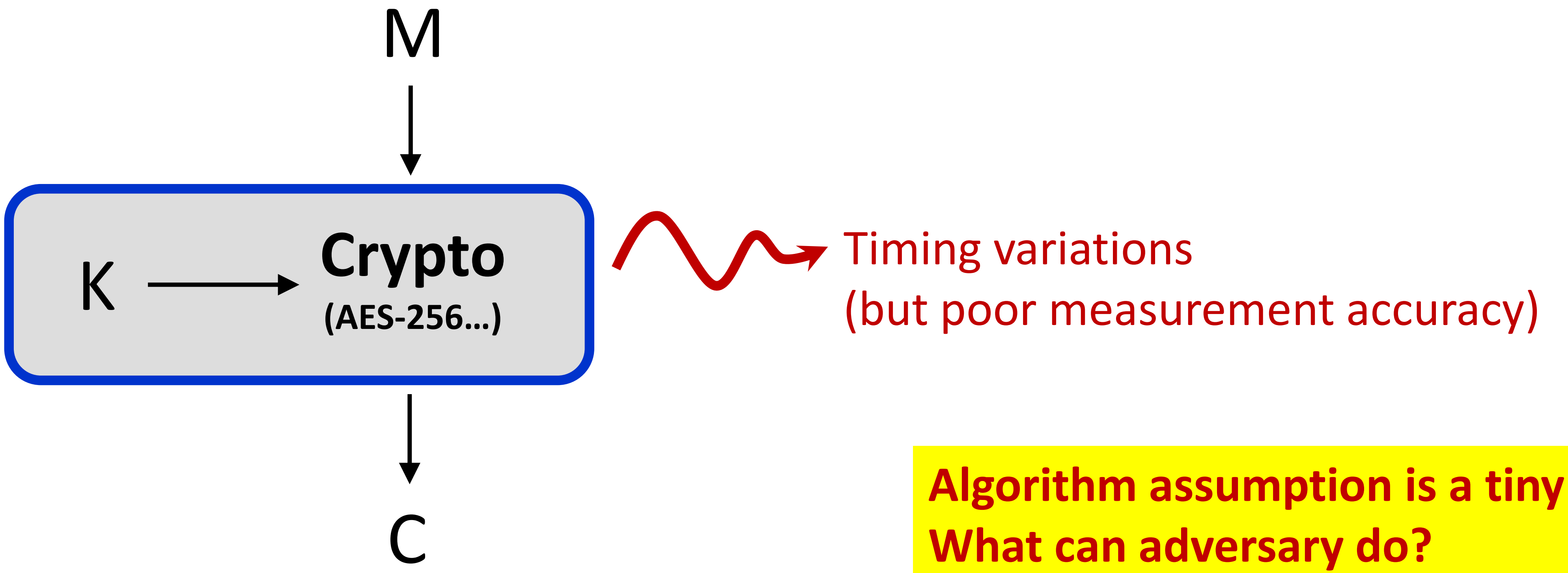
time to compute each **F**(R_i,K[i])

Loop overhead…

Measurement error, timing jitter…

Does it matter?

Algorithm assumptions:
Attacker sees many plaintexts and/or ciphertexts, but nothing else



Algorithm assumption is a tiny bit wrong.
What can adversary do?

Attack inputs: n messages ($M_{1..n}$) and their timings ($T_{1..n}$)

Attack goal: Given K[0..i-1], find the next bit K[i]

```
Let $R_0$ = input message M
for $i$ in 0..keysize-1:
    Let $R_{i+1}$ = F($R_i$,K[$i$])
Output $R_{keysize}$
```

Attacker knows K[0..i-1], so can find R_i for each M

Attacker doesn't know K[i], so doesn't know if:

R_{i+1} = **F**(R_i , 0) or
R_{i+1} = **F**(R_i , 1)

For each M, attacker can estimate how long victim would take to compute F() for each key bit value:

$t_{i,0}$ = time to compute **F**(R_i , 0) ← *if K[i]=0*
$t_{i,1}$ = time to compute **F**(R_i , 1) ← *if K[i]=1*

One of these this is a (tiny) part of the overall time T

The observed time is:

Either $t_{i,0}$ or $t_{i,1}$

$$T = \text{constant} + \text{noise} + t_0 + t_1 + t_2 + \ldots + t_i + \ldots + t_{keysize-1}$$

Observed time for a message

Correlation

Either $t_{i,0}$ or $t_{i,1}$

$$T = \text{constant} + \text{noise} + t_0 + t_1 + t_2 + \ldots + t_i + \ldots + t_{\text{keysize-1}}$$

Observed time for a message

If $\mathbf{F}(R_i, 0)$ was actually computed, T will be correlated to $t_{i,0}$ over messages $M_{1..n}$

If $\mathbf{F}(R_i, 1)$ was actually computed, T will be correlated to $t_{i,1}$ over messages $M_{1..n}$

Simple attack to find key bit i:

- Predict $t_{i,0}$ and $t_{i,1}$ for each $M_{1..n}$
- For M where $t_{i,0}$ is faster than average, how many also have T faster than average? Votes for $K[i] = 0$
- For M where $t_{i,1}$ is faster than average, how many also have T faster than average? Votes for $K[i] = 1$

Given enough messages, the actual value for $K[i]$ will get more votes

Overcoming noise

How many timing measurements do we need?

- σ_1 = noise = standard deviation among T (measurement error, unknown t_i)
- σ_2 = signal = average difference between $t_{i,0}$ and $t_{i,1}$
- Need $N \propto \left(\frac{\sigma_1}{\sigma_2}\right)^2$ measurements

Example: σ_1 = 1µs variation, σ_2 = 1ms, N ≈ 10^6

Noise 1000X signal

- One set of measurements only (reuse for all key bits)
- Error detection: if a bit is guessed wrong, subsequent correlations go away (= back up + fix)

More noise = attack still works (but need more measurements)

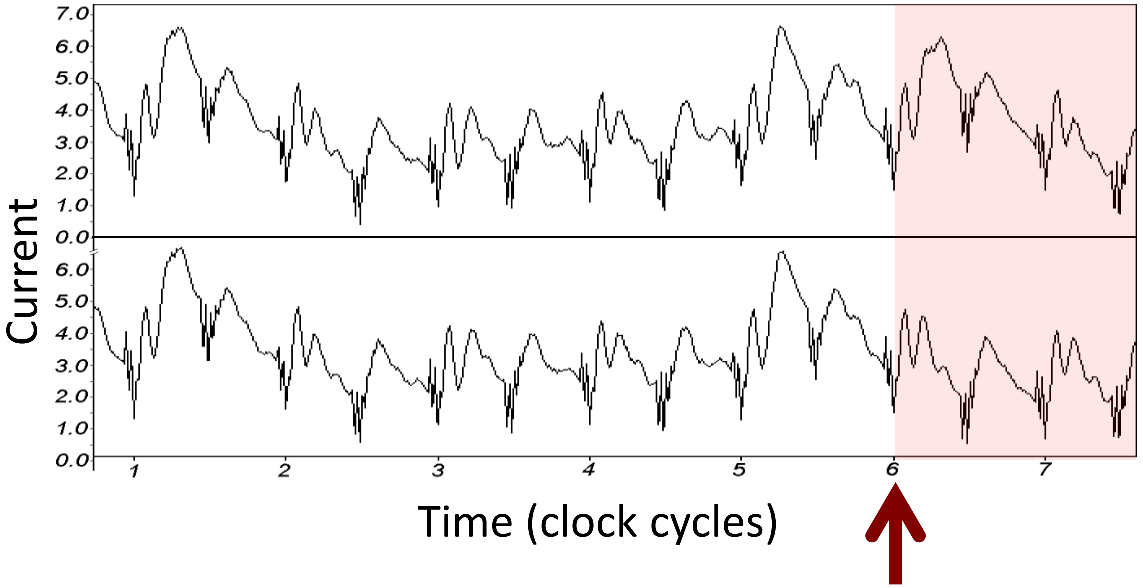
Generally not sufficient to add modest random delays or limits on timer resolution

Simple Power Analysis

Attackers can measure more than transaction time

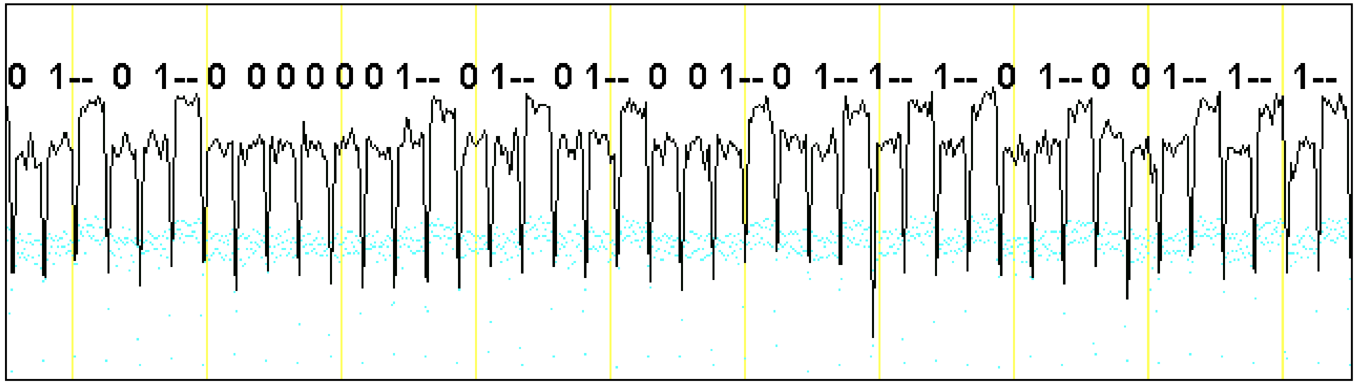
Example: Power consumption or RF

- Easy to measure if physically near device
- High bandwidth (many measurements during computation)



Simple case: power consumption is visibly different for `K[𝑖]=0` **vs** `K[𝑖]=1`

```
for $i$ in 0..keysize-1:
    Let $R_{i+1}$ = $\mathbf{F}(R_i,K[i])$
```



Signal-to-noise ratios for power & RF can also be poor

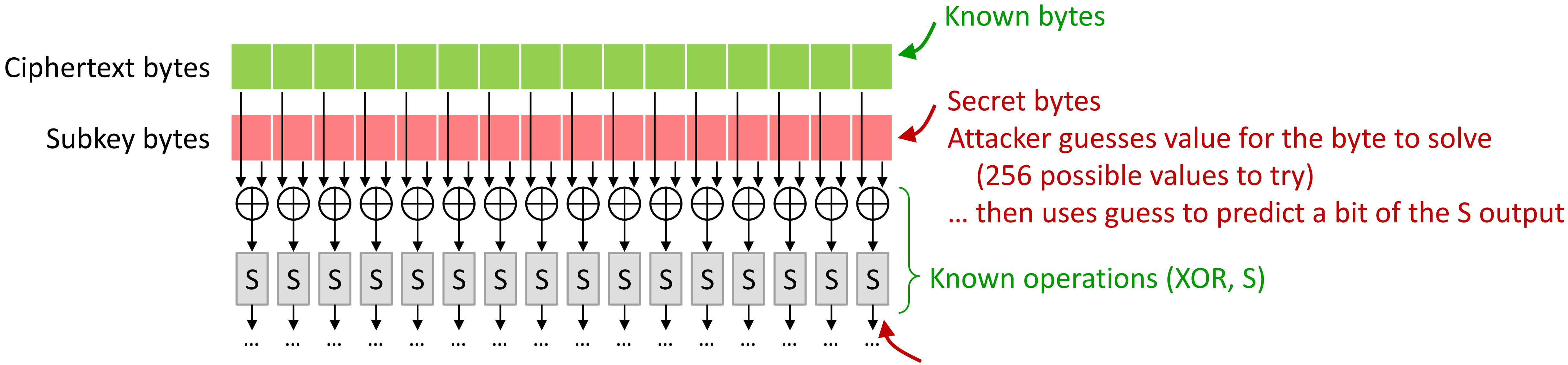
- Signals of interest may just be a few logic gates/wires in a large chip

Can these noisy power measurements be leveraged using statistics?

Overcoming noise: AES decryption

Attack scenario: AES decryption, where adversary can observe ciphertexts & power measurements

In first decryption round of AES:



Known bytes

Secret bytes

Attacker guesses value for the byte to solve

(256 possible values to try)

… then uses guess to predict a bit of the S output

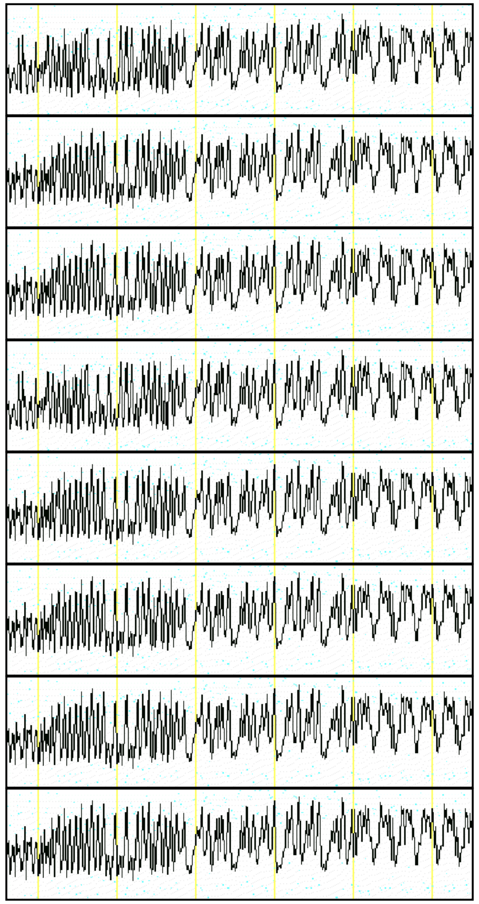
Correct key byte guess = predicted bit correct for 100% of messages

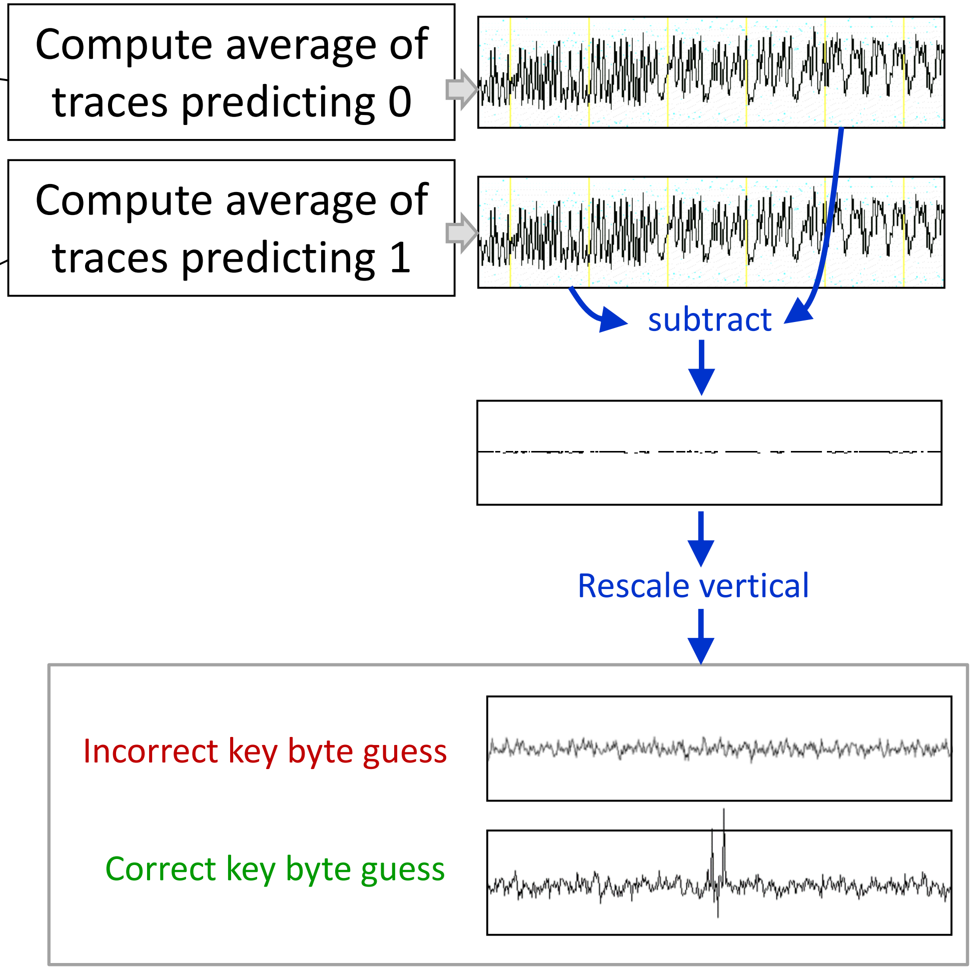
Wrong key byte guess = predicted bit correct 50%, wrong 50%

Overcoming noise

Input message (ciphertext)	Predicted S output bit
399632A62CA005E77710E3B947B1B65F	0
C06854D3EA68B82B8BFAADF73154ED17	1
01CC39A29605DD66A881D5128AC7FFA5	1
8FFC2EF2E831EF098514CA537F95EFA0	0
F02C294367605EB6ED393275C280587F	1
EEEE63394CE9DF029C52CCC9D69D8834	0
621E0462C66D4609AC0318A0284E1B7F	0
2FCA8B635D3DABBD8FEFC2AD7694D456	1

Power trace





Compute average of traces predicting 0

Compute average of traces predicting 1

subtract

Rescale vertical

Incorrect key byte guess

Correct key byte guess

DPA: Overcoming noise

Input message (ciphertext) | Power trace | Predicted S output bit

39

C0

01

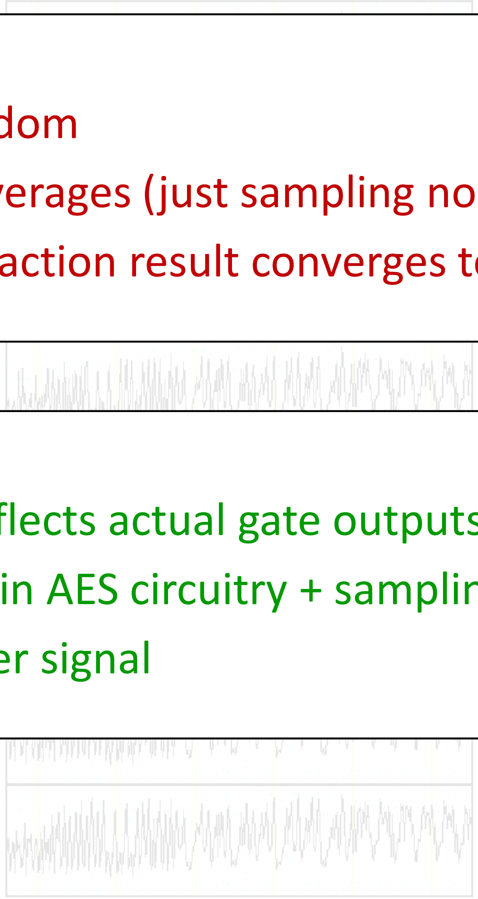
8FFC2EF2E831EF098514CA537F95EFA0 0

F0

EE

621E0462C66D4609AC0318A0284E1B7F

2FCA8B635D3DABBD8FEFC2AD7694D456 1

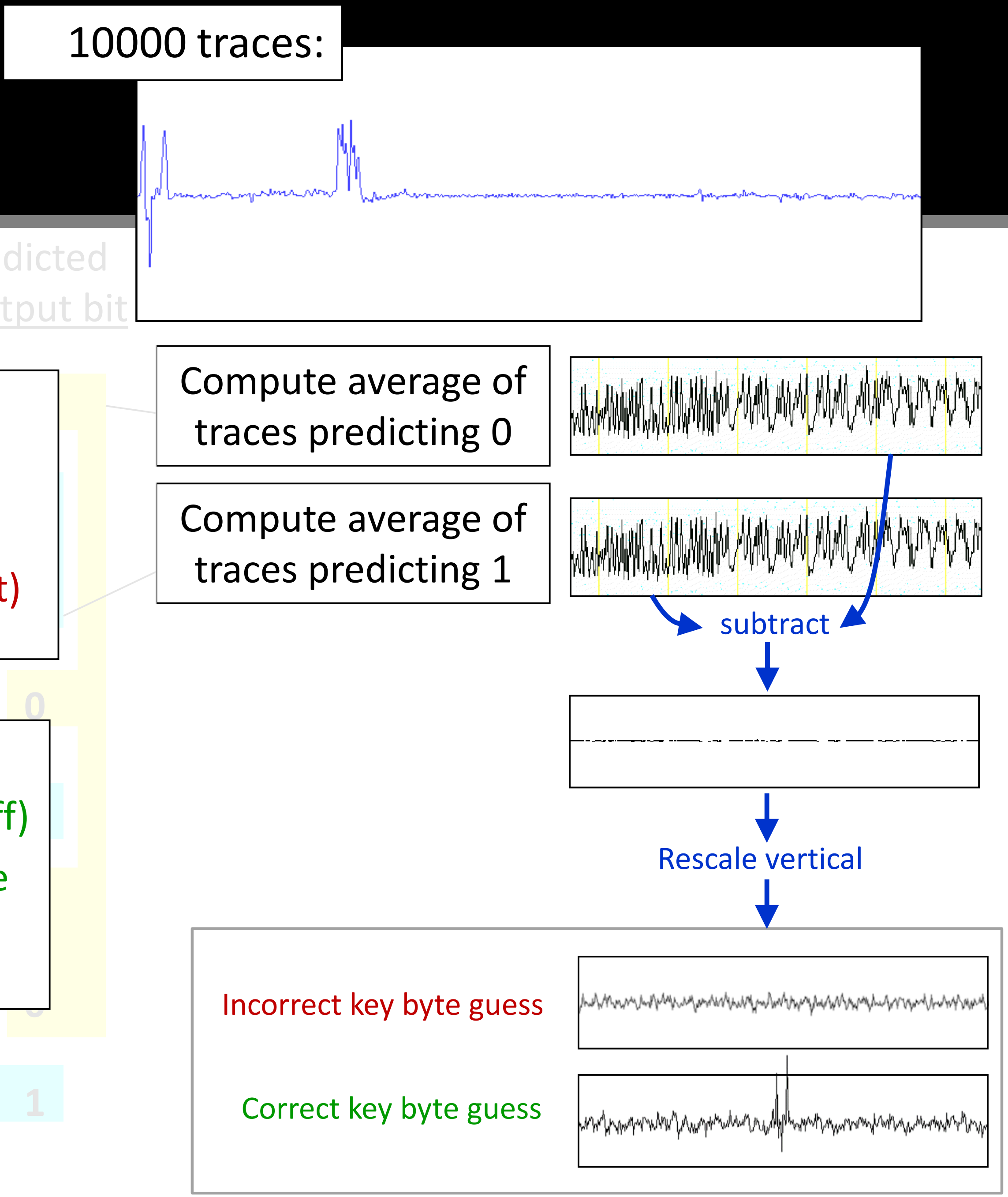


Incorrect guess:

- Predictions are essentially random
- No significant differences in averages (just sampling noise)
- More messages/traces = subtraction result converges to 0 (flat)

Correct guess for key byte:

- Assignment of power traces reflects actual gate outputs (on/off)
- Difference in averages = effect in AES circuitry + sampling noise
- More inputs = less noise, clearer signal



Sharing hardware

Power & timing attacks assumed an external adversary

But what if attackers can run on the victim's computer…?

- Cloud
- Mobile
- Browser

JavaScript

WebAssembly

amazon web services™

Google Cloud

Microsoft Azure

Available on the App Store

GET IT ON Google Play

Transmitters & receivers

Shared resources => channels

- Program A uses resource -> program B observes use of resource

Transmitter: Victim code leaks sensitive data into channel

- Natural result of using system resources

Receiver: Adversary code reads from the channel

- Analyzes measurements to infer victim secret

Examples of side channel transmitters & receivers

Channel	Transmit: Secret affects victim's…	Receive: Attacker measures…
CPU utilization	CPU usage	how much time attacker gets
DRAM usage	DRAM allocations	free memory
DRAM bandwidth	DRAM access rate	memory performance
cache state	memory accesses pattern	cache additions/evictions (see next slides)
GPU time	use of GPU	GPU availability
thermal	heat generation	effects of temperature on CPU speed
swap space	size of (compressed?) swap	disk/swap usage or free space
RNG seed pool	timing/amount of random data us	RNG latency (e.g. rdrand instruction)
disk utilization	timing/amount of file read/writes	disk/file system performance
signal coupling	crosstalk to analog circuitry	LSBs from A/D converters, oscillators…

… many others: acoustic effects, LED brightness, debug/performance register state…

Microprocessor performance

CPU market driven has been by speed

$$\text{Time} = (\text{clock rate}) \times (\#\ \text{clock cycles})$$

Rates grew fast to ~4 GHz… then plateaued:

- 1990: 80486 at 33MHz
- 2004: Pentium 4 at 3.8 GHz
- 2018: i7-8086K ‘up to’ 5.0 GHz

+40.36% / year

+1.98% / year

Performance gains => reducing # clocks

- Getting more done per clock

Moore’s Law allows huge complexity

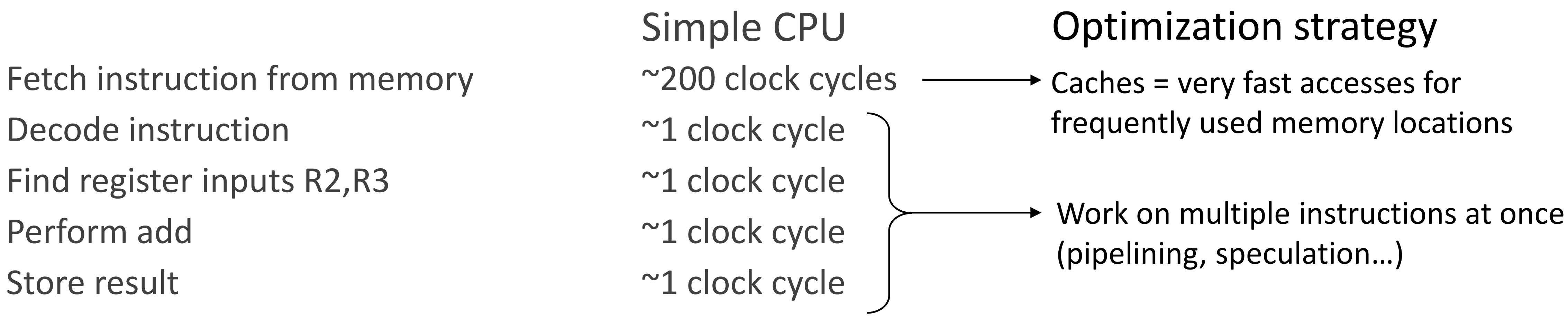
- Squeeze gains from cumulative effect of many complex optimizations

Getting more done per clock

Program = series of instructions

- Arithmetic/logic operations (add, multiply, divide…)
- Memory operations (load, store)
- Flow control (jump = decide which instruction to do next)

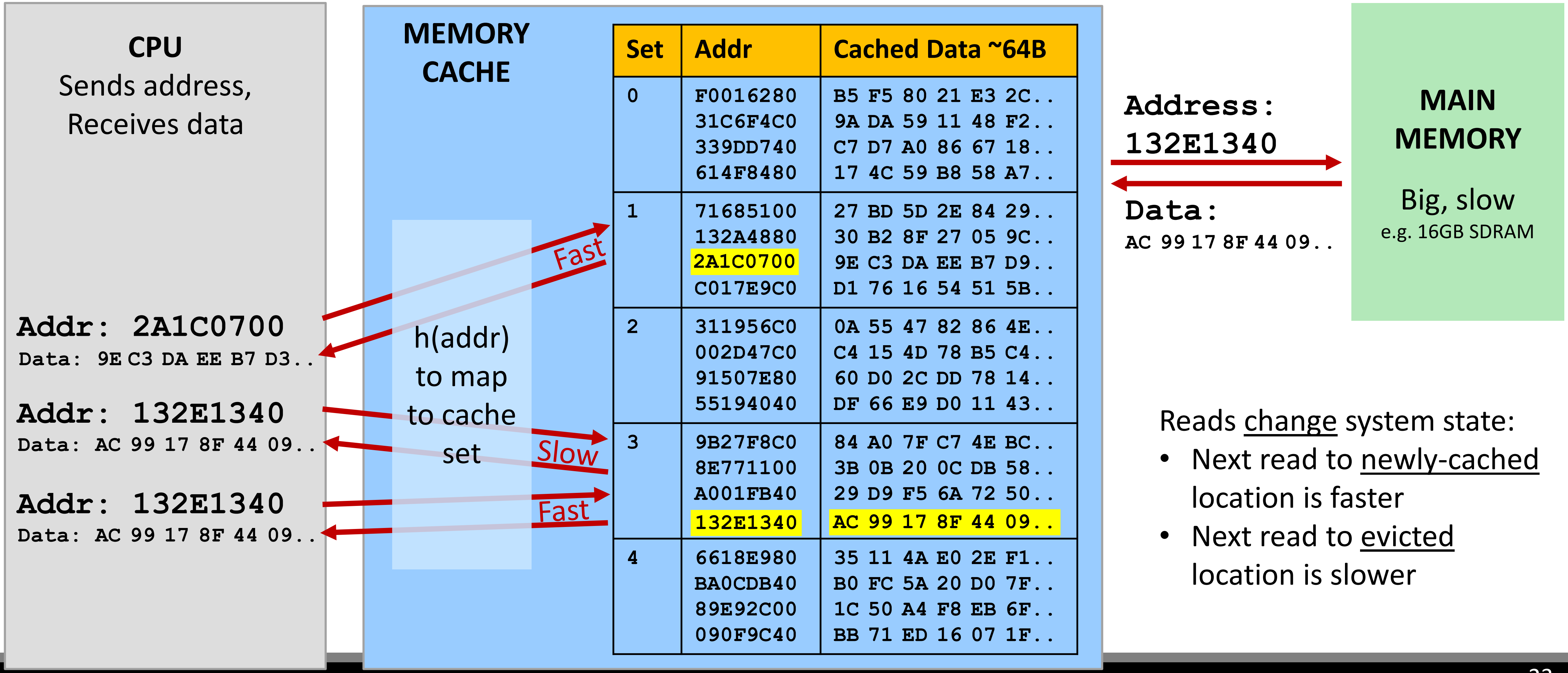
Example operation “R1 = R2 + R3”



	Simple CPU	Optimization strategy
Fetch instruction from memory	~200 clock cycles	Caches = very fast accesses for frequently used memory locations
Decode instruction	~1 clock cycle	Work on multiple instructions at once (pipelining, speculation…)
Find register inputs R2,R3	~1 clock cycle	
Perform add	~1 clock cycle	
Store result	~1 clock cycle	

Memory caches

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



CPU
Sends address,
Receives data

Addr: 2A1C0700
Data: 9E C3 DA EE B7 D3..

Addr: 132E1340
Data: AC 99 17 8F 44 09..

Addr: 132E1340
Data: AC 99 17 8F 44 09..

MEMORY CACHE

Set	Addr	Cached Data ~64B
0	F0016280	B5 F5 80 21 E3 2C..
	31C6F4C0	9A DA 59 11 48 F2..
	339DD740	C7 D7 A0 86 67 18..
	614F8480	17 4C 59 B8 58 A7..
1	71685100	27 BD 5D 2E 84 29..
	132A4880	30 B2 8F 27 05 9C..
	2A1C0700	9E C3 DA EE B7 D9..
	C017E9C0	D1 76 16 54 51 5B..
2	311956C0	0A 55 47 82 86 4E..
	002D47C0	C4 15 4D 78 B5 C4..
	91507E80	60 D0 2C DD 78 14..
	55194040	DF 66 E9 D0 11 43..
3	9B27F8C0	84 A0 7F C7 4E BC..
	8E771100	3B 0B 20 0C DB 58..
	A001FB40	29 D9 F5 6A 72 50..
	132E1340	AC 99 17 8F 44 09..
4	6618E980	35 11 4A E0 2E F1..
	BA0CDB40	B0 FC 5A 20 D0 7F..
	89E92C00	1C 50 A4 F8 EB 6F..
	090F9C40	BB 71 ED 16 07 1F..

Address:
132E1340

Data:
AC 99 17 8F 44 09..

MAIN MEMORY

Big, slow
e.g. 16GB SDRAM

Reads change system state:

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

Cache attacks

Basic cache attack:

- Step 1: Prepare cache (erase or fill lines known data)
- Step 2: Run victim code
- Step 3: Detect changes in cache state (additions or removals)
- Step 4: Infer victim's secret data

Common variants include:

FLUSH+RELOAD

- Attacker flushes an address it shares with the victim (e.g. `clflush` instruction on an address in a shared library)
- Victim runs
- Attacker measure time to read from address (fast=victim accessed an address in the cache line)

PRIME+PROBE

- Attacker loads cache with attacker data, e.g. by reading data in attacker address space
- Victim runs
- Attacker reads back its data (slow=attacker data got evicted, e.g. because victim read data from an address mapping to same cache set)

EVICT+TIME

- See if evicting a location changes how long victim takes to run (e.g., run victim repeatedly, optionally evicting an address in between)

+ others (e.g., EVICT+RELOAD, FLUSH+FLUSH…)

Implications of shared caches:

- Adversary can infer the pattern of memory addresses accessed by adversary
- Adversary can infer the control flow, i.e. the destination of branch/jump instructions

Various responses/mitigations:

- Write code so secrets do not affect control flow & memory accesses [used for crypto libraries]
- Disable SMT (aka Hyperthreading)
- Page coloring, cache partitioning (e.g., Intel Cache Allocation Technology (CAT))
  … but implementations have limits + some workarounds (e.g., see CATalyst) + other side channels
- Don’t multitask attacker code on computers with sensitive data

Fault attacks

RSA Signing: $S = M^d \bmod n$

RSA is normally optimized using Chinese Remainder Theorem (faster)

- $S_p = (M \bmod p)^{d_p} \bmod p$
- $S_q = (M \bmod q)^{d_q} \bmod q$
- $S = S_q + q(k(S_p - S_q)M \bmod p)$

M is padded message hash
d is secret exponent
n is public key (= p·q, where p and q are secret)

What if the CPU makes a mistake?

- Computation of S_p or S_q use most of the computation time, so are the most likely places for an error
- Example: Defective signature S’ with erroneous S_p but correct S_q
  - Result: S’ is wrong by a multiple of q
  - Attacker computes gcd(M – S'^e, n), which reveals q, which factors N

Fault attacks, continued

Not just RSA -- applicable to any crypto algorithm

- AES, 3DES, ECDSA…

Or bypass crypto altogether

- Example: Corrupt length entering output API

Various analog tricks to cause computation errors

- Pay TV glitchers: Send timed clock/ground glitches to smart card
- CLKSCREW: Misconfigure frequency/voltage management circuitry to induce errors
- Rowhammer: Corrupt data in DRAM via malicious memory access patterns
- Xbox 360 reset glitch attack
- Etc

Nobody would <u>intentionally</u> design a CPU that made errors… right?

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
- Before uncached value known, save old registers/state and start executing compute_something() speculatively
- When memory read finishes, if() can be evaluated definitively:
  - Guess correct: Save speculative work – performance gain
  - Guess incorrect: Discard speculative work

The CPU is making erroneous computations – effectively implementing a fault attack on the software

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 <u>without</u> speculation is safe

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

What about with speculative execution?

Conditional branch (Variant 1) attack

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

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 at `array1` base address:

8 bytes of data (value doesn't matter)

[… lots of memory up to `array1` base+N…]

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 (where 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)

Memory & Cache Status

array1_size = 00000008

Memory at array1 base address:

8 bytes of data (value doesn't matter)

[… lots of memory up to array1 base+N…]

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 (where 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 at array1 base address:
    8 bytes of data (value doesn't matter)
    [... lots of memory up to array1 base+N...]
    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

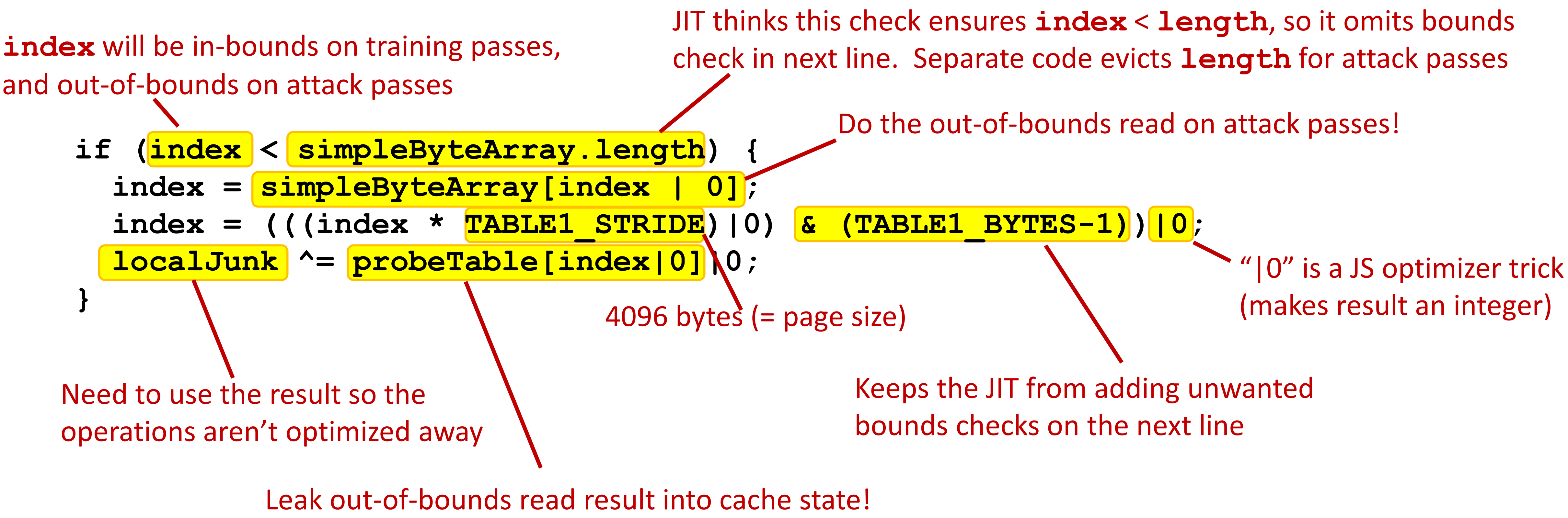
Browser runs JavaScript from untrusted websites

- 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



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

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

JIT thinks this check ensures **`index`** < **`length`**, so it omits bounds check in next line. Separate code evicts **`length`** for attack passes

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

"|0" is a JS optimizer trick (makes result an integer)

4096 bytes (= page size)

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

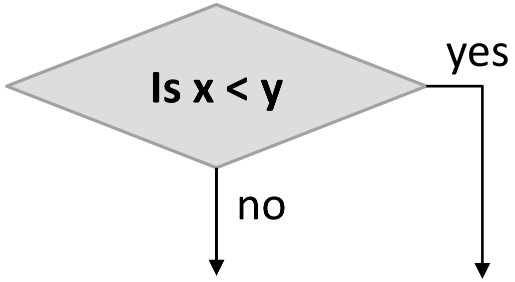
Need to use the result so the operations aren't optimized away

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

Indirect branches

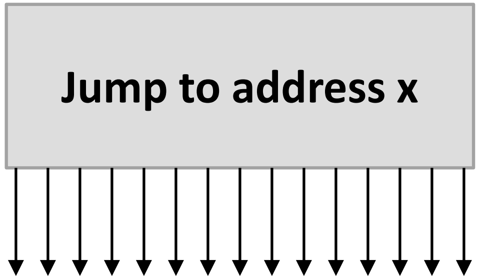
Conditional branches: Fork in the road

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



Indirect branches: Teleport 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

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”)

- Instructions in victim code/libraries usable to leak victim’s memory 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

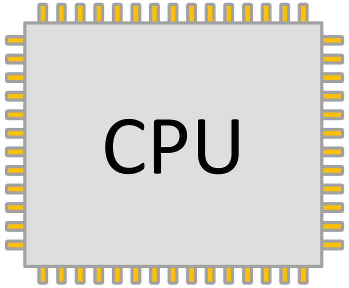
Mitigations

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

Not necessarily a complete solution

PANSEMENTS ADHÉSIFS DE MARQUE
BAND-AID®
BRAND ADHESIVE BANDAGES
Johnson & Johnson
STERILE
Do not use if open or damaged.
N'utilisez pas si ouvert ou endommagé.
Caution: The Packaging of This Product Contains Natural Rubber Latex Which May Cause Allergic Reactions.
Mise en garde: L'emballage de ce produit contient du latex d'élastomère-caoutchouc pouvant provoquer des réactions allergiques.

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



Variant 1 Mitigation: Speculation-barrier instruction (e.g. `LFENCE`)

- Idea: Software developers insert barrier on all vulnerable code paths in software
- Simple & effective (for CPU developer)

CPU architecture
Machine language
Compiler
Higher-level language

Abstraction boundaries

Software

- operating systems
- drivers
- web servers
- interpreters/JITs
- databases

:

Insert LFENCEs manually?

Often millions of control flow paths

Too confusing - speculation runs 188++ instructions, crosses modules

Too risky – miss one and attacker can read entire process memory

Put LFENCES everywhere?

Abysmal performance - LFENCE is <u>very</u> slow (+ no tools)

Not in binary libraries, compiler-created code patterns

Insert by smart compiler?

Protect all potentially-exploitable patterns = too slow

- Compilers judged by performance, not security

Protect only known-bad bad patterns = unsafe

- Microsoft Visual C/C++ `/Qspectre` unsafe for 13 of 15 tests

https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

Untestable: Effects are intentionally hidden by CPU + can differ in new CPUs

- Transfers blame to software developer ("you should have put an `LFENCE` there")

App image: CC0 Creative Commons, Free for commercial use , https://pixabay.com/en/app-software-contour-settings-1013616/

Mitigations: Indirect branch variant

> *"All of this is pure garbage"*
> *-- Linux Torvolds*
> *https://lkml.org/lkml/2018/1/21/192*

x86:

- New MSRs created via microcode
  - Low-level control over branch target buffer
  - Intel + AMD: Microcode updates for many (but not all) CPUs: messy (stability problems + recalls…)
  - Available from kernel mode only (use by applications unclear)
  - Performance impact
- Retpoline proposal from Google
  - Messy hack -- replace indirect jumps with construction that resists indirect branch poisoning on Haswell
  - Microcode updates to make retpoline safe on Skylake & beyond
- Really hard to test

ARM: No generic mitigation option

- Fast ARM CPUs broadly impacted, e.g. Cortex-A9, A15, A17, A57, A72, A73, A75...
- Often no mitigation, but on some chips software may be able to invalidate/disable branch predictor (with "non-trivial performance impact")
  - See: https://developer.arm.com/support/security-update/download-the-whitepaper

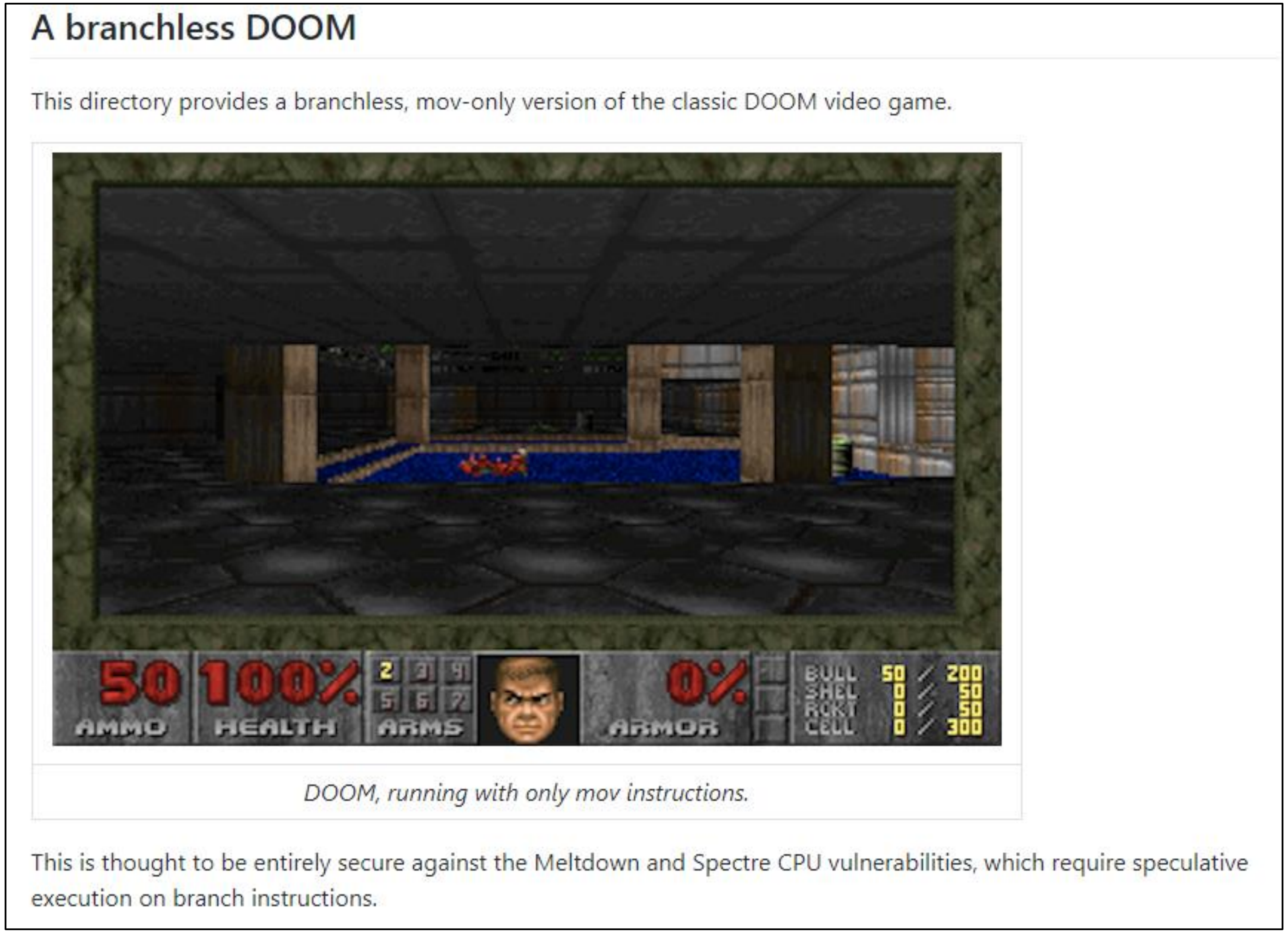
DOOM with only MOV instructions

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.

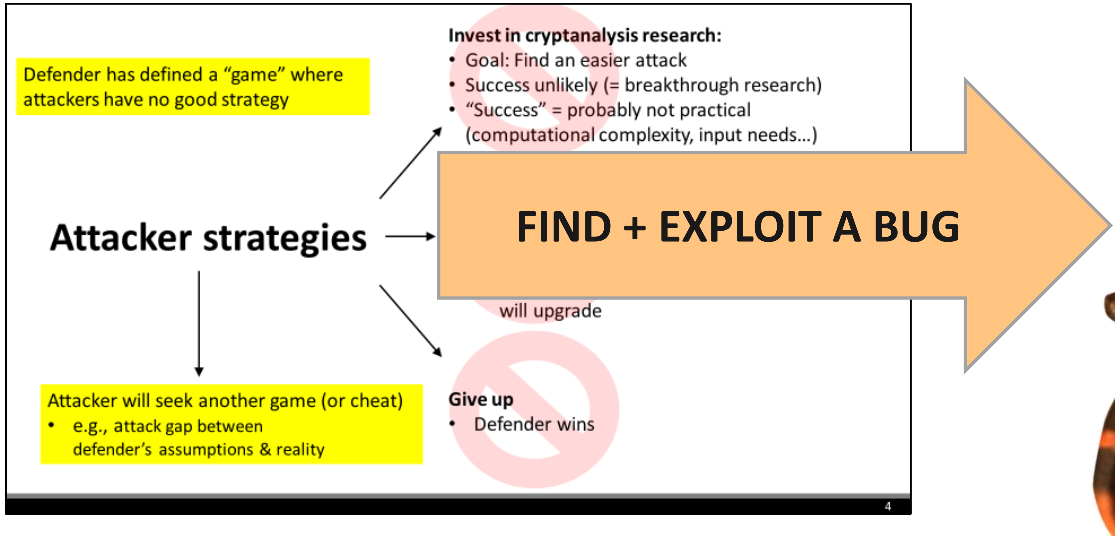
This is thought to be entirely secure against the Meltdown and Spectre CPU vulnerabilities, which require speculative execution on branch instructions.

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

Implications + Looking to the Future

Risk in context

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



Defender has defined a "game" where attackers have no good strategy

Attacker strategies

Invest in cryptanalysis research:
- Goal: Find an easier attack
- Success unlikely (= breakthrough research)
- "Success" = probably not practical (computational complexity, input needs…)

will upgrade

Give up
- Defender wins

Attacker will seek another game (or cheat)
- e.g., attack gap between defender's assumptions & reality

FIND + EXPLOIT A BUG

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

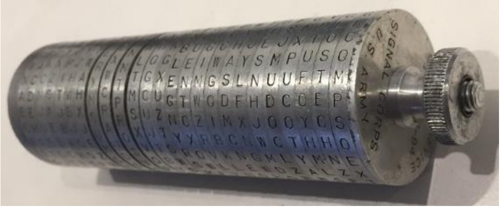
Complexity of fixes -> new risks

Psychology of unfixed vulnerabilities

Image licensed under Creative Commons Attribution 4.0 International license, author QZD, cropped from source: https://commons.wikimedia.org/wiki/File:Three_species_of_kissing_bugs.PNG

Kerckhoffs's principle (1883) :

A system should be secure even if everything about it, except the key, is public knowledge.



1. Insecure: The designer knows the security is weak
   P(secure) = 0

2. Ignorance: Attackers can almost certainly succeed using what designers don't know
   P(secure) < ϵ (updates can maintain security at designer ignorance)

…

3. Probably secure: Reasonable chance of no bugs
   50% < P(secure) < 99+%

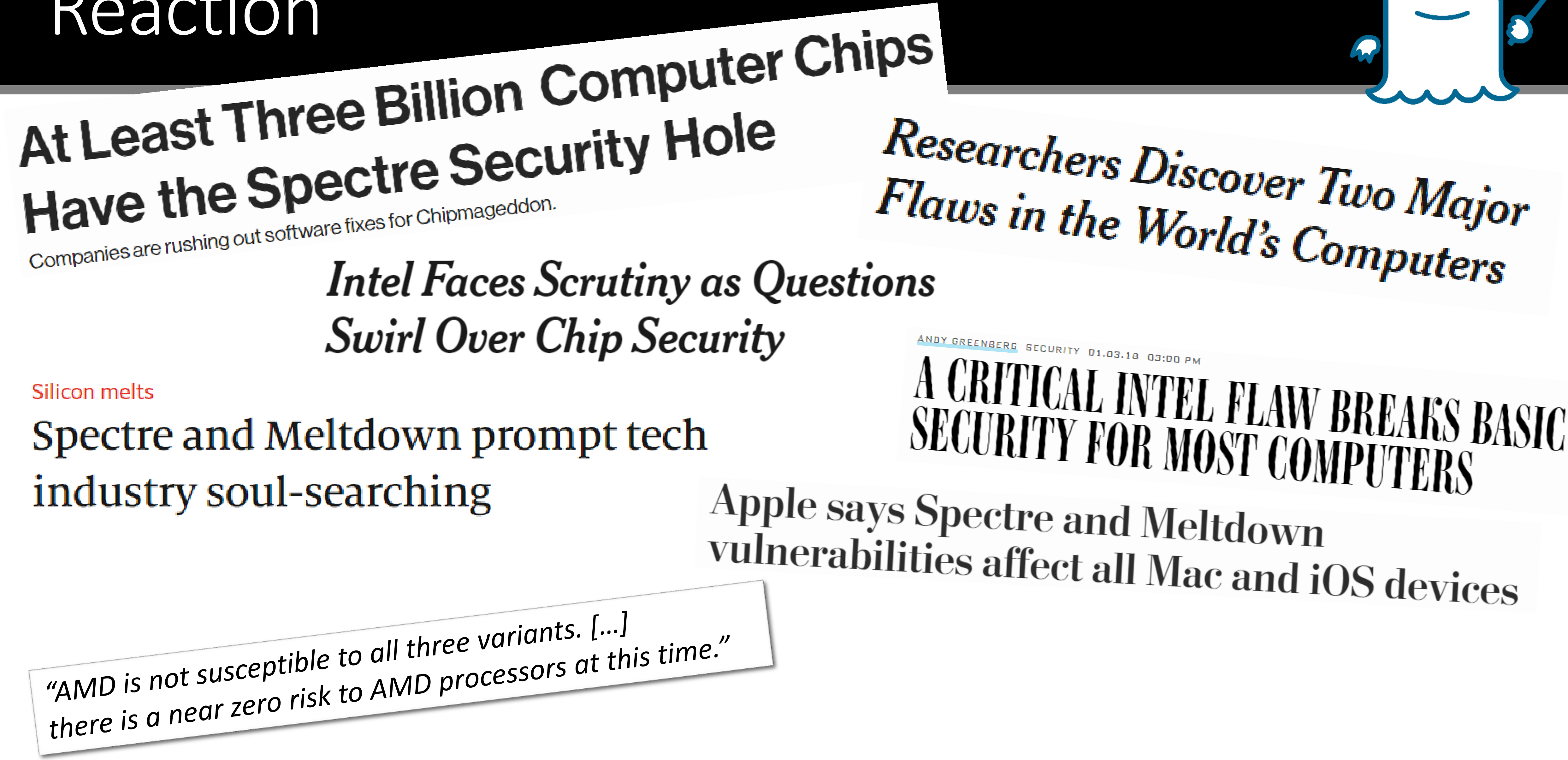
⇦ AES, seL4, TLS 1.2+(?), simple hardware designs??

If the assumptions correct… ☹

4. Reliably secure: Security properties which anyone can easily verify and are widely checked
   P(secure) = 100% - ϵ

Hopelessly difficult?

Reaction



At Least Three Billion Computer Chips Have the Spectre Security Hole

Companies are rushing out software fixes for Chipmageddon.

Researchers Discover Two Major Flaws in the World's Computers

Intel Faces Scrutiny as Questions Swirl Over Chip Security

ANDY GREENBERG SECURITY 01.03.18 03:00 PM

A CRITICAL INTEL FLAW BREAKS BASIC SECURITY FOR MOST COMPUTERS

Silicon melts

Spectre and Meltdown prompt tech industry soul-searching

Apple says Spectre and Meltdown vulnerabilities affect all Mac and iOS devices

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

??!

Why panic when a vulnerability is identified?

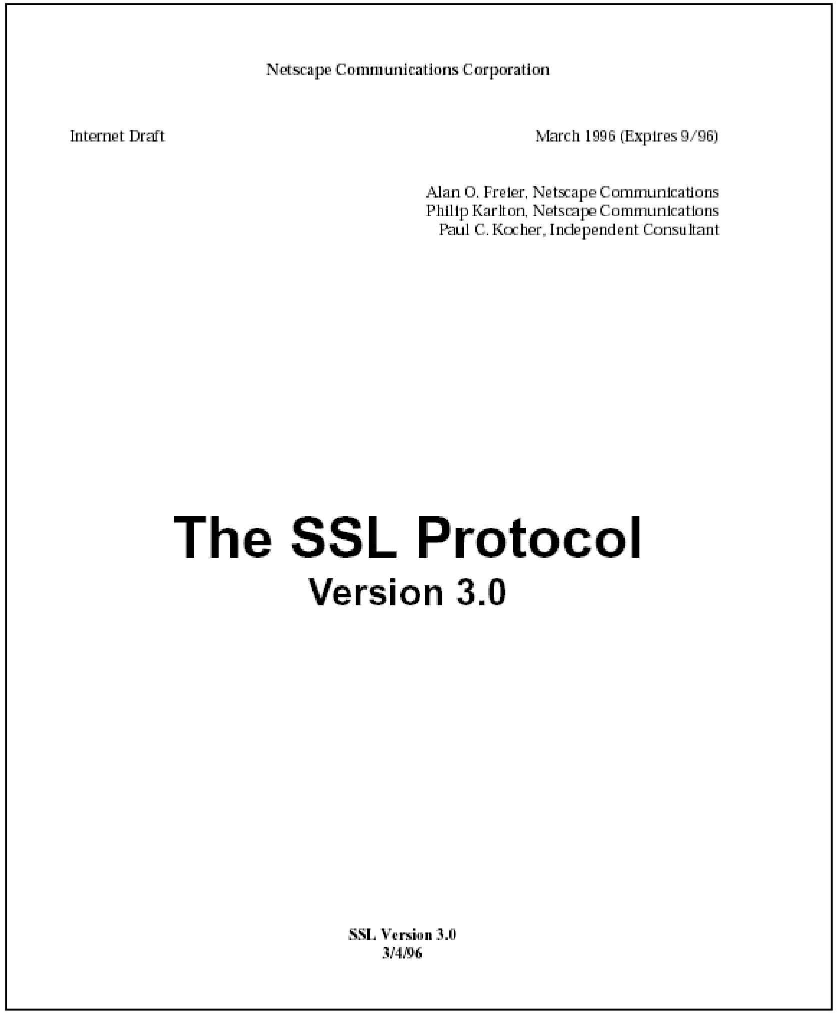
P(secure) fell from ϵ to 0.

but ϵ was usually negligible

Optimist's security = $\lceil \text{P(secure)} \rceil$

fell from 100% to 0%

Long history of over-optimism



Netscape Communications Corporation

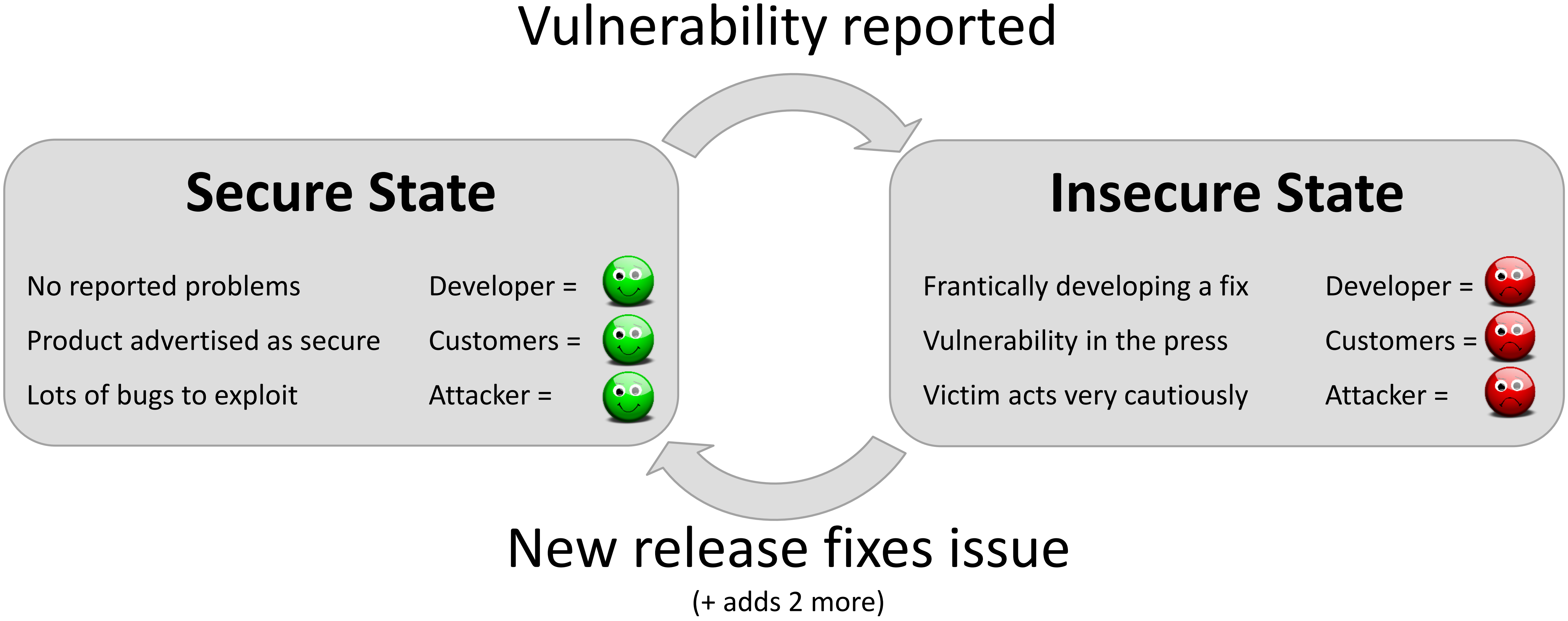
Internet Draft — March 1996 (Expires 9/96)

Alan O. Freier, Netscape Communications
Philip Karlton, Netscape Communications
Paul C. Kocher, Independent Consultant

The SSL Protocol
Version 3.0

SSL Version 3.0
3/4/96

State machine for the security cycle



Vulnerability reported

Secure State

No reported problems	Developer =	☺
Product advertised as secure	Customers =	☺
Lots of bugs to exploit	Attacker =	☺

Insecure State

Frantically developing a fix	Developer =	☹
Vulnerability in the press	Customers =	☹
Victim acts very cautiously	Attacker =	☹

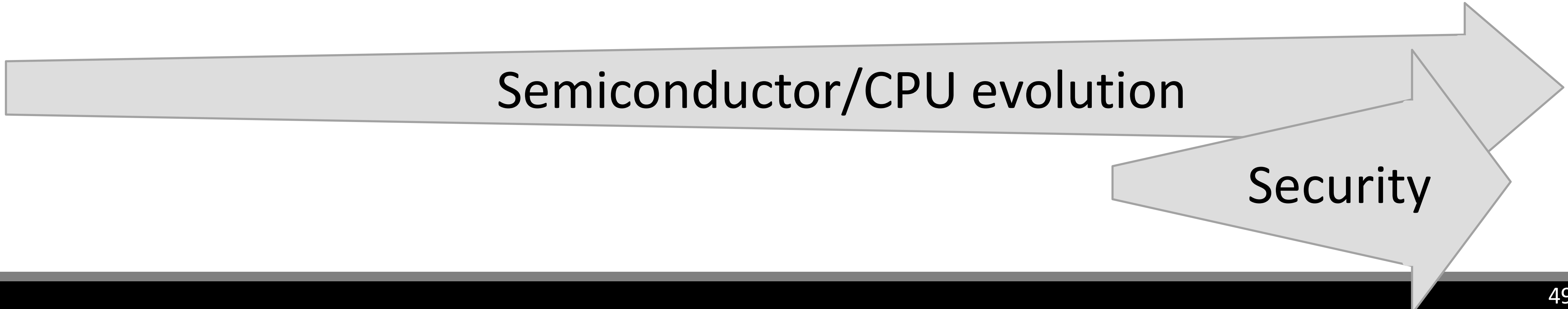
New release fixes issue

(+ adds 2 more)

A great question

> Are there any security implications from speculative execution?
>
> *-- Mike Hamburg*

at the intersection of two trends



Semiconductor/CPU evolution

Security

Why a great question?

> Are there any security implications from speculative execution?
>
> -- *Mike Hamburg*

Explores gaps…

Architecture vs. CPU

- Programmers' mental model: simple in-order CPUs
- Reality: Enormous hidden complexity of actual CPUs

Correctness assumptions

- Security efforts rely on CPUs executing code faithfully
- Reality: Actual CPUs are making then discarding errors

Expertise

- Few security experts specialize in CPU details
- Few CPU designers specialize in security

Hidden in plain sight

> *"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)*

Obvious… in hindsight:

- Component behaviors well known
  (taught in textbooks on electronics/CPU design)

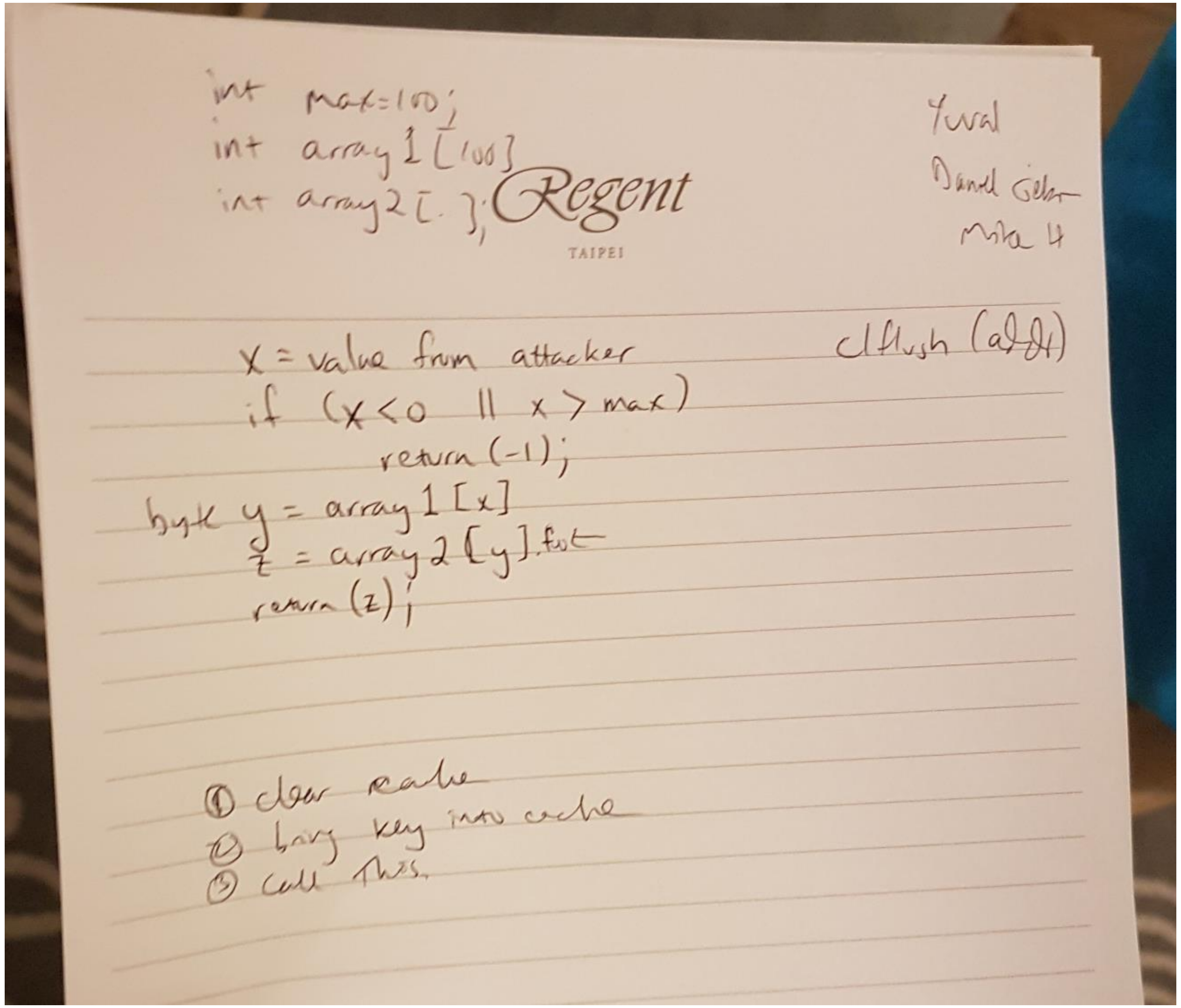
Why missed until now?

- Specialization of technical teams
  - Implications (or existence) of properties not obvious to others
- Few communications channels, different terminology, perspectives, assumptions, goals
  - Hardware designers <-> software developers
  - Software developers <-> security specialists
  - Security specialists <-> cryptographers

  } Focus on minutiae (bugs, individual blocks) rather than risks
  Few cross-disciplinary people, conferences…
- Assumption that old approaches meet current changing requirements
- Failure to explore implications of prior side channels results
- Performance-first priorities

Are there any security implications from speculative execution?

-- Mike Hamburg



```
int max=100;
int array1[100]
int array2[.];
```

Regent

TAIPEI

Yuval

Daniel Gelbr

Mike H

clflush (addr)

```
  x = value from attacker
  if (x<0 || x > max)
         return (-1);
byte y = array1[x]
     z = array2[y].fot
  return (z);
```

① clear cache
② bring key into cache
③ call This.

+ asked before an uninteresting conference talk

What are some other areas we could ask questions about?

Confidentiality (leaking data)

- Side channel attacks: Spectre variants + AI/ML analysis…
- Analog effects (voltage sensors, PLLs, RF leaks to radios…)
- Incomplete zeroization on context switch (debug regs…)

Integrity (corrupting information)

- Crosstalk effects in chips
- Clock control logic
- Misconfigure power mgmt/SW-induced glitches
- Rare failures on big distributed systems

Availability (crashing/destroying hardware)

- NVMs fail after 10K-1M writes - is write-leveling secure?
- Overheating (disable thermal shutoff + cause heating)
- Electromigration & other physical effects
- Malicious blowing of (anti-)fuses
- FPGA failures after to loading malicious bitstreams

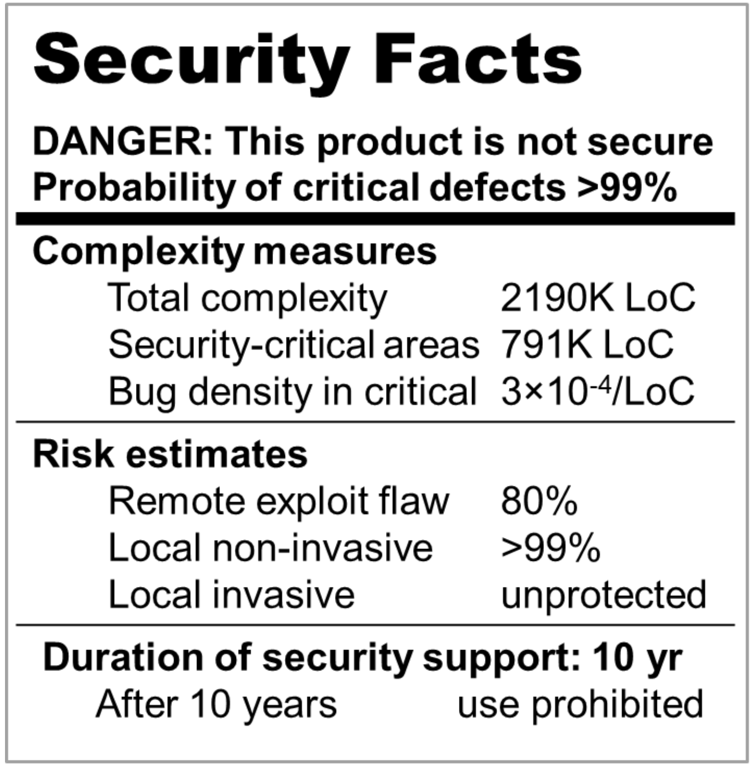
Manufacturing, infrastructure, legal

- Insecure test modes (global JTAG passwords common)
- Insecure factory key programming
- Political/legal threats (complex mandatory backdoors…)
- Attacks on firmware/microcode signing keys

Can we better quantify risk?

Many aspects: data quality, analysis cost, reproducibility, incentives:

- Vendors – want to support/fund work that highlights security (not risk)?
- Customers – lack access to needed data, uncoordinated spending, limited choices… and are optimists!
- Regulators?



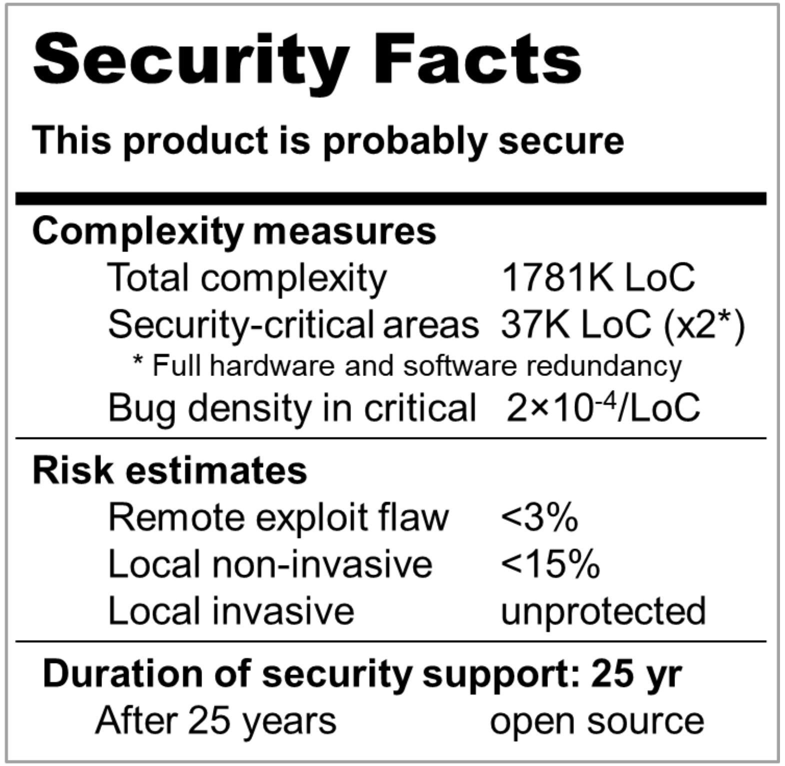
Security Facts

DANGER: This product is not secure
Probability of critical defects >99%

Complexity measures	
Total complexity	2190K LoC
Security-critical areas	791K LoC
Bug density in critical	3×10^{-4}/LoC

Risk estimates	
Remote exploit flaw	80%
Local non-invasive	>99%
Local invasive	unprotected

Duration of security support: 10 yr	
After 10 years	use prohibited



Security Facts

This product is probably secure

Complexity measures	
Total complexity	1781K LoC
Security-critical areas	37K LoC (x2*)
Bug density in critical	2×10^{-4}/LoC

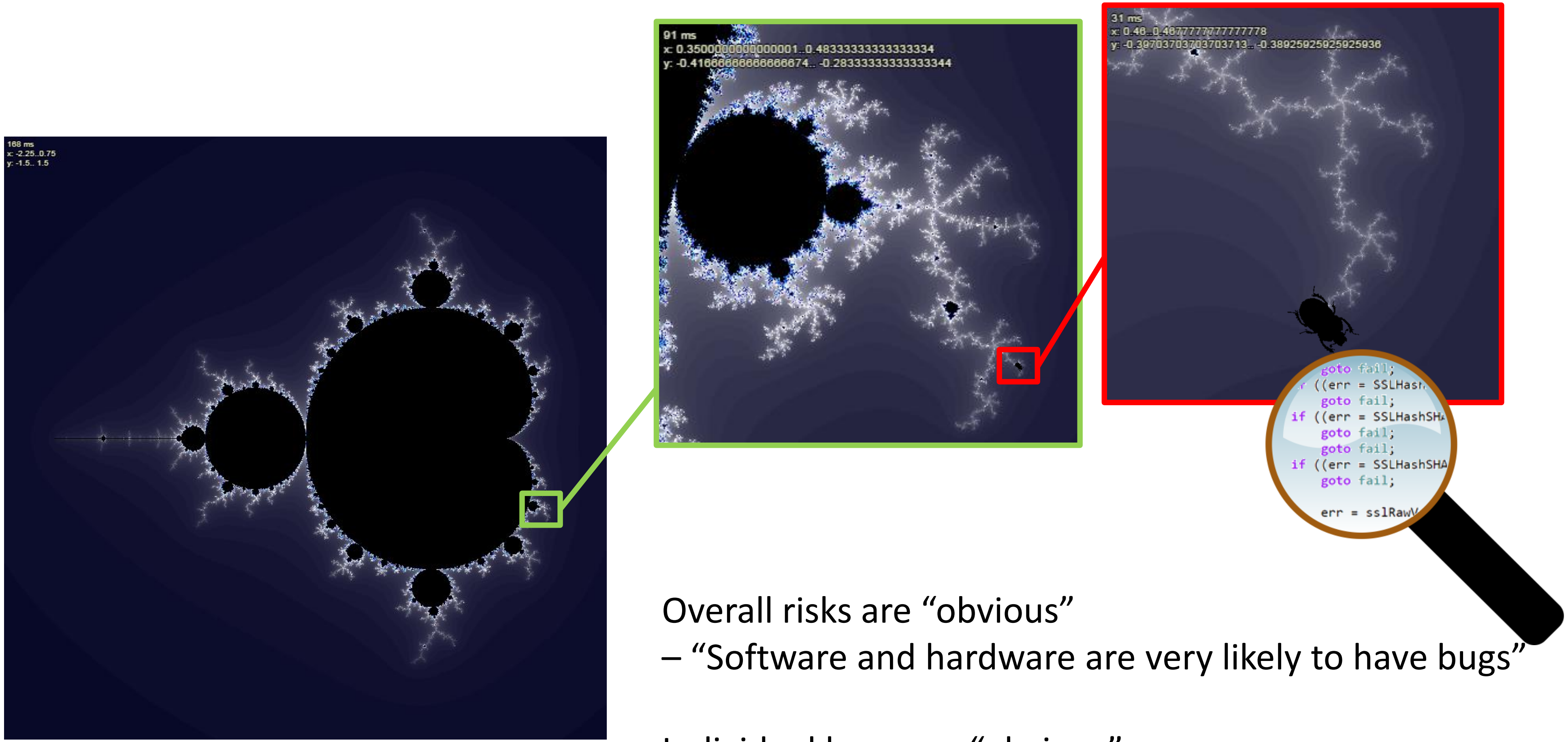
* Full hardware and software redundancy

Risk estimates	
Remote exploit flaw	<3%
Local non-invasive	<15%
Local invasive	unprotected

Duration of security support: 25 yr	
After 25 years	open source

New & Improved!
Now only a 10% chance of being completely hackable.

Problems = Opportunities



```
168 ms
x: -2.25..0.75
y: -1.5.. 1.5
```

```
91 ms
x: 0.3500000000000001..0.4833333333333334
y: -0.4166666666666674.. -0.2833333333333344
```

```
31 ms
x: 0.46..0.4677777777777778
y: -0.3970370370370371З.. -0.38925925925925936
```

```
    goto fail;
r ((err = SSLHash
    goto fail;
if ((err = SSLHashSH
    goto fail;
    goto fail;
if ((err = SSLHashSHA
    goto fail;

    err = sslRawV
```

Overall risks are “obvious”
– “Software and hardware are very likely to have bugs”

Individual bugs are “obvious”
– when we stare directly at them

What can we do to reduce the obvious risks without having to understand every logic gate & line of code?

The vulnerabilities in this talk **all** arise from critical details of a component not described by abstraction compromising the security a complex system.

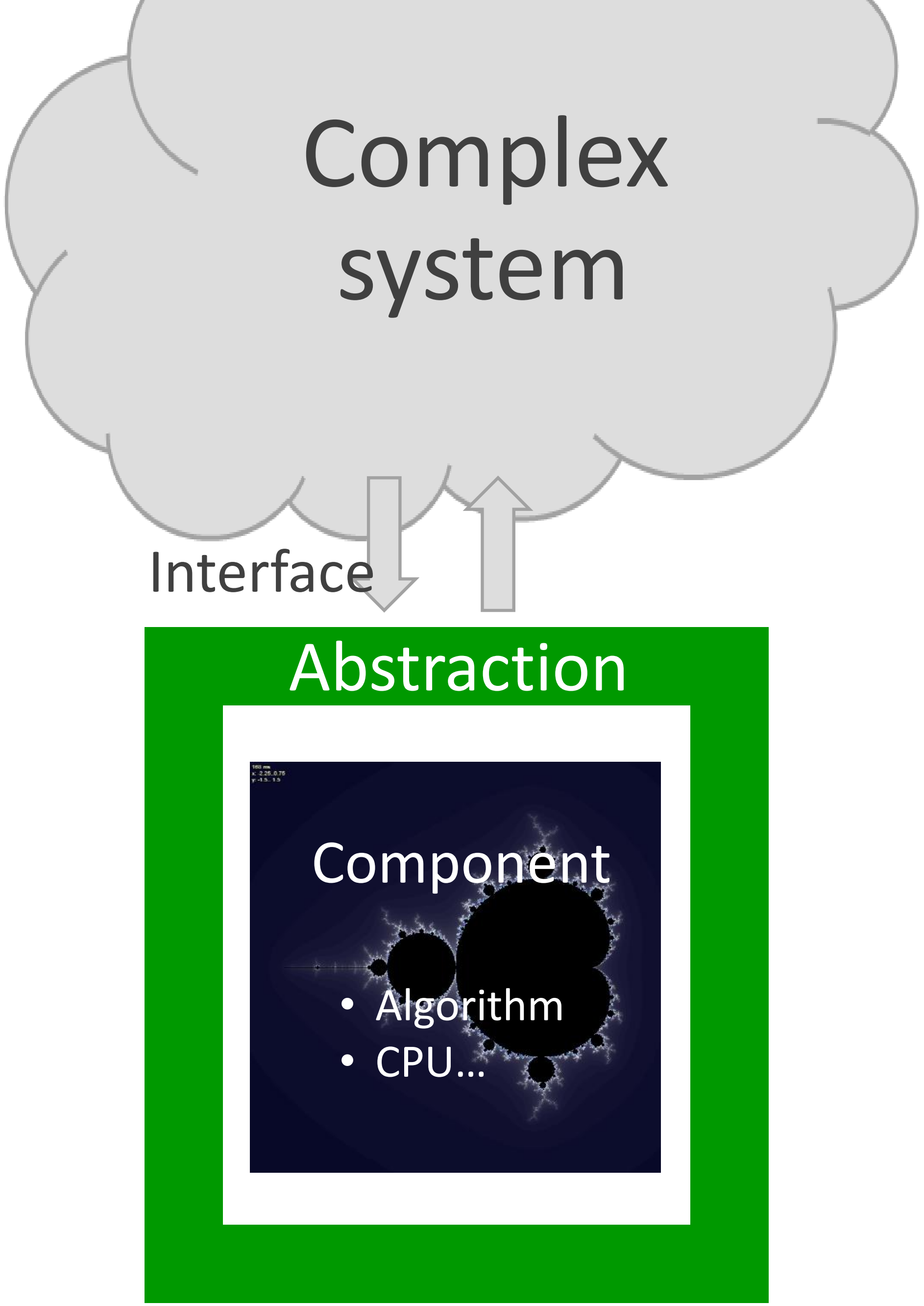
We need 4 things:

Better components
= Better matched to system security needs

Better abstraction/interface descriptions
= Conveys security properties

Better ways to instantiate components
= Use components with less risk to overall system

Better reasoning about compositions
= Understand combinations of components



Complex system

Interface

Abstraction

Component

- Algorithm
- CPU…

Strategies from yesterday's talk

- Abstraction collapsing
- Temporal separation
- Spatial separation
- Redundancy

Software determinism

- Can we guarantee (most) SW is deterministic?
  - Output = f(source, input) … and nothing else
  - Functional programming attributes for large functional blocks
  - Repeatable, checkable

Implies:

- Same output on all CPUs
- No buffer overflows
- No race conditions
- No side channel receivers
  … but can transmit

Note: Determinism != correctness
Narrower but more achievable

- Can use for a lot (but not everything)
  - Codecs, decompression, font rendering, math…
  - Much lower overhead (+ better compartmentalization) than microservices

- Strategies for realizing
  - **sound** static analyzers
  - safe subsets of unsafe languages (C, assembly)
  - safer compilers
  - safer languages

Challenge: Poor hardware support!

Example: Big performance cost for dynamic checks due to lack of support

We need better hardware!

In essentially all practical situations, computations can only be as secure as the hardware that performs them

- Technical: Better hardware can address technical barriers to safer systems & safer software. [Example: CHERI]

- Stronger foundational layers (e.g. HW) is needed to make security investments in higher layers make economic sense

No “best” architecture or implementation

Fastest != safest

- Market (mis-)trained to expect optimal everything
- Result: same CPUs for video games, funds transfers, …

Trade-offs: Need to bifurcate **faster** vs. **safer**

- ‘Safer’ needs to be much less complex HW and OS -- not just a different mode (like TrustZone/SGX)
- Can co-exist in one device (or chip)

Many areas of difference:
- performance needs
- tolerance for complexity-induced risks
- choice of risky optimizations
- safety margins
- side channel/fault attack countermeasures
- backward compatibility with unsafe software
- feature choices
- verification/test processes
- …

Volvo image public domain by OSX, source: https://en.wikipedia.org/wiki/Volvo_XC90#/media/File:2005_Volvo_XC90_(P28_MY05)_2.5_T_wagon_(2011-11-18)_01.jpg
Race car by Andrew Griffith CC BY 2.0, source: https://en.wikipedia.org/wiki/Ferrari_F10#/media/File:Felipe_Massa_Ferrari_Bahrain_2010_GP.jpg
SR-71 image in public domain, source: https://commons.wikimedia.org/wiki/File:Lockheed_SR-71_Blackbird_(modified).jpg
787 image by Dave Sizer, CC BY 2.0, source: https://simple.wikipedia.org/wiki/Boeing_787_Dreamliner#/media/File:787_First_Flight.jpg

10 areas for work

① Side channel attacks

- What analog & digital effects leak information?
- What new optimizations will leak information?

② Fault attacks

- How might errors be induced (crosstalk, glitches…)? Detected? Mitigated?

③ Denial of service attacks

- Can software induce permanent failures (e.g. damage hardware via NVM exhaustion, fuses, electromigration…)?

④ Architectures

- What guarantees “should” architectures make to meet the needs of SW? How should guarantees be documented? How do guarantees flow to code in higher level languages?

⑤ Memory models

- How should memory work in secure systems? What support is needed in higher-level languages & legacy code?

⑥ Verification & test

- How to minimize (or just estimate) the likelihood of vulnerabilities in architectures? Implementations? Manufactured devices?
- Are there safer ways to test manufactured devices (e.g. scan chains can enable attacks)?

⑦ Metrics for security

- What metrics might help buyers, developers, and regulators make informed security decisions?

⑧ Legacy migration

- How can existing architectures (x86, ARM), operating systems, drivers, and applications be migrated to something safer?

⑨ Incident response

- How should hardware vulnerabilities be handled, including in cases where devices cannot be fixed?

⑩ Education

- How can we meet the need for security talent (research, architecture, implementation, verification, operation…)?

Q&A

My email: paul@paulkocher.com