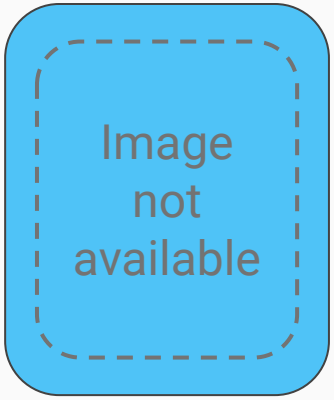
Data Dependent Instruction Timing Channels

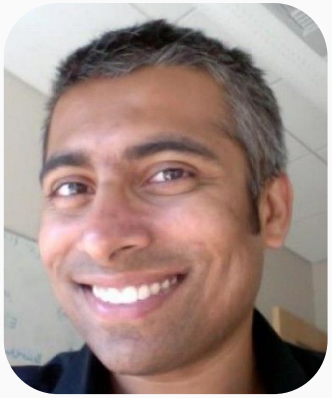
Hovav Shacham
The University of Texas at Austin

Work with ...



Marc Andrysco

Dongseok Jang



Ranjit Jhala

David Kohlbrenner

Sorin Lerner

Keaton Mowery

NSF

moz://a

Google

Let’s run some code! (on an Intel E3-1271v3 @ 3.60 GHz, GCC 5.4.0, -O0)

Normal floating point

```
#include <stdio.h>
#include <stdint.h>

int main() {
  double x = 1.0;
  double z, y = 1.0;
  uint32_t i;

  for (i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

0.160 s

Subnormal floating point

```
#include <stdio.h>
#include <stdint.h>

int main() {
  double x = 1.0e-323;
  double z, y = 1.0;
  uint32_t i;

  for (i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

Let’s run some code! (on an Intel E3-1271v3 @ 3.60 GHz, GCC 5.4.0, -O0)

Normal floating point

```
#include <stdio.h>
#include <stdint.h>

int main() {
  double x = 1.0;
  double z, y = 1.0;
  uint32_t i;

  for (i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

0.160 s

Subnormal floating point

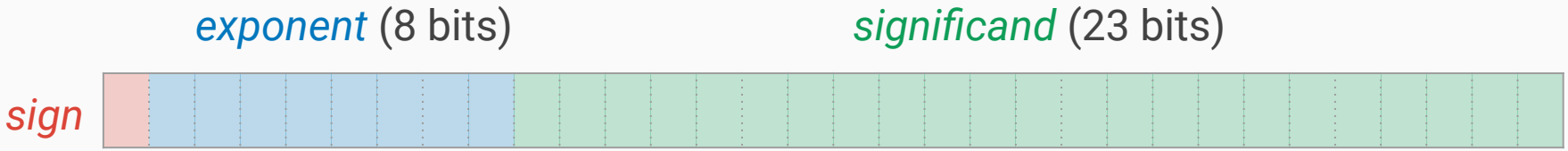
```
#include <stdio.h>
#include <stdint.h>

int main() {
  double x = 1.0e-323;
  double z, y = 1.0;
  uint32_t i;

  for (i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

3.588 s

20 times slower?!

- Who knew?
  - Numerical analysts
  - CPU designers
  - Game engine authors
- Who should have known?
  - Every computer scientist: “What Every Computer Scientist Should Know about Floating Point Arithmetic,” Goldberg ‘91

Background: Floating point and subnormals



Value = $(-1)^{sign} \times significand \times 2^{(exponent - bias)}$

Normal values have nonzero *exponent*, implicit leading 1. before *significand*

Subnormal values have all-zero *exponent*, implicit leading 0. before *significand*

⇒ slow-path special handling in hardware

In fact, FP ops show timing variation beyond subnormals

Dividend	Divisor 0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.56	6.59	6.58	6.55	6.57	6.58	6.57	6.57	6.59
1.0	6.58	6.58	12.19	12.17	12.22	12.24	6.57	12.24	165.76
1e10	6.58	6.55	12.25	12.20	12.23	12.25	6.57	12.22	165.81
1e+200	6.60	6.60	12.25	12.20	12.22	12.22	6.58	12.24	165.79
1e-300	6.59	6.57	175.22	12.24	12.17	12.22	6.52	12.23	165.83
1e-42	6.60	6.53	12.23	12.22	12.21	12.24	6.58	12.21	165.79
256	6.57	6.55	12.24	12.20	12.20	12.20	6.53	12.22	165.79
257	6.55	6.58	12.24	12.22	12.24	12.23	6.56	12.21	165.80
1e-320	6.56	150.73	165.79	6.59	165.78	165.76	150.66	165.80	165.78

double-precision SSE scalar division on Intel i5-4460

Different processors exhibit different timing behavior

	0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	7.01	7.01	7.01	7.01	7.01	216.22	216.16	7.01	7.01
1.0	7.01	7.01	7.01	7.01	7.01	48.07	48.06	7.01	7.01
1e10	7.01	7.01	7.01	7.01	7.01	48.06	48.06	7.01	7.01
1e+30	7.01	7.01	7.01	7.01	7.01	48.06	48.06	7.01	7.01
1e-30	7.01	7.01	7.01	7.01	7.01	48.07	48.06	7.01	7.01
1e-41	216.17	48.05	48.05	48.06	48.06	216.20	216.17	48.05	48.05
1e-42	216.22	48.06	48.05	48.05	48.05	216.16	216.16	48.05	48.05
256	7.01	7.01	7.01	7.01	7.01	48.06	48.06	7.01	7.01
257	7.01	7.01	7.01	7.01	7.01	48.06	48.06	7.01	7.01

single-precision SSE addition on AMD Phenom II X2 550

Not just x86 processors; GPUs, too …

Dividend	Divisor								
	0.0	1.0	1e10	1e+30	1e-30	1e-41	1e-42	256	257
	Cycle count								
0.0	5.17	5.85	5.85	5.85	5.85	5.89	5.89	5.85	5.85
1.0	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
1e10	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e+30	6.19	2.59	2.59	2.59	5.96	8.64	8.64	2.59	2.59
1e-30	6.19	2.59	7.82	6.51	2.59	8.40	8.40	2.59	2.59
1e-41	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
1e-42	6.19	10.21	8.92	8.92	8.13	8.41	8.41	10.23	10.23
256	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59
257	6.19	2.59	2.59	2.59	2.59	8.64	8.64	2.59	2.59

single-precision division on NVIDIA GeForce GT 430

… and ARM processors!

1,000,000,000 double-precision divisions on Ampere eMAG 8180:

1.0/1e20:	~	8.8 s	
1e-300/1.0:	~	8.7 s	
1e-300/1e20:	~	10.7 s	(subnormal out)
1e-320/1e-20:	~	13.1 s	(subnormal in)
1e-320/1.0:	~	15.0 s	(subnormal in+out)
1e-300/1e100:	~	3.6 s	
1e-320/1e100:	~	3.6 s	(subnormal in!)

Security implication of timing variability: A side channel

A side channel is a leak of secret information through a channel other than intended system output

Floating point timing channels allow us to mount cross-origin pixel stealing attacks against Firefox 24–27 [AKMJLS'15]

… and, in followup work [KS'17], in recent Firefox, Safari, Chrome

First data-dependent timing side channel attack demonstrated on desktop-class processors, nearly 20 years after first hypothesized [K'96]

Browser security in one slide

Browsers allow a user to interact with multiple, mutually distrusting origins

Browsers manage and display secrets shared between the user and each origin, which must not be exposed to other origins

Browsers allow programs supplied by an origin to run on the user's CPU and to interact with the user's system through a rich API

A single document may incorporate content from multiple origins, e.g., through <img> or <iframe> elements

Pixel-stealing attacks

Many secrets shared between the user and an origin are expressed visually in documents in that origin: login name, bank balances, inbox contents, etc.

Cross-origin pixel-stealing attack: attacker learns other-origin private info
(victim site must allow itself to be framed)

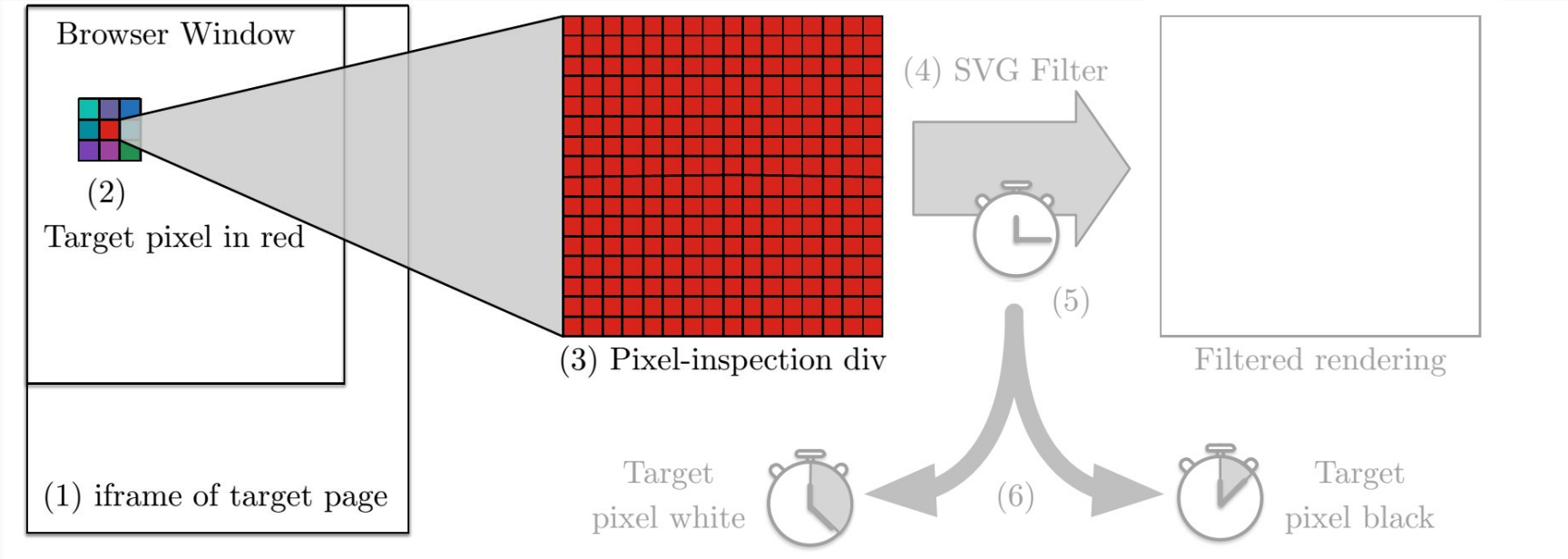
Same-origin pixel-stealing attack: history-sniffing (at circa 60 links / second)
(by checking whether a link was rendered in :visited colors)

Cross-origin SVG filters: Turn this …

… into this!

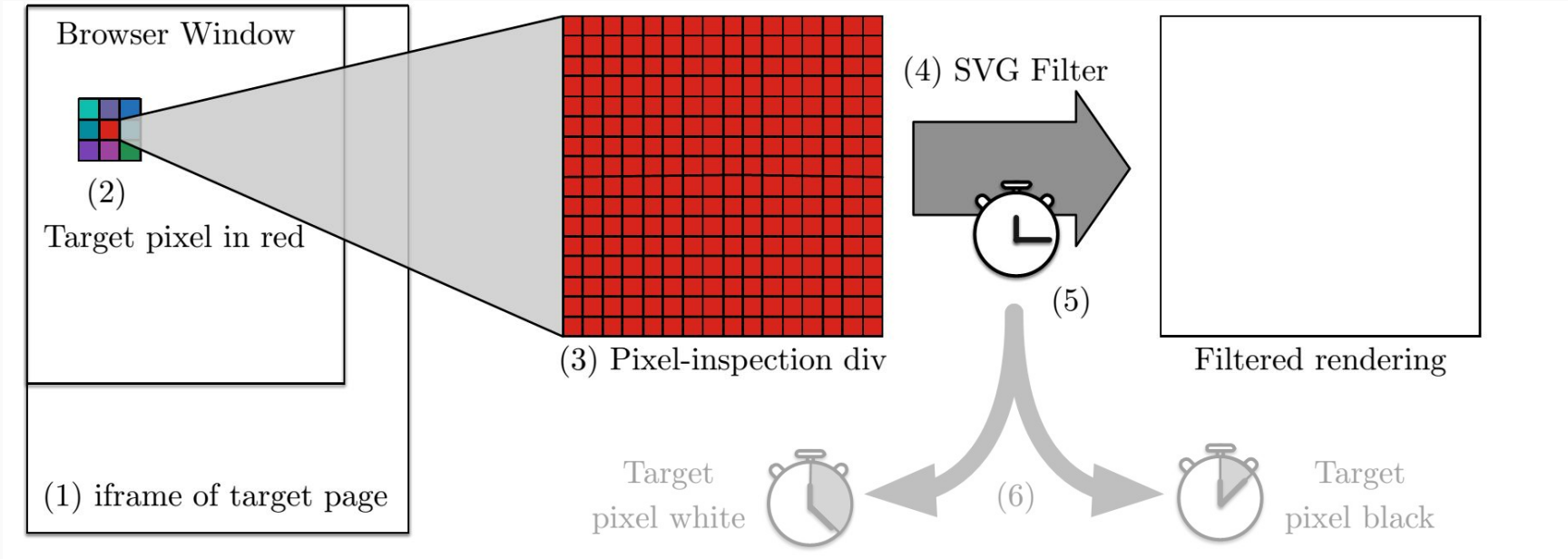
```
<svg><filter id="f">
<feGaussianBlur
    stdDeviation="3"/>
</filter></svg>
```

SVG filter timing attack methodology



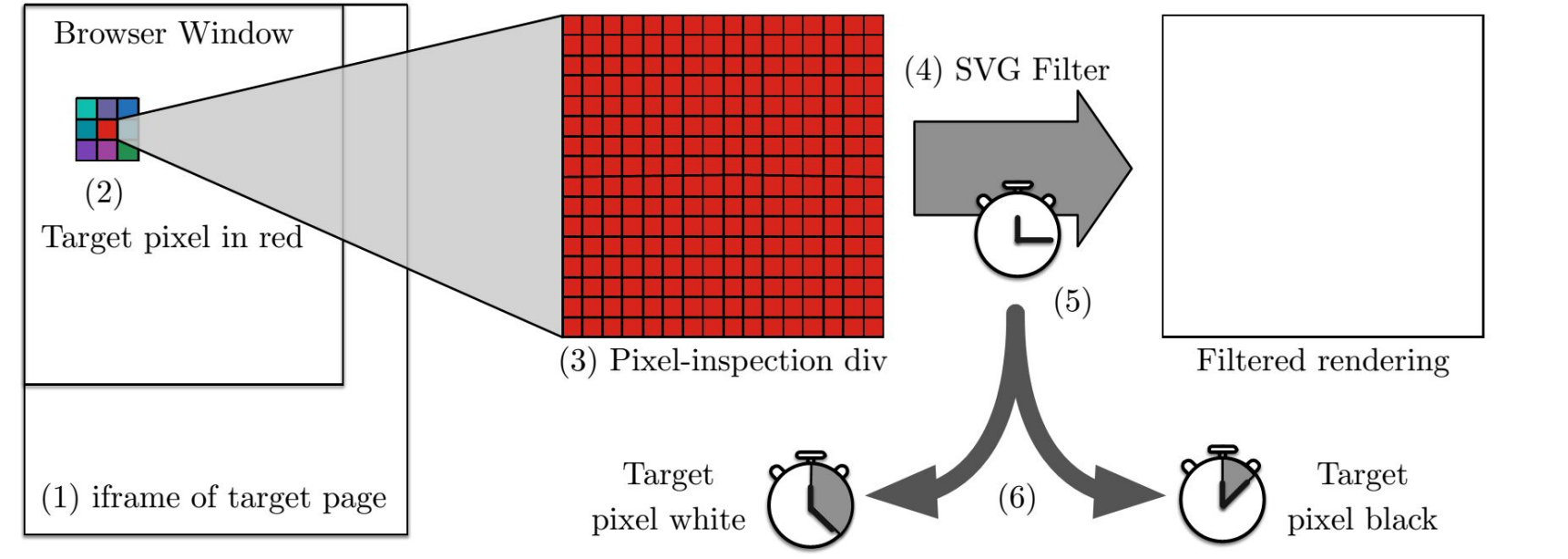
See Paul Stone's "Pixel Perfect Timing Attacks with HTML5"

SVG filter timing attack methodology



See Paul Stone's "Pixel Perfect Timing Attacks with HTML5"

SVG filter timing attack methodology



See Paul Stone's "Pixel Perfect Timing Attacks with HTML5"

Paul Stone's `feMorphology` timing channel

Firefox implementation for feMorphology had a special case:

```
// We need to scan the entire kernel
if (x == rect.x || xExt[0] <= startX || xExt[1] <= startX ||
    xExt[2] <= startX || xExt[3] <= startX) {
  [...]
{ else { // We only need to look at the newest column
  [...]
}
```

Mozilla fix: try to write constant-time filter code

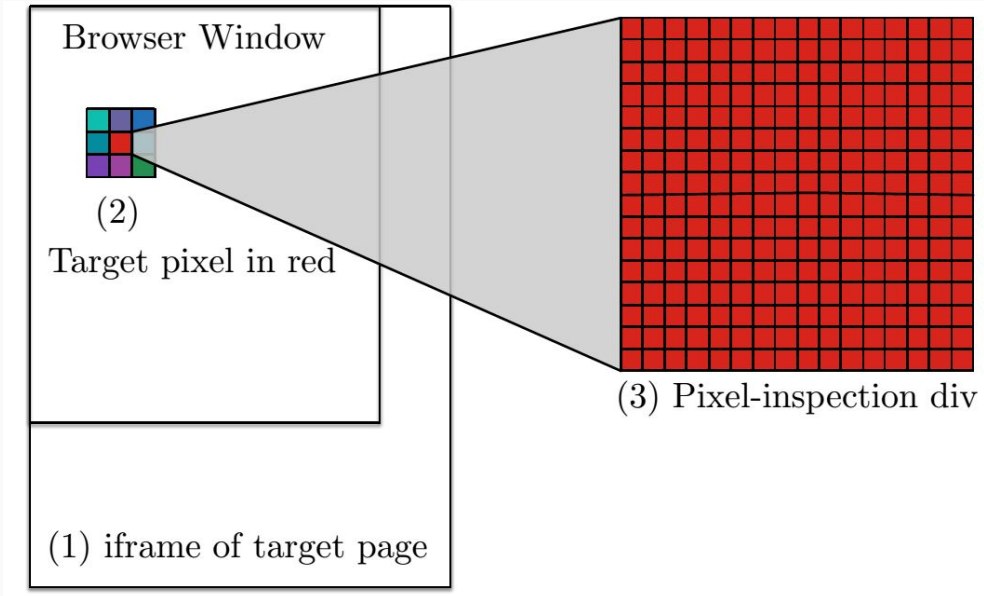
(took two years to land, and a 150-comment Bugzilla thread)

Our pixel-stealing attack on Firefox (versions 23–27)

Firefox's "constant-time" SVG filters still used floating point!

Attacker chooses filter and settings so black pixels generate fewer subnormal ops than white pixels

Slowdown is just a few cycles per instruction; need to amplify it:



Example filter: `feConvolveMatrix` (in pseudocode)

```
def do_one_convolve(kernel, subimage):
  for x,y in subimage:
    tmp[x,y] = kernel[x,y] * subimage[x,y]
  result = 0.0
  for x,y in tmp:
    result = result + tmp[x,y]
  return result

for x,y in input:
  output[x,y] = do_one_convolve(kernel,
                    subimage_around(input, x, y,
                        kernel.width, kernel.height))
```

s × 0. = 0. if black
s × 1. = s if white

0. + 0. = 0. if black
s + s = s if white

Use a kernel consisting of all subnormal values.

Revisiting browser SVG filter implementations

Since our 2015 paper:

- Firefox SVG filters rewritten mostly to use fixed-point math
- Chrome SVG filters moved to GPU or run with FTZ/DAZ enabled
- Safari implementation not meaningfully changed

We found and disclosed new floating-point pixel-stealing attacks on all three.
Firefox: CVE-2017-5407; Safari: CVE-2017-7006; Chrome: CVE-2017-5107

Safari no longer applies SVG filters to cross-origin iframes.

Some operations still show variable timing even with FTZ and DAZ enabled

Dividend	Divisor								
	0.0	1.0	1e10	1e+200	1e-300	1e-42	256	257	1e-320
	Cycle count								
0.0	6.58	6.59	6.58	6.55	6.59	6.54	6.54	6.56	6.56
1.0	6.55	6.55	12.23	12.19	12.22	12.22	6.56	12.25	6.56
1e10	6.58	6.59	12.22	12.22	12.21	12.21	6.59	12.23	6.59
1e+200	6.57	6.59	12.22	12.20	12.17	12.21	6.58	12.17	6.57
1e-300	6.59	6.57	12.18	12.23	12.24	12.22	6.59	12.24	6.57
1e-42	6.58	6.56	12.21	12.25	12.23	12.18	6.56	12.21	6.58
256	6.57	6.60	12.20	12.22	12.24	12.24	6.57	12.23	6.54
257	6.57	6.58	12.22	12.23	12.25	12.20	6.57	12.23	6.58
1e-320	6.57	6.58	6.60	6.51	6.59	6.57	6.58	6.55	6.58

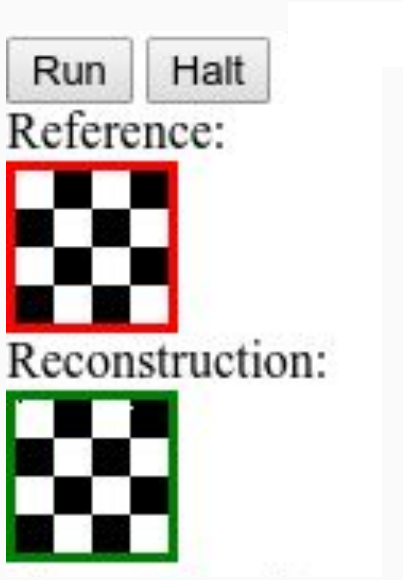
double-precision SSE division on Intel i5-4460, FTZ and DAZ enabled

Our attack on Chrome/Skia

On i5-4460, only 32-bit divide and square root show timing variation when FTZ and DAZ are enabled

Unable to find a filter that uses these operations unsafely

Skia turns FTZ/DAZ off when a filter is sent to the GPU, failed to turn FTZ/DAZ back on when it changes its mind



Chrome/Skia bugfix

```
680  680
     681  // Big filters can sometimes fallback to CPU. Therefore, we need
     682  // to disable subnormal floats for performance and security reasons.
     683  ScopedSubnormalFloatDisabler disabler;
681  684  SkMatrix local_matrix;
682  685  local_matrix.setTranslate(origin.x(), origin.y());
683  686  local_matrix.postScale(scale.x(), scale.y());
684  687  local_matrix.postTranslate(-src_rect.x(), -src_rect.y());
```

In QA for patch, developers discovered FTZ/DAZ never enabled on Windows!

Lesson: processor flags offer brittle security, are difficult to manage

Our approach to closing floating-point timing channels: libfixedtimefixedpoint

The floating-point unit is too dangerous to use in security-relevant code.

Instead, use integer unit to implement fixed-point math

We wrote libfixedtimefixedpoint

- Implements most common math operations (not all guaranteed precise)
- Between 1 and 61 fractional bits, selected at compile time
- No data-dependent branches, no data-dependent table lookups, no instruction with runtime known to be variable (e.g., div)

Writing constant-time code is a battle against the processor and the compiler.

```
int64_t fix_to_int64(fixed op1) {
  return ({ uint8_t isinfpos = (((op1 )&((fixed) 0x3)) == ((fixed) 0 x2));
uint8_t isinfneg = (((op1 )&((fixed) 0x3)) == ((fixed) 0 x3)); uint8_t
isnan = (((op1) &((fixed) 0x3)) == ((fixed) 0x1 )); uint8_t ex = isinfpos
| isinfneg | isnan; fixed result_nosign = (({uint64_t SE_m__ = (1ull <<
((64 - ((60 + 2)))-1)); (((uint64_t) ((op1) >> ((60 + 2)))) ^ SE_m__) -
SE_m__;}) + !!( (!!((op1) & (1 LL << (((60 + 2))-1))) & !!(( op1) & ((1LL
<< (((60 + 2))-1)) -1))) | ((((op1) >> (((60 + 2)) -2)) & 0x6) == 0x6) ));
((({ uint64_t SE_m__ = (1ull << ((1) -1)); (((uint64_t) (!!(isinfpos ))) ^
SE_m__) - SE_m__;}) & (92233720368547758O7LL)) | (({ uint64_t SE_m__ =
(1ull << ((1) -1)); (((uint64_t) (!!(isinfneg ))) ^ SE_m__) - SE_m__;}) &
((-92233720368547758O7LL -1))) | (({uint64_t SE_m__ = (1ull << ((1)-1));
(((uint64_t) (!!(!ex ))) ^ SE_m__) - SE_m__;}) & ( result_nosign))); });
}
```

LibFTFP recently proved to be constant time as LLVM bytecode [ABBDE'16]; may or may not be constant-time on actual processors

What is to be done?

Browser vendors:

- Eliminate unnecessary cross-origin interactions
- Reduce resolution of reference timers [Wray ‘91, KS’16]

Compiler vendors:

- Make constant-time programming less miserable

CPU vendors:

- Document variable-time and constant-time instructions
- Implement opt-in constant-time mode

ARMv8.4-A added PSTATE.DIT, “Data-Independent Timing”… but not to FP operations!

```
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Conclusion

Floating-point instructions exhibit data-dependent timing variation

… ARM processors included

More software than just crypto libraries keeps secrets

… and manipulates those secrets using floating point ops

Today, no provably secure way to use FPU if data timing channels are a threat

PSTATE.DIT should be extended to cover floating point operations

Questions?

In this talk:

- D. Jang, R. Jhala, S. Lerner, and H. Shacham. “An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications.” In proc. CCS 2010.
- M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. “On Subnormal Floating Point and Abnormal Timing.” In proc. Oakland 2015.
- D. Kohlbrenner and H. Shacham. “Trusted Browsers for Uncertain Times.” In proc. USENIX Security 2016.
- D. Kohlbrenner and H. Shacham. “On the effectiveness of mitigations against floating-point timing channels.” In proc. USENIX Security 2017.

https://www.cs.utexas.edu/~hovav/