moz://a

Servo

Spectre and the interfaces between browser, operating system and hardware

2.28.2019

Alan Jeffrey
Research Engineer

Agenda

1. Servo browser architecture

   What is Servo, and what is its architecture?

2. Spectre and multiprocess

   What is Spectre, and how does it impact a browser?

3. Hardware/OS support for browsers

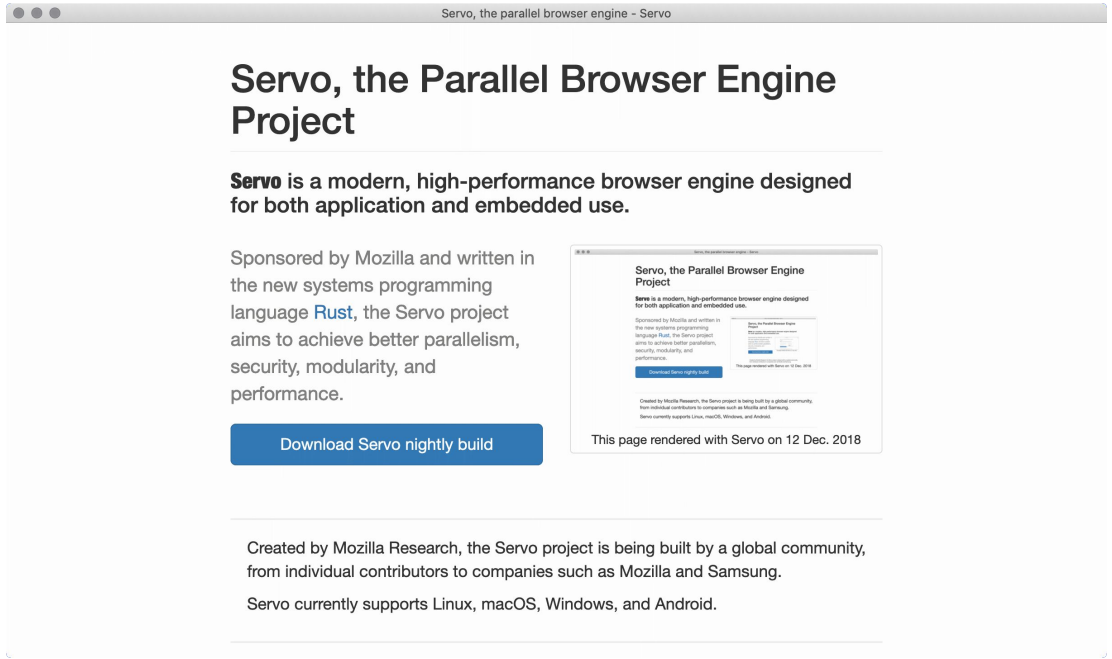
   What could a browser hope from its environment?

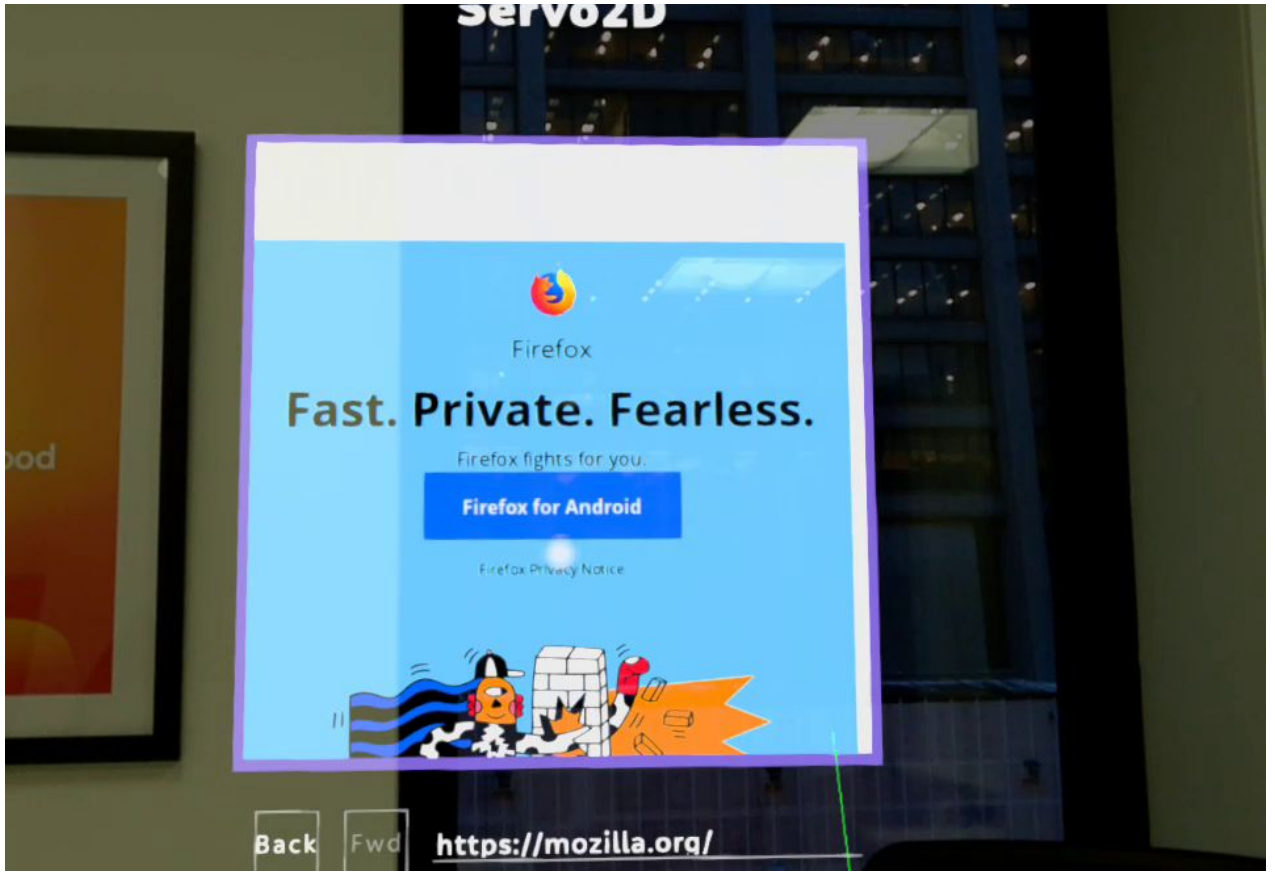
Servo browser architecture

What is Servo?

Mozilla’s next-generation browser engine

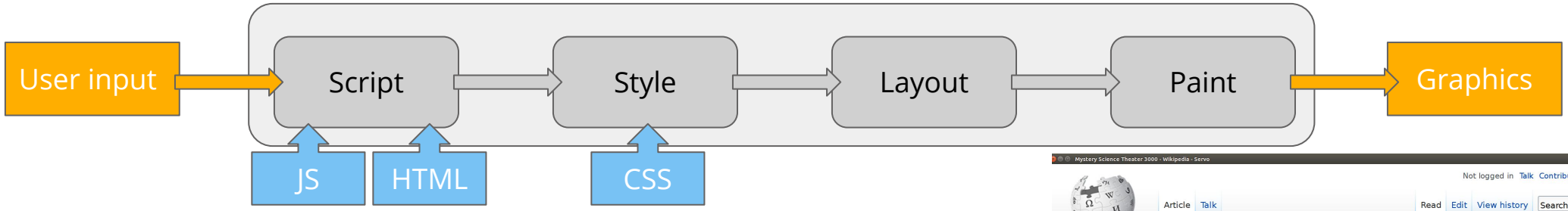
- Designed for modern CPUs (multicore) and GPUs (shaders)
- Implemented in Rust (memory safety)
- Targeting Mixed Reality (VR, AR, XR,…)

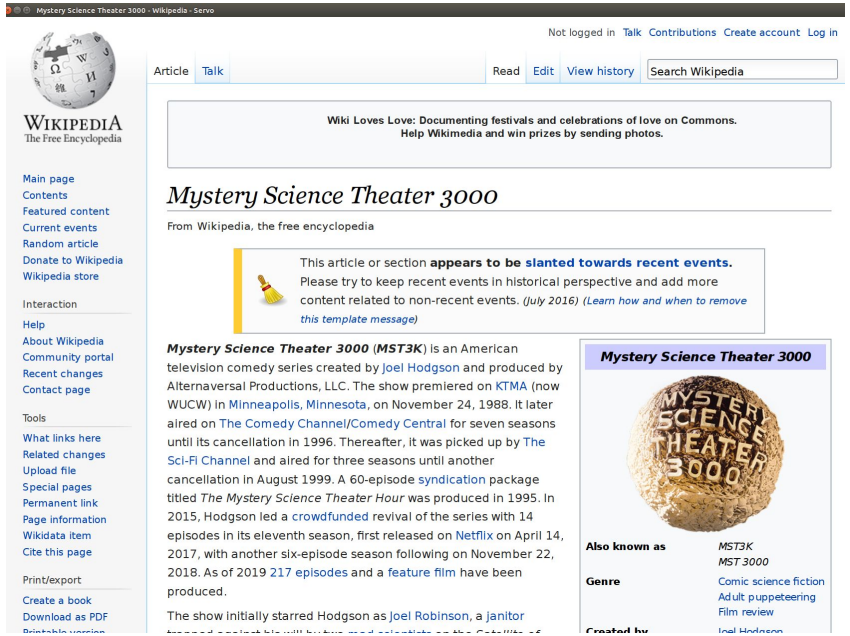




Browser architecture

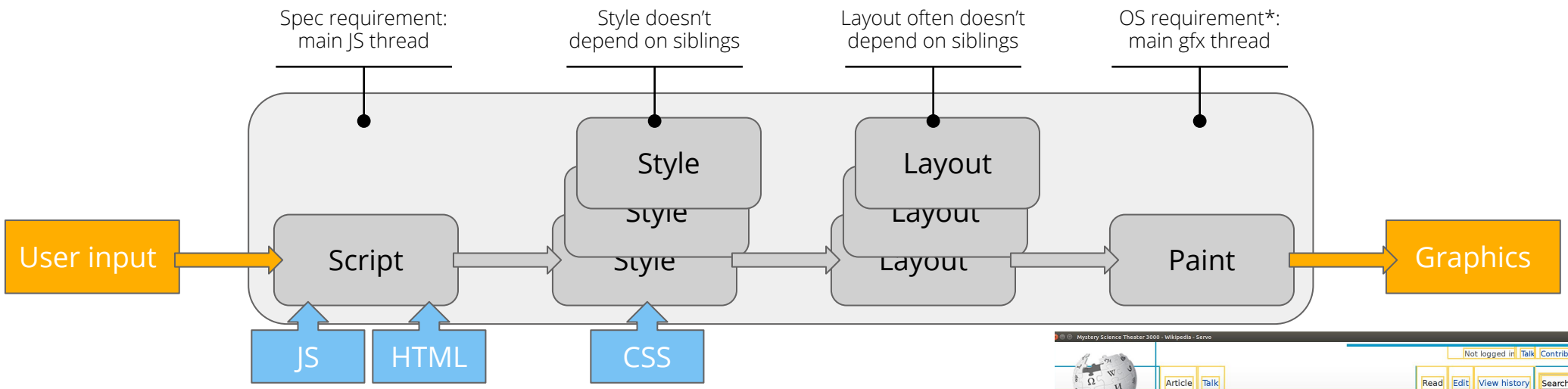
Highly simplified (i.e. contains lies)

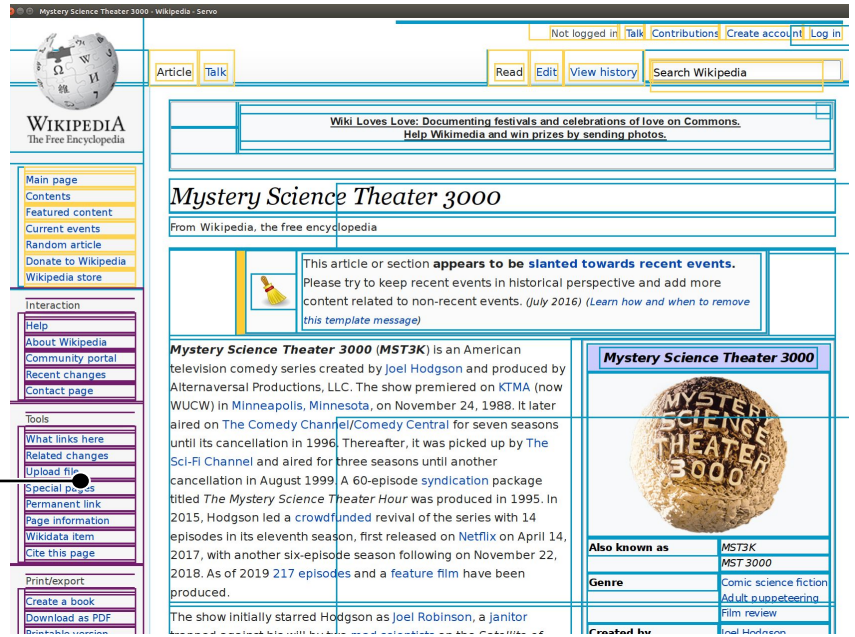




Multithreaded browser architecture

Independent computations can be run concurrently



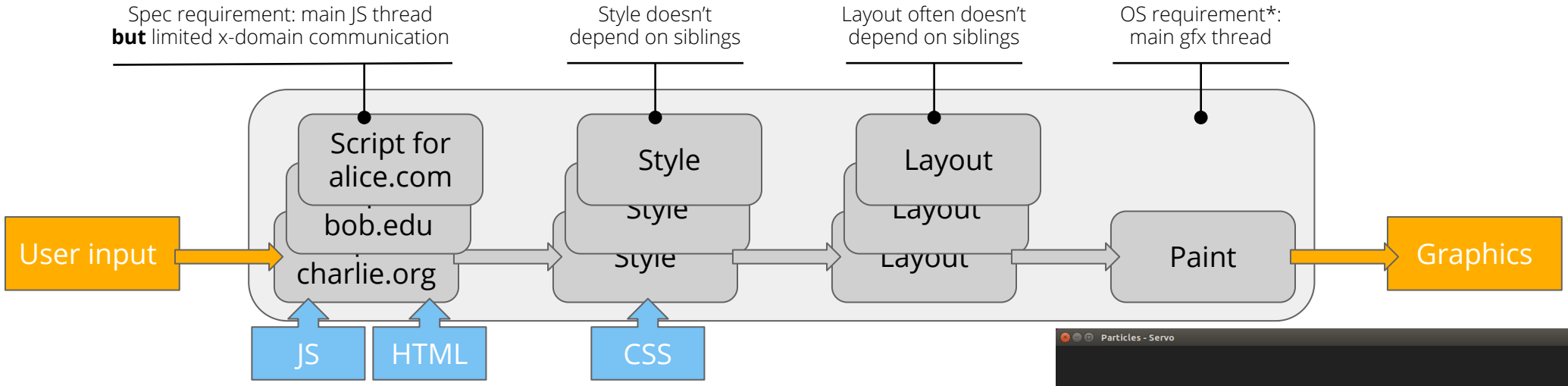


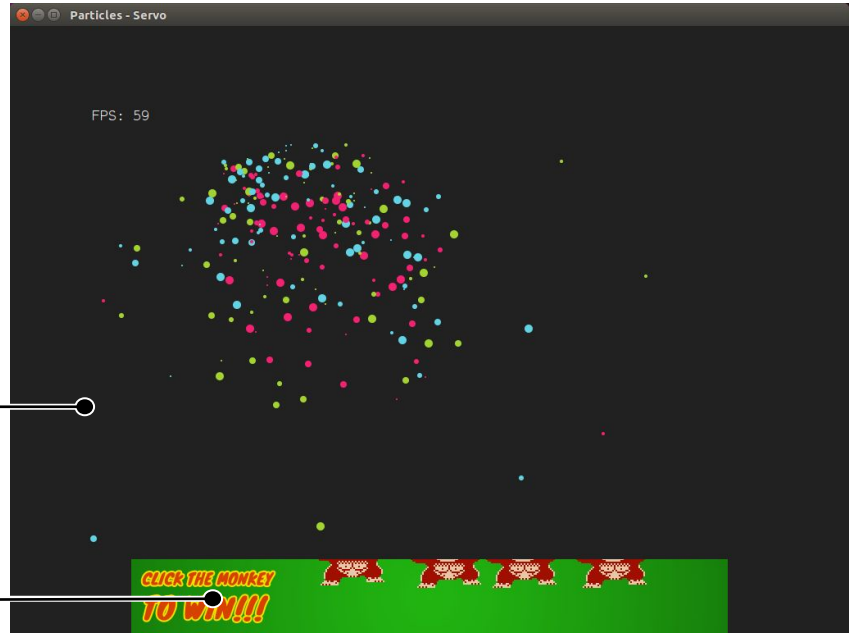
Different colors represent different threads

*on some OSs

Multithreaded browser architecture

Different security domains are independent





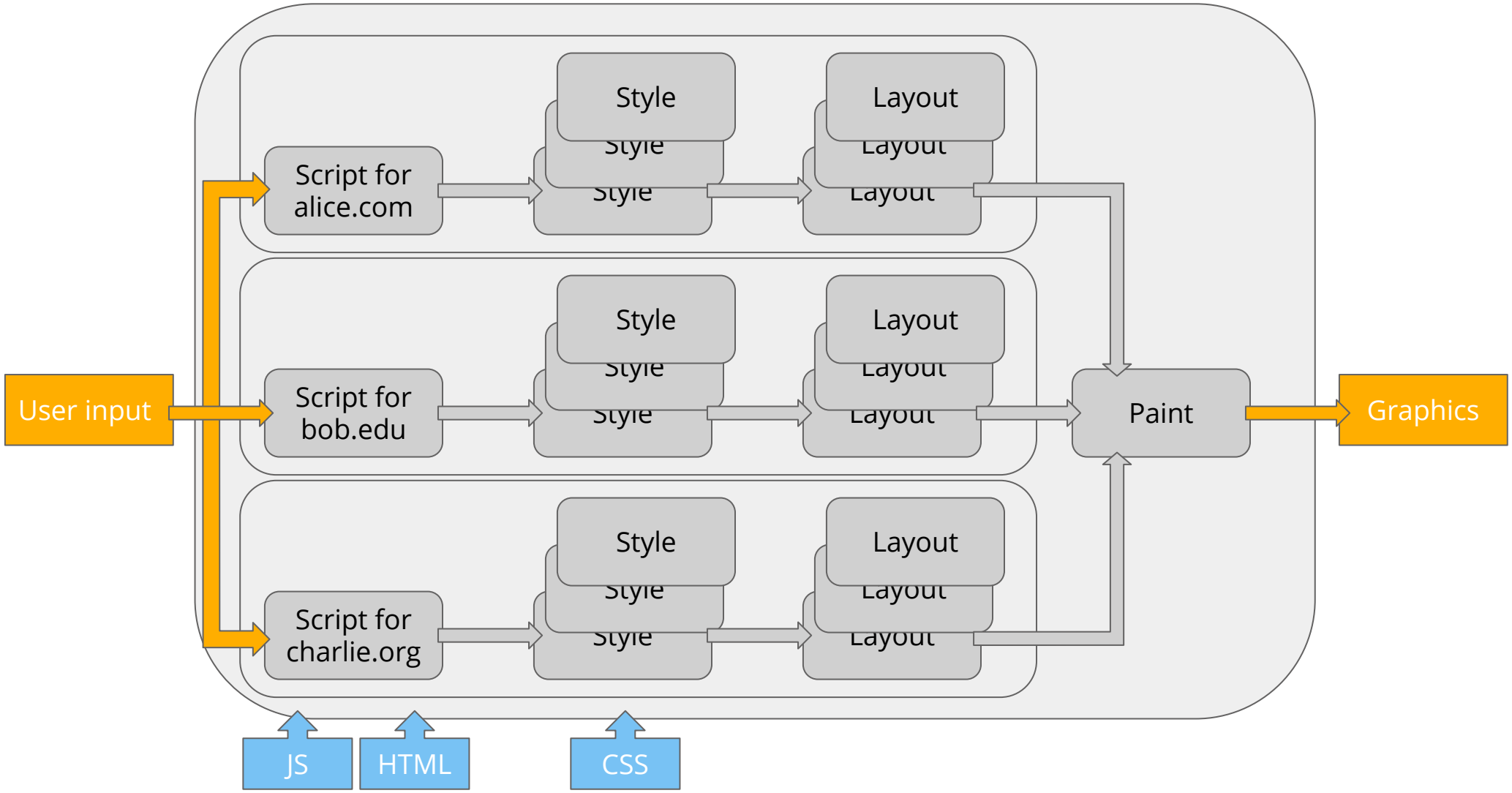
Content the user wants to see

Adverts, trackers, bitcoin miners, etc.

*on some OSs

Multiprocess browser architecture

Different security domains need to be in different address spaces*



*reasons coming up

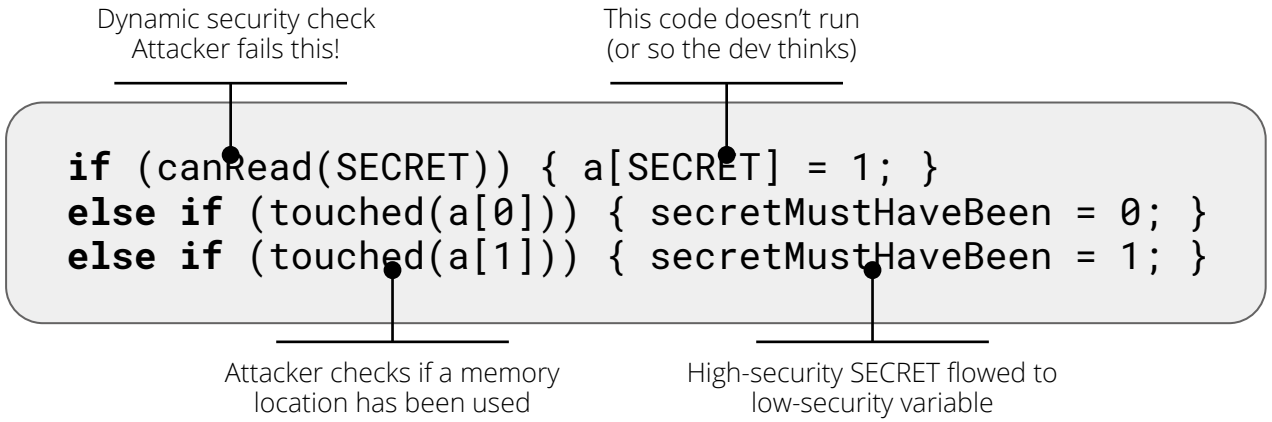
Spectre and multiprocess

What is Spectre?

In which I tell you things you already know

- Exploits μarchitecture (caching and branch prediction)
- Abstraction has leaked
- Allows bypassing of dynamic security checks

Simplified Spectre v1 attack:



Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

***Abstract*—Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.**

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

While makeshift processor-specific countermeasures are possible in some cases, sound solutions will require fixes to processor designs as well as updates to instruction set architectures (ISAs) to give hardware architects and software developers a common understanding as to what computation state CPU implementations are (and are not) permitted to leak.

I. Introduction

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side-channel attacks focus on exploiting these side effects to extract otherwise-unavailable secret information. Since their introduction in the late 90's [43], many physical effects such as power consumption [41, 42], electromagnetic radiation [58], or acoustic noise [20] have been leveraged to extract cryptographic keys as well as other secrets.

Physical side-channel attacks can also be used to extract secret information from complex devices such as PCs and mobile phones [21, 22]. However, because these devices often execute code from a potentially unknown origin, they face additional threats in the form of software-based attacks, which do not require external measurement equipment. While some attacks exploit software vulnerabilities (such as buffer overflows [5] or double-free errors [12]), other software attacks leverage hardware vulnerabilities to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [8, 30, 48, 52, 55, 69, 74], branch prediction history [1, 2], branch target buffers [14, 44] or open DRAM rows [56]. Software-based techniques have also been used to mount fault attacks that alter physical memory [39] or internal CPU values [65].

Several microarchitectural design techniques have facilitated the increase in processor speed over the past decades. One such advancement is speculative execution, which is widely used to increase performance and involves having the CPU guess likely future execution directions and prematurely execute instructions on these paths. More specifically, consider an example where the program's control flow depends on an uncached value located in external physical memory. As this memory is much slower than the CPU, it often takes several hundred clock cycles before the value becomes known. Rather than wasting these cycles by idling, the CPU attempts to guess the direction of control flow, saves a checkpoint of its register state, and proceeds to speculatively execute the program on the guessed path. When the value eventually arrives from memory, the CPU checks the correctness of its initial guess. If the guess was wrong, the CPU discards the incorrect speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. However, if the guess was correct, the speculative execution results are committed, yielding a significant performance gain as useful work was accomplished during the delay.

From a security perspective, speculative execution involves executing a program in possibly incorrect ways. However, because CPUs are designed to maintain functional correctness by reverting the results of incorrect speculative executions to their prior states, these errors were previously assumed to be safe.

A. Our Results

In this paper, we analyze the security implications of such incorrect speculative execution. We present a class of microarchitectural attacks which we call *Spectre attacks*. At a high level, Spectre attacks trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. As the effects of these instructions on the nominal CPU state are eventually

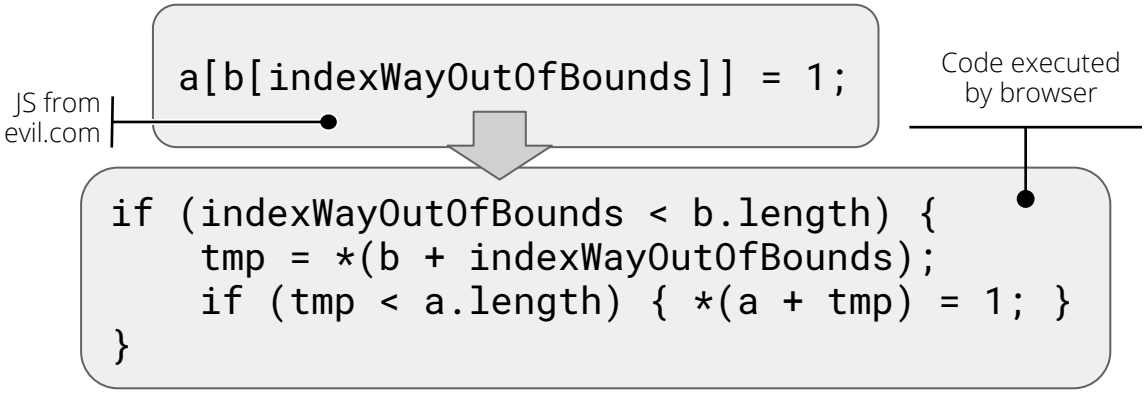
Spectre Attacks: Exploiting Speculative Evaluation, IEEE Security & Privacy 2019

Why do browser vendors care?

We're in the "sandboxed execution of untrusted code" business

Spectre is implementable in JavaScript.

Dynamic security checks are on array bounds.



Allows scanning of entire address space!

Spectre Attacks: Exploiting Speculative Evaluation, IEEE Security & Privacy 2019, p.8

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * 4096)|0) & (32*1024*1024-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

Listing 2: Exploiting Speculative Execution via JavaScript.

```
1 cmpl r15,[rbp-0xe0]          ; Compare index (r15) against simpleByteArray.length
2 jnc 0x24dd099bb870           ; If index >= length, branch to instruction after movq below
3 REX.W leaq rsi,[r12+rdx*1]   ; Set rsi = r12 + rdx = addr of first byte in simpleByteArray
4 movzxbl rsi,[rsi+r15*1]      ; Read byte from address rsi+r15 (= base address + index)
5 shll rsi,12                  ; Multiply rsi by 4096 by shifting left 12 bits
6 andl rsi,0x1ffffff           ; AND reassures JIT that next operation is in-bounds
7 movzxbl rsi,[rsi+r8*1]       ; Read from probeTable
8 xorl rsi,rdi                 ; XOR the read result onto localJunk
9 REX.W movq rdi,rsi           ; Copy localJunk into rdi
```

Listing 3: Disassembly of JavaScript Example from Listing 2.

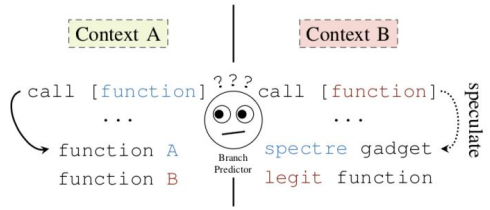


Fig. 2: The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space.

powerful means for attackers, for example exposing victim memory even in the absence of an exploitable conditional branch misprediction (cf. Section IV).

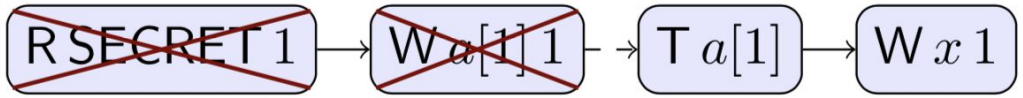
For a simple example attack, we consider an attacker seeking to read a victim's memory, who has control over two registers when an indirect branch occurs. This commonly occurs in real-world binaries since functions manipulating externally-received data routinely make function calls while registers contain values that an attacker controls. Often these values are ignored by the called function and instead they are simply pushed onto the stack in the function prologue and restored in the function epilogue.

The attacker also needs to locate a "Spectre gadget", i.e., a code fragment whose speculative execution will transfer the victim's sensitive information into a covert channel. For this

Dealing with a leaky abstraction

Spectre is not the first time this has happened

- Instruction reordering in μarchitecture
- Used to be behind an abstraction barrier
  (devs thought in terms of sequential consistency)
- This abstraction leaked: weak memory models
- Will speculative evaluation go the same way?



Paper also contains a Spectre-like attack
on optimizing compilers (plug plug)

The Code That Never Ran:
Modeling Attacks on Speculative Evaluation

Craig Disselkoen
University of California San Diego
Mozilla Research Internship
cdisselk@cs.ucsd.edu

Radha Jagadeesan
DePaul University
rjagadeesan@cs.depaul.edu

Alan Jeffrey
Mozilla Research
ajeffrey@mozilla.com

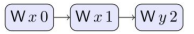
James Riely
DePaul University
jriely@cs.depaul.edu

***Abstract*—This paper studies information flow caused by speculation mechanisms in hardware and software. The Spectre attack shows that there are practical information flow attacks which use an interaction of dynamic security checks, speculative evaluation and cache timing. Previous formal models of program execution are designed to capture computer architecture, rather than micro-architecture, and so do not capture attacks such as Spectre. In this paper, we propose a model based on pomsets which is designed to model speculative evaluation. The model is abstract with respect to specific micro-architectural features, such as caches and pipelines, yet is powerful enough to express known attacks such as Spectre and PRIME+ABORT, and verify their countermeasures. The model also allows for the prediction of new information flow attacks. We derive two such attacks, which exploit compiler optimizations, and validate these experimentally against gcc and clang.**

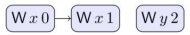
I. Introduction

This paper is about some of the lies we tell when we talk about programs.

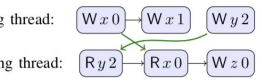
An example lie (or to be more formal, a "leaky abstraction") is the order of reads and writes in a program. We pretend that these happen in the order specified by the program text, for example we think of the program $(x{:=}0; x{:=}1; y{:=}2)$ as having three sequentially ordered write events:



However, due to optimizations in hardware or compilers, instructions may be reordered, resulting in executions where the accesses of x and y are independent, and the hardware or compiler is free to reorder them:



Instruction reordering optimizations are not problematic as long as they are not visible to user code, that is if programs are sequentially consistent. Unfortunately, multi-threaded programs can often observe reorderings. For example running the above writing thread concurrently with an observing thread $(\mathsf{if}\,(y)\,\{\,z{:=}x\,\})$ can result in a sequentially inconsistent execution (where we highlight the matching write for each read):

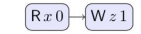


This leaky abstraction has resulted in a literature of *relaxed memory models* [29, 44, 19, 28, 5, 20, 21], which try to state precisely the memory guarantees a compiler is expected to provide, without requiring the use of expensive memory barriers to ensure sequential consistency.

Relaxed memory is an example of how simple models can become complex. Instruction reordering was originally intended to be visible only to the microarchitecture or compiler, not to the architecture or user code. Reordering optimizations are so important to the performance of modern systems that hardware and programming language designers have now accepted the complexity of relaxed memory models as the price that has to be paid for acceptable performance.

This paper looks at another leaky abstraction: *speculative evaluation*. This is similar to reordering, in that it is an optimization that was intended to be visible only to the microarchitecture, but the arrival of Spectre [23] shows that not only is speculation visible, it has serious security implications.

The simplest example of speculative evaluation is branch prediction. The expected observable behavior of a conditional such as $(\mathsf{if}\,(x)\,\{\,y{:=}1\,\}\,\mathsf{else}\,\{\,z{:=}1\,\})$ is that just one branch will execute, for example:



To improve instruction throughput, hardware will often evaluate branches speculatively, and roll back any failed speculations. For example, hardware might incorrectly speculate that x is nonzero, speculatively execute a write to y, but then roll it back and execute a write to z:

Speculation is intended only to be visible at the microarchitectural level, but as Spectre shows, this abstraction is leaky, and in a way that allows side-channel attacks to be mounted.

Instruction reordering and speculative evaluation are similar leaky abstractions. Both were intended originally not to be visible to user code, but both abstractions have leaked. This opens some possible areas of investigation:

- *Using ideas from relaxed memory for speculation.* There is a significant literature showing how to build models of

The Code That Never Ran: Modeling Attacks on Speculative Evaluation, IEEE Security & Privacy 2019

What are browser vendors doing about it?

Spectre mitigation

Short term fixes:

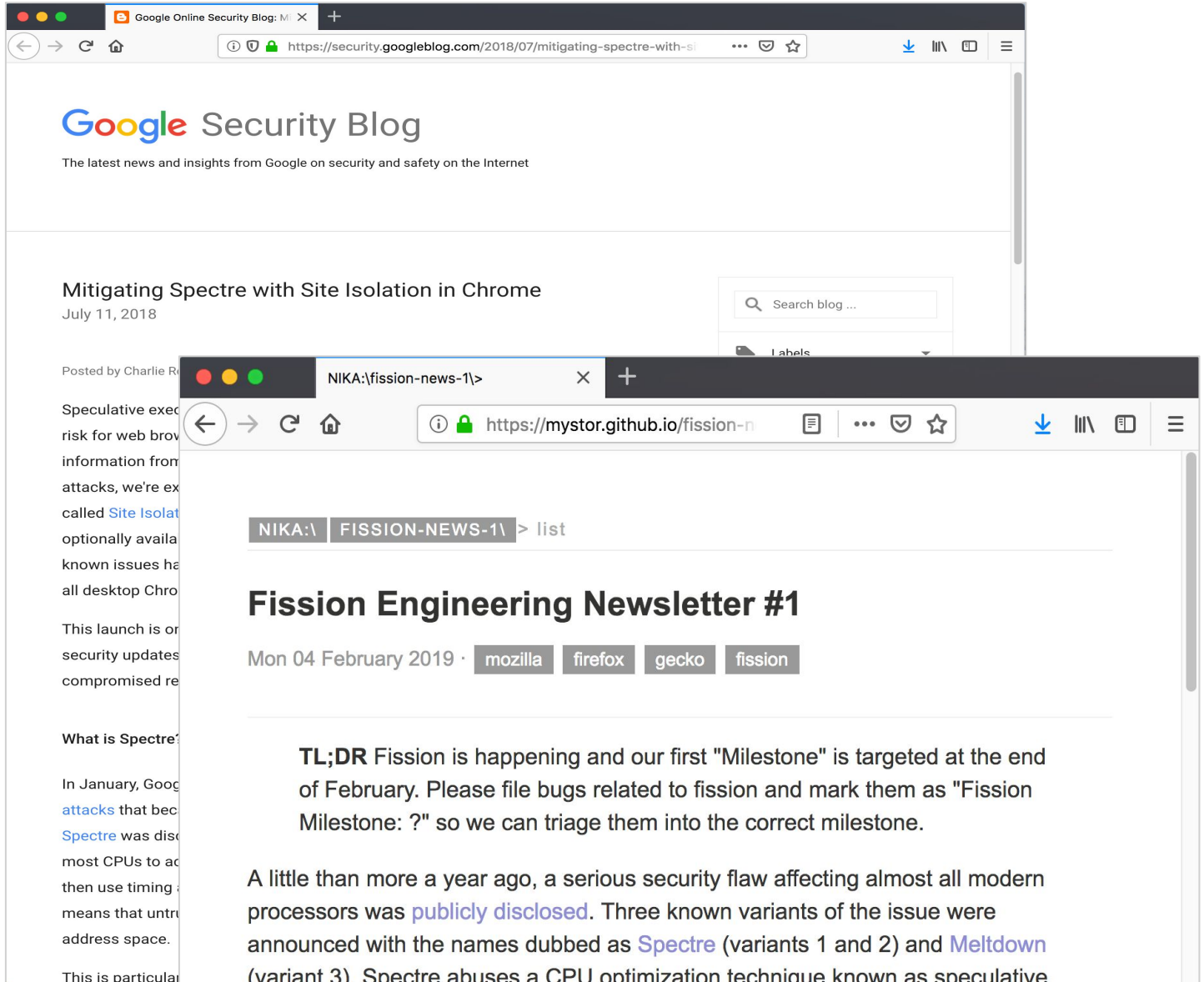
- Disable high-precision timers
- Disable shared-memory concurrency

Medium term fixes:

- Run security from different security domains in different address spaces

Long term fixes:

- Whatever hardware/OSs come up with



Hardware / OS support for browsers

Hardware/OS support for browsers

My wishlist by me

CPUs

- Spectre mitigation
- Security boundaries inside a process
- Call stack protection
- Async all the things
  (blocking → polling + callbacks)

GPUs

- GPUs/drivers as stable as CPUs/compilers
- Cross-platform open gfx stack
- Fast shared memory between CPU and GPU
- Video encoding support
  (e.g. Servo in the cloud)

Thank You