Browser-based CPU Fingerprinting

Dan Plămădeală

Jim van Vliet

Jakob Naucke

Cassie Xu

Konrad Ponichtera

Hacking Lab project, supervised by Harm Griffioen

Browser-based CPU Fingerprinting

Leon Trampert, Christian Rossow, and Michael Schwarz

CISPA Helmholtz Center for Information Security
Saarbrucken, Saarland, Germany
{leon.trampert,rossow,michael.schwarz}@cispa.de

Abstract. Mounting microarchitectural attacks, such as Spectre or Rowhammer, is possible from browsers. However, to be realistically exploitable, they require precise knowledge about microarchitectural properties. While a native attacker can easily query many of these properties, the sandboxed environment in browsers prevents this. In this paper, we present six side-channel-related benchmarks that reveal CPU properties, such as cache sizes or cache associativities. Our benchmarks are implemented in JavaScript and run in unmodified browsers on multiple platforms. Based on a study with 834 participants using 297 different CPU models, we show that we can infer microarchitectural properties with an accuracy of up to 100 %. Combining multiple properties also allows identifying the CPU vendor with an accuracy of 97.5 %, and the microarchitecture and CPU model each with an accuracy of above 60 %. The benchmarks

Motivation

- Research
    - Determining characteristics of processors based on their behavior under stress test
    - Checking how these characteristics allow classifiers to distinguish between different processors
- Technical
    - Hacking part: bypassing limitations of browser’s sandbox (e.g. lack of high-precision timestamps)
    - Exploring WebAssembly stack
    - Learning full-stack Rust the hard way
- Having fun

```
(module
    (import "env" "mem" (memory 1024 1024 shared))
    (import "console" "log" (func $log (param i32)))
    (export "iterate" (func $iterate))
    (export "check" (func $check))
    (export "write" (func $write))

    (func $iterate (param $start i32) (param $iterations i64) (result i64)
        (local $head i32)
        (local $i i64)
        (local $t0 i64)
        (local.set $i (i64.const 1))
        (local.set $head (local.get $start))

        (local.set $t0 (i64.load (i32.const 256)))

        (loop $iter
            (local.set $head (i32.load (local.get $head)))
```

Approach

1. Create Rust full stack project
    a. Backend for collecting data to the PostgreSQL database
    b. Frontend for running benchmarks and sending them to backend
2. Analyze original benchmarks, written in raw WebAssembly
3. Implement high-precision clock
4. Replicate the benchmarks (preserving original data format)
5. Deploy benchmarks and collect data through crowdsourcing
6. Visualize and compare results of original solution and the new one
7. Run classification on the collected data

Used technologies

WA

Trunk

ACTIX

Yew

Technical difficulties

1. Yew-agent lacks feature for sending *SharedArrayBuffers* to workers
    a. Opinionated library design - requires every payload to be serializable
    b. *SharedArrayBuffer* is not serializable

```
error[E0277]: the trait bound `for<'de> SharedArrayBuffer: Deserialize<'de>` is not satisfied
  --> frontend/src/worker/mod.rs:72:18
   |
72 |     type Reach = Private<Self>;
   |                  ^^^^^^^^^^^^^ the trait `for<'de> Deserialize<'de>` is not implemented for `SharedArrayBuffer`
   |
```

Technical difficulties

1. Yew-agent lacks feature for sending *SharedArrayBuffers* to workers
2. Still required to program in a JavaScript type of way (i.e. many callbacks)
    a. Event-driven development
    b. No way to sleep the thread - limiting for some benchmarks, like the single core one

```rust
let scope = DedicatedWorkerGlobalScope::from(JsValue::from(js_sys::global()));

let scope_clone = scope.clone();
let onmessage = Closure::wrap(Box::new(move |msg: MessageEvent| {
    info!("Clock worker received shared array buffer");
    let buffer = SharedArrayBuffer::from(msg.data());
    let clock = Clock::from(buffer);

    scope_clone
        .post_message(&JsString::from(CLOCK_MESSAGE_STARTED))
        .expect("posting started message succeeds");

    loop {
        clock.increment();
    }
}) as Box<dyn Fn(MessageEvent)>);

scope.set_onmessage(Some(onmessage.as_ref().unchecked_ref()));
onmessage.forget();
```

Technical difficulties

1. Yew-agent lacks feature for sending *SharedArrayBuffers* to workers
2. Still required to program in a JavaScript type of way (i.e. many callbacks)
3. No direct interface between WASM and browser, need to use JavaScript under the hood
4. The *web_sys* crate does not fully implement WebIDL
    a. For instance, no access to Performance API from within the worker scope in Rust code

Demo time

Results - Benchmarks

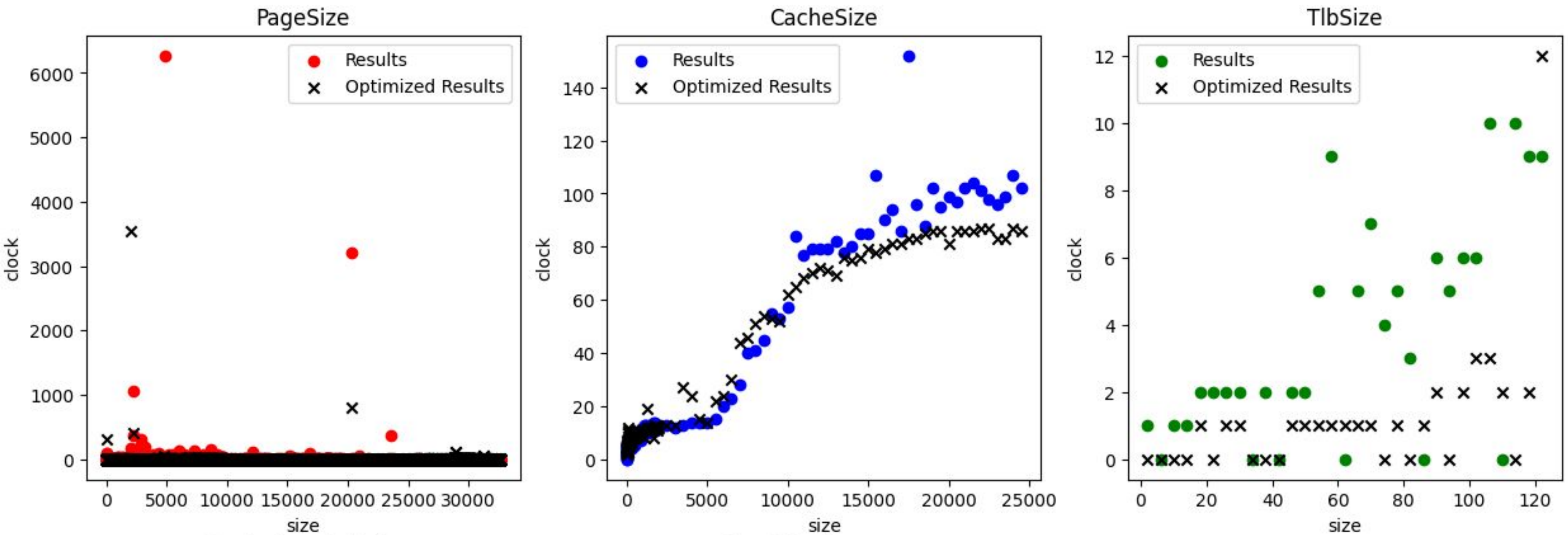
What is information is targetted by your benchmarks?

- Number of CPU cores
- Single core performance
- Multi core performance
- Memory latencies
- Data cache sizes
- L1D cache associativity
- L1D TLB size
- Page size
- Hyper-threading availability
- Data cache prefetcher presence
- Load buffer size
- `SharedArrayBuffer`-based timer precision (read more)

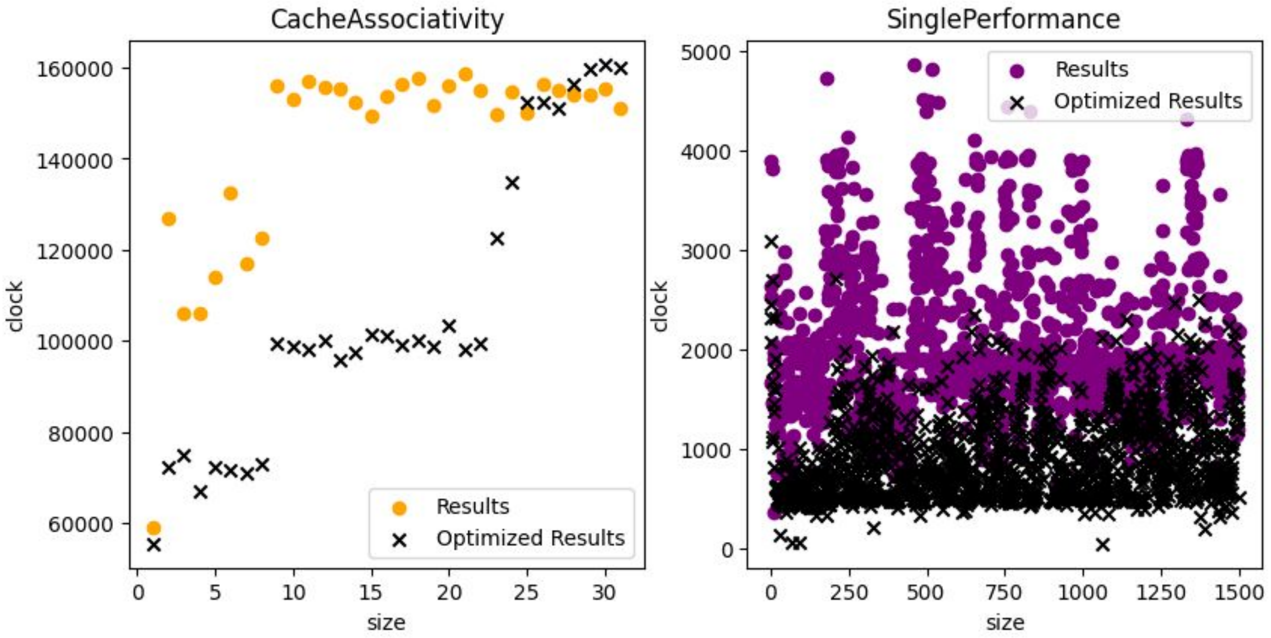
Our implementation:

- Cache size
- Cache associativity
- Page size
- Single core performance
- TLB size

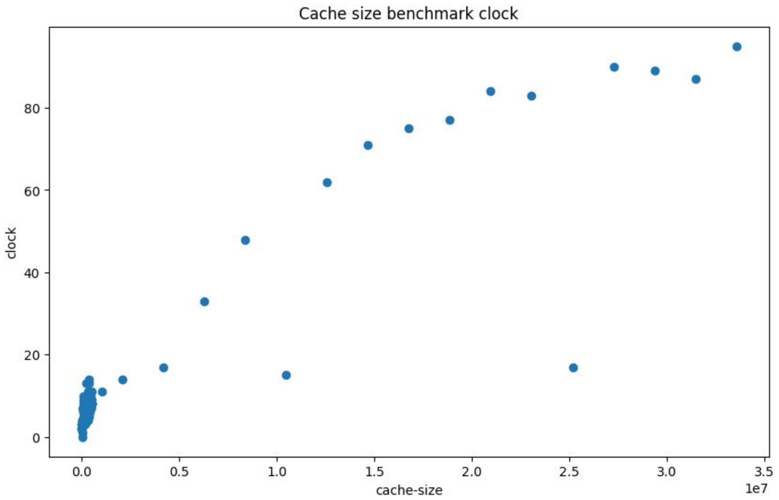
Rust modes - debug vs release optimized



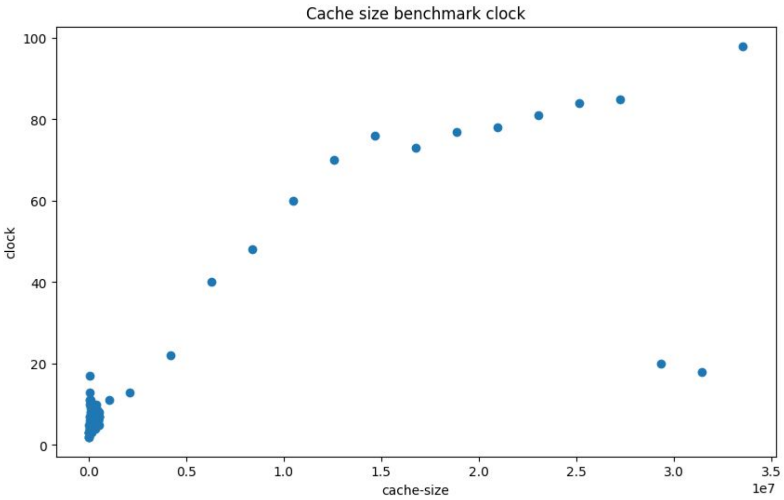
Rust modes - debug vs release optimized



Results - Cache Size



Pure WASM solution



Rust solution

Results - Single Core Performance

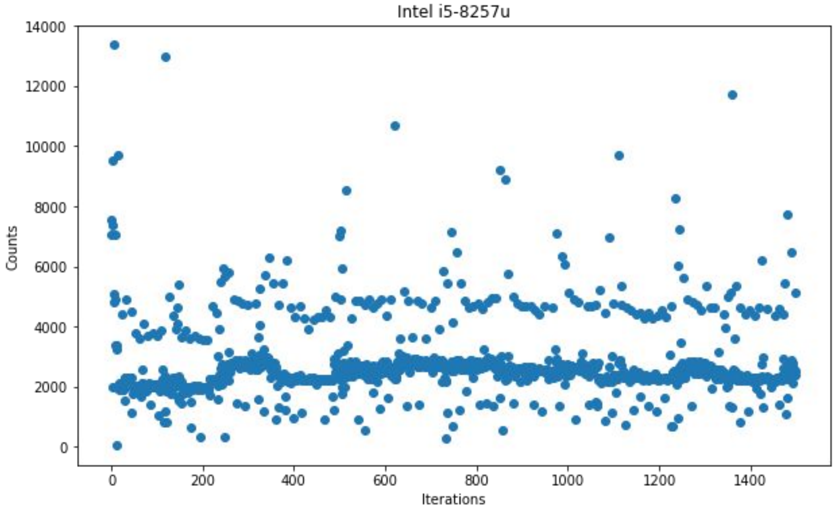
1.

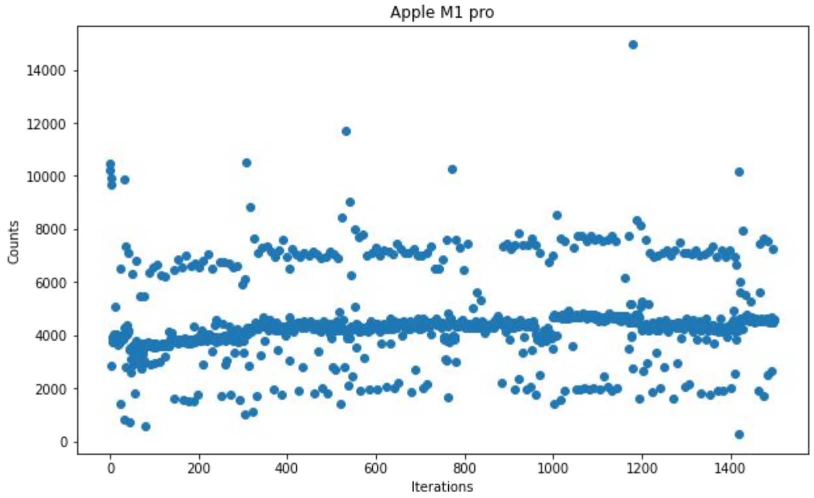
- Measurement: How many loops in 1000 clock cycles
- Not a direct property, but able to support the model classification
- However, the cycle of SharedArrayBuffer clock is dependent on the hardware, makes this not a uniform comparison.

```
for i in 0..iterations {
    counter = 0;
    end = clock.read() + 1000;
    while end > clock.read() {
        counter += 1;
    }
    data_array.push(DataPoint {
        x: (i),
        y: (counter),
    });
}
```

Results - Single Core Performance

1.





Average Count: 2712

Average Count: 4442

- We could not observe distinct results between different Intel CPUs.

Results - Page size

1. Page size benchmark did not give usable results
    a. Rust implementation of *SharedArrayBuffer* clock did not have high enough resolution

```
!!! Intel(R) Core(TM) i7-10510U - [0, 0]
!!! Intel(R) Core(TM) i7-8559U - [0]
!!! Intel(R) Core(TM) i7-5500U - [1024]
!!! Intel(R) Core(TM) i7-6700HQ - [0]
!!! Intel(R) Core(TM) i5-8257U - [0]
!!! Intel(R) Core(TM) i3-8100 - [1024]
!!! Intel(R) Core(TM) i5-10400F - [0]
!!! AMD Ryzen 7 4800H - [0, 0, 0]
!!! Intel(R) Core(TM) i7-10870H - [0]
!!! Intel(R) Pentium(R) CPU 4415Y - [0]

0 / 76

Process finished with exit code 0
```

Results - classification

	old acc	new acc	old F1	new F1
L1 cache size	100%	83%	100%	63%
L2 cache size	90%	**92%**	86%	**90%**
L3 cache size	20%	**42%**	14%	**37%**
L1 associativity	83%	**100%**	83%	**100%**
L1D TLB size	93%	90%	48%	**72%**
HTT availability	100%	100%	100%	100%
SMT availability	88%	**100%**	47%	**100%**
Boost availability	-	94%	-	48%
AMD vs Intel	100%	66%	100%	58%
ARM vs Intel vs AMD	90%	82%	64%	**82%**
M1 vs Rest	100%	100%	100%	100%
CPU model	36%	14%	27%	6%
CPU model with timings	30%	**46%**	21%	**43%**
Microarchitecture	40%	**45%**	26%	**29%**
Microarchitecture grouped	100%	73%	100%	36%

Results

1. Page size benchmark did not give usable results
    a. Rust implementation of *SharedArrayBuffer* clock did not have high enough resolution
2. For several metrics the classification on our results performs better than the original solutions
3. The features in the data from our solution resembles the features from the original one

Conclusions

1. We managed to reimplement part of the benchmarks in Rust
2. Properties classification on our data manages to do better for certain metrics
3. CPU model classification has worse performance (due to fewer benchmarks)
4. Support of browser WebAssembly with Rust is a mess
5. Implementing super high frequency clock directly in Rust may not currently be possible
    a. Not accurate enough to help determine page faults

Future work

1. Implement remaining benchmarks from the original suite
2. Collect more data for classification (e.g. with Amazon Mechanical Turk)
3. Find a way to implement more accurate clock
    a. Or just use raw WebAssembly clock and write benchmarks in Rust
4. Use LLVM's profile-guided optimization to further increase the accuracy
    a. Compilation optimizations, derived from dynamic execution of a program

Questions?

Thank you!