Browser-based CPU Fingerprinting with Rust and WebAssembly

Dan Plămădeală
Delft University of Technology
The Netherlands
d.plamadeala@student.tudelft.nl

Jakob Naucke
Delft University of Technology
The Netherlands
j.j.naucke@student.tudelft.nl

Cassie Xu
Delft University of Technology
The Netherlands
w.xu-6@student.tudelft.nl

Jim van Vliet
Delft University of Technology
The Netherlands
j.i.m.vanvliet@student.tudelft.nl

Konrad Ponichtera
Delft University of Technology
The Netherlands
k.j.ponichtera@student.tudelft.nl

ABSTRACT

CPU fingerprinting is a technique used to identify unique characteristics of a device's CPU based on its behaviour while executing specific tasks. In this paper, we explore the implementation of a browser-based CPU fingerprinting benchmark in Rust, compiled to WebAssembly (WASM), and compare it to an existing benchmark written in JavaScript and native WASM. We highlight the challenges faced in the implementation process and discuss the issues encountered with Rust and WASM. Our findings show that the data obtained from the Rust and WASM implementation are slightly better for some classifiers and worse for others. Still, more work is needed to refine the implementation and understand the limitations of this approach. This paper contributes to the ongoing research in the field of CPU fingerprinting, particularly in the context of using Rust and WASM for browser-based benchmarks.

1 INTRODUCTION

In addition to software vulnerabilities, hardware-based exploitation poses critical security risks to computer systems. In recent years researchers have discovered Spectre [4] and Meltdown [5], which both utilise the speculative execution of modern CPUs. In order to optimise performance, modern processors use branch prediction to guess the following instructions and preload them in memory. However, this may unwantedly leak private data to attackers. Rowhammer [2] is another microarchitectural vulnerability that could allow the attacker to flip bits inside the DRAM chip due to the physical proximity of memory cells. By repeatedly accessing a row of memory cells, it is possible to cause electrical interference that can flip the values of adjacent rows, which could be exploited and potentially allow an attacker to bypass security controls or execute arbitrary code.

In order to successfully mount such attacks, specific knowledge of the target processor is required, including cache hierarchy, cache size, and cache associativity, among other details. Knowing the CPU model would give the attackers all the relevant technical specifics.

In the paper *Browser-based CPU Fingerprinting* by Leon Trampert et al. [7], an approach to obtain the CPU model from the client side of an unmodified browser was designed and implemented. The framework was implemented in Javascript, with the performance-critical part written in WebAssembly, a portable binary-code format that runs in web browsers and outperforms equivalent Javascript code. Even though the performance-critical parts are written in raw WebAssembly, the framework still takes a relatively long time to retrieve all related data and causes noticeable resource usage on the victim's computer. Rust is a modern low-level programming language that emphasises performance, reliability, and safety. Rust can run in the browser after being compiled into WebAssembly. Instead of only a part of the benchmarking code being run in WebAssembly, we attempt to increase the precision and performance by running the entire code in WebAssembly, by recreating the benchmarking suite in Rust.

2 BACKGROUND AND RELATED WORK

2.1 Browser fingerprinting

Browser fingerprinting is a technique used to identify and track individual users across the web. This method relies on collecting a variety of information through the user's browser, such as the User-Agent header, installed plugins, and the returned value from several client-side APIs (such as HTML5 Battery Status, WebGL and Canvas, or AudioContext API). This information is combined to create a unique identifier (the "fingerprint") that can be used to track the user. Browser fingerprinting could be used for identification or as an additional safety feature during authentication [1]. Online advertisers also often use it to build user profiles for targeted advertising and malicious actors for identity theft or other nefarious purposes.

2.2 CPU fingerprinting

Like browser fingerprinting, CPU fingerprinting is a technique used to identify the specific model and configuration of a computer's CPU. It relies on collecting a range of information about the system's hardware, such as the CPU's clock speed, cache size, and instruction set architecture. CPU fingerprinting can be used for a variety of purposes, including software optimisation, hardware profiling, and security analysis. However, it can also be used to gather information for targeted attacks, such as exploiting known vulnerabilities in a specific CPU model.

When having native access to the system, such CPU properties could be quickly queried or measured, while it becomes difficult in a web setting. The sandbox environment in browsers gives the attacker only limited control over the instructions executed. The side-channel leakage does not indicate the underlying characteristic of the processors due to the noise generated by the browser engine itself.

2.3 Rust and WebAssembly

Just like in the case of native assembly, WebAssembly code can be written directly. Such an approach gives developers much control over executed low-level instructions, which is beneficial in settings where code is executed repeatedly in rapid succession, like in the case of benchmarks. However, the code written this way is significantly less maintainable and error-prone than that written in a high-level programming language. That is why, in the case of assembly and WebAssembly, the most common way is to use other languages like C/C++, Go, or Rust and have the compilation target configured for a specific processor architecture and operating system. Or, in the case of the browser's WebAssembly, to an operating system and processor agnostic target.

The fact that the compilation target is not bound to the processor nor to the underlying system means that several limitations are in place. We will address them in Section 3.1.

3 METHODOLOGY

In order to collect CPU-related information, measurement algorithms are running in the background of a website, retrieving the performance regarding several benchmarks. After obtaining the data, machine learning classification models are used to reveal the exact specifics of each benchmark. Subsequently, the results of multiple benchmarks are combined to deduce the specific CPU vendors and models. The original website was written in Javascript with partial WebAssembly. We reimplement the website in Rust and aim to compare the usability and performance of Rust and Javascript in browser-based side-channel attacks.

3.1 Benchmarks

The original code and dataset have been published by the paper's authors[1]. During their examination, we noticed an inconsistency between the code implementation and the paper's explanation regarding the benchmarks. The paper mentioned six benchmarks to identify the CPU model: the number of CPU cores, data cache sizes, L1D cache associativity, L1D TLB size, single-core performance, and page size. By examining the open-sourced code, we observe that 12 benchmarks were implemented. However, we could not clearly determine whether all of them were used for the data classification. This is due to the fact that in the repository that contains the code used for classification, some things are commented-out. In addition to the six benchmarks above are multi-core performance, memory latencies, hyper-threading availability, prefetcher presence, load buffer size, and `SharedArrayBuffer`-based time precision. Due to the difficulty of directly reading the low-level algorithms without any explanation and time constraints for this project, we only adopted the benchmarks explained in the paper.

Data cache sizes A CPU cache is a hardware cache that is directly available to the CPU in order to reduce the time of fetching data from memory. Modern CPU's cache is divided into multiple levels: L1, L2, often L3, and seldom L4. As in the original paper, we use the algorithm pointer chasing[2] to determine the data cache size. In short, we see the variance between different cache levels by timing the running duration through linked lists of various sizes. Typically, the lower the cache level, the lower the latency. In other words, running through linked lists of smaller sizes will yield lower results until one point, when the linked list does not fit into the L1 cache, and a cache miss happens, which suddenly increases the latency. This way, we can approximate the sizes of different cache levels.

Algorithm 1 Pointer Chasing to determine cache size

```
Input: sizes
Output: timestamps
  N ← $16 * 1024 * 1024$
  timestamps ← []
  for size in sizes do
      Prepare randomised circular linked list of size KB
      head ← head of linked list
      startTime ← getTimestamp()
      for 1 upto N do
          head ← head->next
      end for
      timeDifference ← getTimestamp() - startTime
      timeStamps.insert(timeDifference)
  end for
```

L1D cache associativity Most CPUs have the cache associativity of 8, except ARM CPUs; thus, this associativity will help to distinguish ARM CPUs from the rest. The algorithm for determining cache associativity is very similar to cache size. We use the same pointer-chasing algorithm but only aim to fill in a single cache set instead of the whole cache. We space the nodes in the linked list cache-size bytes apart for all memory accesses mapping into the same cache set. Once the total data size exceeds the associativity, we will observe an increased memory latency due to the L1 cache misses. We perform the test from 1 to 32 associativity.

L1D TLB size A translation look-aside buffer (TLB) is a memory cache that stores the recent translations of virtual memory to physical memory. Modern CPUs usually store the most recent 64 x 4KB pages, i.e., L1D TLB has 64 entries. The only deviation values are in ARM CPUs and older x86 CPUs. Thus, this benchmark is able to distinguish ARM CPUs from modern x86 CPUs. The algorithm used here is also pointer-chasing, as in the data cache size. We aim to fill in the L1D TLB and notice an increased latency after exceeding L1D TLB. We separate the node by page size. This way, an address translation is needed whenever the next point is visited, and a new entry in TLB will be created. We tested between 2 and 128 entries. The page size is 4KB since all other modern CPUs have this default value except for Apple M1 CPUs (16KB).

Single-core performance The CPU performance benchmark measures how many calculations the CPU can perform in a given time cycle. Since different CPU models have different performances, this will be an indicator for the classification. We increase the clock cycle by 1000 and observe how many loop iterations it could enter. We repeat this step 500 times in a round and have *three* rounds in total.

Page size. The page size benchmark measures the time taken to execute pointer read instructions to detect page faults and, from this, determine the page size. The actual page size can then be

[1]https://github.com/CISPA/browser-cpu-fingerprinting
[2]https://github.com/afborchert/pointer-chasing

used to differentiate between different CPU families [3]. Since a single pointer read instruction takes little time, it is required to have a timer with high resolution. In the paper *Browser-based CPU Fingerprinting* by Leon Trampert et al. [7], they managed to implement this as described by Michael Schwarz et al. [6]. How we implemented this timer in Rust is explained in section 3.2.2.

3.2 Implementation

The primary design choice during development was trying to make our solution replicate the original one as closely as possible while still providing a maintainable structure of the Rust project.

3.2.1 *Technological Stack.*

We replicated the original solution by developing two applications: a backend and a frontend. Both were written in Rust. The backend uses the Actix[3] web framework to expose RESTful API, and store received benchmark results in the PostgreSQL database using SQLx[4] toolkit.

The frontend was developed with the use of Yew[5] framework, which allows for creating single-page web applications (SPA), in a similar way it is done in React or Angular. The logic responsible for rendering the page is written in Rust and compiled into a WebAssembly binary, which is then loaded by the served HTML page and takes care of building the page content.

The concept of web workers is an essential element of the web browser stack in developing benchmarks. The Javascript execution engine within the browser is single-threaded by design. Workers allow developers to load Javascript code asynchronously and execute it in a separate thread. The loaded code can also execute WebAssembly binaries. Yew allows for easy development and integration of web workers into the lifecycle of user interface components, allowing one to develop responsive web applications without blocking the main thread.

Both developed components, as well as the PostgreSQL database, were containerised with Docker for easy deployment.

3.2.2 *High-precision Clock.*

Web browsers do not provide high-precision clocks to developers in order to mitigate timing attacks and fingerprinting. Common methods for obtaining timestamps, such as through the Performance API[6], intentionally provide imprecise results, with the highest level of accuracy being 5 microseconds. This makes it impossible to use these APIs to obtain time differences for the majority of benchmarks, which require nanosecond-level precision.

Michael Schwarz et al. [6] devised a way to work around that limitation by creating a web worker, which increments a number in an infinite loop. The way of exchanging data between the main thread and the workers is to send and receive messages. This creates an overhead, which would render the clock results infeasible to use. Instead, the number is stored in `SharedArrayBuffer`[7] - an object holding a raw binary data buffer, which can be shared between the main Javascript thread and the workers. Since the data is stored in the same place in memory, reading its content is much faster than performing the cycle of querying the clock worker for its state and waiting for the response.

Yew uses the gloo[8] toolkit under the hood, which provides an opinionated implementation of worker spawning. The approach has proven itself suitable for creating workers responsible for the benchmark execution - each execution of the benchmark sent a message to Yew, allowing for updating the progress bar and starting the next benchmark. However, one of its limitations is that all messages must be serializable and deserializable, making it impossible to pass the `SharedArrayBuffer` to the newly instantiated worker since it is not a serializable object. This means that we had no way of creating the `SharedArrayBuffer` in a way that would be accessible to both the benchmark and the clock worker. Consequently, this forced us to implement spawning the clock worker using raw `web_sys` Javascript bindings, which required a different way of handling the message lifecycle than in Yew. First, the clock worker is instantiated with the script that loads its WebAssembly binary. Unlike pure Javascript workers, WebAssembly ones cannot queue the messages while the worker is still starting. It means we are not able to send the newly-created `SharedArrayBuffer` straight away since there is a chance that the message will be lost. Instead, we wait for the clock worker to send the message about it being ready to the spawning thread. After receiving that message, we know that the clock worker is able to handle the messages, so we send the message with the buffer. Upon receiving it, the clock worker sends another message to the spawning one, informing it that the clock has started. Then it begins incrementing the value in the `SharedArrayBuffer` in an infinite loop. Once the spawning worker receives the start message from clock one, it can proceed with running the benchmark. The clock worker is then forcibly terminated after the benchmark finishes executing.

Integration with Javascript objects like `SharedArrayClock` can be done from Rust through the web_sys crate[9], which uses the WebIDL[10] standard to generate the Rust bindings. However, the crate is missing certain elements which are present in the WebIDL. For instance, there is no way to obtain Rust representation of the Performance API object from the web worker's scope. This means that we had to rely solely on the `SharedArrayBuffer`-based clock without any way to access coarse real-time clock data or to measure how many clock cycles are executed within a constant timespan.

3.2.3 *Benchmark Implementation.*

Cache size: Our Rust implementation of the cache size implementation differs slightly from the original paper's Javascript + WASM implementation. Due to the nature of Rust, we could not work with direct pointers that would point to the head, tail and next element directly. Instead, we created a simple vector containing `size` elements, where `size` is the number of elements required to take up a given size in bytes in memory. For example, for the array to occupy 1024 bytes, it would have to contain $\frac{1024}{\texttt{sizeof(usize)}}$, where `usize` is the type that the array contains. With one `usize` occupying 4 bytes on a WASM target, the array ends up having 256 elements.

[3]https://github.com/actix
[4]https://github.com/launchbadge/sqlx
[5]https://github.com/yewstack
[6]https://developer.mozilla.org/en-US/docs/Web/API/Performance_API
[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

[8]https://github.com/rustwasm/gloo
[9]https://crates.io/crates/web_sys
[10]https://developer.mozilla.org/en-US/docs/Glossary/WebIDL

Next, in order to randomise the propagation of the linked list, we store all the indices of the array in a separate array, after which we shuffle that array. Next, we execute a warmup step to ensure that most of the data is contained in the CPU cache. After that, we run through the indices and compute the passed time. We also make use of the `black_box`[11] function, which hints to the compiler not to apply any unneeded optimisations to the benchmarking code.

TLB size: The central part of the TLB size implementation is the same as the cache size benchmark. The same array has been built; however, we only select the indices every $\frac{PAGESIZE}{\texttt{sizeof(usize)}}$ and build the linked list from the elements at these indices. In this way, each node is separated from another node by a page size, and visiting the next node requires an address translation. We tested with 2, 4, ...128 nodes and the time cost to run each experiment is returned together with the node amount.

Cache Associativity: The foundation algorithm of this benchmark is still the pointer-chasing algorithm and has a similar implementation to the data cache size and TLB size benchmark. We only select the indices every $\frac{32}{\texttt{sizeof(usize)}}$ and build the linked list from the elements at these indices. We tested with 1, 2, ..., and 32 nodes.

Single-core performance: The original implementation used `performance.now()`, which is a Javascript API that returns the current timestamp in milliseconds. In our implementation, we no longer have access to this clock; we still use the `SharedArrayBuffer` high-precision clock as in other benchmarks. However, the cycle in this clock is related to hardware and thus is not uniform across different CPUs, but we still can notice the difference in performance from the loop count in 1000 cycles.

Page size: The page size benchmark repeatedly measures the time it takes to execute pointer read instructions in a preallocated block of memory. Each iteration increases the pointer offset by 4. Each individual pointer read instruction is timed separately to detect page faults, which allows for determining page size. This technique can be used to differentiate between different CPU families [3].

To actually determine the page size from the collected measurements, we use a different way of classifying than most other benchmarks. Namely, a basic algorithm is used which works as follows[12]. It starts by iterating through the measurements in steps of 32 while ignoring all the other collected measurements. Then the ten largest measurements are iterated through two at a time. For each pair, we calculate the *gcd* of the memory offsets and check if it is greater than 1023 and has a power of 2; if so, it gets added to a list. After iterating through all the pairs, the *gcd* of all the elements in the list is returned as the predicted page size.

3.2.4 *Release Optimisation.*

Rust provides two primary modes of build optimisation - debug and release. Debug mode produces relatively large binaries with debugging symbols present and most optimisations disabled. Release mode strips them off and produces highly-optimised binary for use in production.

Since we aim to replicate the results of a project in which the benchmarks were implemented in pure WebAssembly, we need to optimise the produced binaries as much as possible. This includes enabling the highest code optimisation level, striping debug symbols, disabling debug assertions, and integer overflow checks. The default behaviour of unwinding the stack upon panic is replaced by terminating the process, which removes additional instructions. Finally, link time optimisations are performed across all the crates within the dependency graph, not just the one with a project.

In our testing, the binary compiled in the debug mode not only executes much longer but also leads to results from which no meaningful features can be identified. The comparison between debug and release optimised is shown in 1 and 2.

Figure 1: Optimisation comparison in cache size benchmark

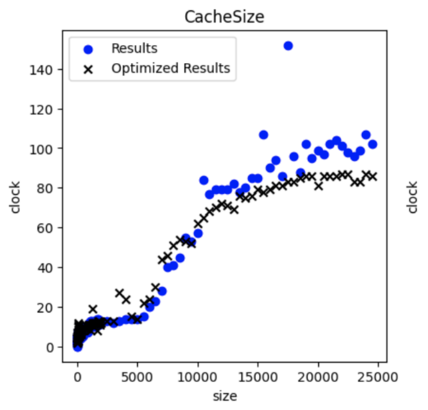
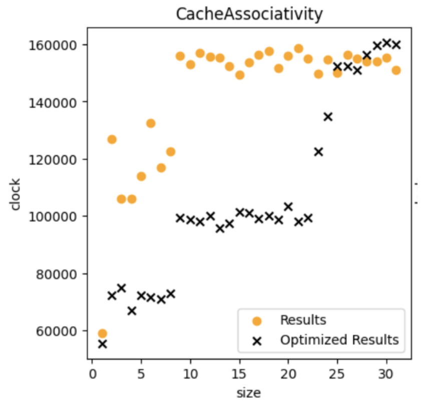


Figure 2: Optimisation comparison in cache associativity benchmark



3.3 Classification

For the classification part, we have used the existing infrastructure initially set up by the original paper's authors while modifying it to support our data. In the original infrastructure, the cache size classifiers differentiate between large cache sizes and small cache sizes and treat them differently. Some classifiers take as input features only one of the two, while others take both cache sizes' benchmarks

[11]https://doc.rust-lang.org/stable/std/hint/fn.black_box.html
[12]https://github.com/kponichtera/browser-cpu-fingerprinting-rs/blob/main/classification/pagesize.py#L15

results. Moreover, most classifiers in the original infrastructure take as input feature the number of cores benchmark. We did not implement this benchmark; thus, we had to remove it as a feature when training the classifiers based on our data. Furthermore, we have entirely omitted the classifiers that take as input features the results from benchmarks that we did not implement.

We have trained the classifiers on both the data collected from the original implementation ($n = 89$) and the data collected from the new implementation ($n = 110$). As mentioned earlier, the training with the data collected from the original implementation included the "number of cores" benchmark, while the data collected from our implementation did not. This may give an advantage for some classifiers, where this feature may improve the accuracy of the classifier.

3.4 Datasets

We have deployed our solution[13], and the original solution[14] on the server. In order to test them, we took the same approach as the original paper's authors and used pseudo-crowdsourcing. The links were then provided to various groups of people with varying levels of technical expertise. We managed to collect 110 results for our solution and 89 for the original one. As can be seen, approximately one-fourth of the participants in our benchmark either missed the initial instructions or did not bother to finish the second one. One could also argue that the counter in the original implementation, which requires the user to click a button every 30 seconds, may have been an annoying factor that led to people not completing the benchmarks.

4 RESULTS

4.1 Benchmark Comparison

We first mention a few benchmarks and give the comparison and discussion. Then the numerical classification result is given and discussed.

4.1.1 Cache Size. The cache size benchmark aims to determine the size of the L1, L2, and L3 size. Our implementation is able to obtain these features accurately. We can observe a similar data trend in the original implementation and our implementation, as seen in figure 3 and figure 4. The two turning points are visible, indicating the point jumping from the L1 cache to the L2 cache and from the L2 cache to the L3 cache.

4.1.2 Single Core Performance. The single core performance measure how many loops it can enter within the given time cycle. In our implementation, we measure 500 x 3 iterations, with 1000 clock cycles for each iteration. In our implementation, we no longer have access to the javascript performance.now() clock and are only able to use the `SharedArrayBuffer` clock. Since this clock is dependent on the hardware itself, it is not a uniform clock and loses part of the accuracy in the measurement. We are able to compare the performance partially but cannot have the full-order result between different CPU models.

The data from an Intel Core i5-8257U CPU and an Apple M1 Pro CPU is shown in figure 5 and figure 6. The average loop count

[13]https://benchmark.ponichtera.dev/
[14]https://benchmark2.ponichtera.dev/

Figure 3: Cache size results in pure WASM implementation

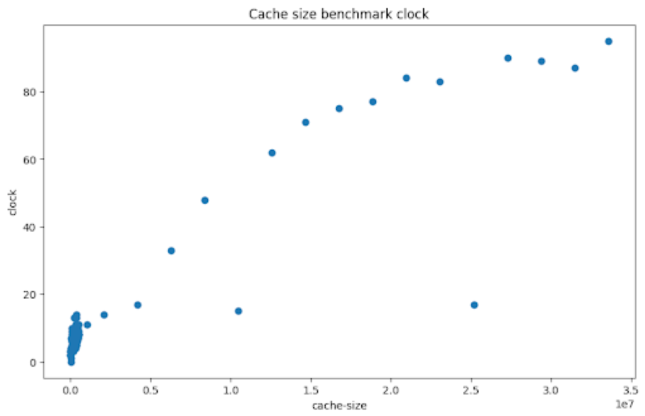
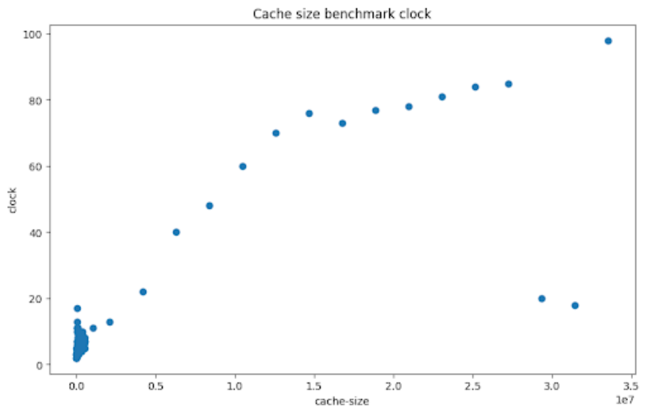


Figure 4: Cache size results in Rust implementation



for each CPU is respectively 2712 and 4442, which is aligned with the expected performance of the two CPUs. We can observe that an Apple M1 Pro outperforms Intel CPUs but could not observe distinct results between different Intel CPUs.

4.1.3 Page Size. Page size is a benchmark that requires a very high precision clock to receive usable data. Due to the compromised precision of our clock, this benchmark could not contribute to the classification. This is clear to see from the page size classification algorithm results because it does not successfully predict a single correct page size out of 76 distinct CPU models.

4.2 Classification result

The results we collected were exported from the database, and the metrics of the trained classifiers were computed. We present the accuracy and F1-score of all the tested and evaluated classifiers in Table 1. It can be seen that for 6 out of 15 classifiers, the data we have collected has resulted in much better accuracy and F1-score. For the "ARM vs Intel vs AMD" classifier, we managed to achieve a superior F1-score. It is important to mention that for the L3 cache

Table 1: Accuracy and F1 results for different classifiers, trained on original data (old) and our data (new). The classifiers, which yielded a better result with our data, are highlighted in bold.

	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%

Figure 5: Single core performance of Intel i5-8257u

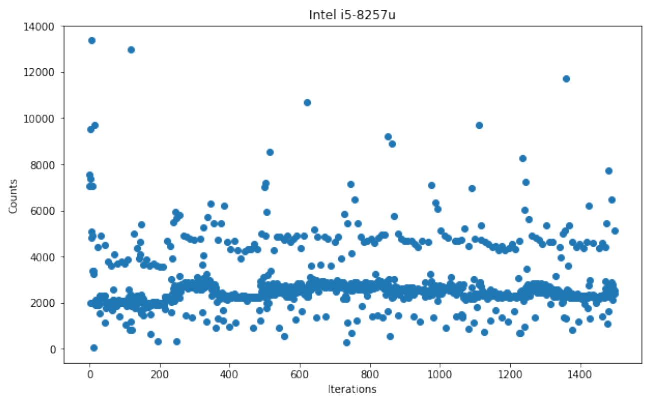
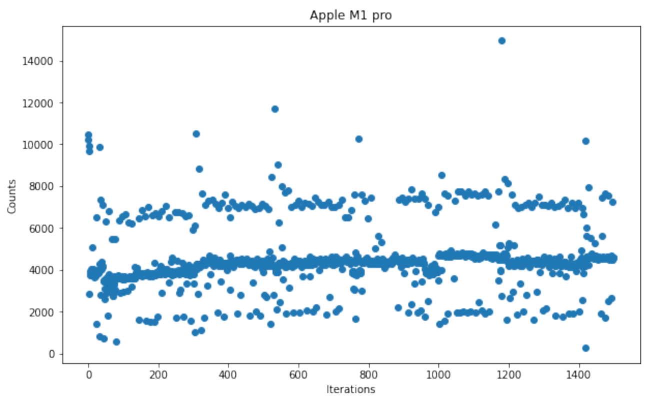


Figure 6: Single core performance of Apple M1 Pro



size, the classifier trained on our data yielded twice the performance of the data collected using the original implementation.

Considering the fact that the page size benchmark did not return any usable results, we missed the opportunity to have that as an additional feature that would have learned the model, hopefully, a bit better. Furthermore, considering that our data does not contain the "number of cores" benchmark, the performance of the new classifiers may have suffered for this reason.

5 DISCUSSION

Both the original paper and our implementation show that it is feasible to predict the CPU microarchitecture within a browser sandbox, which poses a security risk for possible microarchitecture-related attacks, such as Spectre and Meltdown. In the experiment, the user was asked not to perform any other tasks during the meantime and leave only this website tab open. This is not a very realistic scenario in the real world. We could expect that a typical user will always have other activities going on in the background, which contribute significant noise to affect the accuracy of CPU profiling. The multiple layers between the hardware and the browser also add noise to this attack.

The availability of a high-precision clock inside the browser sandbox is critical to perform such timing attacks. A mitigation strategy from the browser's perspective is disabling such clocks or only making them available with specific headers present from the client request (making it still possible for specific applications).

6 CONCLUSION

In this project, we developed a suite of WebAssembly benchmarks based on the existing research in the field. The developed benchmarks analyse different components and behaviours of the CPUs. In addition to that, we created a system for collecting the results with the use of a homogeneous technological stack based on Rust.

Through crowdsourcing, we obtained a dataset in a similar way as the authors of the original project did. The results collected by our benchmarks are characterised by a similar distribution of data points to the ones from the original solution.

Given enough time and resources, we think this project could be taken to the next level with much better results. Future work could attempt to further increase the accuracy of benchmarks by using profile-guided optimisation[15], where the compilation optimisations are derived from the traces, collected from dynamic execution of the program. It would be beneficial also to implement the rest of the benchmarks from the original suite, which will provide classifiers with additional information, allowing them to distinguish processors more accurately. More data could also be collected and used for classification by making use of large-scale crowdsourcing services, such as Amazon Mechanical Turk[16].

REFERENCES

[1] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. [n. d.]. FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2021) *(Lecture Notes in Computer Science)*, Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves (Eds.). Springer International Publishing, 237–257. https://doi.org/10.1007/978-3-030-80825-9_12

[2] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. [n. d.]. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2016) *(Lecture Notes in Computer Science)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, 300–321. https://doi.org/10.1007/978-3-319-40667-1_15

[3] Intel Corporation 2023. *Intel 64 and IA-32 Architectures Optimization Reference Manual - Revision 46.* Intel Corporation.

[4] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[5] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[6] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Aggelos Kiayias (Ed.). Springer International Publishing, Cham, 247–267.

[7] Leon Trampert, Christian Rossow, and Michael Schwarz. 2022. Browser-based CPU Fingerprinting. In *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III.* Springer, 87–105.

[15] https://doc.rust-lang.org/rustc/profile-guided-optimization.html
[16] https://www.mturk.com/