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 are unaffected by current side-channel and browser fingerprinting mitigations, and can thus be used for more targeted attacks and to increase the entropy in browser fingerprinting.

Keywords: Microarchitecture · Fingerprinting · Side Channel · JavaScript

1 Introduction

Knowing the CPU of a target system, or its properties, may allow attackers to craft severe tailored attacks. For example, research in recent years has revealed multiple critical CPU vulnerabilities related to speculative and out-of-order execution [16, 20, 4, 32, 35, 40, 27]. These transient-execution attacks [16, 20, 4] leverage CPU-specific inner details to leak data in the same process or even across security domains [4]. Likewise, Rowhammer [15] is a vulnerability in modern DRAM that leverages knowledge of a CPU's cache architecture to flip bits in memory without accessing them. While such attacks are already a security risk when mounted in native code, they reach the masses when successfully launched in a browser. In fact, despite the limitations of the restricted browser environment, previous research mounted Rowhammer [11] and the transient-execution attacks Spectre [16, 39, 37, 1], RIDL [32], and ZombieLoad [6] in the browser.

All these attacks have in common that they require knowledge of specific properties of a CPU. However, attackers cannot easily obtain the CPU model (or CPU properties) in a Web setting. Consequently, in contrast to native attacks, browser-based attacks face several challenges. The sandboxed code execution does not give the attacker full control over the instructions executed.

The JavaScript and WebAssembly code is just-in-time-compiled by the browser engine to the instruction set architecture (ISA) of the particular CPU. It is often difficult for an attacker to distinguish the noise naturally generated by the operating system and other processes from the actual target data. This is made more difficult because these attacks do not deliver any relevant data at all on CPUs that are not vulnerable. Moreover, as side-channel leakage rates for these browser-based implementations are often low, with typically a few bits per second [32, 6], knowing whether the leaked values are actually sensitive data is tedious. Identifying potential victims or even choosing the most effective attack for a concrete target is thus valuable to an attacker. Likewise, a successful Rowhammer attack requires knowledge of cache architecture and replacement policies of the different cache levels, as implied by the CPU model or family. Knowing these parameters simplifies generating code for memory accesses that purposely miss the caches and can thus be used in a Rowhammer attack. All these vital CPU properties can be trivially extracted from knowledge of the CPU model in a native attack setting. However, lacking knowledge of the CPU model from within the browser often complicates—if not even impedes—an attack.

In this paper, we explore if attackers can determine attack-relevant CPU properties from within the browser. To this end, we present six benchmarks designed to reveal CPU-specific properties and behaviors. Our benchmarks are implemented in JavaScript and WebAssembly and run in unmodified browsers. We target microarchitectural elements relevant to microarchitectural attacks, including the cache and the TLB. In addition, they also reveal the number of CPU cores and profile the performance of the CPU in single- and multi-threaded scenarios. All these properties can be inferred reliably, even with state-of-the-art mitigations enabled in Mozilla Firefox and Google Chrome. Moreover, the benchmarks are independent of the operating system and instruction set architecture. We optimize all benchmarks to work on x86 and ARMv8 CPUs, including low-end devices such as smartphones.

To evaluate the efficacy of our benchmarks, we conduct a study over 834 participants to collect information from 297 CPU models in the wild. Based on this data set, we achieve accuracies of up to 95 % for determining microarchitectural properties such as the L1D size or associativity, or the used page size. Moreover, when combining the microarchitectural properties, we can expand our knowledge to predict the CPU vendor at 97 % accuracy and even identify the exact CPU models and microarchitectures at about 65 % accuracy.

First, our results show that these benchmarks can be used to infer properties useful for microarchitectural attacks. Second, by combining the benchmarks, they can also be used for hardware fingerprinting, e.g., to track users across websites. Hence, our benchmarks can augment state-of-the-art browser fingerprinting, which mainly focuses on the software side to enable web tracking [18]. Our evaluation also shows that current browser-fingerprinting mitigations do not impede the generation of our CPU fingerprints. We publish our data set and benchmarks as open source.[1]

[1] `https://github.com/CISPA/browser-cpu-fingerprinting`

Contribution To summarize, we make the following contributions.

1. We show 6 benchmarks to infer microarchitectural properties from the browser.
2. We evaluate our benchmarks on a set of 834 CPUs found in the wild, showing an accuracy of up to 95 % for inferring microarchitectural properties.
3. We demonstrate that combining these properties can reliably detect CPU vendors, models, and microarchitectures.
4. We show that fingerprinting mitigations do not prevent our benchmarks.

2 Background

A device fingerprint is information collected about the hardware or software of a particular device, typically for the purpose of authentication or identification.

Browser Fingerprinting For browser fingerprinting, a site uses a client-side scripting language to reveal characteristics of the browser, software, or hardware of a system. The seminal work by Eckersley [7] investigated browser fingerprinting using the information transmitted by HTTP, such as the User-Agent header, and information accessible via the browser API exposed to client-side scripting languages. In the past, several browser-provided APIs [23], including the HTML5 Battery Status [24], WebGL and Canvas [19, 22, 5], or AudioContext API [8] have been used to craft accurate fingerprints. Mowery et al. [21] exploit performance differences of the different JavaScript engines used by different browsers and the allowlist of the popular NoScript plugin. Schwarz et al. [33] presented JavaScript Template Attacks, an automated framework to detect differences in browser engines caused by the surrounding environment.

Browser-based CPU Fingerprinting Recent works also investigate hardware fingerprinting of the CPU. Sanchez-Rola et al. [31] observe the accumulated execution times of common functions, such as string manipulation functions or cryptography functions from the HTML5 Cryptography API. This allows the identification of a concrete CPU with adequate accuracy. Saito et al. [29] proposed multiple side-channel related methods to infer the presence or absence of different Intel CPU features, such as Advanced Encryption Standard New Instructions (AES-NI) and Intel Turbo Boost Technology. Saito et al. [30] also proposed algorithms to infer the presence or absence of Hyper-Threading Technology (HTT) and Streaming SIMD Extensions 2 (SSE2). With many of these Intel-specific extensions and technologies omnipresent in modern Intel CPUs, these algorithms generate few distinguishing features for modern Intel CPU models. Furthermore, the proposed features do not necessarily allow to distinguish Intel CPUs from CPUs distributed by other manufacturers since other manufacturers provide their own functionally-equivalent technologies.

3 Methodology

In the following, we present 6 benchmarks to reveal information about the CPU. The output is used to infer microarchitectural characteristics such as the cache

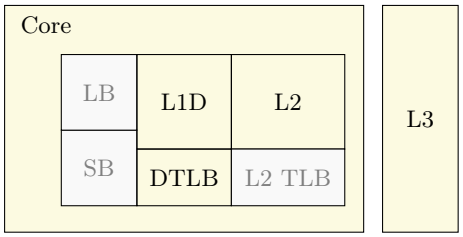

Fig. 1: Parts of the CPU targeted by our benchmarks are highlighted in yellow.

size, which provides information for microarchitectural attacks. Furthermore, by combining multiple benchmarks, we can generate CPU model fingerprints, an extension of device fingerprints, that can reidentify a CPU model or, at the minimum, the CPU vendor. These can be used, e.g., as additional entropy for tracking users on the web. Figure 1 provides an overview of CPU parts that are targeted by our benchmarks. In particular, we have a benchmark to determine the number of cores, the sizes of the different data cache levels, the associativity of the L1D cache, and the size of the L1D TLB. While not shown in Figure 1, we additionally present benchmarks to determine the performance of a single core and the page size.

3.1 Benchmarks

Our benchmarks are written in JavaScript and WebAssembly. The latter is used whenever compiler optimizations could impair the functionality of a benchmark or when highly-optimized instructions are required to ensure the highest level of performance. We operate in a cross-origin-isolated environment that automatically reenables features that have been disabled as a response to microarchitectural attacks, e.g., `SharedArrayBuffer`. Note that this is a server-side setting that does not require any changes in the browser, i.e., benchmarks run in unmodified off-the-shelf browsers if the user visits the attacker's website. All benchmarks, except the single-core performance and the cores benchmark, use a `SharedArrayBuffer`-based timer [36, 9].

Number of CPU Cores The number of CPU cores is typically a power of two, though 6, 10, and 12 have also been prominent in recent years. Many modern CPUs implement Simultaneous Multithreading (SMT) to effectively double the number of threads that can run in parallel. The ability to distinguish actual physical cores from virtual cores based on SMT further allows clustering of CPUs by the number of physical or virtual cores. Both aspects have previously been algorithmically examined for Intel CPUs by Saito et al. [29, 30]. While JavaScript is a single-threaded language, `Web Workers` are a browser feature that allows running scripts in the background without blocking the main UI thread. While the browser API may feature the `navigator.hardwareConcurrency` read-only property, this does not necessarily reflect the number of available cores.

Algorithm 1: Cache Size Benchmark

```
Input: sizes
Output: timestamps
  N ⇐ 16 * 1024 * 1024
  timestamps ⇐ []
  for size in sizes do
    Prepare randomized 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
```

Our benchmark starts more workers than actual threads available, which interferes with the effective multithreading of the workers. Some workers have to operate sequentially, taking turns on a shared hardware thread or even waiting for one worker to finish before starting. This, in turn, increases the average execution time of each worker and the time it takes for all workers to finish. Each iteration, we start `N` Web Workers that each perform the same independent task. If we do not create more workers than hardware threads available, the execution time of each worker is roughly the same. The time it takes all workers to finish their task noticeably increases once the number of workers exceeds the number of available hardware threads. In addition, we can also observe that once the number of workers exceeds the number of available physical cores due to the negative performance impact of microarchitectural components shared between two co-located logical cores. The output of this benchmark is a list of timestamp differences for all `N` from the set of even numbers up to 32.

Data Cache Sizes Since ARM CPUs often only feature two cache levels, while x86 CPUs typically feature three, the number of cache levels allows distinguishing ARM and x86 CPUs with relatively high accuracy. Furthermore, the L2 cache size allows determining the vendor of a recent x86 CPU, and the L3 size allows distinguishing CPU models. This is particularly interesting since many microarchitectural attacks are ISA-specific or even vendor-specific. The class of MDS vulnerabilities [32, 35, 3], for example, only affects Intel CPUs.

We frequently access memory and measure the latency. Over time, we increase the size of the frequently-used memory. First, the used memory easily fits inside the L1 cache, and we observe fast access times. Once the size approaches the limit of the L1 cache, we statistically observe an increased number of L1 cache misses, increasing the average memory latency. Likewise, the latency increases again if the limit of the L2 and then the L3 is reached. This reveals each cache level and their respective sizes. To eliminate the noise generated by the various

prefetchers, we rely on pointer chasing using a randomized linked list. Here, each memory access determines the subsequent pointer to be dereferenced. This generates a series of loads that each depend on the previous load, thus enforcing serialization. We use a circular randomized linked list, where we perform a fixed number of pointer advances (Algorithm 1). Over time, we increase the memory allocated by the linked list. The output of the benchmark is a list of timestamp differences that are sampled at different list sizes. We test 273 potential sizes from 2 KB to 32 MB. To accurately determine L1 cache sizes, the granularity for smaller sizes is finer than for larger sizes.

L1D Cache Associativity Most modern x86 CPUs have the L1D cache associativity of 8 [14]. Thus, this feature allows us to recognize ARM CPUs where the associativity differs. The general idea behind the algorithm is the same as for determining the cache size. Instead of filling the cache, we now only aim to fill a single cache set. To fill a cache set, we create a randomized circular linked list as in Algorithm 1. By spacing the nodes of our linked list cache-size bytes apart, all memory accesses map to the same cache set, as the cache size is always a multiple of the cache associativity. As long as the accessed memory fits into the same cache set, we observe repeated L1 cache hits. After exceeding the associativity, the set can no longer accommodate all memory locations, and we thus observe an increase in execution time due to an increase in L1 cache misses. The output of the cache associativity benchmark is a list of timestamp differences sampled at different linked list sizes. We test for associativities between 1 and 32.

L1D TLB Size CPUs usually have an L1 TLB that stores translations for 64x 4 kB pages [14], with only older x86 (e.g., Intel Nehalem, 2008) and ARM CPUs deviating from this value. Thus, this property allows distinguishing modern ARM CPUs from modern x86 CPUs fairly accurately. The general idea behind the algorithm is to fill the L1 TLB. We use Algorithm 1 but space the nodes of our linked list at least page-size bytes apart. Hence, all memory accesses map to different pages and thus require an address translation. While the number of pages used by our linked list does not exceed the number of L1D TLB entries, we observe relatively low execution times due to repeated L1D TLB hits. Once this number is exceeded, we statistically observe more L1D TLB misses and thus an increase in execution time. As almost all modern CPUs have a standard page size of 4 kB we hardcode this value to eliminate this error source. In our data set, this hardcoded value only negatively affects this benchmark on the Apple M1 chip, as it features a standard page size of 16 kB [12]. The output of the L1D TLB size benchmark is a list of timestamp differences that are sampled at different linked list sizes. We test for TLB sizes between 2 and 128 entries.

Single-Core Performance To estimate the single-core performance of the CPU, we increment a counter for the duration of 1 ms (measured using the `performance.now` function). We repeat this step for a fixed number of iterations and collect the counter's value after each iteration. To better observe the

difference between boost and base frequency, we repeat this process three times and wait for 100 ms between each time to reset the frequency.

Page Size The page size is usually determined by the processor architecture [14], with a default of 4 kB for x86 CPUs in laptops and desktops. In our data set, only the Apple M1 has a different default page size of 16 kB. The use of certain page sizes directly reveals the CPU family [14]. As the resolution of our `SharedArrayBuffer`-based timer is high enough to differentiate cache hits and misses, it also allows detecting page faults. Iterating the memory in 256 B strides while measuring the access times detects page faults due to noticeably higher execution times [11, 36]. We take the offsets of the 10 highest timing differences and iterate these offsets in pairs. If the greatest common divisor of an offset pair is greater than 1023 and a power of two, the greatest common divisor is added to a list. This step eliminates the majority of outliers.

3.2 Data Set

Study To obtain a real-world data set for further investigation, we conduct a voluntary study. We implement all benchmarks on a website that allows collecting measurements from web clients. Before starting the execution of our benchmarks, the participant is informed about and has to agree on the purpose of the study. The collected data is then stored in our database. This data only contains the output of the 6 benchmarks, the User-Agent value to determine the browser version and the self-reported CPU model string. It does not suffice for the identification of a person or device as the k-anonymity (i.e., the number of persons using a given CPU model in the global population) is sufficiently large. In addition, we implemented strict access control and removed all personally identifiable information. The participant is instructed to connect a mobile device to a charger, to leave this tab running in the foreground and to ensure a low system load during the study. This website was first presented to professional computer scientists. During this phase of the study, we obtained benchmark results from about 120 participants. To collect a representative set of measurements, we use the crowdsourcing marketplace *Amazon Mechanical Turk (MTurk)*[2] to distribute our study to a larger audience. The setup of the study remains the same, except for the addition of a mechanism to ensure the study is run in the foreground. This mechanism simply consists of a button, that has to be pressed at least every 30 s.

Structure The final data set consists of benchmark results from 834 participants featuring 297 different CPU models. Our set exclusively consists of CPUs currently used in desktops and laptops, and two AWS Graviton CPUs. The majority of benchmarks are collected on Google Chrome version 91-93 and Firefox version 89-91. Almost 75 % of the CPUs in the set are manufactured by Intel.

[2] `https://www.mturk.com/`

The remaining quarter contains AMD CPUs and 21 ARM CPUs. 19 ARM CPUs of our data set are recently released Apple M1 chips. Our data set reflects the consumer market shares of x86 CPU vendors for the past few years. Optionally, users could leave out certain parts of benchmarks requiring a large amount of memory, such as the cache size benchmark for cache sizes larger than 1 MB if they used a low-end device. Additionally, the collection of benchmark execution times was introduced later in the study, such that they are not always available. Thus, not all data set entries may be used in all scenarios. Each classification only considers entries that feature the required benchmark results.

3.3 Classification

The outputs of our benchmarks are mostly a collection of measurements, not single values. Hence, we rely on machine learning algorithms to perform the actual classification of the different CPU properties. We also use the combination of properties to further detect CPU vendors, models, and microarchitectures.

Algorithms We use three supervised learning classifiers as implemented by the *scikit-learn* [26] Python module. In particular, we use the `KNeighborsClassifier` (KNN), `SVC`, and the `MLPClassifier` (MLP). These classifiers were chosen as they showcase varying levels of complexity, after experimenting with a variety of popular readily-available classifiers. Simple thresholds and statistical methods did not lead to adequate results, we suspect the high levels of noise to be responsible for this, and are thus not discussed further. Each classification scenario may use a different balanced subset of our data set for testing and training to give equal priority to all classes. We detail this balancing further in the following sections. All subsets consist of labeled samples. A labeled sample uses the results of one or more benchmarks as features. As the output of a benchmark is a list of timestamps or timestamp differences, the list directly represents the list of features. If multiple benchmarks are used, their results are concatenated. The label can be the vendor, microarchitecture, model, or property.

Property Classification To evaluate our benchmarks for their ability to discriminate between the different instances of targeted CPU features, we introduce property classifiers. These classifiers are each trained based on data from a single CPU property, such as the number of threads. Each property classifier is trained and tested only using the results of the benchmark designed to reveal information about this property. It is important to note that the distribution of properties is rarely balanced, as the dataset tends to feature recent CPU models, and vendors often reuse established parts of the microarchitecture. To counteract this bias, our property classifiers operate on balanced subsets of the dataset.

In addition to specific properties, we can also use a combination of properties. While single properties might not be unique, a combination of properties reduces the set of possible CPU models with such a combination. Hence, by combining multiple properties, we can infer higher-level information, such as the CPU vendor, microarchitecture, and model, as described next.

Vendor Classification For the classification by CPU vendor, we use the results of the cache-size benchmark and the TLB-size benchmark. We first examine the capability of distinguishing two vendors producing CPUs of the same ISA. Since the vast majority of our data set consists of x86 CPUs, we aim to distinguish CPUs manufactured by Intel from CPUs manufactured by AMD. As our data set generally features more Intel CPUs, we use a balanced subset of it for training and evaluating the machine learning algorithms. The subset features data of 165 Intel and 165 AMD CPUs.

Secondly, we extend the classes to include all vendors present in our data set. As the data set does not contain a large variety of ARM CPU vendors, we decided not to distinguish between specific ARM manufacturers but rather regard the ARM ISA as a group. The balanced subset used includes 21 AMD CPUs, 21 ARM CPUs, and 21 Intel CPUs.

Microarchitecture Classification The second-coarsest clustering of CPUs is based on their microarchitecture. The subset used for training and testing contains 16 different microarchitectures as classes. Each class contains at least 17 and a maximum of 25 samples to keep the influence of the imbalance as small as possible without reducing the size of our set drastically. The final set contains the data of 368 CPUs. For this classification, we again use the results of the cache-size benchmark and the TLB-size benchmark and also add the results of the cores and cache-associativity benchmark.

As many microarchitectures are, however, based on the same base microarchitecture and thus indistinguishable by our benchmarks, we also classify microarchitectures by their base microarchitecture. A prominent example is the Skylake base architecture by Intel, on which eight microarchitectures (e.g., Kaby Lake and Comet Lake) from 2015 to 2020 are based. We consider 10 different groups with 18 to 22 samples each, with a total of 211 different CPUs.

Model Classification We prepare multiple subsets of our data set to investigate the performance of inferring the specific CPU model using machine learning. The first subset features 18 different CPU models with at least 7 and a maximum of 9 samples each. In total, the resulting set contains the data of 153 different CPUs, thus only using about 22 % of the data set suitable for this classification. Here we use the results of different benchmarks for training and testing. The first approach uses the results of the cache-size benchmark, the cache-associativity benchmark, the TLB-size benchmark, and the cores benchmark.

The second approach uses the execution time of each benchmark as a feature. Here we use a slightly different subset of our data set, as the capturing of execution times per benchmark was only introduced later in our study. Each class of the set has at least 7 samples with a maximum of 9 samples from 14 different CPU models. In total, the set contains data of 114 different CPUs.

3.4 Classification Evaluation

In each classification scenario, we perform a Grid Search on 75 % of the corresponding data set for each of the three classifiers. This process uses a K-Fold with $k = 5$. Finally, the best-performing classifier is evaluated on the held-out test set with the best-performing hyperparameter configuration.

4 Evaluation

In this section, we present the metrics achieved by different classification algorithms for the scenarios presented in Section 3.3. The metrics show the ability of our benchmarks to fingerprint CPU vendors, microarchitectures, CPU models, and certain CPU properties (e.g., L1-cache size). We evaluate the efficiency of our current implementation by analyzing the benchmark execution times.

4.1 Classification

We compare the classification algorithm achieving the best accuracy to the best-performing `DummyClassifier` (DC). The dummy classifier uses the *most-frequent* or the *uniform* strategy. The most-frequent strategy predicts the most-frequent element of the training set. The uniform strategy chooses a random label.

Property Classification Table 1 shows the accuracies the best-performing classification algorithms achieve when classifying properties such as the L1 cache size. Especially accurate are the classifiers for the L1D cache size, L2 cache size, and the L1D cache associativity. The L1 and L2 cache sizes can be classified with an accuracy of above 95 % each. The L3 cache size classification achieves an accuracy of slightly more than 60 %. This is most likely because the last-level caches are usually shared and exhibit greater noise generated by the system. The accuracy of the L1D cache associativity classification is close to 94 %.

Although the cores benchmark is also negatively affected by the system noise, the `SVC` classifier achieves an accuracy of almost 75 % when classifying the number of threads. Mispredictions are often just off by two, indicating the system noise to be responsible for most of them. The same benchmark results are additionally used for the SMT and HTT availability classification. As HTT is the proprietary SMT implementation of Intel, the data set used for this classification is restricted to Intel CPUs. Here, the best-performing classifiers achieve accuracies of 69.5 % and 84.2 %. Increasing the maximum number of workers used by the cores benchmark should yield better results but would also drastically increase the execution time for CPUs featuring a low number of cores.

Finally, the boost-technology availability of a CPU can be classified with an accuracy of 72.7 % by the `SVC` classifier. The result is most likely negatively affected by external factors discussed in Section 5.2, as well as system noise.

The page size does not require a classifier, as the algorithm directly outputs the correct page size. The correct page size was inferred in 151 out of 158 test cases, resulting in an accuracy of 95.5 %.

Table 1: Property classification results

Property	Classifier	Accuracy	macro-F1	Test Set Size ($\|T\|$)
L1 Cache Size	MLP	1.000	1.000	34
L2 Cache Size	SVC	0.965	0.966	29
L3 Cache Size	SVC	0.629	0.629	62
L1D Cache Asso.	KNN	0.937	0.955	16
Number of Threads	SVC	0.741	0.661	62
HTT Availability	MLP	0.842	0.842	70
SMT Availability	SVC	0.695	0.695	105
Boost Availability	SVC	0.727	0.723	33

Table 2: AMD vs. Intel classification results

	Accuracy	macro-F_1	$\|T\|$
`DC`	0.481	0.479	83
`MLP`	0.975	0.975	

Vendor Classification For the combined properties, we first evaluate the capabilities to distinguish CPU vendors, specifically AMD and Intel. When distinguishing these vendors, the random dummy classifier achieves accuracies of around 50 % as the data set is balanced. In this scenario, the `MLPClassifier` achieves the highest accuracy at 97.5 %, as shown in Table 2.

The vendor classification most likely shows these results due to the observable differences in L2 cache sizes (512 kB for AMD, 256 kB for Intel). This effect also translates to ARM CPUs. Most ARM CPUs do not feature an L3 cache and share a large L2 cache among all cores. As most ARM CPUs in our data set are Apple M1, they are also distinguishable due to their large TLBs [14, 12]. Here, the accuracy is 100 % with a small test set containing 15 samples.

Microarchitecture Classification The microarchitecture classification to evaluate the microarchitecture fingerprinting capabilities of our benchmarks is not performed on a fully-balanced set. The randomly operating dummy classifier achieves an accuracy of 4.2 %, while the `KNN` classifier achieves an accuracy of 65.6 %. The results are listed in Table 3.

Table 3: Microarchitecture classification results

	Accuracy	macro-F_1	$\lvert T\rvert$
`DC`	0.041	0.043	96
`KNN`	0.656	0.640	

(a) not grouped

	Accuracy	macro-F_1	$\lvert T\rvert$
`DC`	0.074	0.077	54
`MLP`	0.925	0.888	

(b) grouped by base microarchitecture

The classification struggles to differentiate microarchitectures that do not differ in their L1 cache associativity or their L1 cache, L2 cache, or L1 TLB sizes. The Coffee Lake, Comet Lake, and Whiskey Lake microarchitectures by Intel are, for example, often confused with each other. They do not differ in their cache or TLB hierarchies, as all three are based on the Skylake microarchitecture, which makes it impossible to distinguish them based on these features. While there are smaller differences, they are not addressed by our benchmarks. Grouping all microarchitectures by their base microarchitecture, the classification yields an accuracy of 92.5 %. The results of this classification are listed in Table 3.

Model Classification For the CPU model fingerprinting, we use the results of our benchmarks and compare them to benchmark execution times. The benchmark execution times basically compress the benchmark results into one number. Using the same benchmarks as used for the microarchitecture classification, the `SVC` classifier achieves an accuracy of 58.9 %. Here, the dummy classifier only achieves an accuracy of about 10 %. The resulting metrics are shown in Table 4. Many CPU models do not differ in their cache or TLB hierarchy, with the only exception being the shared LLC, as its size usually depends on the number of cores. Should the number of cores also be the same, the ability to distinguish such CPU models is often comparable to guessing at random.

Table 4: Model classification results

	Accuracy	macro-F_1	$\lvert T\rvert$
`Dummy`	0.102	0.074	39
`SVC`	0.589	0.590	

(a) using benchmark results

	Accuracy	macro-F_1	$\lvert T\rvert$
`Dummy`	0.033	0.020	30
`SVC`	0.700	0.568	

(b) using execution times

This is further supported by a small-scale experiment. Here, a classifier is evaluated to distinguish an Intel Core i5-8250U from an Intel Core i7-8550U.

These two CPU models feature the same microarchitecture and the same number of cores. They only differ in their frequencies, with the Core i7 model having a slightly higher base and boost clock. The best-performing classifier does not outperform the random dummy classifier. In contrast, running the same experiment using two CPUs that differ by at least one microarchitectural property yields an accuracy of 100 %. For example, the Ryzen 5 2600 has an 8 MB L3, while the Ryzen 5 3600 has a 16 MB L3. They also differ slightly in their frequency. Note that both classifiers were only evaluated on test sets of size 8. This indicates that the used benchmarks do not contain much information about the performance of the CPU and only accurately profile their respective microarchitectural property. The classification using benchmark results can identify CPU models with unique properties. In our data set, this applies to the Apple M1 chip. A classification algorithm can distinguish this CPU model from all others in our set with 100 % accuracy due to the unique TLB and page size.

The model classification using only the vector of execution times, i.e., the execution time of each benchmark, achieves an accuracy of 70 % with the best-performing `SVC` classifier. The results of this classification are listed in Table 4. It is also important to note that the data sets used in both experiments are not identical. As the collection of execution times was only introduced later in our study, the second experiment contains different CPU models and may contain fewer samples for some classes.

The execution time of a benchmark is a compression of the benchmark results for non-constant-time benchmarks. For example, in the L1 cache-associativity benchmark, we observe lower execution times in iterations that do not exceed the associativity. However, a low execution time of this benchmark can indicate at least one of two things. The execution time could have been generated by a high-performance CPU or high L1 cache associativity, or a combination. Ultimately, the execution time of a benchmark thus is a compression of the benchmark results and the performance of the CPU, enabling the feature to be used to distinguish microarchitectural properties and performance.

A combined approach using benchmark results and their execution times performs similarly to the approach only using the execution times. Here, the best-performing `SVC` achieves an accuracy of 60 %. We assume that the combination scales better to data sets containing more classes, as the compression of benchmark results and performance may lead to confusion.

4.2 Efficiency

The runtime of our benchmarks depends largely on the performance of the CPU that is being profiled. In addition, as discussed in Section 4.1, it also depends on the properties of the CPU (e.g., L1 cache size). On a fast CPU (e.g., AMD Ryzen 9 5900X), the total runtime amounts to about 1.1 min, while slower CPUs (e.g., Intel Core2 Duo P8600) in our tests sometimes take more than 6.5 min to finish. The median of all total runtimes is slightly over 2 min. Figure 2 shows a box plot of the execution times per benchmark in seconds. Note that this plot does not show outliers to improve the readability.

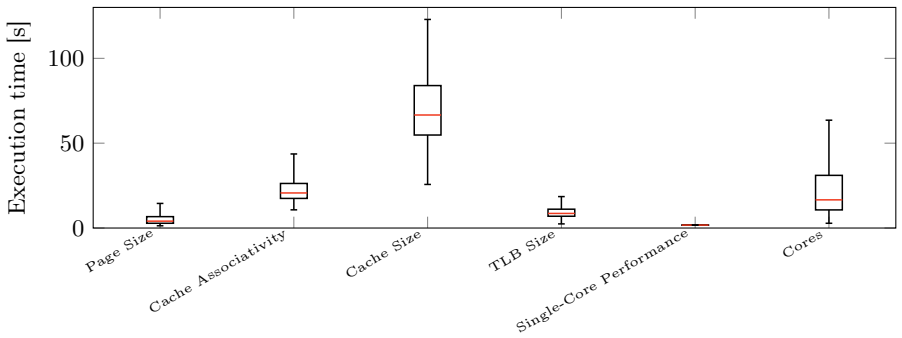

Fig. 2: Benchmark execution times box plot. Unused benchmarks are grayed out.

The single-core performance benchmark has a constant execution time of about 1.75 s, which rarely varies due to the scheduler. As shown by the box plot, the execution times of the page and TLB size benchmark almost always stay below 20 s each. The median of both benchmarks, however, is below 10 s. The cache associativity benchmark has a median value of about 20 s, with almost all execution times being in the range of 10 s to 50 s. As the runtime of the cores benchmark largely depends on the number of available threads, the execution times of this benchmark vary in the range from 5 s to 60 s. The execution time of CPUs with more than 8 available threads is generally below the median time of 20 s. Our benchmark with the highest median execution time is the cache-size benchmark with a median value of about 60 s. The maximum value is slightly more than 120 s. Its current implementation dominates the overall runtime.

The runtime of our benchmarks can, however, still be reduced, especially since they currently implement exhaustive search. For the cache size benchmark, it might, e.g., be possible to implement a binary search. Similarly, the number of cores benchmark could implement an early-abort mechanism to reduce the runtime drastically. In many cases, the number of iterations could also be reduced by a large margin without sacrificing discriminative power. It is not uncommon for users leave a tab open for more than 10 minutes (e.g., in case of streaming portals, or online games). Furthermore, our benchmarks may be interrupted and resumed later to circumvent the issue of running in the background.

4.3 Noise Resilience

To evaluate our benchmarks for noise resilience, we collect a small dataset using four different CPUs where each CPU is sampled in 7 different noise scenarios. The first scenario is a baseline, where no additional noise is added to the environment. We consider CPU noise by using the *stress-ng*[3] tool to run 1, 2, or 4 CPU stressors. Separately, we consider memory noise by running 1, 2, or 4 Virtual Memory stressors. Here, each stressor uses 10 % of free memory.

[3] `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`

The collected dataset is then used as a test set for the property classifiers trained as described in Section 3.4. Since the size of the dataset is small, we cannot draw any certain conclusions. However, the accuracies indicate that all benchmarks except for the cores benchmark and the large sizes for the cache size benchmark are largely unaffected by noise. Property classifiers using the output of the cores benchmark or output of the cache size benchmark targetted at cache sizes that exceed the L2 cache thus perform noticeably worse under noise. This also explains the performance of the L3 cache size and the Number of Cores classifier, which perform slightly worse than other property classifiers.

5 Discussion

5.1 Use for Microarchitectural Properties

Many microarchitectural attacks have implicit assumptions about the underlying microarchitecture. For example, page-deduplication attacks in the browser [10, 2, 6] assume a page size of 4 kB. While this was common when the attacks were published, new (micro-)architectures, such as the Apple M1, have different default page sizes, such as 16 kB. Assuming a page size that is too small breaks the attack. Hence, our page-size benchmark reenables this attack on the M1. Similarly, some attacks rely on initial assumptions about the microarchitecture, e.g., the page size [43, 36], the cache associativity and cache size for cache eviction sets [42], the L1 associativity for evicting L1 sets [39], or the number of CPU cores when optimizing MDS attacks [28]. With our benchmarks, these values can be inferred in the browser with a high accuracy, improving such attacks.

In addition to improving microarchitectural attacks, the information about the CPU can be used for browser fingerprints. The advantage over software properties is the long-term stability of hardware fingerprints, as users do not upgrade hardware that often. While the CPU as single property is not very unique, it can help linking less stable fingerprints [41]. Moreover, as there is no dependency on a specific browser API, our CPU fingerprints are difficult to mitigate without impacting the usability of websites (cf. Section 5.3).

We believe that our benchmarks also translate to newly released CPU models, as long as their respective features fall within the addressed ranges. Otherwise the ranges of our benchmarks would have to be adjusted and our respective property classifiers would have to be retrained. The model may have to be included in the training of the vendor and microarchitecture classifiers. In addition, the model must be included in the training for the model classification.

5.2 Limitations

Data Set The data set used for our evaluation is fairly small as we could not rely on any existing data set. Especially the number of samples per class is low for the model classification. The same problem also affects property classifiers with dominating classes. Since the data was collected using a study involving

the manual reporting of the CPU model, we cannot guarantee the correctness of all reported CPU models. In some cases, the labels used for our algorithms might contain wrong information. This is due to the fact that this large data set containing almost 300 different CPU models was collected manually using information from *cpu-world*[4] and the official vendor information.

Runtime Influences While different browsers use different JavaScript engines, we do not see a significant negative impact of that in our results. Even though Chrome provides timestamps with a lower resolution than Firefox, the benchmarks work well in both browsers. We cannot enforce a low system usage, such that some data might be very noisy due to demanding background tasks. Similarly, running the benchmarks on a mobile device running on battery interferes with power-saving mechanisms. Furthermore, we do not consider non-default CPU settings, such as deactivation of features (e.g., HTT), manual overclocking, and undervolting. Lastly, we do not perform any hyperparameter optimization of our classification algorithms and instead mostly use default parameter values. We leave this optimization to future work.

5.3 Mitigations

As our benchmarks only measure timing effects of microarchitectural elements and the performance of the CPU, it is difficult to mitigate them fully.

Disabling JavaScript The simplest solution to stop attacks involving client-side scripts is to disable the execution of all scripts. While this approach completely mitigates all of our benchmarks, it also disrupts benign functionality using scripts and thus is often not an option. Moreover, recent work [38] shows that limited microarchitectural attacks are possible without code execution.

Disabling Features Our benchmarks rely on shared memory and high-resolution timers. Disabling these features does, in fact, mitigate our current implementation. The benchmarks used by our classification algorithms, however, do not necessarily require these features. By aggregating timing differences instead of measuring single event timings, everything can be implemented using the timers with reduced precision. Some benchmarks profiling multi-threaded scenarios require the Web Worker API. Disabling this feature would fully mitigate the core benchmark and associated classifications.

Adding Randomness Another possible mitigation is to add random noise to the JavaScript engine (e.g., random memory prefetches, instruction reordering, low-resolution timer) [17, 34]. This does, however, not fully mitigate benchmarks but only adds noise, resulting in worse classification results. We observe that our benchmarks are largely unaffected by a slower `SharedArrayBuffer`, buffer ASLR, array preloading, and message delay.

[4] `https://www.cpu-world.com/index.html`

Detection Another countermeasure is to detect the benchmarks on the system or the browser level using, e.g., performance counters [13, 25, 44]. Future work has to evaluate if these approaches can be used to detect browser-based attacks.

6 Conclusion

We presented 6 JavaScript and WebAssembly benchmarks with a total median runtime of just over 2 min designed to reveal different CPU properties. The individual benchmarks allow determining their respective target properties with high accuracies of up to 100 %. As a result, microarchitectural attacks from the browser can be better tailored to the specific CPU. Moreover, the results of these benchmarks can be combined to accurately infer the vendor of a CPU, the microarchitecture, or the CPU model. Our benchmarks allow the identification of the vendor of a CPU with accuracies above 97 %. Moreover, current browser mitigations do not prevent our benchmarks. Hence, this information can also improve state-of-the-art browser fingerprinting techniques.

7 Acknowledgments

We would like to thank all participants of our study. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 491039149. We further thank the Saarbrücken Graduate School of Computer Science for their funding and support.

References

1. Agarwal, A., O'Connell, S., Kim, J., Yehezkel, S., Genkin, D., Ronen, E., Yarom, Y.: Spook.js: Attacking chrome strict site isolation via speculative execution (2022)
2. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In: S&P (2016)
3. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS (2019)
4. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (2019), extended classification tree and PoCs at https://transient.fail/.
5. Cao, Y., Li, S., Wijmans, E.: Browser Fingerprinting via OS and Hardware Level Features. In: NDSS (2017)
6. Easdon, C., Schwarz, M., Schwarzl, M., Gruss, D.: Rapid Prototyping for Microarchitectural Attacks. In: USENIX Security (2022)
7. Eckersley, P.: How unique is your web browser? In: PETS (2010)
8. Englehardt, S., Narayanan, A.: Online tracking: A 1-million-site measurement and analysis. In: CCS (2016)
9. Gras, B., Razavi, K.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)

10. Gruss, D., Bidner, D., Mangard, S.: Practical Memory Deduplication Attacks in Sandboxed JavaScript. In: ESORICS (2015)
11. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA (2016)
12. Handley, M.: M1 Exploration - v0.70 (2021)
13. Herath, N., Fogh, A.: These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In: Black Hat Briefings (2015)
14. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2019)
15. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: ISCA (2014)
16. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
17. Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium (2016)
18. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: A survey. In: ACM Transactions on the Web (2020)
19. Laperdrix, P., Rudametkin, W., Baudry, B.: Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In: S&P (2016)
20. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (2018)
21. Mowery, K., Bogenreif, D., Yilek, S., Shacham, H.: Fingerprinting information in JavaScript implementations. In: W2SP (2011)
22. Mowery, K., Shacham, H.: Pixel Perfect: Fingerprinting Canvas in HTML5. In: W2SP (2012)
23. Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In: Security and privacy (SP) (2013)
24. Olejnik, L., Englehardt, S., Narayanan, A.: Battery Status Not Included: Assessing Privacy in Web Standards. In: Workshop on Privacy Engineering (IWPE) (2017)
25. Payer, M.: HexPADS: a platform to detect "stealth" attacks. In: ESSoS (2016)
26. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research (2011)
27. Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C.: CrossTalk: Speculative Data Leaks Across Cores Are Real. In: S&P (2021)
28. Röttger, S.: Escaping the Chrome Sandbox with RIDL (2020), `https://googleprojectzero.blogspot.com/2020/02/escaping-chrome-sandbox-with-ridl.html`
29. Saito, T., Yasuda, K., Ishikawa, T., Hosoi, R., Takahashi, K., Chen, Y., Zalasiński, M.: Estimating cpu features by browser fingerprinting. In: International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS) (2016)
30. Saito, T., Yasuda, K., Tanabe, K., Takahashi, K.: Web browser tampering: Inspecting CPU features from side-channel information. In: International Conference on Broad-Band Wireless Computing, Communication and Applications, BWCCA (2017)

31. Sanchez-Rola, I., Santos, I., Balzarotti, D.: Clock around the clock: Time-based device fingerprinting. In: CCS (2018)
32. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)
33. Schwarz, M., Lackner, F., Gruss, D.: JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In: NDSS (2019)
34. Schwarz, M., Lipp, M., Gruss, D.: JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: NDSS (2018)
35. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)
36. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC (2017)
37. Schwarzl, M., Borrello, P., Kogler, A., Varda, K., Schuster, T., Gruss, D., Schwarz, M.: Dynamic process isolation. arXiv:2110.04751 (2021)
38. Shusterman, A., Agarwal, A., O'Connell, S., Genkin, D., Oren, Y., Yarom, Y.: Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In: USENIX Security Symposium (2021)
39. Stephen Röttger and Artur Janc: A Spectre proof-of-concept for a Spectre-proof web (2021), `https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html`
40. Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yuval, Y., Sunar, B., Gruss, D., Piessens, F.: LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: S&P (2020)
41. Vastel, A., Laperdrix, P., Rudametkin, W., Rouvoy, R.: Fp-stalker: Tracking browser fingerprint evolutions. In: S&P (2018)
42. Vila, P., Köpf, B., Morales, J.: Theory and Practice of Finding Eviction Sets. In: S&P (2019)
43. VUSec: RIDL test suite and exploits (GitHub) (2020), `https://github.com/vusec/ridl`
44. Wang, H., Sayadi, H., Sasan, A., Rafatirad, S., Homayoun, H.: Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks. In: ICCAD (2020)