Clock Around the Clock: Time-Based Device Fingerprinting

Iskander Sanchez-Rola
Deustotech, University of Deusto
iskander.sanchez@deusto.es

Igor Santos
Deustotech, University of Deusto
isantos@deusto.es

Davide Balzarotti
Eurecom
davide.balzarotti@eurecom.fr

ABSTRACT

Physical device fingerprinting exploits hardware features to uniquely identify a machine. This technique has been used for authentication, license binding, or attackers identification, among other tasks. More recently, hardware features have also been introduced to identify web users and perform web tracking. A particular type of hardware fingerprint exploits differences in the computer internal clock signals. However, previous methods to test for these differences relied on complex experiments performed by running native code in the target machine.

In this paper, we show a new way to compute a hardware fingerprinting, based on timing the execution of sequences of instructions readily available in API functions. Due to its simplicity, this method can also be performed remotely by simply timing few seemingly innocuous lines of JavaScript code. We tested our approach with different functions, such as common string manipulation or widespread cryptographic routines, and found that several of them can be used as basic blocks for fingerprinting.

Using this technique, we implemented a tool called CryptoFP. We tested its native implementation in a homogeneous scenario, to distinguish among a perfectly identical (both in software and hardware) set of computers. CryptoFP was able to correctly discriminate all the identical computers in this scenario and recognize the same computer also under different CPU load configurations, outperforming every other hardware fingerprinting method. We then show how CryptoFP can be implemented using a combination of the HTML5 Cryptography API and standard timing API for web device fingerprinting. In this case, we compared our method, both in the same homogeneous scenario and by performing an experiment with real-world users running heterogeneous devices, against other state-of-the-art web device fingerprinting solutions. In both cases, our approach clearly outperforms all existing methods.

KEYWORDS

device fingerprinting; web privacy

ACM Reference Format:
Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2018. Clock Around the Clock: Time-Based Device Fingerprinting. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243734.3243796

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243796

1 INTRODUCTION

A large number of *physical device fingerprinting* techniques have been proposed over the years to uniquely identify a device based on its physical features [3, 5, 14, 27, 32, 33]. The application of these techniques also varies, and includes device authentication, software license binding, attackers identification [12, 23], and wireless network identification [2, 13].

More recently, hardware-level features have also been adopted to create more precise forms of web tracking. In what is normally called *web device fingerprinting*, the owner of a website computes a unique identifier for each visitor's machine, without storing any information on the client side — thus making these techniques easier to hide and harder to block or mitigate. Its stateless nature is what makes device fingerprinting particularly relevant for web tracking. Since the user's unique identification is computed every time she visits a website, it is not possible for the user to remove the fingerprint, making this more difficult to avoid than older stateful web tracking approaches. We can distinguish between two types of device fingerprinting techniques: we refer to those that are based on browser artifacts as *attribute-based device fingerprinting* and to those based on hardware-level features as *hardware-level device fingerprinting*. Attribute-based techniques relies on different accessible browser attributes such as the list of installed fonts, the `UserAgent` string, and the screen resolution. Since these attributes change often and are easy for the user to modify, the resulting fingerprint also rapidly evolves – thus preventing a stable, long-lasting tracking [41]. In contrast, hardware-level techniques exploit subtle differences in the underlying hardware that are detectable by invoking certain APIs to compute the differences between devices. For instance, it is possible to compute differences in the way text is rendered by the HTML5 Canvas API or by using the WebGL API [30]. Even though these techniques are very promising and less prone to periodic changes than attribute-based solutions, all the hardware-based techniques proposed to date depend not only on the hardware itself, but also on the particular APIs implementation in the target browsers.

In this paper, we propose to look at code execution time as a way to precisely identify different devices. The time a computer spends to execute an instruction depends on how many clock cycles the instruction requires, and on the duration of each cycle. Internal clocks use oscillators based on quartz crystals, and small variations in those crystals can result in extremely small, but measurable, differences in the clock frequency. Researchers have already proposed to use these differences to uniquely fingerprint different devices [23, 34], but previous measurements were difficult to take, as they needed to analyze network traffic, and required an external reference time to compare with. Salo [35] proposed a solution to this problem by comparing two different clocks: the one used by the CPU and the independent one used to maintain the internal timer. However, the proposed methodology strongly depends on

specific hardware, relied on custom snippet of assembly code, and required a long execution time to generate a stable fingerprint.

Our idea relies instead in the identification of readily available functions that, when repeated a sufficient number of times, can be used to amplify the small differences between different clocks. Those functions should contain enough instructions to achieve a sufficient precision, but not too many to be regularly interrupted by the OS scheduler. We then measure the execution time by using the `datetime` APIs, which rely on a separate clock than the one used by the CPU to execute code. Our experiments show that this approach can be used to precisely fingerprint a machine, even when performed by using a snippet of JavaScript embedded in a web page. After testing with a set of candidate functions, we settled our proof of concept implementation on a simple cryptographic routine to generate pseudo-random numbers, as it is widely available and it is commonly used as basic block in many popular applications.

Our experiments demonstrate that subtle differences in the execution times of this cryptographic function are sufficient to capture the differences among different machines, outperforming all hardware device fingerprinting techniques proposed to date. To obtain a baseline to compare the fingerprinting capabilities of our approach, we first implemented a native version of our method in C. The tool was stress-tested in a scenario with hundreds of computers equipped with the exact same hardware and software. Then, in order to verify that our solution can also be used for web device fingerprinting, we implemented a web version of our algorithm using the HTML5 Cryptography API — that ultimately invokes the same operating systems functionalities we relied upon in the C version. This web implementation was tested using the same homogeneous scenario composed of computers with same software and hardware configuration as well as a real-world scenario including different users who visited our public experiment website, making a total of 565 different users.

In summary, our main contributions are:

- We show that a timing side-channel present in all modern computers can be used to uniquely identify a machine among a large number of possible (identical or not) candidates.
- We present a specific implementation of our time-based fingerprinting technique based on simple cryptographic functions. We tested our solution in a homogeneous scenario for device fingerprinting evaluations that tackles the main limitations of previous tests by including measures to deal with homogeneous targets.
- We ported our technique to the Web using the HTML5 Cryptography API library. This makes our solution available as a web device fingerprinting technique. We show that our technique overcomes the state-of-the-art hardware-based device fingerprinting techniques both in a homogeneous scenario and in a real-world web fingerprinting experiment.

The remainder of the paper is organized as follows. §2 proposes a new set of features for assessing fingerprinting methods, tackling the current limitations regarding fingerprinting methods evaluation. §3 details the proposed hardware machine fingerprinting method, explaining the reasons that make these new techniques to work accurately. §4 evaluates the native method in a homogeneous scenario environment, showing that it can discriminate between identical hardware machines. §5 details the specific implementation in the web, for a web device fingerprinting technique, evaluating this method and comparing it with current state-of-the-art in hardware-level web device fingerprinting techniques, both in a homogeneous scenario environment and in an in-the-wild real world scenario experiment. §6 discusses the major implications of this work. §7 provides the reader with the required background on device fingerprinting and timing attacks, critically analyzing existing methods. Finally, §8 provides the concluding remarks.

2 FINGERPRINT ASSESSMENT

The goal of fingerprinting techniques is to uniquely identify a target entity. This entity can be a browser, a physical machine, or even a user across different personal devices. Despite the large number of fingerprinting techniques proposed by both academia and industry (see §7 for more details) — or already discovered in the wild, the security community has not come yet to a consensus on which characteristics need to be measured to properly evaluate and compare between fingerprinting solutions.

For instance, for web-based fingerprinting approaches, the *cross-entropy* and the size of the *anonymity set* are used as the "de facto" evaluation standard procedure across multiple papers [6, 9, 26]. While certainly important, they fail to capture many important aspects of a fingerprinting procedure, such as its resilience to changes in the user browser (e.g., due to a software update) or the overall efficiency of the computation process.

In this paper, we propose a rich set of metrics to be used as a new basis to measure the quality of different fingerprints. This set includes six "desired" characteristics of a fingerprint:

- **Discrimination Power:** The discrimination power of a fingerprint is defined as its ability to produce different fingerprints for different targets. This can measure the ability to uniquely identify a target among a set of possible candidates.
- **Stability:** The stability of a fingerprinting technique is its ability to always produce the same fingerprint for the same target over multiple measurements.
- **Homogeneous Discrimination:** This property measures the discrimination power of a technique in the case in which the targets belong to the same homogeneous family, and they are therefore similar among each other.
- **Efficiency:** This feature simply measures the time required to generate the fingerprints and check them against a database of previous candidates.
- **Resilience to Evasion:** Since there exist methods to avoid fingerprinting or at least to reduce its consequences, this feature takes into account whether a fingerprinting technique is resilient to known or possible evasions.
- **Resilience to Change:** This final characteristic captures the ability of a fingerprinting technique to remain stable over time. Some techniques use features that naturally evolve, thus resulting in a fingerprint that can be associated to a target only for a limited time window. Indeed, a recent paper [41] has studied the evolution of existing techniques and has found that the vast majority of the general fingerprints changes in less than 10 days.

Unfortunately, current evaluation procedures usually assess only the discrimination power (by measuring both the cross-entropy and the anonymity sets) and sometimes the efficiency of a solution. However, we believe that all features are equally important to comprehensively assess a new fingerprinting technique.

In particular, the stability has been surprisingly omitted from many evaluations presented to date. If a fingerprint lacks stability, it means that the procedure may generate erroneous fingerprints or that the result includes a certain level of noise, misleading the identification. Please note that stability should not be mistaken with resilience to change. The latter deals with the natural fingerprint evolution over time, rather than the fact that a technique may return different values for the same device when executed multiple times.

In a similar vein, the most common way researchers used to compile a test set for a new fingerprinting method and to compute its discriminatory power is to host it on a web page and share its URL with a large number of users. In this case, the number of different configurations both in hardware and software is high, even more if we consider that most of the machines will be commodity user computers. This setting results in some attributes, such as the `UserAgent`, to show a very high cross-entropy [26], against intuitive observations. A more homogeneous environment (e.g., the set of many similar or identical computers that form many company networks) would provide a much more challenging environment to assess the fingerprinting precision.

Finally, with the notable exception of Cao et al. [6], no fingerprinting has taken into account the error introduced by possible changes in the user browser or operating system. This is another fundamental aspect of the problem, as even the most accurate solutions is of limited use if the fingerprints changes every time a user reboots the computer or installs a software update.

To better understand the importance of these metrics, we reviewed the characteristics of a number of current state-of-the-art fingerprinting methods, namely (i) Attribute-based FP, (ii) CanvasFP, (iii) WebGL FP and (iv) AudioFP. It is important to remark that this comparison is only based on what has already been tested by the original authors (in the particular case of WebGL, we conducted experiments using the available open-source tool) or based on the design of the fingerprinting method.

Interestingly, none of the methods described so far is resilient to changes in the target environment — with the exception of the aforementioned work by Cao et al. [6]. For instance, a simple graphic driver update can completely modify the fingerprint obtained by CanvasFP or WebGL. Evasion can also be easily implemented for all the methods, and actually most of them are already completely ineffective against the Tor Browser. According to the results presented in their respective papers, all techniques are capable of discriminating different targets (note that we have grouped attribute-based fingerprinting together as they are usually utilized altogether). WebGL was poor on the efficiency axis, as the version we tested required several seconds to build a single fingerprint. Unfortunately, the homogeneous discrimination and stability are difficult to estimate. Since we consider all of these features equally important, we will compare our method and the state-of-the-art hardware-level fingerprinting methods along all these dimensions in §7.

3 HOST FINGERPRINTING BASED ON CLOCK IMPERFECTIONS

In this section we present a new machine fingerprinting technique based on timing the execution of several invocations, performed using different parameters, of a properly selected function. The main assumption behind our solution is that it is possible to measure small variations on the execution time of a sufficiently long sequence of instructions that are introduced by imprecisions and imperfections (also known as "process variation" in the VLSI and architecture communities) of the machine clock crystal .

3.1 Threat Model and Use Cases

We test our time-based fingerprinting method in two different yet complementary scenarios. In the first one, we implement our technique in C and use it to tune our algorithm, while providing a baseline for comparison for the remote scenario. In the second use case, we port our technique to the web device fingerprinting scenario, implementing it in HTML5 and, thus, testing its ability to fingerprint machines over the web.

Host-based Fingerprinting. In this first scenario (detailed and tested in §4), we test the accuracy of our time-based device fingerprinting technique running natively in the target operating system. We perform this test even when it is known that you can fingerprint the clock natively to show the capabilities of our method and provide a baseline for the web version. Here, we assume the entity interested in computing the fingerprint is able to run arbitrary code with user privileges in the physical machine. For instance, this is the case of (i) malicious applications that want this information to perform selective attacks against certain victims, and (ii) proprietary applications that want to bind a license to a single machine.

Web-based Fingerprinting. The second scenario is more challenging, as we imagine that the entity who wants to compute the fingerprint is now an arbitrary website containing JavaScript code. In this case, it is not possible to run arbitrary instructions on the CPU, as modern browsers introduce numerous intermediate layers between the JavaScript code and the final CPU instructions. The goal of this scenario is to test if our approach can also be remotely executed over the web, thus resulting in a very powerful new technique for device fingerprinting. Also in this case, we can envision two different scenarios: (i) advertisers or tracking companies can use it to obtain the browsing history of their users, and (ii) websites that require strong authentication (e.g., banking and shopping) can use this technique to include an additional verification to their process.

3.2 Existing Approach

The detection of clock imperfections for fingerprinting purposes has already been exploited on a single CPU by Salo [35], but this solution required complex native experiments (which made the technique difficult to use in the real world) and were not able to successfully discriminate all machines involved in the test. To detect the imperfections, Salo proposed to compare the CPU clock cycles of ticks in the clock with the cycles needed for the digitalization of an analog signal using the sound card (all validated by an external GPS receiver). Afterwards, the author computed different statistical

tests to distinguish among different machines. Several factors play a crucial role for this technique to work:

(1) The program needs to have access to the CPU clock cycles, which is not a big problem for a low-level programming language as C or C++, but is not a common option in high-level languages as JavaScript. Furthermore, some specific tuning needs to be done depending on the specific type of CPU used in the experiments.
(2) The sound card used for the digitalization must not rely on the CPU clock and should use an independent crystal-controlled oscillator.
(3) To obtain enough data to successfully distinguish between two or more machines, the experiment needed to run for approximately one hour.

These limitations show that the technique strongly depends on some specific hardware, tuning, and a long computation time — making the entire approach poorly usable in practice. Even when these requirements are satisfied, the method can only be used with low-level programming languages that can obtain direct control over the CPU clock cycles. Moreover, the results obtained show that despite many machines (from the 38 analyzed) could be differentiated, not all were correctly identified.

3.3 Our Approach: Time-Based Device Fingerprinting

We now present our approach, which takes just some milliseconds to execute, can be used both in low or high level programming languages, and is not dependent on any specific hardware. Our algorithm is divided into two different phases: the generation of the fingerprint performed by timing a given function, and the comparison phase in which we test whether a pair of fingerprints (which consists of a matrix of time results) belong to the same machine.

3.3.1 Fingerprint Generation. In this phase, the algorithm computes the time required to execute different invocations of a target function (see Figure 1 for the detailed pseudo-code of the algorithm). The algorithm takes one parameter n that indicates the number of calls to measure. Moreover, for the sake of simplicity, in the example in Figure 1 we have assumed that this number is also used as parameter for the function itself. For instance, if we use a function that generates random numbers, we will consecutively create different number of random values, allowing us to time the functions in different situations depending on the input.

There are many factors that can cause performance variability in non-deterministic ways. Pure hardware-level factors as Cache/TLB misses and sharing the pipeline resources with other threads co-scheduled on the same core (hyper-threading) or even OS's DVFS (Dynamic Voltage Frequency Scaling) decisions. Because of all these possible non-deterministic factors, a single measure is insufficient to obtain a stable measurement. In order to obtain stable fingerprints, our method uses an additional parameter m that determines the number of times this process is repeated, to achieve a real representation of the machine independently of different specific situations. As a result, the final fingerprint is a $n * m$ matrix of execution times. To sum up, there are m function calls, with specific values as input, computed for each of the n rows of the timing matrix.

Input: n, number of timings to perform
Input: m, number of arrays of these timings to generate.
Output: fp, array of arrays of numbers representing the fingerprint: each position are the result of timings with a different parameter for a function.

```
1  Function FPGeneration (n, m)
2      i ← 1;
3      fp ← float[][] of size n × m;
4      while i ≤ m do
5          j ← 1;
6          while j ≤ n do
7              startTime ← GetCurrentTime() ;
8              Function(j);
9              endTime ← GetCurrentTime();
10             logTime ← endTime − startTime;
11             fp[j][i] ← logTime;
12             j ← j + 1;
13         end
14         i ← i + 1;
15     end
16     return fp;
```

Figure 1: Fingerprint Generation Algorithm.

As the technique is not based on computing the same function with the same input all the time, but executing the same function with different inputs, the matrix structure allows a quick comparison with other fingerprints. For example, following the case of generating random numbers presented before, we can easily check the differences between the fifth execution of the function that generated 20 random numbers in one computer with exactly their counterpart on another computer.

3.3.2 Fingerprint Comparison. In this phase, the system compares two previously-computed fingerprints and determines whether or not they belong to the same machine (for the detailed pseudo-code of the algorithm refer to Figure 2). To this end, we compute the most frequent timing values (the mode) for each call parameter over all iterations. Afterwards, the mode of the first fingerprint is compared with all the generated values for the same call in the second fingerprint. If one match is found, a counter is incremented. This process is then repeated, inverting the order and checking the most common values in the second one with all the values from the first one. If the number of matches divided by the number of comparisons surpasses a fixed threshold, then our algorithm concludes that the two fingerprints belong to the same machine.

For example, suppose we want to compare the following two fingerprints fp_1 and fp_2, each composed of three repetitions of three different timing results of the invocation of a given function:

$fp_1 = [\{0.1; 0.12; 0.14\}, \{0.1; 0.12; 0.13\}, \{0.1; 0.12; 0.13\}]$
$fp_2 = [\{0.1; 0.12; 0.14\}, \{0.11; 0.12; 0.14\}, \{0.1; 0.12; 0.13\}]$

We start by generating the mode of the timing values of fp_1: $\{0.1; 0.12; 0.13\}$ and comparing each of the three values with the values in the three value sets of fp_2, resulting in three positive

Input: $fp1$, $1st$ array of arrays of timing results sized $n \times m$.
Input: $fp2$, $2nd$ array of arrays of timing results sized $n \times m$.
Input: n, number of timings for different parameters.
Input: m, number of arrays of timings generated.
Output: indicates the number of coincidences

```
1  Function GetNumCoincidences (fp1, fp2, n, m)
2      num_coindences ← 0;
       /* Compute the mode of each number in fp1    */
3      fp1_mode ← float[];
4      i ← 1;
5      while i ≤ n do
6          fp1_mode[i] ← ComputeMode(fp1[i]);
7          i ← i + 1;
8      end
       /* We compute the number of coincidences     */
9      i ← 1;
10     while i ≤ n do
11         check ← false;
12         j ← 1;
13         while (j ≤ m) ∧ (¬check) do
14             if fp1_mode[i] = fp2[i][j] then
15                 num_coindences ← num_coindences + 1;
16                 check ← true;
17             end
18             else
19                 j ← j + 1;
20             end
21         end
22         i ← i + 1;
23     end
24     return num_coincidences;
```

Input: $fp1$, $1st$ array of arrays of timing results sized $n \times m$.
Input: $fp2$, $2nd$ array of arrays of timing results sized $n \times m$.
Input: n, number of timings for different parameters.
Input: m, number of arrays of timings generated.
Input: t, threshold to consider the fingerprint the same
Output: indicates if $fp1$ corresponds to $fp2$

```
25 Function FPCheck (fp1, fp2, m, n, t)
       /* We compute the coincidences amid the most
          frequent values in fp1 in fp2             */
26     num ←GetNumCoincidences(fp1, fp2, n, m) ;
       /* We compute the coincidences amid the most
          frequent values in fp2 in fp1             */
27     num ← num+ GetNumCoincidences(fp2, fp1, n, m) ;
       /* We check if the threshold is surpassed    */
28     return ( num/(n·2) ) ≥ t);
```

Figure 2: Checking Algorithm.

matches. The first value appears in the first and third iteration of fp_2, the second value appears in the all the iterations, and the third value appears in the last iteration. Then, we will do the same with fp_2 being their mode values: $\{0.1; 0.12; 0.14\}$ and also getting all of them matched in the fp_1 set. The first and seconds values appear in all the iterations of fp_1, and the third value appears in the first iteration. In conclusion, vectors do not need to be identical, but match each of the values of the mode with, at least, one of the value in the same position on another fingerprint. In this case, the percentage of similarity would have been 100% which, as a perfect match, would be above the threshold and our method would have determined that both fingerprints belonged to the same computer.

By using this procedure, we are computing and comparing the most common timing values — and, therefore, the most representative ones — among the measurements conducted on the two machines. This reduces the inevitable noise introduced in the timing measurements and reduces the impact of unusual values.

Table 1: Results of the Function Viability Test.

Function	Stable Fingerprint
`string::compare`	✓
`std::regex`	✓
`std::hash`	✓
`crypt`	✗

3.3.3 Function Selection. Before settling on a final choice, we decided to perform a preliminary set of tests to assess the different candidate functions. In particular, we evaluated the functions `string::compare`, `std::regex`, `std::hash`, and `crypt`. While our technique would work also by using a custom, system-independent function, we decided to base our tests on a set of common routines that can be easily found in many different systems. This increases the portability of our approach as it does not require to install or inject any additional code. The evaluation was performed on a set of ten different machines, half of which installed with Microsoft Windows and the other half installed with GNU/Linux. We also computed different tests with the aforementioned functions to empirically validate the best size of the measurement matrix, taking into account the generation time and the fingerprint discrimination capabilities. Based on these preliminary tests, we found that $n = 1000$ and $m = 50$ (i.e., a total number of 50,000 invocations) are sufficient to provide stable results.

Table 1 shows the obtained results. `crypt` was the only function whose fingerprint was not stable because, due to its complexity, it was often interrupted by the operating system scheduler — thus preventing our algorithm to accurately time its execution. For the remaining functions, it is important to note that simpler functions required to compute the execution time of multiple consecutive invocations to find a stable fingerprint. This issue is controllable by simply adding more iterations.

In summary, we investigated and evaluated if our fingerprinting algorithm can be built on top of multiple, diverse functions. According to our results, different candidates provided good results, in particular when they were sufficiently complex but not too long to be often interrupted by the scheduler.

3.3.4 Stability Tests. In order to determine the viability of the proposed approach for machine fingerprinting, we conducted three

additional tests. The setup for these stability tests is the same as the one used for the function selection. We checked if the obtained fingerprint of each machine can still identify the machine in the following cases:

- **CPU Load:** We tested the influence of different CPU load conditions on the fingerprint generation process. In our experiments, we controlled the CPU workload by using the `stress` generator included in the Debian distribution [15] and the corresponding tool part of Windows Sysinternals [19]. We discovered that even in the scenario of 100% CPU load, the resultant fingerprint was always correctly associated. This is a consequence of the fact that each function invocation gets executed in a single CPU with no interruption, and therefore without any side-effect introduced by other concurrent processes.
- **CPU Temperature:** We also tested whether significant environmental temperature changes would invalidate the fingerprint, as previous works have observed that the frequency of the quartz crystal increases with temperature [31]. During our normal experiments, the regular CPU temperatures were generally around 38 degrees Celsius. Hence, we tried to stress the CPU for 20 minutes at 100% load, successfully doubling the internal temperature (as reported by the internal sensor). However, even if under these conditions the clock skew reported in previous studies [25] should have resulted in a measurable difference in our timing experiments, we did not observe any variations or errors in our fingerprint identification. A possible explanation for this discrepancy is that, while the increase in temperature can impact clock-based measurements, our approach relies on the difference of two clocks physically located in the same machine. Therefore, both are likely impacted by the temperature change, thus reducing the effect of the higher temperature and compensating the changes introduced in their frequency. As a result, while the difference introduced by the temperature in one single clock may be relevant, the difference in the delta between two closely-located clocks may be too small to affect our fingerprint.
- **Long-term Stability:** We evaluated if the generated fingerprint remains stable over time during a normal use of the machine. In this case, we repeated our tests respectively one and two months after the fingerprint was first generated and found no problem in the identification process.

We selected fingerprinting functions that can be executed without interruption on a CPU. This guarantee that the collected timing information is not affected by side-effects introduced by other concurrent processes, making the measure independent from the CPU and/or I/O workload of the machine. When running the native measurement, we checked it was executed without interruption by using transactional memory. However, we could not guarantee the same property when the fingerprint is executed remotely over the web. Therefore, the scheduler might have interrupted some of the executions, but this is mitigated by the multiple calls to the function performed in the fingerprinting generation phase.

3.4 CryptoFP

Since this clock-based fingerprinting method works with virtually any simple function, we selected one based on its general availability and on the possibility to generalize our results and compare our host-based and web-based approaches.

According to these criteria, the selected function should be available in different forms but in all possible system. In fact, since one of our goals is to implement a web version of this device fingerprinting technique, it should be available also in JavaScript, called by a wrapper in this scripting language.

Based on the results of our preliminary tests, we decided to implement our prototype by timing the execution of the pseudo-random generator APIs (e.g., `CryptGenRandom/RtlGenRandom` in Microsoft Windows). These cryptographic functions are available in every system and also are accessible through JavaScript, which meet all our requirements.

4 HOST-BASED FINGERPRINTING OF IDENTICAL TARGETS

Since the common evaluation procedure used to measure fingerprinting techniques does not take into account several important features, we first propose our own methodology (detailed in §4.1) that is able to capture the two main omissions of previous approaches: (i) the impact of targets heterogeneity and (ii) the actual stability of a fingerprint within the same machine.

To evaluate CryptoFP, we implemented a native version of the algorithm. This version calls directly the function that generates a series of random numbers. We also repeated the tests described in §3.3.4, confirming that there was not effect introduced by the CPU load, internal temperature, or long-term stability of the fingerprint. We also conducted several experiments with a subset of different computers in order to properly tune the similarity threshold used by our algorithm, resulting in a value of 0.5 (i.e., two fingerprints are considered to belong to the same machine if there is at least 50% of positive matches when comparing them, as shown in Figure 2).

4.1 Methodology

The current evaluation methodology for fingerprinting techniques measures two features: the entropy of the fingerprinting and the size of the anonymity sets [26]. These are often used to replace other widely accepted and more conventional metrics, such as precision and recall, that are rarely used in this specific area as they provide a less precise image of the discrimination power of a fingerprinting technique. Therefore, we also decided to use similar measurements to be able to compare our results with those obtained in previous studies. In fact, since the fingerprint generation process in all major techniques results in a hash or in an identifier, it is possible to compute the entropy — i.e., a representation of global uniqueness — among a set of tested devices. Moreover, due to the nature of these methods, if a particular machine A has the same fingerprint of B and B matches a third machine C, C will always match with both A and B. This transitivity allows the computation of anonymity sets.

However, CryptoFP works differently and does not generate a unique identifier. Instead, it produces fingerprint information that needs to be compared with the one collected on other machines to identify possible matches. In other words, it produces some sort of

fuzzy hash, which cannot be simply matched against other candidates, but requires a comparison routine to compute the similarity among two values. Also, in our case, the final result is not a direct comparison of identifiers but a similarity score based on the described matching procedure. This approach has been intentionally designed to be more resilient to noise in the timing of the generation of random numbers and results in a greater accuracy. However, due to this design, the transitivity property does not hold anymore — thus making CryptoFP difficult to evaluate using entropy or anonymity sets as the obtained results (e.g., the entropy of our time matrix) would not be comparable with the entropy values of previous approaches. In our evaluatuon, we will use an adaptation of the anonymity sets.

4.1.1 Homogeneous Scenario. Previous experiments were performed by asking users to visit a website hosting the fingerprinting code. Therefore, users were likely using a browser running on commodity computers with different hardware, software, and configurations. While this is a realistic experiment (we will also use the same to further evaluate the web version in §5.2), it fails to capture the discrimination capability of the fingerprinting method, as the check strongly depends on the heterogeneity of the tested machines. For instance, if there are no computers with the same specific set of characteristics in the dataset, a simple hardware test can differentiate each client with 100% certainty. However, both companies and universities often rely on large numbers of identical machines, which can greatly complicate fingerprinting. To take this into account, we propose a homogeneous scenario evaluation that includes the next points:

- **Homogeneity:** In order to provide homogeneity and test our fingerprinting technique with the same hardware computers rather than with different computers, we performed our experiments using two groups of machines with perfectly identical software (installed through a disk image) and hardware components. The groups included 176 and 89 computers, respectively. Thanks to this setup we can identify whether our fingerprinting algorithm is really distinguishing *hardware imperfections* and to what extent it is possible to discriminate exactly identical hardware.
- **Stability:** We define the stability of the fingerprint as the ability to identify the same computer repeatedly. This measure has not been tested before in many previous studies, as authors assumed the property to be true by default. However, there may be some circumstances, such as specific hardware availability, general CPU workload, and number of concurrent process, that can affect and jeopardize the identification. Therefore, we repeated the CryptoFP generation phase three times for each computer. Each measurement was performed ten minutes apart. We then compared all results to check if the extracted fingerprints were always matching.
- **Discrimination:** Since our fingerprinting does not produce a hash but it needs a comparison phase, we cannot use the common measures like entropy or anonymity sets. Instead, we adapted the anonymity set measurement to an *identical comparison set size* that translates the idea behind anonymity sets to the comparisons performed by our method. In this way, the size is no longer the number of computers with the same fingerprint, but the number of computers with the same number of positive matches with other computers. To make it more clear, we are presenting a simple example. Imagine four different machines: A, B, C and D.
  - A matches B
  - B matches A and D
  - C matches D
  - D matches C

  In this case A, C and D have a set size of one, and B a set size of two (because it matches two other machines).

We run our CryptoFP native implementation in the two different sets (commodity computers running Microsoft Windows 7) and measured the properties introduced above. Using the threshold empirically computed in §3.3.3 ($n = 1000$ and $m = 50$) the test took just a few milliseconds, although obviously the exact computing time depends on the specific machine.

4.2 Results

As described above, we present our results using the *Identical Comparison Sets* metric, which is an adaption of the well-known anonymity set method for fingerprint evaluation, obtained by substituting "identical fingerprints" by "identical fingerprint comparisons". Therefore, in our particular cases we have a 0–175 possible values for identical comparisons for the first set of computers and 0–88 in the other, where 0 means that the particular computer had no match and the maximum value meaning the computer matched every other machine in the group.

Furthermore, we tested the stability of our method repeating the generation of the fingerprinting three times in each computer and validated that, in all cases from both scenarios, CryptoFP was always able to identify the computer. Regarding the discrimination capabilities, the native version of CryptoFP with a similarity threshold of 50% was able to distinguish every computer in each group. In other words, the uniqueness of our method in both tests is 100%, even when both hardware and software in the computers are identical. This shows that CryptoFP is capable of detecting clock crystal imperfections in order to accurately distinguish machines.

Please note that even thought we did not observe any in our experiments, collisions may occur on larger sets of identical targets. However, in most of the possible use cases, this is an acceptable result. In fact, if a user has a license bound to some machine, it is not very likely that she can test the software on tens of thousands of other identical machines just to find another one in which the software can be used. Our algorithm had no collisions in a lab containing 176 identical machines and another with 89 identical machines, which is a sufficient guarantee in most use cases.

5 WEB IMPLEMENTATION OF CRYPTOFP

The HTML5 Web Cryptography API is able to interact with cryptographic keys and functions managed by users. A very important aspect for our hardware-level device fingerprinting to work at native level even from the web is that "*the API itself is agnostic of the underlying implementation of key storage*" [42]. Its main objective is to provide just an interface or wrapper that allows system-level

```
void RandBytes(void* output, size_t output_length) {
  char* output_ptr = static_cast<char*>(output);
  while (output_length > 0) {
    const ULONG output_bytes_this_pass = static_cast<
        ULONG>(std::min(
    output_length, static_cast<size_t>(std::
        numeric_limits<ULONG>::max())));
    const bool success =
    RtlGenRandom(output_ptr, output_bytes_this_pass)
        != FALSE;
    CHECK(success);
    output_length -= output_bytes_this_pass;
    output_ptr += output_bytes_this_pass;
  }
}
```

Figure 3: Extract from the Chrome Implementation of `generateRandomNumbers`.

```
size_t RNG_SystemRNG(void *dest, size_t maxLen)
{
  size_t bytes = 0;
  if (RtlGenRandom(dest, maxLen)) {
   bytes = maxLen;
  }
  return bytes;
}
```

Figure 4: Extract from the Firefox Implementation of `generateRandomNumbers`.

cryptographic operations such as hashing, encryption, or decryption.

This API offers several interfaces to cryptographic functions through the `window.crypto` or `window.crypto.subtle` properties. The implemented methods can be very simple such as `getRandomValues` to generate a set of random numbers, `digest` to generate hashes, or `generateKey` that generates keys for encryption.

5.1 Implementation

We selected the simplest method available in the API, namely `getRandomValues`, for our device fingerprinting technique. Since our method is a timing side-channel attack, a complex cryptographic method — although the actual operations are performed at native level — may obscure our timing and make our fingerprint dependent not only in the underlying cryptographic functions, but also in the Web Cryptography API itself.

We analyzed the implementations of this method in two major open-source browsers, Firefox and Chrome, and inspected the native cryptographic function calls which were performed when the function was invoked. For example, when running Microsoft Windows, in both Chrome and Firefox , the `generateRandomNumbers` call finally leads to the native function `RtlGenRandom` to generate random numbers. For our experiments it is important, as shown in Figure 3 and Figure 4, that the browser API is just a basic wrapper for the native version, so the browser will not make other operations or memory accesses that may pollute the time measurement.

Regarding the values for n and m, we will use the empirically computed values of 1000 and 50 as indicated in §3.3.3. The computing time needed for the generation and checking of the fingerprint is just a few milliseconds. In order to determine the specific threshold for the web implementation of CryptoFP, we performed various preliminary tests. As the timing precision offered by HTML5 is smaller than the native timing functions, the threshold was finally set to 100% for the comparison of time matrix.

5.2 Evaluation

In this case, we compare CryptoFP with the other three state-of-the-art web hardware-level device fingerprinting techniques: (i) the famous canvas fingerprinting [30], (ii) the improved version of WebGL fingerprinting [6], and (iii) the recently discovered audio fingerprinting [10]. This allows us to compare the discrimination capability and stability of the four different techniques.

As the web implementation is devoted to track users on the Internet, we analyzed the fingerprinting techniques both in the homogeneous scenario presented in §4 and by using a classical web evaluation where users were asked to visit a website that performed all the techniques (making a total of 565 different users). In this case, we informed the users about our experiments, and ask permission to collect the information that was going to be gathered by our tool. Users where using their own machines and had no restriction on what computer they were using, so therefore our dataset can contain both GNU/Linux and Microsoft Windows in many different versions. In addition, in order to protect the users privacy, all the data collected was anonymous. We disseminated the URL of the website through social networks and friends, asking them to participate in the study and further re-disseminate the link among their contacts.

As described in §4, all results are shown using the *Identical Comparison Sets* metric, that is an adaptation of the extensively used anonymity set technique to evaluate fingerprinting methods. Zero indicates that there is not other match in the dataset, and the maximum number indicates that the fingerprint is the same in all the computers.

5.2.1 Homogeneous Scenario. In our experiments, we tested the stability of our technique by repeating the fingerprint generation three times in each computer. We found that all methods correctly generate the same fingerprint in all our tests, with the exception of audio fingerprinting, that failed the stability test in 21% of the cases, thus raising serious doubts about its possible use as fingerprinting technique with a basic hash comparison, regardless of other factors. For this reason, audio fingerprinting was removed from the following discrimination capability tests.

All methods took just few milliseconds to execute, with the exception of WebGL that required several seconds. Regarding the possible overhead, all methods are simple enough to result in no observable slowdown, with again the exception of WebGL, which relies on complex graphics checks and can therefore slow down navigation while it is being executed.

We divided the comparison sets in five groups, one containing computers that did not share any fingerprint, then three equally divided groups containing respectively 1-58, 59-117, and 118-174 positive matches in the 176 computer group and 1-28, 29-57, and 58-87 positive matches in the 89 computers set, and finally, one group with computers that shared their fingerprint with all the rest. CryptoFP was able to cover around 18% of the computers with

the two first sets for each of the computer groups (0-58 and 0-28 matches) and the percentage increases until 85% if we include the third set (0-117 and 0-57 matches). Even if these results are far from the perfect identification capability provided by the native method, current top state-of-the-art hardware-level fingerprint methods (canvas fingerprinting and the improved version of WebGL fingerprinting) could not differentiate any of the computers in none of the two homogeneous groups, resulting in the same fingerprint for all computers. Therefore, our solution clearly outperforms all previous state-of-the-art hardware technique in this particular settings.

Finally, the result of this experiment show that the web implementation of our technique is less precise than the native implementation, due to a more coarse-grain precision offered by the HTML5's `performance.now` timer. We will discuss different solutions in order to improve the results of the web implementation in §6. However, it is important to note that despite this limitation, CryptoFP is still capable of distinguishing completely identical hardware and software computers.

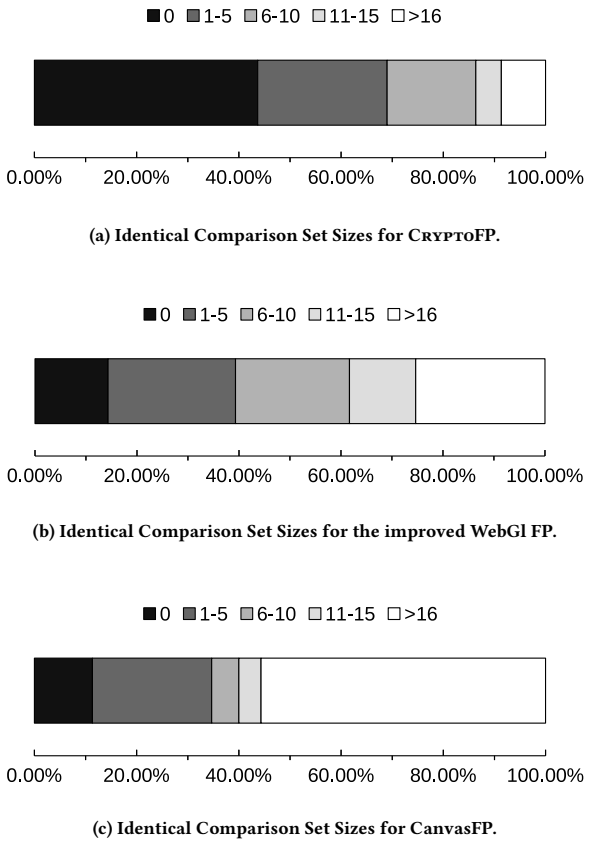
5.2.2 Heterogeneous Scenario. In this case, we also divided the comparison sets in five groups, but instead of separating the sets equally, we divided the sets every 5 matches, starting from 0 up to 15. The first group means that no additional matches were detected apart from its own, the second group counts the number of computer with 1-5 matches in the dataset of 300 computers, the next groups between 6-10 and 11-15 matches, and the last group counts the computers with more than 16 matches. In contrast to the homogeneous analysis, in this scenario, all the fingerprinting techniques are able to differentiate computers, so this more fine-grained set sizes will allow us to compare the methods more precisely.

Looking at the results collected thought our public website, reported in Figure 5, we can see that CanvasFP obtains only around 10% of completely unique fingerprints and the improved WebGl FP around 15%, whereas CryptoFP achieves around 45% in exactly the same dataset. More in detail, CryptoFP covers 70% of all the involved computers with just the two first identical comparison sets (0-5). Specifically, more than half of the computer were either completely unique or only matched another computer. However, both CanvasFP and improved WebGl FP obtain only around 40% with the two first identical comparison sets, which is less than just the first set, unique fingerprints, of CryptoFP.

The obtained results show the capabilities of the web version of CryptoFP, which is outperforming all existing hardware device fingerprinting solution, being able to obtain a better discrimination also in a heterogeneous scenario.

Fingerprinting combinations. CryptoFP, as any other device fingerprinting techniques, does not necessary need to work as a standalone solution. Instead, it can be easily combined with other different techniques, as other approaches already proposed to date. As a case study, we decided to combine all the hardware-level device fingerprinting methods with ours in order to increment the size of the discrimination rate by cross-referencing the results of the different methods.

Figure 6 shows that the combination of the hardware-level device fingerprinting techniques (the stable ones) achieved a uniqueness of around 80% and nearly a 100% coverage by just including the second comparison set (1-5). This simple combinations of CryptoFP with



(a) Identical Comparison Set Sizes for CryptoFP.

(b) Identical Comparison Set Sizes for the improved WebGl FP.

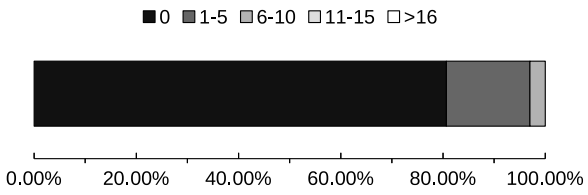
(c) Identical Comparison Set Sizes for CanvasFP.

Figure 5: Identical Comparison Set Sizes for CryptoFP, improved WebGl FP and CanvasFP in-the-wild web evaluation (300 different users involved). The colors represents the number of identical comparisons whereas the X axis represents the percentage of computers in the ranges.

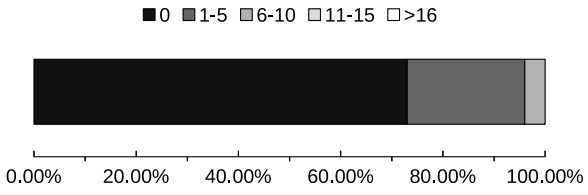
the improved WebGL FP and CanvasFP follow a similar fashion, with a 70% and 60% of uniqueness and nearly 100% and 90% coverage when the second comparison set is included.

6 DISCUSSION

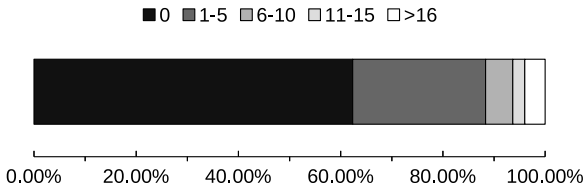
Generality. The assumption behind our approach is that any function can be timed and that this timing information can then be used to fingerprint subtle clock differences in the underlying machine. To confirm this hypothesis, we tested several functions in order to find out how generic the function selection can really be. After these preliminary tests involving functions of different nature, we realized that our method needs the function to be uninterrupted by the OS scheduler because, otherwise, the timing values would obviously be polluted by other processes. We also found that the timing of very small functions is also harder to measure, requiring a higher number of iterations to obtain a stable value. Therefore, we can conclude that our method require a function that includes a sufficient number of instructions, but not long enough to be often interrupted by the scheduler.



(a) Identical Comparison Set Sizes combining CryptoFP, the improved WebGl FP and CanvasFP.



(b) Identical Comparison Set Sizes combining CryptoFP and the improved WebGl FP.



(c) Identical Comparison Set Sizes combining CryptoFP and CanvasFP.

Figure 6: Identical Comparison Set Sizes for the different combinations of CryptoFP with the rest stable hardware-level device fingerprinting techniques (300 different users involved). The colors represents the number of identical comparisons whereas the X axis represents the percentage of computers in the ranges.

The confirmed generic nature of our approach makes it adaptable to different environments and situations. For instance, if a certain installation of a particular operating system uses a restricted version of the standard C library, our method can easily be changed to use another installed function. Similarly, if the target uses a completely different version of the operating system, even dedicated to IoT systems or critical infrastructures, if we can learn which functions are available, we can easily adapt our method in order to work under this new environment.

If we can execute native code, we can also create our own function and perform the timing using this function — making our code completely independent from the system libraries, as long as we have access to a timing operation that does not use the CPU clock signal.

Fingerprint Evaluation. In §2, we introduced a set of features that we hope can serve as guidelines for future fingerprinting evaluation. In addition, instead of testing our method against random machines, our evaluation procedure (described in §4 and §5.2) was designed to stress the algorithm in a scenario in which all machines have identical software and hardware components.

Table 2 summarizes the characteristics of different device fingerprinting techniques proposed to date, and compare them with our approach. Our method was the only one to discriminate all the computers (in the machine version) and the many of them (in the web version). In fact, the other methods could not differentiate any of the computers in any of the two sets. Stability was 100% for all methods, except of the Audio FP technique that returned different fingerprint values on the same computer. In addition, our method was the only one resilient to both changes and evasion techniques. In fact, since the method does not necessarily rely on a specific function, the only reliable way to affect its measurement is to insert noise in the time measurement — something that can have serious side effects on many web pages. Similarly, our fingerprint can survive even a complete re-installation of the operating system.

The only negative aspect of our solution, if used as a way to track users on the Web, is the back-end efficiency. On the one hand, computing a single fingerprint is extremely fast. On the other hand, existing fingerprints cannot be just indexed in a database for a fast retrieval. Instead, our solution require to compare a new fingerprint with all those collected for other machines. However, each comparison is fast (200 milliseconds in our current Python prototype), completely independent, and easily parallelizable. Moreover, an incremental comparison can be implemented to optimize the process, stopping the algorithm and removing candidates when a difference is found.

Application to Web Device Fingerprinting. The web-based implementation of our algorithm was not as precise at discriminating identical hardware and software machines as the native implementation. The reason behind this fact is the granularity of the HTML5 timing API, which does not allow for a more precise measurement. However, there are several improvements that can be implemented in the web version to enhance the timing precision.

First of all, instead of using the standard HTML5 timing API, there are improved timing techniques that can achieve more precise timing values, such as the clock interpolation technique presented by Schwarz et al. [37]. The timing precision we can obtain with some of this timers is similar to the timer used in the machine version. Therefore, it is logical to think that the fingerprint should also be as precise. Even in this particular case, the evasion would be difficult to implement since the functions used can be easily modified.

In addition, *WebAssembly* [17], a project that aims at introducing a new binary format for web applications, can also be used. In this case, we may not only improve the precision of the web version of CryptoFP but also implement a web version using any function. This API will allow to compile C/C++ code, amid others, as well as execute it at native speed using common hardware capabilities. The technology is currently in an early stage but it can be used in the future to fully implement the native fingerprinting method.

Countermeasures. Regarding possible evasions, we did not test those in which users were performing specific actions to tamper with the results – such as underclocking/overclocking the CPU, but

Table 2: A comparison of current state-of-the-art methods according to the proposed features. ✓ indicates that the method has, to a certain extent, that characteristic. ✗ implies that either the method has been tested and does not meet the feature or that, because of its design, it is unlikely to meet that requirement.

	Methods				
Feature	**Attribute-based FP**	**Canvas FP**	**WebGL FP**	**Audio FP**	**Our method**
Discrimination Power	✓	✓	✓	✓	✓
Stability	✓	✓	✓	✗	✓
Homogeneous Discrimination	✗	✗	✗	✗	✓
Efficiency	✓	✓	✗	✓	✓
Resilience to Evasion	✗	✗	✗	✗	✓
Resilience to Changes	✗	✗	✗	✗	✓

we focus instead on techniques implemented by browsers to avoid fingerprinting. In fact, some of the existing fingerprints are ineffective against existing browsers countermeasures. As our technique does not necessarily rely on a specific function, such protection is more difficult to implement.

Nevertheless, there are few countermeasures that can be adopted in order to avoid our new fingerprinting method. Since the basis of our method is the precision of the timing process itself, countermeasures need to focus on this aspect. While this is possible in the context of a browser, major browsers have already reduced the precision of their timers to avoid several of these attacks performed by JavaScript. Reducing it even further would definitely be an unpopular solution, as more and more applications are pushing for better timing capabilities in JavaScript and HTML5.

Another countermeasure could rely on the use of secure timers, several of which have been proposed in the literature [22, 28, 39]. Their goal is precisely to control timers to make attacks more difficult. These methods are, nevertheless, costly to implement [16].

7 RELATED WORK

Physical Device Fingerprinting. Physical device fingerprinting relies on variations in physical features of devices for their identification. Originally intended for authentication, other uses appeared over the years, such as license binding or statistically determining the source of an attack [12]. Another work focused on wireless device fingerprinting [2, 13] tries to identify a network source rather than a machine. Other techniques have been proposed to physically identify hardware. Examples include the variation in the process in semiconductor foundries [3, 5, 32], Physical Unclonable Functions (PUFs) [14, 27, 33], and exploiting motion sensors embedded on smart devices [7, 8].

Another line of work [34] focused on fingerprinting computers based on the system clock skew extracted by analyzing the different types of timestamps present in the generated traffic. Kohno et al. [23] exploited the TCP and ICPM timestamps to identify computers. Later, Jana and Kasera [20] used the timestamp present on WLAN beacon packets to identify unauthorized wireless access points. More recently, Huang et al. [18] proposed to use the Bluetooth included in some devices to identify the skews. These techniques are really interesting, but the information they rely upon are optional and not always enabled by default in various operating systems and can be easily spoofed by the user. Moreover, they can be easily disabled by users, thus completely preventing the fingerprint computation. Our approach follows instead a schema that allows to obtain a fingerprint without relying on any specific options in the system and without needing to analyze any traffic data, and still allowing a precise identification of computers, even if they share the same hardware and software.

The works closest to our is the recent proposal to use Flash memory to produce both random numbers and generate unique device fingerprints [43] and the proposal to use a clock crystal fingerprinting technique that by using another time reference [35]. However, these approaches differ from ours, because ours only relies on timing functions to fingerprint hardware, being less dependent on the specific hardware configurations. In addition, we have been able to create a generic and simple version of clock fingerprinting that can be used both in simple native code and in the web environment.

Browser Timing Attacks. Timing attacks were first introduced by Felten and Schneider [11] to acquire users' information. Bortz et al. [4] categorized timing attacks into two different categories. The first attacks consisted in measuring the time differences through direct timing. The second ones use information from different sites to obtain client-side data.

The usage of CSS properties can also be a source for timing attacks [24]. Van Goethem et al. [40] proposed the usage of the size of cross-origin resources to detect previous access. Sanchez-Rola et al. [36] discovered installed extensions in all major browsers based on access control settings by means of a timing attack. Mowery et al. [29] presented a method using JavaScript engine benchmarks.

Web Fingerprinting. Web fingerprinting is a method to retrieve user or browser information, typically for tracking. *Cookies* [38] were their first form. Later, it started to be more complex e.g., *evercookies* [21], *cookie syncing*, or *ETags* [1]. Finally, *device fingerprinting* computes a unique identifier for each machine without client-side storage.

As aforementioned, there are two types of device fingerprinting: *attribute-based* and *hardware-level*. The first one uses several browser attributes [9] (e.g.,installed fonts or plugins, `UserAgent`, or screen size and resolution). Unfortunately, these attributes change rapidly, rendering the fingerprint obsolete in less than 10 days according to [41]. The second one, however, uses browser implementations of different APIs to compute the differences between

devices that are based in hardware features (e.g., HTML5 Canvas API or the WebGL API [30]).

8 CONCLUSIONS

Device fingerprinting is an active research topic within web security, specially web device fingerprinting, in the last years. These methods can be used for a wide variety of tasks such as user access control, web tracking or analytics, or targeted attacks.

In this paper, we introduced a time-based device fingerprinting technique. This fingerprinting technique is generic and can work with different functions, making the method adaptable to different environments. In addition, we introduced a set of properties to properly assess the functionality of fingerprinting techniques, filling the gap in current fingerprinting evaluation and proposing a new homogeneous scenario evaluation procedure.

We built a specific native version of our method, CryptoFP, using the function for generating random numbers and evaluating it in a homogeneous scenario with two large sets of machines with the exact same hardware and software installed, showing that is capable of distinguishing every machine. Based upon this implementation, we built an application to web device fingerprinting using the HTML5 Cryptography API that internally uses the same native functions that the machine-version, evaluating and comparing it with state-of-the-art hardware-level web fingerprinting techniques. In a homogeneous scenario evaluation CryptoFP was not as accurate as its native counterpart due to the timing limitations of the JavaScript engine, but still capable of discriminating several of the identical hardware and software machines, outperforming the state-of-the-art methods that were not able to uniquely identity none of the machines. The heterogeneous in-the-wild evaluation shows that the percentage of unique computers identified by CryptoFP was much higher than any other existing method.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments and our shepherd Yinzhi Cao for his assistance to improve this paper. This work is partially supported by the Basque Government under a pre-doctoral grant given to Iskander Sanchez-Rola.

REFERENCES

[1] M Ayenson, DJ Wambach, A Soltani, N Good, and CJ Hoofnagle. 2011. Flash cookies and privacy II: Now with HTML5 and Etags respawning (2011). *Social Science Research Network Working Paper Series* (2011).

[2] Suman Banerjee and Vladimir Brik. 2011. Wireless device fingerprinting. In *Encyclopedia of Cryptography and Security*. Springer, 1388–1390.

[3] Duane S Boning and James E Chung. 1996. Statistical metrology: Understanding spatial variation in semiconductor manufacturing. In *Proceedings of the Microelectronic Manufacturing*. International Society for Optics and Photonics.

[4] Andrew Bortz and Dan Boneh. 2007. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web (WWW)*. ACM, 621–628.

[5] Keith A Bowman, Steven G Duvall, and James D Meindl. 2002. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of solid-state circuits* 37, 2 (2002), 183–190.

[6] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Proceedings of the Network and Distributed System Symposium (NDSS)*.

[7] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses.. In *Proceedings of the Network and Distributed System Symposium (NDSS)*.

[8] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable.. In *Proceedings of the Network and Distributed System Symposium (NDSS)*.

[9] Peter Eckersley. 2010. How unique is your web browser?. In *Proceedings of the Privacy Enhancing Technologies (PETS)*.

[10] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1388–1401.

[11] Edward W Felten and Michael A Schneider. 2000. Timing attacks on web privacy. In *Proceedings of the 7th ACM conference on Computer and Communications Security (CCS)*. ACM.

[12] Russ Fink. 2007. A statistical approach to remote physical device fingerprinting. In *Proceedings of the Military Communications Conference (MILCOM)*.

[13] Jason Franklin, Damon McCoy, Parisa Tabriz, Vicentiu Neagoe, Jamie V Randwyk, and Douglas Sicker. 2006. Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting.. In *Proceedings of the USENIX Security Symposium (SEC)*.

[14] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. 2002. Silicon physical random functions. In *Proceedings of the ACM Conference on Computer and CBommunications Security (CCS)*.

[15] GNU/Linux. 2018. Stress, tool to impose load on and stress test systems. https://linux.die.net/man/1/stress.

[16] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings of the Network and Distributed System Symposium (NDSS)*.

[17] WebAssembly W3C Community Group. 2018. WebAssembly. http://webassembly.org/.

[18] Jun Huang, Wahhab Albazrqaoe, and Guoliang Xing. 2014. Blueid: A practical system for bluetooth device identification. In *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2849–2857.

[19] Clint Huffman. 2014. *Windows Performance Analysis Field Guide*. Elsevier.

[20] Suman Jana and Sneha K Kasera. 2010. On fast and accurate detection of unauthorized wireless access points using clock skews. *IEEE Transactions on Mobile Computing* 9, 3 (2010), 449–462.

[21] Samy Kamkar. 2018. Evercookie – virtually irrevocable persistent cookies. http://samy.pl/evercookie/.

[22] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *Proceedings of the USENIX Security Symposium (Sec)*.

[23] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. 2005. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing* 2, 2 (2005), 93–108.

[24] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. 2013. Cross-origin pixel stealing: timing attacks using CSS filters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1055–1062.

[25] Fabian Lanze, Andriy Panchenko, Benjamin Braatz, and Thomas Engel. 2014. Letting the puss in boots sweat: Detecting fake access points using dependency of clock skews on temperature. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 3–14.

[26] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.

[27] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. [n. d.]. A technique to build a secret key in integrated circuits for identification and authentication applications. In *Proceedings of the Symposium on VLSI Circuits*. IEEE.

[28] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[29] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. 2011. Fingerprinting information in JavaScript implementations. In *Proceedings of the Web 2.0 Workshop on Security and Privacy (W2SP)*.

[30] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of the Web 2.0 Workshop on Security and Privacy (W2SP)*.

[31] Steven J Murdoch. 2006. Hot or not: Revealing hidden services by their clock skew. In *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 27–36.

[32] Sani R Nassif. 2000. Modeling and forecasting of manufacturing variations. In *Proceedings of the International Workshop on Statistical Metrology*.

[33] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. 2002. Physical one-way functions. *Science* 297, 5589 (2002), 2026–2030.

[34] Libor Polčák and Barbora Franková. 2014. On reliability of clock-skew-based remote computer identification. In *Security and Cryptography (SECRYPT), 2014 11th International Conference on*. IEEE, 1–8.

[35] Timothy J Salo. 2007. Multi-Factor Fingerprints for Personal Computer Hardware. In *Proceedings of the Military Communications Conference (MILCOM)*. IEEE.

[36] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *Proceedings of the USENIX Security Symposium (Sec)*.

[37] Michael Schwarz, ClÃĪmentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript . In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*.

[38] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. 2010. Flash Cookies and Privacy. In *Proceedings of the AAAI Spring Symposium: Intelligent Information Privacy Management*, Vol. 2010.

[39] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. 2013. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer.

[40] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. The Clock is Still Ticking: Timing Attacks in the Modern Web. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[41] Antoine vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. https://hal.inria.fr/hal-01652021

[42] W3C. 2018. Web Cryptography API. https://w3c.github.io/webcrypto/Overview.html.

[43] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G Edward Suh, and Edwin C Kan. 2012. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*.