Time Will Tell: Exploiting Timing Leaks Using HTTP Response Headers

Vik Vanderlinden[0000−0002−6142−8057], Tom Van Goethem[0000−0001−6846−9081], and Mathy Vanhoef[0000−0002−8971−9470]

imec-DistriNet, KU Leuven
{first.last}@kuleuven.be

Abstract. To execute timing attacks, an attacker has to collect a large number of samples to overcome network jitter. Methods to reduce such jitter, which is a source of noise, increase the signal-to-noise ratio and thereby improve the attack performance. In this paper, we propose an attack technique that uses timing information exposed in HTTP responses by backend servers or services to reduce the jitter included in an attacker's samples. By first synchronizing the attacker clock to the target, sources of low-accuracy timing data can still be used to exploit timing leaks. We collect real-world data on network latencies and use this data in millions of simulations of the synchronizations and attacks. Our simulations indicate that the synchronization can happen with less than 300 samples for accuracies of 1 ms and works down to an accuracy of under 13 μs. When comparing the performance of our novel synchronization-based timing attack to a classical timing attack which is simulated using the same dataset, our attack is able to reduce the exploitable difference in runtime by an order of magnitude to 1 μs. For equal difference in runtime our attack reduces the required samples up to a factor of 20.

Keywords: Timing Leaks · Side-channels · Network Security

1 Introduction

Timing leaks arise when the execution time of a program depends on private information. By measuring the execution time for a specific private input or state of the program, this input or state can be inferred solely based on the measured duration. Remote timing attacks do exactly the same, with the one difference that they exploit timing leaks over a network such as the Internet. It has been shown over the past twenty years that (remote) timing attacks are possible using a multitude of techniques, initially extracting cryptographic information (keys or coefficients) and later also personally identifiable information (PII) from web applications [15,6,5,4,10,31]. Due to browser mechanisms such as the Same-Origin Policy (SOP), multiple works have explored alternative mechanisms to time the round-trip as accurately as possible using JavaScript callbacks when triggering an error, or by constructing a highly accurate counter in the browser [18,29,34,13]. The first timing attacks simply used the round-trip timings of requests, but more recently other sources of information and more complex attack

scenarios are being used [35,12]. Defenses have been subsequently designed as a response to, or (in an attempt) to prevent these types of attacks [19,1,3,2].

In this paper, we introduce a novel technique that abuses timestamps in network responses to improve the performance of timing attacks. The abused timestamp can be coarse-grained and, for instance, might only increment ("tick") every second. We first synchronize the attacker and victim machines' clocks based on this timestamp by assuring that network requests arrive right before the timestamp is about to increment. After this synchronization process, the attack is executed by measuring how many responses are generated before or after the increment of the (coarse-grained) timestamp. We show that, for various timing differences, our attack technique improves the performance of timing attacks, requiring less samples than a classical timing attack that uses round-trip timings.

All combined, we make the following contributions:

- We introduce a novel timing attack that improves attack performance by utilizing timing information in a target's HTTP responses. We show that such timing information can leak the runtime of a server-side program, even if the accuracy of the exposed timestamp is low.
- We evaluate our clock synchronization method and analyze the performance of this synchronization process between two machines over a network, both in terms of the required number of network requests and the accuracy of the acquired synchronization. We also list the main difficulties of this process.
- We collect real-world network data to perform millions of attack simulations and compare our novel attack against a classical timing attack.

Responsible Disclosure: We have disclosed our findings to the developers of a number of web servers that are mentioned in this paper to warn them about the generation of the date-header after handling a request.

2 Background and Related Work

2.1 Timing Attacks

When timing attacks were introduced two and a half decades ago by Kocher in 1996, most of these attacks were aimed at exploiting cryptographic implementations to extract exponents or keys of specific operations [15]. By measuring the execution time of the cryptographic operations it was shown that these attacks were feasible. Not much later, in 2000, the first timing attack in a web browser was shown by Felten et al., who found a method to determine which pages were visited by a user, thus leaking their browsing history, steering more towards a privacy-related attack [10]. The attacks that are executed in these works always measure local runtimes in a browser or of smartcard-like devices and it was believed that timing attacks against general purpose web servers were infeasible due to the large amount of network noise that packets encounter when traveling over the network to the server [6].

To show that attacks over a network are in fact practical, Brumley and Boneh exploited cryptographic operations in OpenSSL running on a web server in a local network [6]. Two years later Bortz et al. defined the differences between direct timing attacks and cross-site timing attacks, and at the same time also showed that they could expose private information from a web application over a network [4]. These publications led to a multitude of attack vectors, such as the cross-site search attacks by Gelernter et al., which allow an attacker to test for the existence of search results on another site [11]. Other works explored different ways to more accurately time round-trip times or how to time cross-site requests while the necessary information is blocked by the browser [29,34]. Additionally, it was shown that arbitrary memory could be read over the network using a timing attack in a remote Spectre attack [30]. Over the last few years, other attack scenarios have been used to exploit timing leaks, by misusing the features of networking protocols to the advantage of the attacker [12]. Besides practical attacks, Crosby et al. tested multiple statistical methods to compare distributions of measurements and defined the box test to distinguish between roundtrip time distributions, which is now commonly used in timing attacks [8].

2.2 Clock Synchronizations

Every machine with an internal time source suffers from a concept known as clock drift [37]. Clock drift refers to the observation that a clock does not advance at the same rate as a reference clock, i.e., the speed at which a clock ticks differs from the reference clock. Some of the factors that make the clock tick at a different rate are: temperature, current CPU load, etc. Over time, the clock drift may shift the internal representation of time significantly, leading to confusion for the user or problems with the software running on the machine.

A solution is to regularly resynchronize the internal clock with a reference clock. The network time protocol (NTP) implements such a synchronization and is designed to sync the time of two machines over a network. It works with a hierarchy of devices where devices with a higher accuracy timestamp are located higher up in the hierarchy, as indicated by the lower so-called stratum number that is associated to them. A stratum number of zero is reserved for the reference clocks that hold the highest accuracy of time (atomic clocks and the like) [20]. This type of synchronization can be accurate down to the sub-millisecond level, but is very dependent on the network connection and conditions [22]. The threshold for a "synchronized" server is indicated by `MAXDIST` (distance threshold) with a value of 1 s [20]. Another protocol designed for the synchronization of clocks is the Precision Time Protocol (PTP). PTP is much more accurate than NTP but is usually only used on local area networks, and its accuracy is dependent on the latency and jitter of the network connections [23].

Additionally, the signals of a Global Navigation Satellite System (GNSS) such as GPS or Galileo could be used to synchronize multiple clocks over a large area. The signals received from a GNSS satellite can be used to triangulate a position, but also to derive the current time with high accuracy. A disadvantage is that all machines attempting to synchronize their clocks should have access

to a physical receiver, which is usually not the case for on-premise servers or virtual instances in a (public) cloud [33,9].

These processes synchronize clocks from devices on a certain sized network to a specific accuracy depending on the network conditions, but the implementation or accuracy of the synchronization is not good enough for the purpose of using our technique to exploit timing leaks. NTP's accuracy is too low and PTP only works for small networks, both conditions that make our attack impossible. In addition, for NTP and PTP, client and server work together to synchronize their clocks, which would not be a feasible requirement for our attack. GNSSs on the other hand require no collaboration between multiple devices and are able to provide receivers with highly accurate timing information. Despite the advantages, synchronizations using these systems are not generally feasible due to the lack of hardware-support on most victim machines. As in every synchronization algorithm, clock drift may invalidate the correct synchronization requiring the process to be repeated regularly.

3 Threat Model and Attack Mechanism

In this section, we first discuss the threat model and outline the attack mechanism in detail, after which we introduce a formal model for timing attacks and use this model to illustrate the expected benefit of our synchronization-based timing attacks.

3.1 Threat Model

We assume the attacker is remote and will position themselves to have the most reliable upstream path possible. This is in contrast with typical timing attacks where the adversary locates themselves near the target to have both reliable upstream *and* downstream network characteristics. Since our attack eliminates downstream jitter, the goal is not necessarily to be as close to the target as possible, but to have the best possible network characteristics on the upstream path. Although being located near a target may help in achieving this goal, this may not necessarily be the closest position to the target because network characteristics can be different for up- and downstream paths [26,7]. Since only the upstream network characteristics matter, our threat model is more relaxed compared to typical classical attacks. An attacker can easily monitor multiple network connections and select the most advantageous one to use in the attack.

We assume the attacker can read responses that contain timing values and thus do not consider cross-origin attacks (which would require the SOP to be relaxed by the CORS header) [18,36].

In comparison with other attacks such as the Timeless Timing Attacks, this attack does not require the use of HTTP/2 [12]. In contrast, our attack is not limited to a specific HTTP version. We assume an attacker that can exploit timing leaks on endpoints that will leak data about the website state or about website users. In order to leak information, the timing values that are being

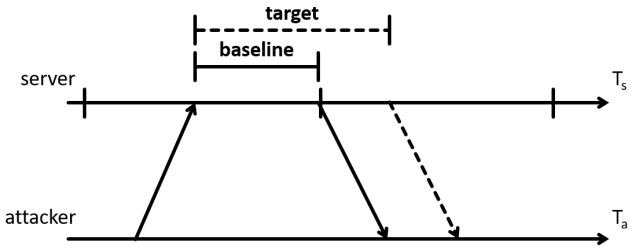

Fig. 1. Overview of the attack mechanism where T_s and T_a represent the server and attacker local time, respectively. The vertical bars in the server's timeline indicate a rollover of the observed timing value on the server.

returned have to be generated after a security-sensitive operation, otherwise private information can never be exposed by means of those timings. The attacker does this from a machine under their control in a direct attack.

3.2 Attack Mechanism

Our attack uses timing values in HTTP responses to gain knowledge about the relative runtimes of two endpoints on a server. By looking at the HTTP response and counting the number of times that the response has been sent by the server before or after a timing value rolls over to the next value, we can derive whether the execution time on the server took longer compared to another (default) page which will be referred to as the baseline page or request (see Figure 1). The attack starts with a synchronization step in which we attempt to find an offset for the attacker to send baseline requests such that out of all respective responses 50% are sent before, and 50% after the server's clock rollover. By finding such an offset we are able to compare the baseline and a target endpoints. If the number of responses that were generated after the clock rollover for a specific target endpoint is significantly higher (say 90%) than in the baseline case (ideally 50%), it is clear that the request took longer to process.

We emphasize again that in order for this (and any) attack to work, the timing information used has to be generated after a security-sensitive operation has been executed. Otherwise there is no possibility that it leaks private data about that operation. This is inherently also depicted in Figure 1. If in this figure a request's arrival time is reflected in the respective response, the value in both responses will be identical.

Also important is that this method of exploiting timing leaks does not facilitate the direct observation of the absolute difference in runtime between baseline and target endpoints, which will eliminate some attack variants that depend on the absolute runtime to estimate for instance sizes of server-side items [4]. Whether the difference in number of requests before and after the target clock rollover can be used to estimate an approximate difference in runtime is left for future work.

3.3 Formal Model

We first create a theoretical model of timing attacks to illustrate how our novel technique can improve the effectiveness of attacks. To ease comparison with related work, we follow the model of Crosby et al. [8], and split the response time R into the processing time T and propagation time B, leading to $R = T + B$. Both the processing and propagation time can be further split up into sub-operations, e.g., to account separately for decoding network packets, generating a response, the routing operations for every hop in the network path, etc. We can model this as the sum of all sub-operation $k \in \{1..K\}$ before the timestamp is generated, and the sum of all sub-operations $\ell \in \{1..L\}$ after the timestamp is generated. Without loss of generality, we assume the processing and propagation time can be split into the same number of sub-operations. Letting T_t denote the operation that generates the timestamp, we get:

$$R = \sum_{k=1}^{K} (T_k + B_k) + T_t + \sum_{\ell=1}^{L} (T_\ell + B_\ell) \tag{1}$$

In a typical timing attack, the jitter of all sub-operations before and after some security-sensitive operation T_s, where $1 \leq s \leq K$, reduce the performance of a timing attack.

In our attack, instead of using the round-trip response time, we use the timestamp in the response. Let T denote the received timestamp in a response, normalized to 0 or 1 to represent either before or after the clock tick. We use $\lceil \cdot \rceil$ to represent this normalization. This implies T is a random value depending on the propagation and processing time preceding the generation of the timestamp:

$$T(d) = \left\lceil d + \sum_{k=1}^{K} (T_k + B_k) + T_t \right\rceil \tag{2}$$

Parameter d represents when the attacker sends the request, i.e., the offset relative to the attacker's clock tick. The above formula highlights that the timestamp must be generated after the security-sensitive operation T_s, otherwise the value of T would not depend on T_s, meaning T would not leak sensitive info. Additionally, we can see that the result is independent of any latency that occurs after the generation of the timestamp. In the synchronization phase of our attack, we search for an offset d such that $E[T(d)] = 0.5$. In other words, we search for an offset d such that half of the requests arrive before the clock tick and the other half afterward. The value of d will be between zero and the precision (granularity) of the timestamp, e.g., if the timestamp is rounded down to seconds, then d lies between zero and one second. In practice, achieving exactly 0.5 is infeasible. Instead, in an attack, we determine whether a target request is processed faster or slower than a reference requests. That is, we are trying to answer whether:

$$E[T_{\text{baseline}}(d)] \stackrel{?}{=} E[T_{\text{attack}}(d)] \tag{3}$$

Here T_{baseline} is the (benign) request that we synchronize with, and the attacker's goal is to determine whether request T_{attack} takes a different amount of time to

process. In practice, the adversary performs two requests *baseline1* and *baseline2* that have the same generation time T_s, and a *target* request with a different generation time T'_s. The attack succeeds when a classification method can be found that distinguishes *baseline1* from *target* while not distinguishing *baseline1* from *baseline2*. The remainder of the paper explores how to determine d in practice and how to distinguish requests with different processing times.

4 Clock Sync

In order to execute our attack, a vital step is to synchronize the attacker and victim clocks as accurately as possible. In this section, we introduce a novel method to perform such a synchronization by only relying on coarse-grained timestamps provided by the server.

4.1 Synchronization Mechanism

To synchronize the clock of the attacker to the victim clock, the same source of timing information as in the attack can be used. We will use the timestamp returned by a server, which is based on the server's internal clock and is usually truncated to a predefined accuracy (called the timestamp's granularity).

The goal of the clock synchronization is to find a local offset d for the attacker such that the server's responses are sent 50% of the time before and 50% of the time after the server's clock rollover. The method we use to find d is illustrated in Figure 2. Initially, we know that $d \in [S_1, E_1]$, for instance, if the timestamp has a granularity of one second, then we start with $d \in [S_1 = 0, E_1 = 1]$. In each *sync iteration* we recursively decrease the size of this interval while assuring the resulting server's clock rollover remains inside the interval. To reduce the size of an interval, the attacker sends multiple requests at different offsets in this interval (see below for details). For instance, given the interval $[0, 1]$, multiple requests will be sent at offsets 0.0, 0.1, and so on up until 0.9. The requests that are sent at these different offsets are called *probes* and the number of probes sent at every offset is a parameter of our synchronization method. The adversary might then observe that the server's clock rolls over between, e.g., offset 0.3 and 0.4, and updates the interval to $[0.3, 0.4]$. In Figure 2 this is illustrated by updating the interval $[S_i, E_i]$ to $[S_{i+1}, E_{i+1}]$. This process continues until a given number of sync iterations have been performed. In practice, each sync iteration reduces the interval by at most an order of magnitude and the attacker must at all times be able to send requests accurately at the offset d.

By repeatedly performing this process we can synchronize clocks as accurately as desired. Within an interval, the estimated location of the clock rollover can be found as follows: At each selected offset within the interval, a number of probes, i.e., network requests, are sent to the server. The number of probes to send is a parameter of the algorithm. Care must be taken when sending these probes: sending too many at once may congest the attacker's network, meaning a small waiting time must be enforced between sending consecutive probes.

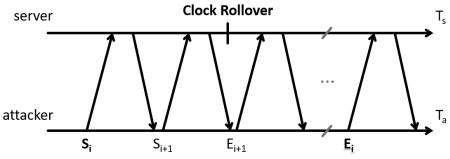

Fig. 2. Clock synchronization where the goal is to find an offset such that the server's response is sent right when its timestamp rolls over to the next value. This implies that 50% of responses will be sent before the server's rollover and 50% after.

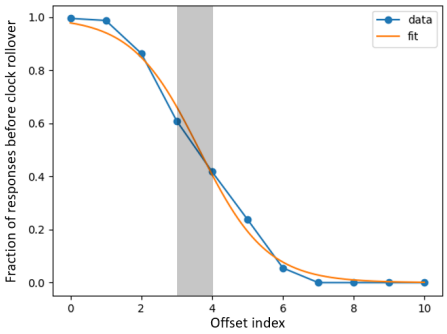

Fig. 3. An example of the analyses of an interval using an inverse *tanh* fit on intervals with a width of 100 μs. The transition can now be defined as the point at which the *tanh* has a value of 0.5.

When all responses are received we can calculate, for each sampled offset, the percentage of responses with a timestamp before its rollover. When the size of the interval is large, meaning the sampled offsets are relatively far away from each other, there will be a clear instant transition from 100% to 0% and we can reduce the interval's size based on this. For higher accuracy synchronizations, e.g., when the interval is in the microsecond range, there will no longer be a clear transition. In that case, to determine the new smaller interval, we plot the percentage of responses containing a timestamp before the clock tick as a function of the sampled offset. An example of this is shown in Figure 3. To now determine the new smaller interval, an inverse *tanh* function is fitted onto to the data due to the observed shape of the curve that approximates an inverse *tanh* and the computational ease of fitting this function. The estimated value of the clock tick is then the point where the fitted *tanh* function reaches 50%, and the new interval is where the *tanh* function is, for example, in the 40% to 60% range. Overall, the synchronization quickly reaches an accuracy of a millisecond and can go on into the microsecond range.

4.2 Influence of Clock Drift

The major difficulty when synchronizing two clocks is the clock drift. Any machine inherently has a specific amount of clock drift. This drift is often determined by external factors such as the temperature, processing load on the server (particularly important for a shared VM in the cloud) and other factors [21]. When two machines are used in the attack (one attacker, one victim), they experience a relative clock drift between the two machines. This has the implication that when their clocks are synchronized, the synchronization is not valid for an indefinite time, meaning clocks have to be resynchronized periodically. Moreover,

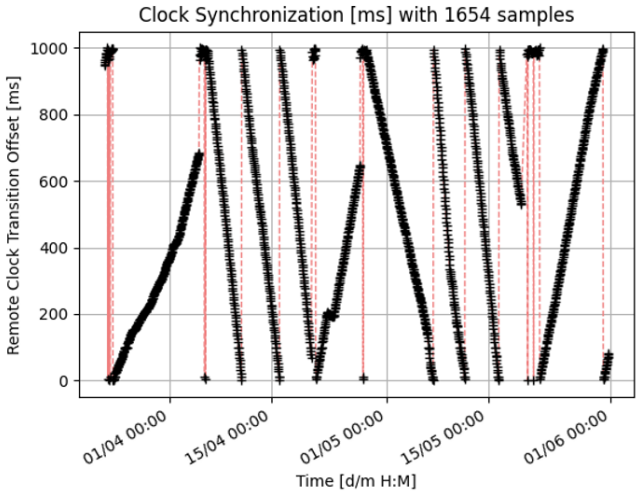

Fig. 4. Clock drift between a machine in our university network and an instance in a public cloud depicted over time.

clock drift may impact the synchronization process itself, especially when performing many sync iterations. That is, when doing many sync iterations, more probes must be sent, and this significantly increases the duration of the synchronization process. Recall that we cannot send many requests simultaneously, as this would have complicated side-effects such as being blocked by firewalls or with very high amounts of traffic, self-induced network congestion, which decreases the synchronization and attack performances and should of course be avoided. Due to the long duration, the relative clock drift may in principle be so high that the synchronization is invalidated before it is found.

Fortunately, previous work, such as that of Zander and Murdoch [37], has shown that clock skew can be found and thus it can also be compensated for. To independently confirm this, we measured the drift of one machine on our university network and an instance in the public cloud, for over two months. Their clocks were synchronized every hour using our synchronization algorithm to an accuracy of a millisecond, with the resulting drift show in Figure 4. The sudden jumps in the drift are either due to the data wrapping around to the other side of the interval or due to the client machine rebooting. The long, relatively straight tendencies in the drift confirm that compensating the drift is feasible. An attacker can monitor this drift and determine whether it is behaving linearly between some machines and the target and select a machine with optimal conditions to be able to compensate the drift as easily as possible for the attack.

Motivated by these observations, we will assume in the remainder of the paper that an adversary can measure and correct the influence of clock skew while performing our attack, meaning we do not explicitly have to model it in our experiments.

5 Data and Simulation

To fairly evaluate our new attack against classical timing attacks, we first collected real-world roundtrip data from connections between our university network and datacenters in the EU and US, as well as data between those datacenters themselves. This section will outline exactly which data was collected, how that was done, and how the simulations were performed.

5.1 Timing data prevalence

For all evaluations in this paper, we make use of the HTTP *date* header as the source of timing values, which is present for most web pages. When looking at the HTTP Archive dataset of May 2023, 99.76% of HTML pages are served with a date header, and 99.18% of all 1.226 billion responses in the dataset have a date header set, showing that the header might be a reliable choice for an attacker to use as the source of timing information in the attack [14]. It is important to mention that some date headers contain a fixed value, or a value that is updated less than once every second. These values would not be useable in our attack. Because the HTTP standard describes date formats to have a granularity of one second, it is expected that most servers update their date header about once a second [28]. Fixed values can never contain information related to security-sensitive operations on a server, and if the returned values have a too low granularity the attack becomes infeasible due to the amount of time it would take to synchronize the clocks and execute the attack. We crawled a number of pages of the Tranco top 10k domains[1] and evaluated the dynamic nature of the date headers [16]. To do so, each of the 24 928 pages was visited twice with a wait of 30 s in between, after which the detected timing values were matched between the two visits. If a timing value remains unchanged over the 30 s period, it was removed as an interesting value. We found that 92.36% of the crawled response documents were served with a dynamic date header, which makes it an ideal value to use in an attack. In some cases the date header is set to a fixed value, such as the Unix epoch. The results obtained in these experiments should be interpreted as absolute upper bounds on the number of pages that could be vulnerable, because many pages will return timestamps that do not include the duration of sensitive operations.

By using the date header, we also show that the used timestamp does not need to be very accurate, since this header is usually only accurate down to a second. It can be expected that more accurate timing information leads to even better results when using our attack, especially with regards to the synchronization process that becomes more obsolete with growing accuracy of timestamps, since the offset it attempts to find can be derived from the timestamp value up to a certain point.

It remains the case that the timing information has to be generated after a security-sensitive operation in order to be useful to leak private information. We

[1] Available at `https://tranco-list.eu/list/85NV`.

tested multiple popular servers such as nginx, litespeed, Apache httpd and the popular frameworks NodeJS Express and Python Flask [24,17,32,25,27]. Most of these platforms generate their date header values when a response is constructed on the server and thus after the execution of any backend script(s), hereby potentially allowing the leakage of private information depending on the backend operation(s). In fact, Apache's httpd was the only of these platforms that generated its date header values at the moment a request is received, and therefore a different timestamp would have to be used to attack a target that is running the Apache httpd server engine [32]. Such timestamp could for instance be set by the programming language or by a developer because the programming language may still have access to high accuracy clocks. It will, however, mean that the likelihood of finding such timing values decreases.

5.2 Collecting roundtrip times

We first collected real-world roundtrip data so we can accurately simulate both our new and classical timing attacks. For this, we set up servers in the west of the EU and the east of the US with Amazon AWS and in addition used servers from our university's internal network in Europe. Each of the 4 collection servers were located in one of two datacenter locations in the west of the EU and all of them collected information from two hosting servers, one of which was in another (nearby) datacenter in the west of the EU, the second one that was in the east of the US, gathering data over a transatlantic connection. There were no samples collected originating from the US with hosting servers in the EU.

Each of the collection servers collected over 15 million samples for EU-EU cases and over 5 million samples for EU-US cases. Obviously, the latency for packets traveling to the US is much higher resulting in a slower data-gathering process. In order to collect real-life data, we instruct the server to sleep for a number of milliseconds using PHP's builtin sleep function in order to simulate an actual time-consuming task on the server. For each request the roundtrip time was collected with nanosecond accuracy, as would be done in a classical timing attack.

The collected data shows that the roundtrip times at some times do not follow a perfect normal, or log-normal distribution, as can be expected when taking measurements on the Internet. In some cases, the actual distribution of the collected samples seems to be a collection of multiple (log-)normal distributions superimposed. The imperfections in the data are still used in the attack simulations in order to get the most accurate results possible rather than using some form of idealized, generated data. The main characteristic that is important and would be difficult to accurately generate or simulate is the network jitter. By using these real-world samples, the jitter is exactly the same as when an attacker would be performing the attack over these connections. Because the samples will be used in a sequential manner, possible short-term deviations or variations in the latency and jitter will also be mimicked in our simulations.

In our evaluation, we want to control the difference in the upstream and downstream network conditions, so we can evaluate the impact of asymmetric

network paths on the performance of our attack. This difference is hard and arguably even impossible to fully control in real-world networks. Although an attacker would ideally like to be able to, they too cannot control the way the up- and downstream path behave, and will have to resort to monitoring multiple paths and selecting the best one. To nevertheless separately control the downstream and upstream parameters, while at the same time simulating real-world network jitter, we will simulate the server's date header based on the measured round-trip time. This allows us to easily simulate different conditions of the upstream network path, by using a varying fraction of the roundtrip time values.

A second reason to simulate the server's data header based on the round-trip time, is that it allows us to simulate or eliminate clock drift. In other words, real timestamps returned by the server are influenced by the server's clock drift, and this clock drift is practically impossible to afterwards eliminate or change.

During our simulations, we will also configure a random difference between the clock of the server and client. This timing difference, combined with the average upstream network latency, will be learned by our synchronization method. We will continue with a more detailed explanation of the simulations next.

5.3 Synchronization and attack simulations

The goal of the attack simulations is to be as similar to real-life attacks as possible. The simulation software was built in such a way that only the interaction with the server is mimicked and as such, only this interaction should be re-implemented to attack actual target servers.

For every simulation, a random offset between the client and server clocks is selected. It is the job of the synchronization process to find this offset, just as it would be required against an actual target server. Each simulation runs at a very high speed, which is the advantage compared to the real-life attacks. Without using simulations, new network requests would have to be made for every run of an experiment, making it much slower than a simulation. Another advantage of simulations is that each parameter can be evaluated independently if necessary. In order to correctly evaluate our synchronization-based attack against the classical timing attacks, the classical attacks were also simulation using the same data, to eliminate discrepancies between the used datasets. This also allows us to verify that the simulation is correct because the performance of the classical timing attack corresponds with the expected values of this type of attack.

After synchronization, to achieve a successful attack, the adversary should be able to detect that two baseline requests *baseline1* and *baseline2* are similar (baselines case), while being able to distinguish *baseline1* from *target* (baseline versus target case). Here *target* has a different execution time than the baseline. To perform classification, we calculate the following variables, where *attack* can equal either *baseline2* (baselines case) or *target* (baseline versus target case):

$$X_{attack} = \Pr[T_{\text{baseline1}}(d) = 0 \wedge T_{\text{attack}}(d) = 1] \tag{4}$$

$$Y_{attack} = \Pr[T_{\text{baseline1}}(d) = 1 \wedge T_{\text{attack}}(d) = 0] \tag{5}$$

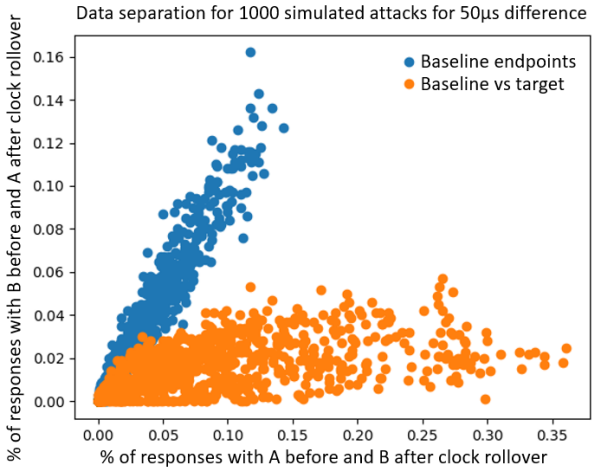

Fig. 5. Example simulation output, each dot is an attack.

Here X_* represents the percentage of times that the baseline response was sent before the clock tick and that the attack response was send after the clock tick (variable Y_* represent the opposite). The values for these variables are determined by sending a requests and calculating the percentage of times that the conditions in the above formulas hold.

Figure 5 shows the resulting data from 1 000 simulated attacks for both the baselines and baseline versus target cases, where the x-axis represents X_* and the y-axis represents Y_*. Each attack is plotted as a point and executing an attack boils down to finding a classification between the baselines and baseline versus target datasets that correctly classifies 95% or more of the attacks. In the baselines case, we see that $X_{\text{baseline2}} \approx Y_{\text{baseline2}}$ (blue dots in Figure 5). This confirms that both baseline responses have the same probability of being generated after or before the clock tick. In the baseline versus target case, we see that $X_{\text{target}} \gtrsim Y_{\text{target}}$ (orange dots in Figure 5). In other words, the attack response is more often generated after the clock tick compared to the baseline response, which is expected since the target response takes longer to generate.

To search for a classification, the regression lines of the two datasets were taken and 10 000 lines spanning the space in between these regressions were evaluated each time. If a line could be found that correctly distinguishes the datasets with the previously mentioned accuracy, the attack was considered successful.

6 Results

6.1 Clock Synchronization

The performance of the clock synchronization is important information for an attacker when they want to select the optimal synchronization parameters. De-

Table 1. The median of the accuracy and number of samples required for synchronizing for a specified number of iterations, which in practice means decreasing the tested offsets to the accuracy specified in this column. At the top, the number of requests sent for each sync probe offset in the synchronization phase are listed (recall Section 4.1).

Metric	Sync Iterations	nb requests per sync probe offset				
		10	20	50	100	200
Required nb samples	1 ms	300	600	1 500	3 000	6 000
	100 μs	400	800	2 500	5 000	10 000
	10 μs	800	2 000	5 500	12 000	24 000
	1 μs	4 200	11 000	32 500	70 000	144 000
Deviation (μs)	1 ms	173.9	161.7	188.1	242.2	348.2
	100 μs	27.5	26.5	24.1	22.5	21.2
	10 μs	38.9	29.6	19.3	14.9	12.9
	1 μs	40.3	29.3	19.9	16.2	14.7

ciding if they want to synchronize for a number of iterations (which in practice means reducing the final offsets that are being tested to an accuracy of a microsecond or a millisecond) has a significant impact on the attack performance (as will be discussed later) but has the disadvantage that it requires a higher number of samples during the synchronization process, which will also take a longer time, thereby possibly counteracting the attack performance improvements. On top of that, for each number of sync iterations, the number of requests sent for each offset can be selected as a separate parameter.

To evaluate the performance of the clock synchronization, each simulation gathered metrics about its accompanying synchronization's performance. Specifically, the number of total requests that had to be made, the deviation from the correct clock rollover offset to be found, and the number of tries that were necessary to synchronize were gathered. From a total of over 2.5 million simulations, the median values for these metrics are shown in Table 1 (top). As expected, a synchronization with a lower number of iterations requires less samples. The increase in the number of samples required is not inversely linear in relation to the number of sync iterations. Due to the complexity of synchronizations with more iterations, the number of samples rises more than linear. The offsets used in the synchronization are now smaller than network jitter, which makes it increasingly difficult to continue synchronizing and adds uncertainty to the process. This means that instead of clearly finding a clock transition, the exact location is hidden by the network noise that cannot be removed. The implementation of the clock synchronization will thus not be able to locate the clock transition

Table 2. The number of samples required to attack a server in the US, provided that clock synchronization has already been performed. In this table, the resulting samples sizes follow from the synchronization to 10 μs. A dash (-) is shown if the attack was unsuccessful or required more than 50 000 samples. Data showing the performance for other synchronization iterations can be found in Appendix A.

Attack	$1\mu s$	$2\mu s$	$5\mu s$	$10\mu s$	$20\mu s$	$50\mu s$	$100\mu s$	$200\mu s$	$500\mu s$	$1ms$
Classical	-	-	-	18 539	1 907	530	121	40	7	7
TWT (RTT/1.5)	-	45 329	6 104	1 830	468	136	96	36	34	33
TWT (RTT/2.0)	-	20 110	3 480	1 025	348	113	63	31	29	29
TWT (RTT/3.0)	48 126	9 204	1 741	519	202	104	46	43	40	40

between two adjacent offsets and has to increase the search space for the next iteration. This also has an impact on synchronization time.

For the deviation shown in Table 1 (bottom) the results show that the accuracy indeed increases when a higher number of synchronization iterations are performed. In the case of the 1 ms entry, the deviation increases with larger samples sizes for each tested probe offset, against intuition. This is because in the 1 ms entry, each iteration recurses on the interval for which the fraction of responses after the clock rollover is between 0% and 100%. For instance, if the current interval is $[0.02, 0.03]$, and only 10 requests per probe offset are used, the procedure may find that 100% of responses at offset 0.022 are sent before the clock rollover, and 0% at 0.023, meaning the new interval becomes $[0.022, 0.023]$. However, when using 200 requests per probe offset, 99.5% of responses at offset 0.022 may now arrive before the clock rollover, and 0.05% at offset 0.023, so the new interval, for instance, becomes $[0.021, 0.024]$ instead. This drives the number of samples required up and decreases the accuracy of the synchronization. In the sub-millisecond range, the *tanh* fit is used, which works better because even if there are a small number of samples returning on the other side of the clock rollover, this will not have a large impact on the fitted curve and thus the interval for the next recursion remains as small as when using fewer samples per probe.

6.2 Attack

Table 2 shows the attack performance after synchronization. That is, the number of requests used for synchronization are not accounted for in this table. Because a synchronization can be maintained against a specific server, an adversary can use one synchronization to perform multiple subsequent attacks, and we therefore evaluate the synchronization and attack separately. The clock synchronization iteration down to 10. μs was used to arrive at each of the sample sizes. In Appendix A, the full resulting data can be found, which shows the performance for each clock synchronization iteration.

The results show that the number of requests necessary to perform an attack are in most cases much lower than in the classic timing attack. Table 2 also shows the performance for three different upstream network conditions. For instance, the bottom row shows the attack performance when the jitter in the upstream path is 1/3rd of the round-trip time jitter that we collected empirically (and analogous for RTT/2.0 and RTT/1.5). We can observe that a better upstream connection, i.e., when the collected round-trip times are divided by a higher constant, less samples are needed to perform our novel attack. Results were similar when using the round-trip times that we collected between different servers and locations. All combined, we can successfully exploit a timing difference of 10, 5 and with a good upstream network path even down to $1\mu s$, which is not possible using a classical timing attack.

6.3 Defenses

The main defense against this type of attack is to expose the time at which the request came into the server instead of the time at which the timing value is actually requested by the program. The compromise that is made is that the time that is communicated to the client may be out of sync for a couple of hundred milliseconds, but that can occur naturally due to high amounts of jitter as well.

Another defense is to decrease the accuracy of the timing information and instead of updating the header value for each second, round these values to five or ten seconds. This does not eliminate the vulnerability, but makes it even more difficult to exploit, due to the high waiting period between each clock tick.

7 Conclusion

While the exposure of timing information such as the current time on a server may be regarded as normal, we have shown that developers should be careful when exposing seemingly harmless information. Timing information that is exposed in HTTP responses can be used to exploit timing leaks by synchronizing the clocks of the attacker and the victim. We gathered real-world data on latencies between servers in the EU and the US, and used this data to run millions of simulations of our complete attack process. Our novel attack was shown to improve a classical timing attack by decreasing the necessary amount of requests and we could successfully exploit a smaller timing difference compared to the classical attack.

To finish we proposed two easy defenses. One defense suggests to always use the time at which a request arrives at the server in the responses. This defense always eliminates the timing leak. Another defense is to decrease the accuracy of the timing information. This is not as effective at the previously proposed defense.

Acknowledgements This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity.

References

1. Alex Christensen: Reduce resolution of performance.now. https://bugs.webkit.org/show_bug.cgi?id=146531 (2015)
2. Boris Zbarsky: Chromium: window.performance.now does not support sub-millisecond precision on windows. https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110 (2015)
3. Boris Zbarsky: Clamp the resolution of performance.now() calls to 5us because otherwise we allow various timing attacks that depend on high accuracy timers. https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab (2015)
4. Bortz, A., Boneh, D., Nandy, P.: Exposing private information by timing web applications. In: 16th International World Wide Web Conference, WWW2007. pp. 621–628 (2007). https://doi.org/10.1145/1242572.1242656
5. Brumley, B.B., Tuveri, N.: Remote Timing Attacks Are Still Practical. In: Lecture Notes in Computer Science, vol. 3523, pp. 355–371 (2011). https://doi.org/10.1007/978-3-642-23822-2_20, http://link.springer.com/10.1007/978-3-642-23822-2_20
6. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks **48**(5), 701–716 (aug 2005). https://doi.org/10.1016/j.comnet.2005.01.010, https://linkinghub.elsevier.com/retrieve/pii/S1389128605000125
7. Cox, B.: Splitting the ping. https://blog.benjojo.co.uk/post/ping-with-loss-latency-split (2022)
8. Crosby, S.A., Wallach, D.S., Riedi, R.H.: Opportunities and Limits of Remote Timing Attacks. ACM Transactions on Information and System Security **12**(3), 1–29 (jan 2009). https://doi.org/10.1145/1455526.1455530, https://dl.acm.org/doi/10.1145/1455526.1455530
9. EUSPA: European GNSS Service Centre. https://www.gsc-europa.eu/, accessed: 2023-05-28
10. Felten, E.W., Schneider, M.A.: Timing attacks on Web privacy. In: Proceedings of the 7th ACM conference on Computer and communications security - CCS '00. pp. 25–32. ACM Press, New York, New York, USA (2000). https://doi.org/10.1145/352600.352606, http://portal.acm.org/citation.cfm?doid=352600.352606
11. Gelernter, N., Herzberg, A.: Cross-Site Search Attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. vol. 2015-Octob, pp. 1394–1405. ACM, New York, NY, USA (oct 2015). https://doi.org/10.1145/2810103.2813688, https://dl.acm.org/doi/10.1145/2810103.2813688
12. van Goethem, T., Pöpper, C., Joosen, W., Vanhoef, M.: Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. Proceedings of the 29th USENIX Security Symposium pp. 1985–2002 (2020)
13. van Goethem, T., Vanhoef, M., Piessens, F., Joosen, W.: Request and conquer: Exposing cross-origin resource size. Proceedings of the 25th USENIX Security Symposium pp. 447–462 (2016)
14. HTTP Archive Contributors: The HTTP Archive. https://httparchive.org/, accessed: 2023-05-28
15. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: CRYPTO - Annual International Cryptology Conference, pp. 104–113 (1996). https://doi.org/10.1007/3-540-68697-5_9, http://link.springer.com/10.1007/3-540-68697-5_9

16. Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., Joosen, W.: Tranco: A research-oriented top sites ranking hardened against manipulation. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium. NDSS 2019 (Feb 2019). https://doi.org/10.14722/ndss.2019.23386
17. LiteSpeed Technologies Inc.: LiteSpeed Web Server. https://www.litespeedtech.com/products/litespeed-web-server, accessed: 2023-06-04
18. MDN contributors: Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (2023)
19. Mehta, A., Alzayat, M., de Viti, R., Brandenburg, B.B., Druschel, P., Garg, D.: Pacer: Network Side-Channel Mitigation in the Cloud (2019), http://arxiv.org/abs/1908.11568
20. Mills, Delaware, Martin, Burbank and Kasch: A Border Gateway Protocol 4 (BGP-4). RFC 5905, RFC Editor (June 2010), https://www.rfc-editor.org/rfc/rfc5905.txt
21. Murdoch, S.J.: Hot or not: Revealing hidden services by their clock skew. pp. 27–36. ACM Press (2006). https://doi.org/10.1145/1180405.1180410, http://dl.acm.org/citation.cfm?doid=1180405.1180410
22. Network Time Foundation: Clock discipline algorithm. https://www.ntp.org/documentation/4.2.8-series/discipline/ (2022)
23. Network Time Foundation: Ieee 1588 precision time protocol (ptp). https://www.ntp.org/reflib/ptp/ (2022), accessed: 2023-05-28
24. Nginx Contributors: Nginx. https://nginx.org/en/, accessed: 2023-05-28
25. OpenJS Foundation: Express - Node.js web application framework. https://expressjs.com/, accessed: 2023-06-04
26. Pucha, H., Zhang, Y., Mao, Z.M., Hu, Y.C.: Understanding network delay changes caused by routing events. ACM SIGMETRICS Performance Evaluation Review **35**(1), 73–84 (jun 2007). https://doi.org/10.1145/1269899.1254891, https://dl.acm.org/doi/10.1145/1269899.1254891
27. Python Contributors: Welcome to Flask. https://flask.palletsprojects.com/en/2.3.x/, accessed: 2023-06-04
28. R. Fielding, M. Nottingham and J. reschke: Rfc 9110: Http semantics - date/time formats. https://www.rfc-editor.org/rfc/rfc9110#http.date (2022)
29. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10322 LNCS, pp. 247–267 (2017). https://doi.org/10.1007/978-3-319-70972-7_13, http://link.springer.com/10.1007/978-3-319-70972-7_13
30. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D.: NetSpectre: Read Arbitrary Memory over Network. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **11735 LNCS**(July), 279–299 (jul 2018). https://doi.org/10.1007/978-3-030-29959-0_14, http://arxiv.org/abs/1807.10535
31. Smith, M., Disselkoen, C., Narayan, S., Brown, F., Stefan, D.: Browser history re:visited. 12th USENIX Workshop on Offensive Technologies, WOOT 2018, co-located with USENIX Security 2018 (1) (2018)
32. The Apache Foundation: The Apache HTTP Server Project. https://httpd.apache.org/, accessed: 2023-05-28
33. United States Space Force: GPS: The Global Positioning System. https://www.gps.gov/, accessed: 2023-05-28

34. Van Goethem, T., Joosen, W., Nikiforakis, N.: The clock is still ticking: Timing attacks in the modern web. In: Proceedings of the ACM Conference on Computer and Communications Security. vol. 2015-Octob, pp. 1382–1393 (2015). https://doi.org/10.1145/2810103.2813632
35. Vanderlinden, V., Joosen, W., Vanhoef, M.: Can You Tell Me the Time? Security Implications of the Server-Timing Header. In: Proceedings 2023 Workshop on Measurements, Attacks, and Defenses for the Web. No. March, Internet Society (2023). https://doi.org/10.14722/madweb.2023.23087, https://www.ndss-symposium.org/wp-content/uploads/2023/02/madweb2023-23087-paper.pdf
36. whatwg contributors: Fetch standard: Cors protocol. https://fetch.spec.whatwg.org/#http-cors-protocol (2023)
37. Zander, S., Murdoch, S.: An improved clock-skew measurement technique for revealing hidden services (2008)

Appendices

A Complete Attack Data

Tables 3 and 4 show the full results that have been tested for a victim in the EU and US respectively. Each row lists the selected synchronization iteration and thereafter the required number of samples to differentiate a specific timing difference. Based on this data, an attacker could select their preferred number of synchronization iterations based on the requirements of their desired attack. In both cases the results from the classical attack are shown in the first row and perform significantly worse than the synchronization-based attack. From these tables the importance of the synchronization is clear. When comparing the performance of the $1ms$ synchronization to the others, the most coarse synchronization clearly performs worse.

Table 3. The number of samples required to attack a server in the EU, provided that clock synchronization has already been performed. Each row indicates what synchronization iteration was used to obtain the results in that row. A dash (-) is shown if the attack was unsuccessful or required more than 50 000 samples.

Attack	Sync	$1\mu s$	$2\mu s$	$5\mu s$	$10\mu s$	$20\mu s$	$50\mu s$	$100\mu s$	$200\mu s$	$500\mu s$	$1ms$
Classical	/	-	-	-	-	2 249	436	151	44	8	7
TWT (RTT/1.5)	1ms	-	-	-	-	-	37 444	9 035	2 285	391	62
TWT (RTT/1.5)	100us	-	-	17 384	4 053	1 093	243	105	44	32	30
TWT (RTT/1.5)	10us	-	-	14 400	3 481	996	221	94	41	28	28
TWT (RTT/1.5)	1us	-	-	13 281	3 173	930	213	102	35	24	24
TWT (RTT/2.0)	1ms	-	-	-	-	-	36 944	9 424	2 197	324	51
TWT (RTT/2.0)	100us	-	-	9 054	2 399	691	198	92	36	29	27
TWT (RTT/2.0)	10us	-	-	7 616	1 954	519	156	60	34	32	32
TWT (RTT/2.0)	1us	-	-	7 418	1 882	556	146	63	30	29	29
TWT (RTT/3.0)	1ms	-	-	-	-	-	27 734	5 001	1 265	394	70
TWT (RTT/3.0)	100us	-	28 701	4 027	1 117	359	119	48	34	31	31
TWT (RTT/3.0)	10us	-	24 798	2 990	878	255	101	40	25	25	25
TWT (RTT/3.0)	1us	-	22 867	2 922	882	262	93	33	27	27	27

Table 4. The number of samples required to attack a server in the US, provided that clock synchronization has already been performed. Each row indicates what synchronization iteration was used to obtain the results in that row. A dash (-) is shown if the attack was unsuccessful or required more than 50 000 samples.

Attack	Sync	$1\mu s$	$2\mu s$	$5\mu s$	$10\mu s$	$20\mu s$	$50\mu s$	$100\mu s$	$200\mu s$	$500\mu s$	$1ms$
Classical	/	-	-	-	18 539	1 907	530	121	40	7	7
TWT (RTT/1.5)	1ms	-	-	-	-	-	18 835	3 244	584	103	71
TWT (RTT/1.5)	100us	-	48 944	7 844	2 101	625	178	120	49	42	41
TWT (RTT/1.5)	10us	-	45 329	6 104	1 830	468	136	96	36	34	33
TWT (RTT/1.5)	1us	-	37 300	6 447	1 736	476	133	91	45	37	37
TWT (RTT/2.0)	1ms	-	-	-	-	-	22 605	3 909	940	84	84
TWT (RTT/2.0)	100us	-	25 782	4 998	1 330	415	143	72	43	36	36
TWT (RTT/2.0)	10us	-	20 110	3 480	1 025	348	113	63	31	29	29
TWT (RTT/2.0)	1us	-	21 092	3 916	1 083	317	125	61	27	27	27
TWT (RTT/3.0)	1ms	-	-	-	-	-	37 500	7 141	1 541	175	164
TWT (RTT/3.0)	100us	-	13 668	2 524	671	262	112	53	42	43	43
TWT (RTT/3.0)	10us	48 126	9 204	1 741	519	202	104	46	43	40	40
TWT (RTT/3.0)	1us	43 816	9 521	1 821	466	190	91	40	35	35	35