Timeless Timing Attacks:
Exploiting Concurrency to Leak Secrets over Remote Connections

Tom Van Goethem[1], Christina Pöpper[2], Wouter Joosen[1], Mathy Vanhoef[2]
[1]imec-DistriNet, KU Leuven
[2]Center for Cyber Security, New York University Abu Dhabi

Abstract

To perform successful remote timing attacks, an adversary typically collects a series of network timing measurements and subsequently performs statistical analysis to reveal a difference in execution time. The number of measurements that must be obtained largely depends on the amount of jitter that the requests and responses are subjected to. In remote timing attacks, a significant source of jitter is the network path between the adversary and the targeted server, making it practically infeasible to successfully exploit timing side-channels that exhibit only a small difference in execution time.

In this paper, we introduce a conceptually novel type of timing attack that leverages the coalescing of packets by network protocols and concurrent handling of requests by applications. These concurrency-based timing attacks infer a relative timing difference by analyzing the order in which responses are returned, and thus do not rely on any absolute timing information. We show how these attacks result in a 100-fold improvement over typical timing attacks performed over the Internet, and can accurately detect timing differences as small as 100ns, similar to attacks launched on a local system. We describe how these timing attacks can be successfully deployed against HTTP/2 webservers, Tor onion services, and EAP-pwd, a popular Wi-Fi authentication method.

1   Introduction

When the execution time of an application or algorithm depends on a secret, it may be possible to leak its contents through a timing attack. Over the last few decades, timing attacks have been applied in various contexts, ranging from extracting private keys by abusing SSL/TLS implementations [11, 12, 30] to revealing the browsing habits of users [7, 20, 43]. In a typical timing attack, the adversary obtains a series of sequential measurements and then performs a statistical analysis in an attempt to infer the actual execution time for varying inputs. The success of such an attack largely depends on the signal-to-noise ratio: the more variation the

measurements exhibit, the harder it will be to correctly determine the execution time.

In remote timing attacks, i. e., over a network connection, the measurements are affected by many different factors. Predominantly, the variations in network transmission time (jitter) can render a timing attack impractical, or may require the adversary to collect an extensive number of measurements. On Internet connections, packets experience jitter that depends on the load of the network connection at any given point in time, for every hop along the network path. In cases where the timing difference is on the order of 100's of nanoseconds or a few microseconds, this network jitter becomes a prohibitive factor for performing an accurate timing attack.

In this paper we introduce a new paradigm for remote timing attacks by exploiting multiplexing of network protocols and concurrent execution by applications. Our proposed concurrency-based timing attacks are completely unaffected by network conditions, regardless of the distance between the adversary and the victim server. In contrast to typical timing attacks, where absolute measurements are obtained sequentially, our novel attacks extract information from the *order* in which two concurrent execution tasks are completed, and in fact do not use any timing information (we therefore call them *timeless*). For the attack to be successful, both tasks must start executing immediately after one another, and the delay between starting both tasks should be unaffected by the network. To achieve this, we leverage several techniques that trick various network protocols to coalesce different requests in a single network packet. As a result, both requests arrive simultaneously at the targeted server, and are processed concurrently. We find that the performance of these concurrent timing attacks over a remote connection is comparable to that of a sequential timing attack on the local system.

Through a formal model, we show how concurrency-based timing attacks are theoretically unaffected by jitter on the network connection. We then show how these attacks can be applied in practice in a variety of scenarios: web applications served over HTTP/2 or by Tor onion services, and Wi-Fi authentication. Based on an extensive evaluation, our

measurements confirm that concurrency-based timing attacks are indeed unaffected by variations in network delay. On web servers hosted over HTTP/2, we find that a timing difference as small as 100ns can be accurately inferred from the response order of approximately 40,000 request-pairs. The smallest timing difference that we could observe in a traditional timing attack over the Internet was 10µs, 100 times higher than our concurrency-based attack. Based on the response order of concurrent EAP-pwd authentication requests, whose timing side-channel was previously deemed infeasible to exploit against a server, it is possible to perform a dictionary attack with a success probability of 86%.

In summary, we make the following contributions:

- We introduce a model for timing attacks and show that, in theory, concurrency-based attacks are unaffected by network irregularities. Through various experiments we confirm that this holds true in practice.
- We show how request and response multiplexing in HTTP/2 can be leveraged to perform concurrency-based timing attacks, both in direct as well as cross-site threat scenarios. Our measurements indicate that concurrency-based timing attacks significantly outperform sequential attacks over the network and have a similar accuracy as when attacks are performed on the local system.
- We discuss how network protocols can make applications susceptible to concurrency-based timing attacks, as evidenced by a practical timing attack against Tor onion services despite the high network jitter introduced by the six relays between the adversary and the server.
- In addition to the attacks against web services, we describe how our novel exploitation techniques can be applied in other contexts: We perform a dictionary attack with high success rate against EAP-pwd, a popular Wi-Fi authentication method.
- Finally, we propose various defenses that reduce the performance to sequential timing attacks performed over the network, and evaluate their real-world overhead based on an extensive dataset.

2 Background & Related work

Timing attacks have been known and extensively studied for several decades [30]. They can be used to leak secret information by exploiting a measurable difference in execution time. As connectivity became more stable over time, it was shown that timing attacks can also be launched over the network: in 2005, Brumley and Boneh showed that it was possible to extract an SSL private key over a local network by exploiting a timing side-channel in OpenSSL [12].

2.1 Timing attacks against web applications

In 2007, Bortz and Boneh showed that timing attacks can be applied to extract sensitive information from web applica-

tions [7]. They introduced two types of attacks: direct attacks, where the adversary directly makes a connection with the webserver, e. g., to test for the existence of an account on the application, and cross-site timing attacks, where the attacker tricks a victim to send requests, e. g., upon visiting an attacker-controlled website.

In their research, Bortz and Boneh show how cross-site timing attacks can be used to reveal whether a user is logged in, or how many items the user has in their shopping basket. More recently, in 2015, Gelernter and Herzberg revisited these cross-site timing attacks and introduced two new techniques to overcome the limitations imposed on the attack by the potential instability of the victim's connection [22]. Their techniques rely on inflating either the size of the response or the computation time required by the server. As a result, the difference in response size or processing times becomes significantly higher, and therefore the signal-to-noise ratio of the measurements is increased. This work is orthogonal to our research: We focus on making it feasible to detect small timing differences in contrast to increasing these time differences.

Other timing attacks in the web platform aimed to determine the size of responses, either by abusing side-channel leaks in the browser [51] or by exploiting TCP windows [55]. Furthermore, it has been shown that the various high-resolution timers that are available in modern browsers [45] can be abused to leak information on which URLs have been visited [47], activities of other tabs [56], and even to create unique device fingerprints [44].

2.2 Other remote timing attacks

Outside of the context of web applications, remote timing attacks have mainly been demonstrated on crypto protocols, such as SSL/TLS. It has been shown that the private key of an OpenSSL-based server could be extracted by exploiting timing side channels [37] in the implementation of RSA [1, 12], AES [6], or ECDSA [11]. Furthermore, Meyer et al. [38] presented timing-based Bleichenbacher attacks on RSA-based ciphersuites that could be exploited over a local network. Another timing attack that was shown to be feasible to exploit over a local network is Lucky Thirteen [2], which leverages timing differences in TLS and DTLS implementations. Since timing attacks against crypto protocols mostly abuse timing differences in operations that are sequential, e. g., during the handshake, our concurrency-based timing attacks cannot be straightforwardly applied. Nevertheless, as we show in Section 5, attacks against crypto implementations can still be applied if there exists an underlying network layer that coalesces different packets.

In 2020, Kurth et al. introduced NetCAT, which targets DDIO, a recent Intel feature that provides network devices access to the CPU cache, to perform a PRIME+PROBE attack that can exploit cache side-channels [32]. Remote mi-

croarchitectural attacks targeting the application layer have been explored by Schwarz et al. [46], who showed that it is feasible to perform Spectre attacks over a network connection. They show that with approximately 100,000 measurements over a local network it is possible to leak one bit from the memory, resulting in 30 minutes per byte, or 8 minutes when the covert channel based on AVX instructions is used. On a cloud environment, leaking one byte took approx. 8 hours for the cache covert channel and 3 hours when leveraging the AVX-based channel. Since the NetSpectre attacks target applications above the network layer, an attacker could, in theory, leverage our concurrency-based timing attacks to improve the timing accuracy. One challenge may be that the concurrent executions required by our attack introduce (microarchitectural) noise or side-effects into the NetSpectre measurements, making exploitation challenging. As such, we consider exploring concurrency in remote microarchitectural attacks an interesting topic for future research.

3 Concurrency-based Timing Attacks

In contrast to classical timing attacks, where an adversary obtains a number of independent measurements over the network and then uses statistical methods to infer the processing time of the request, the concurrency-based timing attacks we introduce in this paper rely on the relative timing difference between two requests that are concurrently executed.

3.1 Classical timing attack model

Before explaining how concurrency-based timing attacks can be executed in practice, we first introduce a theoretical model of timing attacks to show how they can benefit from exploiting concurrency. Inspired by the work of Crosby et al. [15], our model splits the measured response time R into the processing time T and propagation time B, resulting in $R = T + B$. The processing time T is defined as the time required by the server to process the request and form a response, and the propagation time B denotes the latency incurred by transmitting the request and response packets (including, e. g., encryption overhead). Due to dynamic workloads on the server and variations in network conditions, both T and B are random variables. Let $t = E[T]$ and $b = E[B]$, then we can write

$$R = t + b + J, \tag{1}$$

where J represents the sum of the random jitter associated to both the processing and propagation times. To differentiate between requests to different processing tasks $m \in \{1..M\}$ on the same server we write $R_m = t_m + b + J_m$. Here b is independent of m because we assume the propagation time between a specific client and server is independent of the task m being executed, and because the arrival time of the first byte of the response is independent of the response length.

So far, we treated the propagation time as an aggregate over the entire request-response operation. However, this aggregate operation consists of a number of sub-operations, e. g., the routing operations for every hop encountered on the network path, encoding the network packets, decrypting the payload, passing the request to the correct processing unit, etc. As such, we can model the propagation time for a request as the sum of all sub-operations $k \in \{1..K\}$. The formula then becomes:

$$R_m = t_m + \sum_{k=1}^{K} (b_k + J_{m,k}), \tag{2}$$

where the random jitter associated to the processing time t_m is modeled by $J_{m,p}$ for some $p \in \{1..K\}$ with b_p equal to zero .

As an adversary, we have access to the response times for two different requests, and we want to know under which conditions this leaks the order of processing times. That is, if the response time of request x is larger than that of request y, we want to know under which conditions this means that the processing time of x was also higher than that of y. To derive these conditions, we construct the following equivalences:

$$R_x > R_y \Leftrightarrow t_x + \sum_{k=1}^{K} (b_k + J_{x,k}) > t_y + \sum_{k=1}^{K} (b_k + J_{y,k}) \tag{3}$$

$$\Leftrightarrow t_x - t_y > \sum_{k=1}^{K} (J_{y,k} - J_{x,k}). \tag{4}$$

From this we can see that the order of response times correctly leaks the order of processing times if their difference is higher than the combined jitter of **both requests**. The probability of this being the case decreases in function of the sum of jitter variances (assuming the jitter distributions are statistically independent and normally distributed).[1] In practice we can perform multiple measurements to reduce the impact of the random jitter over the difference in processing times.

3.2 Model for concurrency timing attacks

We consider two requests to be *concurrent* when they are initiated at the same time, and where the relative difference of the response times, or their order, reveals information on the execution times. As we will show in the following sections, this allows us to significantly reduce the impact of jitter on our measurements. More concretely, in many cases we can force the jitter values for a subset of sub-operations to be the same for two concurrent requests, e. g., by aggregating the requests in a single network packet. As such, the jitter for the first S sub-operations, i. e., those related to sending the requests from the client to the server, is the same between the two concurrent requests x and y:

$$\forall s \in \{1..S\} : J_{x,s} = J_{y,s} \tag{5}$$

---

[1]This is because $Var[J_{y,k} + J_{x,k}] = Var[J_{y,k} - J_{x,k}] = Var[J_{y,k}] + Var[J_{x,k}]$.

where $1 \le S < K$, and such that the server starts processing the request starting from sub-process $S+1$. When we apply this optimization to the equivalence defined in (4) we get:

$$R_x > R_y \Leftrightarrow t_x - t_y > \sum_{k=S+1}^{K} \left(J_{y,k} - J_{x,k} \right). \qquad (6)$$

As such, by leveraging the aggregation of two concurrent requests in a single network packet, an adversary can observe a difference in processing time if this difference is greater than the jitter incurred by the requests **after** they arrive at the server. Consequently, the probability that the difference in response timing correctly reflects the difference in processing time is higher for concurrency-based timing attacks, and therefore fewer measurements are needed to infer secret timing-based information. If we assume that the jitter of (certain) operations is directly related to the average propagation time, i. e., the longer an operation takes, the higher the absolute jitter values will be (our measurements in Section 4 confirm this), our concurrency-based timing attacks provide the most advantage on (relatively) slow network connections.

Most systems and applications do not support complete concurrency: a network card will read a packet byte by byte from the physical layer, encrypted packets need to be decrypted sequentially in the correct order, etc. Consequently, the processing of the second request will be slightly delayed by operations on the first request. We represent this additional delay before processing the second request using the random variable D_y. Similar to other operations, we define $d_y = E[D_y]$, and we let $J_{y,d}$ represent the random jitter associated to this delay. Note that this delay only exists for the second request. Considering this additional delay, the equivalence becomes:

$$R_x > R_y = t_x - t_y > \sum_{k=S+1}^{K} \left(J_{y,k} - J_{x,k} \right) + \left(d_y + J_{y,d} \right). \quad (7)$$

Since many network protocols, for example TCP, include monotonically increasing sequence numbers in all transmitted packets, we can make another improvement to our model by leveraging the order of responses instead of their differential timing. Specifically, the request that the server finished processing first will have a response with a sequence number lower than the later response. As a result, jitter incurred after a request has been processed will have no effect on the order of the responses. If we let SN_y be the sequence number of the response associated to request y we get:

$$\text{SN}_x > \text{SN}_y \Leftrightarrow t_x - t_y > J_{y,p} - J_{x,p} + d_y + J_{y,d}. \qquad (8)$$

Recall that we defined $J_{y,p}$ as the jitter associated with the processing of request y. From this we can see that the sequence numbers of the responses correctly leak the order of processing times if their difference is higher than the combined jitter of processing both requests plus the total delay (average + jitter). Recall that the delay here refers to the time difference
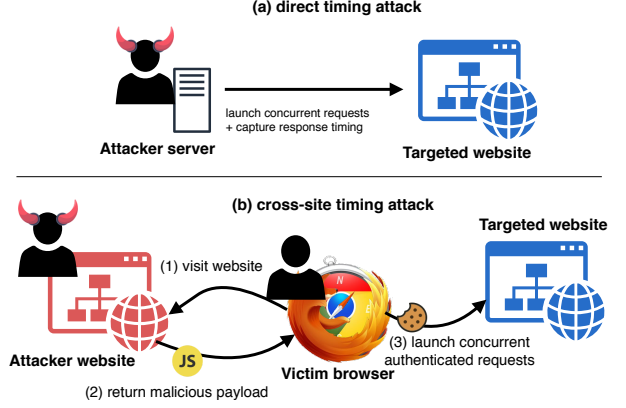

Figure 1: Threat models for web-based attacks.

between when the server started processing each of the two concurrent requests. Importantly, Equation 8 shows that network jitter does not affect concurrency-based timing attacks at all, neither on the upstream nor downstream path. However, a downside of only considering the order of responses is that it may provide less granular information, making statistical analysis less efficient. In the following sections we explore how this affects attacks in practice.

4 Timing attacks on the web

Timing attacks on the web have been studied for well over a decade [7]. To date, researchers have focused on uncovering applications that may leak sensitive data by obtaining several timing measurements and then performing a statistical analysis. Crosby et al. found that the Box Test performs best, and were able to measure a timing difference of 20μs over the Internet and 100ns over the LAN [15]. In the concurrency-based timing attacks of this section, we show that, regardless of the network conditions, it is possible to achieve a performance similar to traditional timing attacks that are launched from the local system.

4.1 Threat model

For web-based timing attacks, we consider two threat models: *direct* and *cross-site* attacks [7], as depicted in Figure 1. In a *direct* timing attack, the adversary will connect directly to the targeted server and obtain measurements based on the responses that it returns. As the adversary is sending packets directly to the server, this provides them with the advantage that they can craft these packets as needed. In the *cross-site* attack model, the attacker will make the victim's browser initiate requests to a targeted server by executing JavaScript, e. g., through a malicious advertisement or tricking the victim into visiting an attacker-controlled web page. Although the same-origin policy prevents the attacker from extracting any

content from the responses, timing is one of the metadata that is still available. More concretely, the adversary could leverage the Fetch API to initiate a request to the targeted website. This will return a `Promise` object, which will resolve as soon as the first byte of the response has been received [52].

It is important to note that the victim's cookies will be included in the request (assuming the adversary passes the `{"credentials": "include"}` option to the Fetch API). As such, the requests are performed under the identity of the victim, effectively allowing the adversary to extract sensitive information that the user has shared with the targeted website. In contrast to the direct timing attack, in which the adversary can choose from which server to launch the attack, preferably one in close vicinity of the targeted server, the network conditions between the victim and server are out of the control of the attacker. For example, the victim could have an unreliable wireless connection, or be located in a different country than the origin server, thereby introducing a significant amount of jitter to the timing measurements in a classical timing attack.

4.2 HTTP/1.1

The most popular protocol on the web is HTTP; for a long time HTTP/1.1 was the most widely used version. Classical timing attacks presented in prior work [7, 15, 19, 22, 51] exclusively targeted this protocol, since HTTP/2 only recently became widely adopted. A major limitation of HTTP/1.1 is its head-of-line (HOL) blocking, causing all requests over the same connection to be handled sequentially. Thus the only way to perform a concurrent timing attack is to use multiple connections.

To evaluate whether concurrency would improve the accuracy of timing attacks, we performed several experiments. We found that concurrently launching requests over two connections increases the jitter on the network path from the attacker to the server. Network interfaces can only transmit one packet at the time; when the attacker sends two concurrent requests in 2 TCP packets, the second one will be delayed until the first one is sent. As such, the jitter that the packets observe during transmission is independent from each other, similarly as with a sequential timing attack. There is a possibility that the two packets would experience jitter such that at the last network hop on the path between the attacker and target server, both packets are buffered and will arrive almost simultaneously at the server. However, as this does not happen consistently, the attacker has no way of knowing that this in fact occurred.

In Appendix A we report on an experiment where two HTTP/1.1 requests were sent concurrently over two different connections. We find that this does not improve the performance of timing attacks, validating our assumption that the jitter observed on the two connections is independent. As such, we conclude that simply sending HTTP requests at the same time over different connections does not improve the performance of a timing attack. Note that this does not mean

that servers using HTTP/1.1 are unaffected by concurrency-based timing attacks (in Section 4.4 we show how Tor onion services running an HTTP/1.1 server can be attacked), but rather that the protocol cannot be abused to coalesce multiple requests in a single network packet.

4.3 HTTP/2

In this section we show how the request and response multiplexing of HTTP/2 can be leveraged to perform concurrency-based timing attacks that allow an adversary to observe a timing difference as small as 100ns, providing a 100-fold improvement over classical timing attacks over the Internet.

4.3.1 Background

A key advantage of HTTP/2 is the removal of the restrictions imposed by HOL blocking. To achieve this, HTTP/2 introduces the concept of streams, a bidirectional flow of data with a unique identifier that is typically associated with a request and response pair. Data is sent in the form of frames; the request and response bodies are sent in `DATA` frames whereas the headers are sent in `HEADERS` frames. Headers are compressed using the HPACK algorithm [40], which uses a static and dynamic table to refer to header keys and values.

Since every frame contains a stream identifier, the web server can process different requests concurrently, despite having only a single TCP connection. Similarly, responses can be sent as soon as they are generated by the server. By default, the web server will prioritize requests equally, although it is also possible to explicitly define a priority for a specific stream, or by declaring dependencies on other streams. However, at the time of this writing, many web servers and browsers either do not support prioritization at all, or only provide limited functionality [18, 35]. Moreover, in our concurrency-based timing attacks, the requests will ideally be handled by the same processing resources, such that their execution time solely depends on the secret information that the adversary aims to infer. The execution of the concurrent requests will typically be performed on different CPU cores/threads; thus the execution of one request will not affect that of the other. Note that if the execution of the request makes use of a shared resource that does not support parallelism, there can be an influence in execution time between the two requests. Depending on the application, this could reduce the timing side-channel leak, or it might amplify it, e. g., when the access to the shared resource occurs after the operation that leaks sensitive timing information and the "slowest" request is delayed further by waiting to access the shared resource.

4.3.2 Direct attacks

As discussed in our formal model of Section 3, the goal of an attacker is to ensure that the server starts processing two
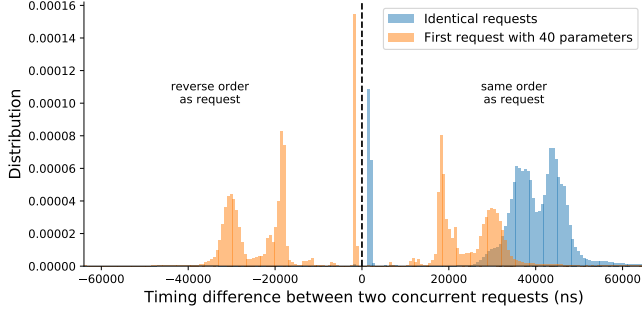
Figure 2: Distribution of the difference in response time for two concurrently launched requests for the same processing task, with and without additional URL parameters.

requests at exactly the same time, such that the order of the responses reveals which request finished processing first. One request is for a known baseline processing task, and the other for a task that would either take the same or a different amount of processing time, depending on some secret information. Thanks to HTTP/2's request multiplexing, multiple requests that are sent at the same time will be processed in parallel, and in contrast to HTTP/1.1, responses are sent as soon as possible independent of the request order. To abuse this behavior, an attacker can embed two `HEADERS` frames containing two HTTP/2 requests in a single TCP packet to ensure they arrive at the same time. As headers are compressed, their size is typically on the order of 100-150 bytes, and thus the resulting TCP packet size is significantly lower than all Maximum Transmission Units (MTUs) observed on the Internet [17].

In our measurements, we made use of the nghttp2 C library [50] and used the `TCP_CORK` option [5, 33] to ensure that the two concurrent requests are added to the same TCP segment[2]. As most browsers only allow HTTP/2 connections over TLS, all measurements that we performed were over a secure connection. In summary, a single TCP segment contains two TLS records that each contain a HTTP/2 `HEADERS` frame. As soon as the TCP segment arrives at the server, the two TLS records are decrypted sequentially, and the server starts processing each immediately after decryption. The $d_y + J_{y,d}$ factor from the equivalence defined in (8) reflects the average duration and jitter of the decryption of the TLS record containing the second request. This forms a problem if we want to leverage the order in which responses are returned: For two requests that have the same processing time, the former will be returned first if the decryption time is higher than the difference in jitter. To measure this impact, we sent one million request-pairs for the same processing task (idle sleep for 100μs) to an HTTP/2-enabled nginx server hosted on Amazon EC2 and captured the difference in response time. In Figure 2, we show the distribution of the differences in response time.

Positive values indicate that the order in which the responses were received is the same as the order in which the requests were sent, and negative values indicate a reversed order. We can clearly see that when two identical requests are sent (blue distribution on the graph), these are virtually always returned in the same order. Note that there is a high peak close to 0^3, which represents the cases where both responses were sent back in a single TCP packet, which may happen when the responses are sent in quick succession.

To overcome this limitation, the processing of the first request needs to be slightly delayed, i. e., for the duration of decrypting the second request. For this, we leverage the fact that the request handler needs to parse the URL to extract the parameters and associated values, and make these accessible to the processing language. Since the execution time of this parsing is directly related to the number of URL parameters, an adversary could arbitrarily extend this execution time by including more parameters. In 2007, Wälde and Klink showed how this mechanism could be abused by including many URL parameters that would result in a hash collision in the hashtable that was constructed in PHP, leading to a Denial-of-Service [29]. In contrast to these DoS attacks, which to date have largely been mitigated, e. g., by reducing the number of allowed parameters to 1,000, we only require a short execution time. In an attack scenario, the adversary can first empirically determine the number of additional URL parameters that need be included in the first request such that both requests are processed at the same time. We found that this value remains stable over time, and depends on the web server application that is used; Figure 2 shows the distribution of the difference in response timing for two requests where the first one contains 40 additional URL parameters (orange distribution). This results in a better balance in the order of the responses; adding fewer parameters causes more responses to be received in the same order as the requests, and when more parameters are included, the second request is more likely to finish first.

In our formal model, this means that we introduce another factor, u_x (and associated jitter value $J_{x,u}$), that can be subtracted from d_y:

$$t_x - t_y > J_{x,p} + J_{y,p} + (d_y - u_x) + J_{y,d} + J_{x,u} \qquad (9)$$

To evaluate the performance of concurrency-based timing attacks in HTTP/2, we perform a series of measurements. In our measurements, we create a set of (baseline, baseline) request-pairs, where both requests were for a PHP endpoint that would perform `time_nanosleep(100000)`, i. e., idle sleep for 100μs. Additionally, we created a set of (baseline, target) request-pairs, where the target requests are for an endpoint that would perform an idle sleep for 100μs+Δ, for multiple values of Δ, ranging from 75ns to 50μs. All requests

---

[3]These values are not equal to 0 as we measure the response time after decryption of the response.

	75ns	100ns	150ns	200ns	500ns	1μs	2μs	5μs	10μs	20μs	50μs
Client-server connection	**Concurrency-based timing attack (HTTP/2)**										
Miscellaneous (Internet)	-	39,342	24,016	9,917	1,610	466	161	52	11	6	6
	Sequential timing attack										
Europe - Europe	-	-	-	-	-	-	-	-	23,220	2,926	333
Europe - East US	-	-	-	-	-	-	-	-	-	16,820	4,492
Europe - South-East Asia	-	-	-	-	-	-	-	-	-	-	7,386
VM - VM (LAN)	-	-	50,463	40,587	14,755	3,052	2,165	498	126	41	20
localhost	-	-	16,031	17,533	3,874	856	220	42	20	16	14
	Concurrency-based timing attack (HTTP/1.1 over Tor)										
Client to onion service	-	-	-	-	-	43,848	3,125	386	96	22	6

Table 1: The average number of requests or request-pairs required to perform a successful timing attack with at least 95% accuracy. If the attack was unsuccessful, or required more than 100k requests, this is indicated with a dash (-).

were made from our university's network (located in Belgium, using a 1Gbit connection), and for every Δ, we obtained 1 million measurements from nine Amazon EC2 servers that ran an nginx web server. We launched three C5.large instances in three geographically distributed datacenters: EU, East US, South-East Asia (nine instances in total); the connection of these instances is 10Gbit. We used a minimal program that only calls a sleep function to minimize the jitter related to program execution. It should be noted, however, that even this minimal program still produces a non-neglibile amount of jitter. In particular, when evaluating the accuracy of the sleep function on the server itself using a high-resolution timer, we still needed 411 measurements to correctly distinguish a timing difference of 75ns.

To compare concurrency-based attacks to traditional timing attacks, we also computed the number of requests needed to perform a classical timing attack, using the Box Test for the statistical analysis [15]. The Box Test considers three distributions of timing measurements: the baseline, that of a negative result (timing matches baseline), and that of a positive result (with a different processing time due to a timing side-channel leak). A timing attack is considered successful if a pair of $i, j \in [0, 100]$ with $i < j$ can be found such that the interval determined by the ith and j^{th} percentile of the distribution of the baseline measurements, overlaps with the interval (determined by the same i and j) of the negative result measurements distribution, while at the same time *does not* overlap with the interval of the positive result. That is, an overlap indicates that the measurements come from requests with the same processing time, whereas a measurements of requests with a different processing time should not have an overlapping interval. If no values for the pair i, j can be found that fulfill these conditions, the timing attack is considered infeasible to exploit.

For this experiment, requests were again launched from our university's network. Additionally, we performed experiments from a VM located in the same datacenter as the server (LAN), as well as a timing attack to localhost. The results of our measurements are shown in Table 1; for the concurrency-based attacks, this table indicates the minimum required number of request-pairs to perform an attack with 95% accuracy (averaged over the nine servers). For the classical (sequential) timing attack we show the total number of requests. Note that the total number of requests that will be launched for the concurrency-based attack is twice the reported number of request-pairs. However, because the pairs are sent simultaneously, it takes approximately the same amount of time to process a request-pair in the concurrency-based timing attack, as to process a single request in the sequential timing attack. If an attack was unsuccessful, i. e., would require more than 100,000 requests, on at least one server, we mark it with a dash (-). Note that because our experiments were performed in a real-world setting, some of the measurements may be affected by temporal variations of load on the network or on the machines that hosted the VMs.

We find that concurrency-based timing attacks provide a significant advantage compared to any sequential timing attack over the Internet: Even for the EU-EU connection, which had the lowest latency (average: 24.50ms) and jitter (standard deviation: 1.03ms), the sequential timing attack can only distinguish an execution time of 10μs, whereas our concurrent timing attack can distinguish a difference of 100ns (2 orders of magnitude more precise). Moreover, our concurrency-based attacks even outperform sequential timing attacks over the local network, which had an average latency between 0.5ms and 1.5ms, and standard deviation of jitter between 15μs and

45μs (depending on the datacenter). Finally, we can see that our novel attacks are comparable to executing a sequential timing attack on the local system, which confirms that, as we determined in our formal model, concurrency-based timing attacks are not affected by network conditions.

In addition to the measurements on nginx, we performed a similar set of experiments against an Apache2 web server, using the same experimental setup. We find that, in general, timing attacks targeting web applications served by Apache2 require more requests compared to nginx, especially in optimal network settings, such as localhost or the local network. This can be attributed to the request handling mechanism of Apache2, which is more computationally expensive compared to that of nginx. Correspondingly, we find that the concurrency-based attacks are also slightly affected by this, as the variation in the computation increases. Nevertheless, we find that the concurrency-based attacks still allow the adversary to distinguish timing differences as small as 500ns. The complete results of these experiments can be found in Appendix B. As web servers need to handle more and more requests, and become increasingly performant, we believe the accuracy of (concurrent) timing attacks will continue to improve.

4.3.3 Cross-site attacks

A key requirement of our concurrency-based timing attacks against HTTP/2 is that we can manipulate the TCP socket in such a way that both HTTP requests are sent in a single packet. For cross-site attacks, where the requests are sent from a victim's browser, this is no longer possible, as the browser handles all connections. To overcome this limitation, we introduce another technique that leverages TCP's congestion control [3] and Nagle's algorithm [8, §4.2.3.4]. The congestion control mechanism determines the number of unacknowledged packets that a host can send, which is initially set to 10 on most operating systems and is incremented with every received acknowledgment, as per TCP's slow start mechanism [48]. Furthermore, Nagle's algorithm ensures that, if there is unacknowledged data smaller than the maximum segment size (MSS), user data is buffered until a full-sized segment can be sent. For example, if an application would consecutively send two small chucks of data, e. g., 20 bytes, these will be combined and only a single packet of 40 bytes will be sent. Consequently, as an attacker we can initiate a bogus POST request with a sufficiently large body that exactly fills the congestion window. Immediately after this POST request the attacker triggers the two concurrent requests, which will be buffered and thus coalesced in a single packet when it is eventually sent to the target website.

An important caveat is that the sending TCP window is incremented with every ACK that is received. Consequently, when (rapidly) performing multiple concurrency-based timing measurements, an attacker would need to send an increasingly

large POST request to fill the sending TCP window. To overcome this limitation, the adversary has two options. First, the request-pairs could be padded such that two requests exactly fit in a single packet. In virtually all cases the exact size of the request can be predicted (the browser will always send the same headers).[4] After sending the first bogus POST request to fill the initial TCP window, the attacker launches several request-pairs, and every pair will be contained in a single packet. As such, for every ACK that is received, two new request-pairs will be sent. In our experiments, we found that on low-latency connections, the ACKs would arrive very rapidly, and thus the server would eventually become overloaded with requests, introducing jitter to the measurements. As a workaround, the attacker could, instead of the initial large bogus POST request, instead send 10 (= initial congestion window) smaller requests with a delay of RTT/10 in between the requests. As a result, the initial TCP window will still be filled and ACKs would only arrive at a rate of RTT/10. Ironically, this means that this technique (and thus the timing measurements) works better on slower network connections.

An alternative technique to overcome the hurdles imposed by the increasing TCP window, is to force the browser to close the connection. Browser have an upper bound on the number of active concurrent connections; if this limit is reached, it will close the least recently used ones. For instance, in Chrome this limit is set to 256 [42], and thus an attacker could make the victim's browser initiate connections to as many IP addresses, forcing it to close the connection to the targeted server. On Firefox, this limit was increased to 900, except for the Android-based browser applications, which remained at 256 [34]. We found that it is feasible to use this technique to close an active connection; other mechanisms may also be abused to do this (e. g., Firefox imposes a limit of 50 concurrent idle connections, and will close active connections when this limit is reached). It should be noted that this technique can be used in conjunction with the first one, if it is required to reset the TCP window.

As the requests are coalesced in a single packet, the performance of these cross-site timing attacks is the same as with the direct attacks. To defend against this type of attack, the webserver could set the SameSite attribute on cookies [36], preventing it to be sent along in the cross-site requests, although certain violations have been discovered [21, 41]. As of February 2020, Google Chrome is gradually rolling out changes that mark cookies as SameSite by default [49].

4.3.4 Limitations

According to HTTPArchive's annual report, which takes into account data obtained from over 3 million regularly visited websites, 37.46% desktop homepages are served over

---

[4]When the cookie constantly changes in length and content, the HPACK algorithm will not be able to refer to it with an entry in the dynamic table, and thus the request length varies along with the length of the cookie.

HTTP/2 [26]. For websites that support HTTPS, a requirement that browsers impose for using HTTP/2, this is percentage is even higher: 54.04%. Although this makes many websites susceptible to our concurrency-based timing attacks, it should be noted that a significant number of websites are using a content delivery network (CDN), such as Cloudflare. HTTPArchive reports 23.76% of all HTTP/2 enabled websites to be powered by Cloudflare. For most CDNs, the connection between the CDN and the origin site is still over HTTP/1.1, and HTTP/2 may not even be supported (as it does not provide performance improvements). Nevertheless concurrency-based timing attacks may still outperform classical timing attacks in this case, as requests are not affected by jitter on the network path between the attacker and the CDN. This is especially valid for cross-site attacks, where the requests are sent by the victim who may have an unreliable Internet connection.

4.3.5 Use-case

To demonstrate the impact of our concurrency-based timing attacks over HTTP/2, we describe how it can be applied in a cross-site attack scenario. More specifically, we found and reported a cross-site timing attack against HackerOne[5], a popular bug-bounty platform where security researchers can report vulnerabilities to a wide range of bug bounty programs. In the dashboard, security researchers and managers of the bounty program can search through the reported bugs; this triggered a GET request where the `text_query` parameter was set to the searched keyword. We found that requests that did not yield any results were processed faster compared to requests where at least one result was returned. As such, a XS-Search attack [22] could be launched against this endpoint: by tricking a manager of a bug bounty program in visiting a malicious web page, the adversary could find out specific keywords that were mentioned in the private, ongoing reports, and potentially reveal information about unfixed vulnerabilities.

After reporting our findings, we were told that a timing attack to this endpoint had been reported a total of eight times. However, our report was the only to qualify for a reward, as it was "the first one to demonstrate a feasible attack". Indeed, with less than 20 request-pairs we could accurately determine if a search query had at least one result. In the meantime, the vulnerability has been mitigated by changing it to a POST request, and requiring a valid CSRF token.

4.4 Tor Onion Services

Tor is a well-known low-latency anonymity network. When a client wants to send a request over Tor to a public web server, this request is first encoded as a Tor cell, which has a fixed length of 514 bytes. These cells are then encrypted once for every relay on the circuit. Most circuits consist of 3 hops, where the last one is the exit node, which sends the request to

the public web server. In addition to protecting the identity of the user when sending outgoing traffic, Tor also provides a feature that hides the identity of the server. To connect to one of these so-called onion services, the client performs a handshake involving the introduction points chosen by the onion service, and a rendezvous point chosen by the client. Upon a successful handshake, a circuit between the client and onion service is created, consisting of 6 intermediate relays.

Due to the extended network path that a request has to traverse to reach a hidden service, the jitter renders almost all sequential timing attacks impractical. Based on 100,000 measurements, we determined an average RTT of 251.23ms to our onion service, with a standard deviation of 32.47ms (approximately 30 times as high as what we observed over a regular Internet connection). If the web server would support HTTP/2, the concurrency-based timing attacks presented in Section 4.3 can be used straightforwardly (the attacker can simply construct a packet containing both requests). However, because of how network packets are transported over the Tor network, it is also possible to perform attacks against onion services that only support HTTP/1.1 (or any other type of service). More specifically, an attacker can create two Tor connections to the onion service, and then simultaneously send a request on each of the connections. This will trigger the attacker's Tor client to create two Tor cells and send these over the 6-hop circuit to the onion service. Because a single Tor cell is only 514 bytes (+ 29 bytes of TLS overhead), two cells will fit in a single TCP segment for virtually every network [17]. Consequently, if the two cells are placed in the same packet on the last hop of the circuit, i. e., between the relay and the onion service, the requests will be processed simultaneously by the server.

Based on our experiments on the live Tor network, we found that when the first request of a request-pair is sufficiently large[6], e. g., by adding padding in the headers, and requests are sent in relatively quick succession, the TCP buffer between the onion service and the last relay becomes filled. Consequently, because of Nagle's algorithm, the last two cells will be joined in a single packet. We found that this would reliably cause an inter-request delay on the order to 10μs, and because the first request was larger, and thus took slightly longer to process, no additional URL parameters had to be added to offset the inter-request delay. Note that the webserver will only start processing a request when the entire request has been received.

We set up several Tor onion services on geographically distributed Amazon EC2 datacenters, these ran an nginx HTTP/1.1 server. The (unmodified) Tor clients were set up on virtual machines on our university's private cloud, using a 1Gbit connection, and used the real Tor network to connect to the web servers. The results of our concurrency-based timing attacks that leverage Tor are shown in Table 1. Again, we

---

[5] https://hackerone.com/

[6] In our tests, we found 1500 bytes to be sufficient.
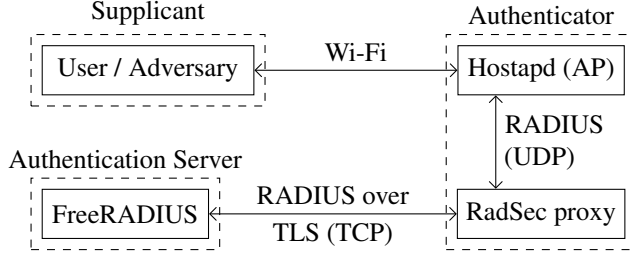
Figure 3: Illustration of an enterprise Wi-Fi setup. Hostapd and the RadSec proxy run on the same device.

```
def hash_to_curve(password, id1, id2, token):
  for counter in range(1, 256):
    seed = Hash(token, id1, id2, password, counter)
    value = KDF(seed, "EAP-pwd Hunting and Pecking", p)
    if value >= p: continue
    if is_quadratic_residue(value^3 + a * value + b, p):
      y = sqrt(x^3 + a * x + b) mod p
      P = (x, y) if LSB(seed) == LSB(y) else (x, p - y)
      return P
```

find a significant increase in the precision by which a timing difference can be distinguished. With 43,848 requests, it is possible to measure a timing difference of 1µs; this is within the range of what is required to perform attacks against crypto protocols (e. g., the Bleichenbacher attack by Meyer et al. exploited a timing difference ranging from 1-23µs). As such, by making a service available as an onion service, it may become possible to perform timing attacks that would not be feasible to exploit over a normal Internet connection.

5 Wi-Fi attacks

In this section we present concurrency-based timing attacks against the EAP-pwd authentication method of Wi-Fi. By abusing concurrency, we exploit a timing leak that was previously considered infeasible to exploit. We also demonstrate how the leaked information can be abused to launch dictionary and password brute-force attacks.

5.1 Background

In enterprise WPA2 and WPA3 networks, the most frequently used authentication methods include EAP-PEAP and EAP-TTLS. Unfortunately, both rely on certificates, which in practice causes security issues because clients often fail to validate server certificates [4, 9]. An authentication method that is based on solely on passwords and avoids certificates, and hence is easier to use and configure, is EAP-pwd [24]. Note that EAP-pwd is almost identical to the Dragonfly handshake of WPA3 [57], and both these protocols were recently shown to be affected by side-channel leaks [54].

With EAP-pwd, the Dragonfly handshake is executed between an authentication server (e. g., FreeRADIUS) and a supplicant (client). During this authentication the Access Point (AP) forwards messages between them. This setup is illustrated in Figure 3, where the AP is called the authenticator. Before initiating the Dragonfly handshake, the AP first sends an EAP identity request, and the supplicant replies with an identity response, which in turn is forwarded by the AP to the authentication server. The server then initiates the Dragonfly handshake by sending a PWD-Id identity frame,

and the supplicant replies using a PWD-Id response. Then Commit frames are exchanged, and finally Confirm frames are exchanged. By default, RADIUS is used to transport all handshake messages between the authenticator and server. However, because RADIUS has design flaws [25], it is often tunneled inside TLS. This tunnel is commonly called RadSec and its precise operation is defined in RFC 6614 [39, 58]. Although FreeRADIUS directly supports RadSec, most APs have to use a proxy that forwards all RADIUS messages over TLS (see Figure 3). In the remainder of this section, we will use the notation RadSec(packet) to denote that a packet is encapsulated in a RadSec TLS record.

Before sending Commit and Confirm frames, the shared password is converted to an elliptic curve point (x,y) using the Hash-to-Curve (H2C) method in Listing 1. This method takes as input the identity of the client and server, the password, and a token that is randomly generated by the server. Note that this random token is sent to the client in the PWD-Commit response. In the H2C method, all four parameters are hashed together with a counter, and the resulting value is treated as a candidate x coordinate. If a corresponding value for y exists on the elliptic curve, the resulting point P is returned. Otherwise, the counter is incremented so a new value for x can be calculated. Several flaws were discovered in this algorithm, with the most critical one that the number of iterations (i. e., for-loops) needed to find P leaks information about the password [54]. It was shown how to exploit this timing leak against EAP-pwd clients. However, attacking a server is harder and deemed practically infeasible. This is because the jitter over Wi-Fi is too high to determine which of two (unique) requests had the highest processing time. Moreover, this jitter cannot be averaged out over multiple handshakes because the server generates a new random token in every handshake, resulting in different H2C executions. Using our classical timing model of Section 3, over Wi-Fi the variance of the jitter components $J_{x,k}$ and $J_{y,k}$ in equivalence (4) are too high to reliably determine whether $R_x > R_y$ holds.

Finally, to quickly reconnect to a previously-used Wi-Fi network, clients can use the Fast Initial Link Setup (FILS) handshake. This handshake is mainly supported in enterprise Wi-Fi networks, and can internally use several authentication methods. We will use it with the EAP Reauthentication Protocol (ERP) [14], which requires that the network contains a central authentication server such as FreeRADIUS.
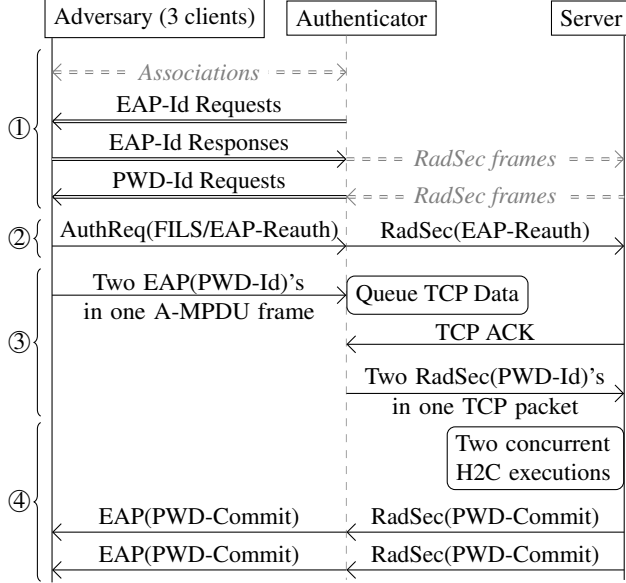
Figure 4: Attacking EAP-pwd servers. Two clients associate, and concurrently send PWD-Id requests in an A-MPDU. The third spoofed client injects a FILS frame so both PWD-Id requests are sent in one TCP frame. Double arrows indicate two (spoofed) clients both *separately* send/receive the frame.

5.2 Attacking an EAP-pwd Server

To exploit the unpatched timing leak in EAP-pwd servers, we will trigger two concurrent executions of the hash-to-curve method. The order of replies then reveals which execution was faster, and this information can be used to recover the password (see Section 5.5). Using our concurrency timing model of Section 3, this means we send concurrent requests x and y, eliminating most of the jitter components as shown in Equation 8. This enables us to determine which execution took longer based on a single concurrent measurement.

Figure 4 illustrates how we trigger two concurrent H2C executions. This is done by impersonating two clients, letting both of them associate to the network, and then replying to the EAP identity requests of the authenticator (stage ①). The authenticator forwards the identity information to the authentication server using RADIUS, which we assume is tunneled over RadSec (i. e., over TLS/TCP). In response, the server initiates the EAP-pwd handshake by sending PWD-Id requests. To trigger two concurrent H2C executions, we now send two PWD-Id frames that arrive at the server simultaneously. This is non-trivial to accomplish, because by default every handshake message is encapsulated into separate RadSec packets, and these will arrive at (slightly) different times. To overcome this, we will induce the authenticator into coalescing two RadSec packets in one TCP packet, assuring both PWD-Id requests arrive simultaneously at the server.

Similar to our previous attacks, we abuse the fact that Na-

gle's TCP algorithm coalesces data if there is unacknowledged transmitted data. To do this, we spoof a third client that sends a FILS authentication request to the authenticator (stage ②). The FILS request contains an EAP reauthentication payload that is forwarded by the authenticator to the server over RadSec. As a result, there will be unacknowledged outstanding data in the RadSec TLS/TCP connection. The adversary now continues the EAP-pwd handshake by sending two PWD-Id frames (stage ③). To assure these packets arrive simultaneously at the authenticator, they are encapsulated in one aggregated Wi-Fi frame (see Section 5.3). Because there is unacknowledged RadSec data due to the FILS request, the two RadSec packets that encapsulate the PWD-Id messages will be queued until a TCP ACK is received. Once the TCP ACK arrives, both RadSec(PWD-Id) records are sent in a single TCP packet.

When the TCP packet with both PWD-Id's arrives at the server, they are processed immediately after one another. Assuming the server is multi-threaded, this processing is done in separate threads that execute concurrently (stage ④). In each thread the server generates a random token and runs the H2C method. The order or PWD-Commit replies now depends on which H2C execution finishes first. The adversary determines this order based on which client receives a PWD-Commit first. In Section 5.5 we show how this information allows an adversary to bruteforce the password.

5.3 Exploiting Frame Aggregation

In our attack, two PWD-Id frames are sent as one aggregated Wi-Fi frame (recall stage ③ in Figure 4). This assures both frames arrive at the same time at the AP. Otherwise the second PWD-Id might arrive after the authenticator received the TCP ACK, meaning the two RadSec(PWD-Id) records would not be aggregated in a single TCP packet. To aggregate Wi-Fi frames one can either use A-MSDU or A-MPDU aggregation.

An Aggregate MAC Service Data Unit (A-MSDU) aggregates frames at the MAC layer, where all subframes must have the same destination and sender address [27, §9.3.2.2.2]. This makes A-MSDU unsuitable for our purpose, because we want to aggregate two frames with different sender addresses. In contrast, an Aggregate MAC Protocol Data Unit (A-MPDU) aggregates frames at the physical layer, where only the receiver address of all subframes must be the same. This means we can use it to aggregate two frames that come from different clients. Moreover, all 802.11n-capable devices are required to support the reception of A-MPDU frames.

Because A-MPDU aggregation happens close to the physical layer, and is commonly implemented in hardware, we cannot easily inject A-MPDU frames using traditional tools. Instead, we extended the ModWiFi framework [53] and patched the firmware of Atheros chips to inject A-MPDU frames.[7]

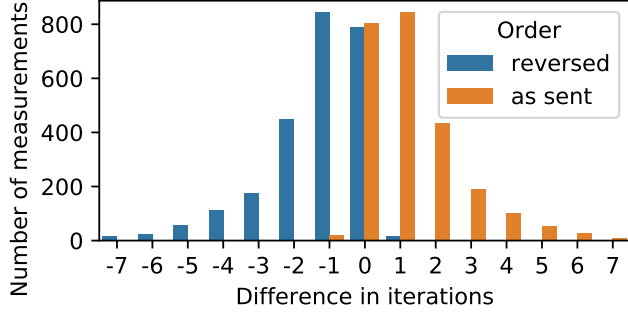[7]This code is available at https://github.com/vanhoefm/modwifi

Figure 5: Results of 5 000 concurrent requests against our EAP-pwd server. The x-axis shows the difference in the number of executed H2C iterations, the y-axis the number of time this occurred, and the color indicates the order of responses.

Our firmware modifications force the Atheros radio to aggregate selected Wi-Fi frames into a single A-MPDU. Note that the adversary has full control over their own hardware, meaning these firmware changes do not limit the applicability of the attack. Moreover, we conjecture that the built-in rate control algorithm of Wi-Fi devices can also be abused to force the aggregation of frames into a single A-MPDU.

5.4 Concurrency Experiments

To perform the attack, we wrote a Python script that controls two modified wpa_supplicant instances. These two instances associate to the AP and execute the EAP-pwd handshake until they have to send the PWD-Id message. When both instances are ready to send the PWD-Id message, the script first injects the FILS authentication request, and then sends both PWD-Id messages in a single A-MPDU. We used a TP-Link TL-WN722N to inject this A-MPDU frame.

We tested the attack against an OpenWRT AP and an Amazon EC2 c5.xlarge instance running FreeRADIUS 3.0.16. The OpenWRT AP was running RadSec 1.6.8 and Hostapd v2.7-devel. Figure 5 shows the results of 5 000 concurrent requests against this setup when using EAP-pwd with curve P-256. We let i_1 and i_2 denote the number of for-loops executed in the H2C method corresponding to the first and second request, respectively. We can see that if the order of the responses matches the order of the requests, then $i_1 \leq i_2$, or in other words then the H2C execution in the second request executed at least as many iterations as the H2C execution in the first request. Otherwise, if the order of responses is reversed and we first receive a response to the second request, then we learn that $i_1 \geq i_2$. All combined, we learn which request needed the most number of iterations. In our experiments the probability that this deduction is incorrect, was below 0.38%.

During our tests we encountered a denial-of-service vulnerability in FreeRADIUS caused by unsynchronized access to a global variable. This can lead to a crash when performing concurrent EAP-pwd handshakes. The flaw was assigned

identifier CVE-2019-17185 and has meanwhile been patched.

5.5 Bruteforcing Passwords

We now perform a dictionary attack by filtering passwords based on the leaked information. Recall that for a single concurrent measurement, the server executes two H2C methods that each use a different random token. For every password, we simulate both H2C methods locally, and reject the password if the difference in executed iterations does not match our concurrent measurement. Based on simulations with elliptic curve P-256 and P-521, on average one concurrent measurement can be used to filter 33% of passwords. To further filter the remaining passwords we can perform the same filtering using another concurrent measurement. This is because the server will use new random tokens in both H2C methods, effectively leaking new information about the password. This means we can perform multiple concurrent measurements, such that in a dictionary of size d, all wrong passwords will eventually be filtered.

We implemented a proof-of-concept of our brute-force algorithm. Since on average a concurrent measurement filters 33% of passwords, we need roughly $n = \log_{1.51}(d)$ concurrent measurements to uniquely identify the password in a dictionary of size d. Taking the RockYou password dump as a reference [16], which contains roughly $1.4 \cdot 10^7$ passwords, on average 40 measurements must be made to uniquely recover the password. Since the probability of a concurrent measurement being wrong is 0.0038, the success probability of this attack against the RockYou dictionary equals $(1 - 0.0038)^{40} \approx 86\%$.

We now estimate the computational cost of our dictionary attack. First observe that filtering a wrong password requires on average $1/0.33 \approx 3$ concurrent measurements. For each measurement we need to simulate two H2C executions, where each execution on average performs 2 iterations. Since we can expect the first candidate password to be found halfway throughout the search, in total $6 \cdot d$ iterations have to be simulated until a candidate password is found. The computational cost of simulating each iteration is dominated by SHA256 operations, since the quadratic residue check can be done efficiently using the Jacobi symbol and the law of quadratic reciprocity. In total three SHA256 operations are executed in every iteration, and based on Hashcat benchmarks we can evaluate $7.48 \cdot 10^9$ hashes per second on an NVIDIA V100 GPU. This means that $2.49 \cdot 10^9$ passwords can be checked per second, and that the estimated cost for various dictionaries matches that of the MODP attack against WPA3's Dragonfly handshake [54, §7.4]. For instance, brute-forcing even the largest public dictionaries costs less than one dollar.

5.6 Countermeasures

A backward compatible defense against the timing leak in EAP-pwd is always performing 40 iterations in the H2C

method, and returning the first valid password element *P*. The probability of needing more iterations equals 2^{-40}, which is considered acceptable in practice [28]. However, this change is non-trivial to implement without introducing other side-channels [54]. As a result, only the development version of FreeRADIUS adopted this defense, and at the time of writing the stable version was still vulnerable to our attack.

A more secure but backward incompatible defense is using a constant-time hash-to-curve method such as Simplified Shallue-Woestijne-Ulas [10]. A specification of this for EAP-pwd has been worked out by Harkins [23].

6 Discussion

Throughout this paper we have shown that our concurrency-based timing attacks are not affected by network jitter and therefore perform significantly better than sequential timing attacks. For the attacks leveraging HTTP/2, we found that the performance is comparable to running them locally from the system the web server is hosted on. Moreover, for the attack on the EAP-pwd authentication method, the concurrency-based timing attacks allow us to abuse a timing leak that was considered infeasible to exploit. Motivated by these impactful findings, in this section we discuss the prerequisites that, when present, can make applications susceptible to concurrency-based timing attacks. Finally, we propose and evaluate various generic defenses.

6.1 Attack prerequisites

Based on our evaluations that were presented throughout this paper, we determine three factors that indicate whether an application may be susceptible to concurrent timing attacks.

1. Simultaneous arrival The first prerequisite is that there needs to be a mechanism available that can be leveraged to ensure that two requests arrive simultaneously at the server. Typically, this will require both requests to be coalesced into a single network packet either directly, e.g. two requests in a single TCP packet for HTTP/2, or by means of an intermediate carrier, such as Tor. We believe that several other network protocols can enable this prerequisite: for instance, HTTP/3 also supports multiplexing of requests and responses, but works over UDP instead of TCP (we did not evaluate this protocol as it is not yet widely deployed or implemented). Furthermore, network tunnels such as VPN and SOCKS may encapsulate packets, allowing packets on two different connections to be coalesced into a single packet on a single connection, similar to the technique we applied to Tor onion services.

2. Concurrent execution As soon as two requests arrive at the target server, they need to be processed concurrently, ideally with as little time in between them as possible. For the protocols, this means that either multiplexing needs to be supported, as is the case with HTTP/2, or multiple connections

need to be used, as we did in case of Tor.

3. Response order An adversary launching concurrency-based timing attacks will leverage the order in which the responses are returned. As such, the order needs to correctly reflect the difference in execution time: if the first request takes longer to process than the second, its response should be generated last. Furthermore, this implies that the difference in execution time of a request-pair reveals information about a secret. Throughout this paper we leveraged a baseline request, which has a known execution time (e. g., a search query for a bogus term); by determining the difference (or similarity) in execution time, we can directly infer information about a secret.

We consider it an interesting avenue for future work to perform a comprehensive survey of network protocols to evaluate whether they provide or enable these prerequisites, and thus make applications susceptible to concurrency-based timing attacks.

6.2 Limitations

In the concurrency-based timing attacks, the adversary relies solely on the order in which responses are returned. This reduces the granularity of timing information that is obtained, compared to sequential timing attacks where absolute timing is used, and thus could pose certain restrictions on the attacks that can be executed. For instance, consider the attack scenario where the timing of a search query is related to the number of returned results. In a sequential timing attack, the adversary would observe higher measurements (barring jitter) for queries returning more results, and from this may be able to estimate the number of search results. Achieving the same with concurrency-based timing attacks is more complicated: instead of inferring the number of search results directly from the timing measurements, the adversary would need to leverage several baseline queries that return a known number of results. When the (target, baseline) request-pair is returned in an equally distributed order, i. e. the target response is received as many times before the baseline response as it is received after, this indicates that the target query returns the same number of results as for the specific baseline request that was used. Note that if the adversary is unable to construct baseline requests that return a given number of search results, it would be infeasible to perform the timing attack by levering concurrency.

For operations that need to be performed sequentially, e. g. the different steps of a cryptographic handshake or the deciphering of a block that depends on the outcome of the previous block, there is no concurrency at the protocol level. In such cases, the adversary would need to leverage the coalescence of packets at the transport layer. For example, for TLS it is not possible to initiate multiple concurrent handshakes over a single TCP connection. As such, to exploit a (hypothetical) timing side-channel leak in the TLS handshake using

concurrency-based attacks, an adversary would need to start two separate TCP connections. Additionally, to ensure that these arrive simultaneously, the requests with the payload should be encapsulated and coalesced at the network layer, e. g. as we showed with the attacks against Tor hidden services and over Wi-Fi.

6.3 Defenses

The most effective counter-measure against timing attacks is to ensure constant time execution. However, this can be very difficult to implement, or virtually impossible, especially for complex applications that rely on third-party components. In this section we propose various defenses that focus on reducing the performance of concurrency-based timing attacks to the that of sequential timing attacks. The defenses that we describe are generic, and can be applied regardless of the application domain.

A straightforward defense is adding a random delay on incoming requests. To mimic network conditions where the standard deviation of the jitter is 1ms (comparable to what we found on a EU-EU network connection), this delay can be sampled uniformly at random from the range $[0, \sqrt{12}]$ms[8], resulting in an average delay of $\approx$1.73ms for every request.

However, only requests that arrive simultaneously at the server need to be padded because all others are subjected to network jitter. To evaluate what percentage of requests would thus need to be delayed in a real-world setting, we performed a simulation based on the crawling data of HTTPArchive. This dataset contains detailed information on 403 million requests that were made while visiting over 4 million web pages with the Chrome browser. For our simulation, we only considered HTTP/2 requests, in total 237 million (58.82%), made to over 2 million different hosts. We considered two requests to be concurrent if they were made within 10ms. Note that this is an overestimation, as we wanted to account for requests that may be coalesced on the network layer. If the server has more knowledge on how the request was transported, the length of the timeframe in which requests are considered concurrent can be significantly reduced. With this improvement, a random padding would still need to be added to 90.95% of the incoming requests according to our simulation. This means that websites are making extensive use of the multiplexing capabilities of HTTP/2.

A further improvement that can be made is to only add padding to concurrent requests to the same endpoint, or to endpoints that have a similar processing time. To estimate the percentage of requests that would be affected by this, we first compute the average processing time for all endpoints on every host by determining the timing difference between sending the request and the arrival of the first byte of the response. This reduces the number of requests that require

---

[8]The standard deviation of a uniform distribution $[a,b]$ is $(b-a)/\sqrt{12}$. So to simulate a jitter of 1ms we can uniformly pick a delay from $[0, \sqrt{12}]$.

padding to 69.81%. By further exploring the requests, we find that most requests with a similar processing time are to static resources. We determined a resource to be static based on the `Cache-Control` and `Expires` headers. With these three improvements, we found that in our simulation in total 20.86% of the requests require padding. Note that this should be considered an upper bound, as not all requests that we considered as concurrent will actually be coalesced in a single packet. When all improvements were applied, we find that 67.53% of the hosts had to apply padding on at most 5% of their requests. For completeness, in Appendix C we show a CDF with the percentage of requests that need to be padded.

Finally, to defend browser users against concurrency-based timing attacks, e. g., in a cross-site attack scenario, a possible defense is to ensure that different requests are not coalesced in a single packet. In a practical setting, this would mean that although the TCP sending window may cause the network stack to buffer data sent by the application, the packets that are sent out never contain more data than what was sent by the application. Essentially, this comes down to disabling Nagle's algorithm. As a result, every request would be sent in a separate TCP packet. Alternatively, packets containing a single request could be padded to the MTU to prevent the network stack and any intermediaries on the path between the client and the server from coalescing two requests in a single packet. Another option is to ensure that the order in which the responses are received, is not observable by the attacker code in the browser, e. g., by leveraging deterministic timing, as proposed by Cao et al. [13]. Note that approaches that limit the accuracy of timers available in the browser [31, 45], are ineffective in defending against concurrency-based timing attacks.

7 Conclusion

With classical timing attacks, an adversary sequentially collects a series of measurements and then applies statistical analysis to try to infer secret information. In this paper, we introduce a new paradigm of performing remote timing attacks that infers secrets based on the order in which two concurrently executed tasks finish. By using a combination of request multiplexing and coalescing of packets, we show how it is possible to ensure that two requests arrive simultaneously at the targeted server. As a direct result, the difference in time when the requests are processed is completely unaffected by network conditions. As we describe in our theoretical model and later show through practical measurements, our concurrency-based remote timing attacks are not subjected to network jitter and thus have a performance comparable to that of an attacker on the local system.

Acknowledgments

References

[1] Onur Aciiçmez, Werner Schindler, and Çetin K Koç. Improving brumley and boneh timing attack on unprotected SSL implementations. In *CCS*, 2005.

[2] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, pages 526–540. IEEE, 2013.

[3] Mark Allman, Vern Paxson, and Ethan Blanton. Requirements for Internet Hosts - Communication Layers. RFC 5681, September 2009.

[4] Alberto Bartoli, Eric Medvet, and Filippo Onesti. Evil twins and WPA2 enterprise. *Comput. Secur.*, May 2018.

[5] Christopher Baus. TCP_CORK: More than you ever wanted to know. `https://baus.net/on-tcp_cork/`, April 2005.

[6] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[7] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW*, 2007.

[8] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, October 1989.

[9] Sebastian Brenza, Andre Pawlowski, and Christina Pöpper. A practical investigation of identity theft vulnerabilities in eduroam. In *WiSec*, 2015.

[10] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology (CRYPTO)*, 2010.

[11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*. Springer, 2011.

[12] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[13] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *CCS*, pages 163–178, 2017.

[14] Zhen Cao, Baohong He, Yang Shi, Qin Wu, and Glen Zorn. EAP extensions for the EAP re-authentication protocol (ERP). RFC 6696, 2012.

[15] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. Opportunities and limits of remote timing attacks. *ACM TISSEC*, 12(3):17, 2009.

[16] Nik Cubrilovic. RockYou hack: From bad to worse. `https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/`, 2009.

[17] Ana Custura, Gorry Fairhurst, and Iain Learmonth. Exploring usable path MTU in the internet. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–8. IEEE, 2018.

[18] Andy Davies. Tracking HTTP/2 prioritization issues. `https://github.com/andydavies/http2-prioritization-issues`, 2019.

[19] Chris Evans. Cross-domain search timing. `https://scarybeastsecurity.blogspot.com/2009/12/cross-domain-search-timing.html`, December 2009.

[20] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *CCS*, pages 25–32, 2000.

[21] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. Who left open the cookie jar? A comprehensive evaluation of third-party cookie policies. In *USENIX Security*, pages 151–168, 2018.

[22] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *CCS*, pages 1394–1405, 2015.

[23] Dan Harkins. Improved Extensible Authentication Protocol Using Only a Password. Internet-Draft draft-harkins-eap-pwd-prime-00, Internet Engineering Task Force, July 2019. Work in Progress.

[24] Dan Harkins and G. Zorn. Extensible authentication protocol (EAP) authentication using only a password. RFC 5931, August 2010.

[25] Joshua Hill. An analysis of the RADIUS authentication protocol. `https://www.untruth.org/~josh/security/radius/radius-auth.html0`, 2001.

[26] HTTPArchive. Web almanac: HTTP/2. `https://almanac.httparchive.org/en/2019/http2`, 2019.

[27] IEEE Std 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec*, 2016.

[28] Kevin M. Igoe. Re: [Cfrg] status of Dragon-Fly. https://www.ietf.org/mail-archive/web/cfrg/current/msg03264.html, December 2012.

[29] Alexander Klink and Julian Wälde. Effective DoS attacks against web application plattforms. https://fahrplan.events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf, 2007.

[30] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[31] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, 2016.

[32] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *IEEE S&P*, 2020.

[33] Linux. tcp(7) - Linux man page. https://linux.die.net/man/7/tcp, 2007.

[34] Patrick McManus. Sockettransportservice socket limits. https://bugzilla.mozilla.org/show_bug.cgi?id=1260218, March 2016.

[35] Patrick Meenan. Better HTTP/2 prioritization for a faster web. https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/, May 2019.

[36] Rowan Merewood. SameSite cookies explained. https://web.dev/samesite-cookies-explained, 2019.

[37] Christopher Meyer and Jörg Schwenk. Lessons learned from previous SSL/TLS attacks-a brief chronology of attacks and weaknesses. *IACR Cryptology ePrint Archive*, 2013:49, 2013.

[38] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *USENIX Security*, 2014.

[39] Open System Consultants. RadSec: a secure, reliable RADIUS protocol. http://www.open.com.au/radiator/radsec-whitepaper.pdf, 2012.

[40] Roberto Peon and Herve Ruellan. HPACK: Header compression for HTTP/2. *Internet Requests for Comments, RFC Editor, RFC*, 7541, 2015.

[41] Renwa. Bypass SameSite cookies default to Lax and get CSRF. https://medium.com/@renwa/bypass-samesite-cookies-default-to-lax-and-get-csrf-343ba09b9f2b, January 2020.

[42] Stephen Röttger. Issue 843157: Security: leak cross-window request timing by exhausting connection pool. https://bugs.chromium.org/p/chromium/issues/detail?id=843157, May 2016.

[43] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. Bakingtimer: privacy analysis of server-side request processing time. In *ACSAC*, pages 478–488, 2019.

[44] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. Clock around the clock: time-based device fingerprinting. In *CCS*, pages 1502–1514, 2018.

[45] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[46] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 2019.

[47] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *USENIX WOOT*, 2018.

[48] Wright Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. 1997.

[49] The Chromium Projects. SameSite updates. https://www.chromium.org/updates/same-site, 2020.

[50] Tatsuhiro Tsujikawa. Nghttp2: HTTP/2 C library. https://nghttp2.org/, February 2015.

[51] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, pages 1382–1393, 2015.

[52] Anne van Kesteren. Fetch - living standard. https://fetch.spec.whatwg.org/, January 2020.

[53] Mathy Vanhoef and Frank Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[54] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*. IEEE, 2020.

[55] Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP encrypted information can be stolen through TCP-windows. In *Black Hat US Briefings*, 2016.

[56] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security*, pages 849–864, 2017.
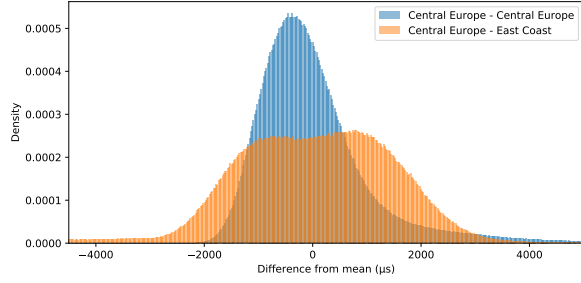
Figure 6: Distribution of how the timing of requests differs from the mean, observed from our university's network.

Figure 7: Distribution of response times of baseline ($100\mu s$) and target ($150\mu s$) requests to the EU server.

[57] Wi-Fi Alliance. WPA3 specification version 1.0. https://wi-fi.org/file/wpa3-specification, April 2018.

[58] Klaas Wierenga, Mike McCauley, Stefan Winter, and Stig Venaas. Transport Layer Security (TLS) Encryption for RADIUS. RFC 6614, May 2012.

A Negative result: concurrent timing attacks against HTTP/1.1

The most popular communication protocol used on the web is HTTP. Over the several decades that it has existed, many improvements have been made, for instance keeping connection alive since HTTP/1.1, allowing multiple requests to be made over the same connection. However, version 1.1 still suffered from a significant performance drawback, namely head-of-line (HOL) blocking, which prevented user agents from sending multiple concurrent requests over the same TCP connection. As a result, the sequence of responses will always be the same sequence as those of the associated requests. The primary way to overcome the consequences of HOL blocking is to initiate multiple TCP connections to the server. Although this allows multiple requests to be processed concurrently, there is no guarantee that when the requests are sent at the same time, i.e., in very rapid succession, these will also arrive at the server at the same time. The jitter values incurred when sending the different requests are largely independent from each other, and thus both affect the time at which the server starts processing them, which in turn affects the order in which they are returned.

To evaluate the impact of network-based jitter and the potential improvements of measuring the timing of two concurrent requests, we performed several measurements. We set up two HTTP/1.1 web servers on two Amazon EC2 C5 instances, one in central Europe, and one on the US east coast. In Figure 6, we show the distribution of how much the timing of each request to the same endpoint differs from the mean, for both instances, based on two million measurements per instance. This clearly shows that the variance is significantly higher for the requests targeting the web server in the US
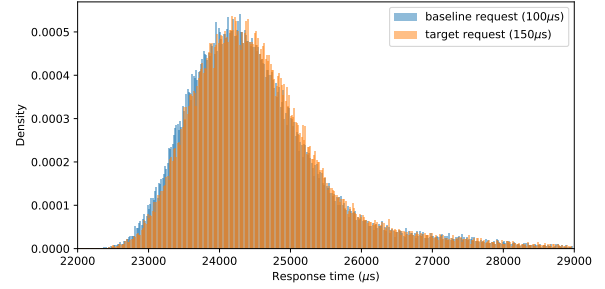
(average RTT: 118.11ms, standard deviation: 1.71ms), compared to the requests to the EU server (average RTT: 24.50ms, standard deviation: 1.03ms).

To evaluate how this variance of response times affects the capabilities of an adversary to infer secret information based on the execution time of a request, we made requests to a PHP file, which would idle for a defined period before returning a response, using the `time_nanosleep()` function. We used a baseline of $100\mu s$, and then obtained measurements for increments of the baseline ranging from 50ns to $50\mu s$, using the Python aiohttp library; we measured the elapsed timing using `time.time_ns`, which has nanosecond resolution. We altered between a baseline request and a request with an increased timing. For each increment value, we obtained 160,000 measurements. Next we applied the box test [15], to determine the number of requests an attacker would need to distinguish a difference in execution time for the varying increments compared to the baseline, with an accuracy of at least 95%. We found that for the EU server, a timing difference of $20\mu s$ could be determined with 4,333 requests; for the US-based server,the attack was unsuccessful (based on the 10,000 upper bound we imposed). A timing difference of $50\mu s$ required at least 1,580 requests for the EU-server, and 3,198 measurements for the server based in the US. Figure 7 shows the distribution of the requests with a $50\mu s$ timing difference to the EU server, indicating a large overlap with the baseline request, but still an observable shift.

Finally, we evaluated whether concurrency using multiple connections in HTTP/1.1 could be used to improve the measurements. For this, we set up two concurrent connections to the targeted server, again using the Python aiohttp package, and sent two requests at the same time, one to the baseline endpoint, and one alternating between the baseline and the endpoint with increased processing time. We then subtracted the timing of each pair of concurrent requests. Interestingly, we found that this differential timing technique leveraging concurrency performed worse than the basic attack: for the EU server, it was no longer possible to distinguish a timing difference of $20\mu s$. Moreover, a timing difference of $50\mu s$ required at least 4,752 measurements to this server, consisting of

					Timing difference			
Attack type	*Connection*	200ns	500ns	1μs	2μs	5μs	10μs	20μs	50μs
Concurrent	Misc. (Internet)	-	45,192	33,394	20,022	2,382	980	313	34
Sequential	Europe - Europe	-	-	-	-	-	-	-	2,711
	Europe - East US	-	-	-	-	-	-	-	12,588
	Europe - South-East Asia	-	-	-	-	-	-	-	-
	VM-VM (LAN)	-	-	29,935	2,333	93	39	33	23
	localhost	-	26,104	6,500	1,087	158	35	22	17

Table 2: The average number of requests or request-pairs required to perform a successful timing attack with at least 95% accuracy against an Apache webserver.
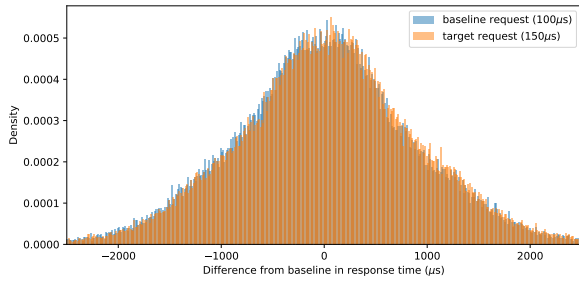

Figure 8: Distribution of the difference in response time for concurrently launched requests to the baseline ($100\mu s$) and target ($150\mu s$) endpoints hosted on the EU server.

two requests each, a six-fold increase compared to the typical attack. The distribution of these measurements is displayed in Figure 8, showing that the two distributions are more difficult to distinguish, compared to the distributions shown in Figure 7. For the US server, the attack only achieved an accuracy of 82.14% for the imposed upper limit of 10,000 requests. We believe that the reason for this degraded performance is that the jitter values are no longer independent, but are affected by the concurrent request. More precisely, requests can not be sent completely concurrently on the network along the same path: only a single request can be put on the wire at once. Consequently, the request that is sent last will also be subjected to the jitter incurred when sending the first request.

B   HTTP/2 measurements for Apache

In Table 2, we show the number of requests or request-pairs that are needed to perform a timing attack against an Apache2 web server with at least 95% accuracy. For smaller timing differences (500ns - 1μs), we can see that the concurrency-based timing attacks perform similarly to a sequential attack on the local network. For timing differences between 2μs and 20μs, the concurrency-based attacks require more requests. This can be attributed to the increased processing time required to

handle requests in Apache2 (as compared to nginx), resulting in higher jitter values. Presumably, this is amplified for concurrency-based attacks because here two requests arrive (instead of just one with the sequential attacks).
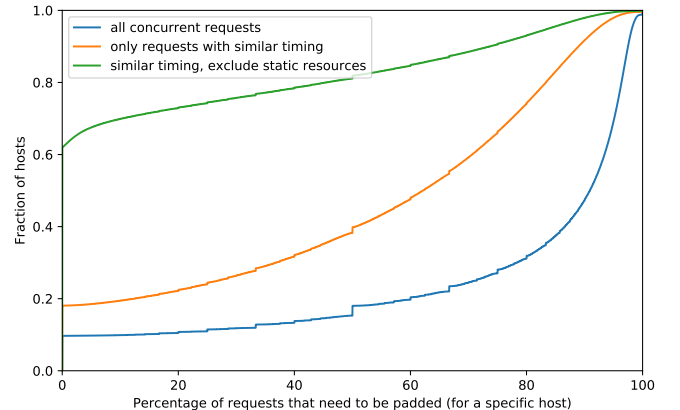
C   Overhead of defenses per host

Figure 9: CDF of the percentage of requests that had to be padded for the three defenses discussed in Section 6.3.

In Figure 9, we show the cumulative distribution function (CDF) for the three variations of defenses that were introduced in Section 6.3: 1) padding all requests that arrived concurrently at the server, 2) only padding requests when within the set of concurrent requests, there is a request with a similar processing time, and 3) using the same optimization but not adding padding to requests for static resources. The CDF shows that for the third defense, more than half of the hosts would not incur any overhead from the defense. The other two defenses perform significantly worse: for half of the hosts random padding needs to be added to at least 90.91% and 62.16% of the requests, respectively.