Off-Path TCP Exploit: How Wireless Routers Can Jeopardize Your Secrets

Weiteng Chen
University of California, Riverside
wchen130@ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Abstract

In this study, we discover a subtle yet serious timing side channel that exists in all generations of half-duplex IEEE 802.11 or Wi-Fi technology. Previous TCP injection attacks stem from software vulnerabilities which can be easily eliminated via software update, but the side channel we report is rooted in the fundamental design of IEEE 802.11 protocols. This design flaw means it is impossible to eliminate the side channel without substantial changes to the specification. By studying the TCP stacks of modern operating systems and their potential interactions with the side channel, we can construct reliable and practical off-path TCP injection attacks against the latest versions of all three major operating systems (macOS, Windows, and Linux). Our attack only requires a device connected to the Internet via a wireless router, and be reachable from an attack server (*e.g.,* indirectly so by accessing to a malicious website). Among possible attacks scenarios, such as inferring the presence of connections and counting exchanged bytes, we demonstrate a particular threat where an off-path attacker can poison the web cache of an unsuspecting user within minutes (as fast as 30 seconds) under realistic network conditions.

1 Introduction

Side channels in networking stacks have recently been demonstrated to precipitate serious attacks. One of the most noteworthy cases is CVE-2016-5696 [18] where a completely blind off-path attacker can infer whether two arbitrary hosts on the Internet are communicating using a TCP connection. The attacker can even infer the TCP sequence numbers in use from both sides of the connection. In addition to this serious vulnerability, other types of side channel vulnerabilities have also been discovered in various scenarios and protocol components [39, 40, 25, 24, 27, 33, 23, 48, 13]. Fundamentally, like any side channel vulnerabilities, these vulnerabilities are introduced by shared resources between the attacker and victim.

In the case of TCP, for example, a server has many kinds of shared resources implemented by operating systems such as a global IP ID counter [1, 25, 23], SYN cache and RST limit [24], SYN-backlog [33], and challenge ACK rate limit [18]. These resources are shared on a host between a connection established with the attacker and a connection with the victim.

When the attacker sends spoofed packets to the server that appear to come from the victim, these shared resources are used differently, depending on the validity of the spoofed packets (*e.g.,* in-window vs out-of-window sequence number). By observing the shared resources, how these spoofed packets are processed are visible to the attacker.

All existing vulnerabilities related to off-path TCP exploit essentially stem from software artifacts. The ones that can lead to serious attacks are already patched primarily by (1) eliminating the shared resources in protocol implementations (or adding randomness to them) [7, 8] and (2) reducing the opportunities that the shared resources leak information, *e.g.,* employing a more stringent acknowledge(ACK) number check [44]). As we will discuss later in §2.3, almost all previously reported off-path TCP attacks no longer work.

Unlike yet another software side channel, we report a fundamental side channel inherent in all generations of IEEE 802.11 or Wi-Fi technology, because they are *half-duplex.* From its definition: when there are uplink wireless frames being transmitted, downlink frames have to wait, and vice versa. This basic and fundamental design seems benign but it creates a timing channel sensitive to the contention between uplink and downlink traffic. For example, a downlink packet measuring the RTT will incur a higher latency if uplink traffic is going on. As we will show in the paper, an attacker can leverage the timing channel to craft clever sequences of packets, creating primitives to infer TCP sequence number and ACK number, ultimately completing a working off-path TCP

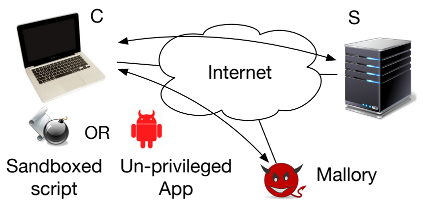


Figure 1: Threat model

exploit.

Through extensive experimentation, we demonstrate that the timing channel is reliable (through amplification) and can be used even when the attacker and victim are far away (with RTTs over 20ms). We implement a realistic blind off-path attack that can achieve web cache poisoning within minutes. The video demo can be found on our project website [3]. We also open sourced the attack implementation at [5] to assist the reproduction and further research of the work.

The contributions of the paper are the following:

- We report the timing side channel inherent in all generations of IEEE 802.11 or Wi-Fi technology. We show the timing channel is reliable and amplifiable and unfortunately almost impossible to eliminate without substantial changes to the 802.11 specification.
- We show that the side channel affects macOS, Windows, and Linux by studying the overlaps and differences in their TCP stack implementations. We construct the only off-path TCP exploit working at the moment based on this new side channel.
- We provide a thorough analysis and evaluation of the proposed attack under different router/network/OS/browser combinations. We also suggest possible defenses to alleviate this attack.

Roadmap. The rest of the paper is organized as follows: we begin with background introduction and the most relevant work in § 2, and then present the timing side channel in Wi-Fi technology in § 3. § 4 describes an overview of the off-path TCP exploit and its goal. In § 5, we elaborate the implementation of the attack against different OSes. In § 6, we evaluate our attack under different conditions. § 7 discusses some potential attacks that exploit the vulnerability. We propose some mitigation schemes at different layers in § 8. We also introduce previous research related to side channels in § 9. Finally, § 10 concludes the paper.

2 Off-Path TCP Exploits

2.1 Generic Threat Model

Fig. 1 illustrates a typical off-path TCP hijacking threat model consisting of three hosts: a victim client, a victim server and an off-path attacker. The off-path attacker, Mallory, is capable of sending spoofed packets with the IP address of the legitimate server. In contrast to Man-in-the-middle attack, Mallory cannot eavesdrop the traffic transferred between a client *C* and a server *S*. Depending on the nature of the side channel, an unprivileged application or a sandboxed script may be required to run on the client side [40, 27] to observe the results of the shared state change and determine the outcome of the spoofed packets (*e.g.,* whether guessed sequence numbers are in-window). In rare cases, if the state change is remotely observable, an off-path attacker can complete the attack alone without the assistance from the unprivileged application or script [18]. After multiple rounds of inferences, starting from whether a connection is established (four tuple inference) to the expected sequence number and ACK number inference, the attacker can then inject a malicious payload that becomes acceptable to the client at the TCP layer.

The side channels typically manifest themselves through the following control flow block:

```
if (in_packet.seq is in rcv_window)
    // shared state change 1
else
    // shared state change 2
```

The example illustrates two variables: (1) the attacker-controlled variable *in_packet.seq* — guessed sequence number in a spoofed packet and (2) the receive window deciding what *in_packet.seq* are valid. Depending on the outcome of the comparison, the shared state may change to different values. The change also has to be observable by the attacker through some side channel. Two necessary building blocks are needed in a TCP off-path side channel attack: (1) existence of vulnerable packet validation logic; (2) the shared state has to be observable by an attacker (*i.e.,* the sandboxed script, unprivileged app, or the off-path attacker). Note that together these two building blocks result in a violation of the non-interference property [29, 50].

Next we give an overview of these two building blocks used by previous attacks and explain why those attacks no longer work. Simply put, they either rely on outdated TCP packet validation logic or shared state that can be easily eliminated.

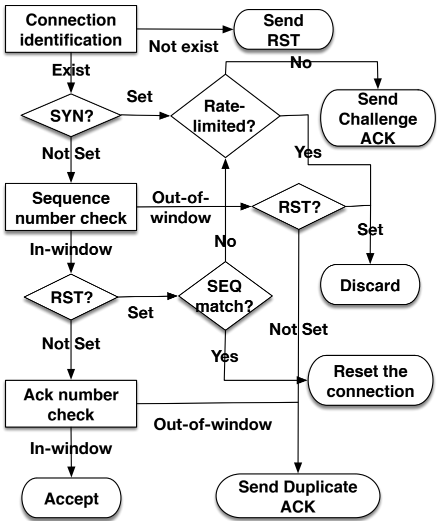


Figure 2: Incoming packet validation logic in RFC

2.2 Latest TCP Incoming Packet Validation Logic

To understand how incoming packets are validated, we refer to the standards of RFC 793 [4] and RFC 5961 [44]. We focus on the latest standard only as it is helpful in understanding why attacks against old versions now fail. Note that even though different operating system implementations may differ in reality, they still try to keep up with the standards (albeit with their own tunings) and overall it provides a foundation for discussion. We discuss the specific operating system implementations in §4.

We distill the latest standard and summarize it in Fig. 2. It involves primarily three types of checks, and each of them has some form of vulnerable logic — different actions are taken depending on the outcome of the check (*e.g.,* a response packet is sent vs. not).

• **Connection (four-tuple) Identification**: The first check tries to identify if an incoming packet belongs to any established connection based on the four tuples – source and destination port numbers as well as IP addresses. If no ongoing connection matches the four tuples, an incoming packet not containing a RST causes a RST to be sent in response. Otherwise, if the SYN bit is set, irrespective of the sequence number, TCP must send an ACK referred to as challenge ACK to the remote peer to confirm the loss the previous connection. Upon receipt of this challenge ACK, a legitimate remote peer who truly lost its connection, after a restart, sends a RST packet back with the sequence number derived from the ACK field of the challenge ACK, which can terminate the connection at that point. The challenge ACK is hence a defense against blind off-path attacks that attempt to terminate a connection forcefully through spoofed SYN packets.

• **Sequence number check**: This check makes sure that the sequence number falls in the receive window. Otherwise, according to the TCP specification RFC 793, an immediate duplicate ACK packet should be sent in reply (unless the RST bit is set, in which case the packet is dropped without reply). If the sequence number is in window and RST bit is on, similar to handling SYN, RFC 5961 suggests the use of challenge ACKs to defend against off-path RST attacks: only if the sequence number matches the next expected sequence number, a receiver terminates the connection; otherwise, the receiver must send a challenge ACK.

• **ACK number check**: Pre-RFC 5961, the ACK number is considered valid as long as it falls in the wide range of [`SND.UNA`$-(2^{31}-1)$, `SND.NXT`][1], which is effectively half of the ACK number space. Thus, an attacker only needs to guess two ACK numbers for every guessed sequence number to successfully inject data into a connection, resulting in a guaranteed successful data injection with up to $2*2^{32}/$`RCV.WND`[2] spoofed data packets. RFC 5961 proposes a much more stringent check suggesting a valid ACK number should be within [`SND.UNA - MAX.SND.WND, SND.NXT`][3], where `MAX.SND.WND` is the maximum receive window size the receiver has ever seen from its peer. If the ACK number is out of this window, the packet is dropped and an ACK should be sent back [44]. If the ACK number is in window yet there is no payload, then the packet should be silently dropped.

Besides, to alleviate the waste of CPU and bandwidth resources caused by challenge ACKs, an ACK throttling mechanism is also proposed. Specifically, the system administrator can configure the maximum number of challenge ACKs that can be sent out in a given interval.

2.3 Prior Attacks and Side Channels

Now that we understand how the generic TCP packet validation logic is envisioned by the standard, we describe the known shared states that lead to side channels, combined with the variants in TCP packet validation logic in different operating systems (sometimes out-of-date), that were leveraged by existing attacks.

• **Global IPID counter**. Until recent years, Windows is the only operating system that chooses to maintain a globally incrementing IPID counter shared across all connections and stamped onto the IPID field in IP header for every outgoing packet [23]. This creates a side channel that allows an attacker to count how many outgoing

[1]`SND.UNA`: the sequence number of the first byte of data that has been sent but not yet acknowledged; `SND.NXT`: the sequence number of the next byte of data to be sent

[2]`RCV.WND`: size of receive window

[3]The window can be as small as a few thousand bytes, which makes the guess much more difficult

Side channel	Requirement	Affected OS	Patch/Mitigation
Global IPID count [1, 25]	Pure off-path or Javascript	Windows	Global IPID counter eliminated
Direct browser page read [27]	Javascript	Any old OS	RFC 5961
Global challenge ACK rate limit [18]	Pure off-path	Linux	Global rate limit eliminated
Packet counter [40, 39]	Malware	Linux,macOS	Namespace / macOS* patch [9, 10]
Wireless contention (this work)	Javascript	Any	N/A

Table 1: Summary of Different Off-Path TCP Side Channel Attacks including the one we propose in this paper

packets have been sent during a time interval, through diffing the queried IPIDs of a Windows machine. This is leveraged in several off-path TCP attacks [1, 25]. Using IP spoofing, an off-path attacker can tell whether the guess is correct based on whether a response is triggered.

However, at the time of writing, we experimentally verify that Windows 10 has finally eliminated this side channel by adopting a safer IPID generation algorithm similar to that used in Linux [33], where connections destined for different IP addresses will no longer share the same IPID counter.

• **Browser page read.** In this attack [27], the shared state is a browser page where an attacker runs malicious Javascript and attempts to inject data into connections to a benign website (both the benign connection and malicious script run under the same page). The successful guess of the TCP sequence number results in a direct feedback from the browser page load. There are three main culprits of the attack: (1) older operating systems follow an earlier standard RFC 793 that considers half of the ACK number space valid. An off-path attacker only needs to guess two ACK values with every guessed sequence number to inject data successfully. Therefore, the feedback about when the injection succeeds is when the malicious payload gets loaded and rendered by the browser. (2) modern browsers are tolerant of response data: if the HTTP response header is missing, the browser simply attaches one automatically. This frees the attacker from having to prepare the header at an exact sequence number (otherwise the browser considers the response invalid and closes the connection). (3) HTTP pipeline is required so that a response arrives ahead of time will be deemed valid.

This attack no longer works because the first culprit is eliminated by most modern operating systems (including Windows, Linux, Android), which adopted a more stringent check on ACK numbers as defined in RFC 5961 where only a much smaller window is considered valid. In addition, from our testing, HTTP pipeline is disabled or not implemented in all modern browsers, eliminating the third culprit as well.

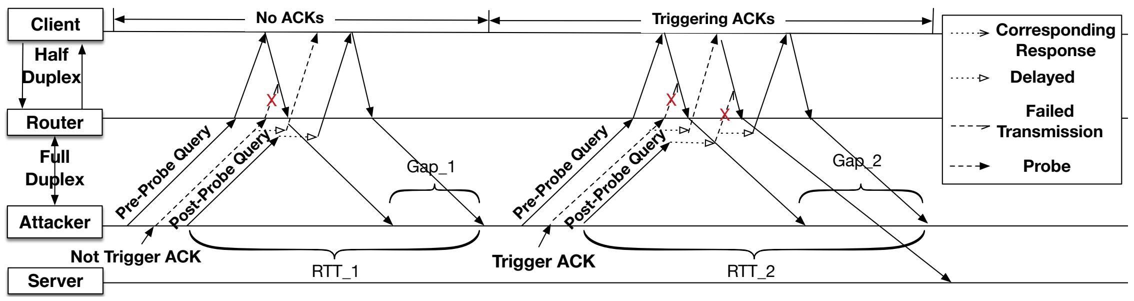
• **Global challenge ACK rate limit**. The Linux kernel first implemented all the features suggested in RFC 5961 in version 3.6 and its TCP packet validation logic closely matches the one shown in Fig. 2. Notably, it implements the recommended ACK throttling feature by introducing a global system variable to control the maximum number of challenge ACKs generated per second. As this limit is shared across all connections, the shared state can be exploited as a side channel. For instance, to infer if an ongoing connection exists, an off-path attacker can initially send a spoofed packet with one guessed port number and SYN bit set; after the attacker sends another 100 [4] non-spoofed in-window RST packets to exhaust the challenge ACK count, it can then observe the number of responses to tell whether its initial spoofed packet matches the four tuples of an ongoing connection and hence triggers a challenge ACK.

Since the shared rate limit is a simple software artifact, shortly after the vulnerability was reported, it was eliminated in a patch introduced in Linux 4.6 [8, 42] where a per-socket rate limit is used instead.

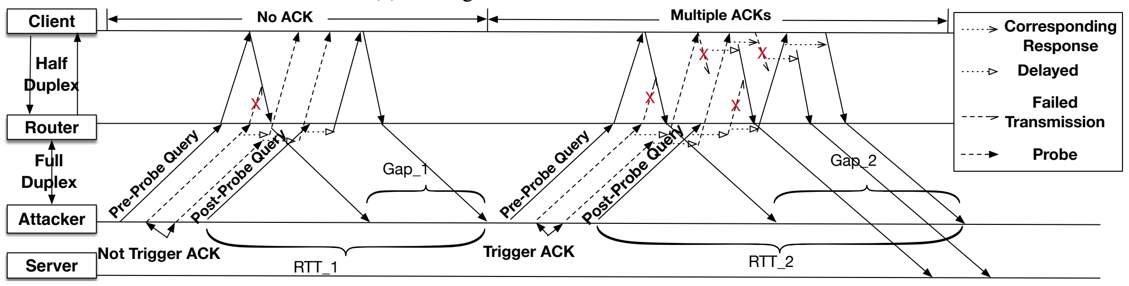
• **System-wide packet counter**. Packet counters report aggregated statistics across all connections and are reliable side channels demonstrated in recent off-path attacks [40, 39]. These attacks require a piece of unprivileged malware to run on the client machine that can access these packet counters and use them as feedback for spoofed packets sent by the off-path attacker. Due to the fact that these counters are internal to TCP implementations, they may leak more diverse and fine-grained information (more than what the standard packet validation logic can leak). In the extreme case, for example, a Linux/Android TCP packet named *DelayedACKLost* is incremented only when it receives a packet with a sequence number smaller than the expected one. This allows an attacker to conduct a binary search on the expected sequence number. Similar dangerous packet counters exist on macOS as well [40].

These packet counters are being mitigated in a number of ways. For Linux, it introduced the mechanism of namespace so that sensitive apps and untrusted apps can run in separate namespaces with isolated counters. For macOS, the side channel vulnerability has recently been assigned CVE-2017-13810 and patches have been pushed out to zero the sensitive counters [9, 10].

[4]It's the default threshold in Linux version 3.6



(a) Timing difference between two cases



(b) Triggering multiple ACKs

Figure 3: Vulnerability caused by wireless contention

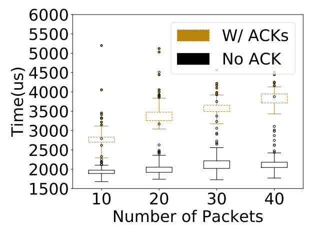
Summary. Overall we listed four different types of software-based side channels that have been exploited to launch off-path TCP attacks. We summarize them in Table 1 for reference. In short, only the packet counter side channels still exist (validated on Linux and Android 8.0). In any event, this side channel requires a high bar to launch because of the malware requirement. In the next section, we describe our newly discovered side channel in detail.

3 Wi-Fi Timing Channel

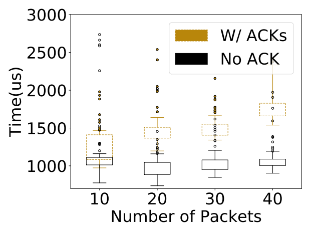
Fundamentally, the *half-duplex* nature of Wi-Fi creates a “shared resource” among uplink and downlink traffic, a prerequisite of any side channel. By sharing the same set of frequency bands with both directions, Wi-Fi relies on carrier-sense multiple access (*i.e.,* CSMA) to share/-divide the channel over time. This means that a node transmits only when the channel is sensed to be idle and thus it has the exclusive right to transmit. This effectively creates a timing channel that delays the local transmission if the opposite direction is transmitting at the same time.

Even worse, this timing difference becomes more visible due to retransmissions caused by contention (collision). Specifically, the protocol starts by listening on the channel and immediately sends the first frame to the transmit queue if the channel is found to be idle; however, this leads to waste of transmissions if collision occurs. If the channel is subsequently sensed to be busy, it waits for a period of time (*e.g.,* usually random or exponential backoff [17]) attempting to avoid collision. Although it might benefit the performance when many nodes are active, it creates a significant overhead when only one is present (plus the AP). In addition to backoffs, Request to Send/Clear to Send (RTS/CTS) [16] may optionally be used to mediate access to the shared medium to solve the hidden-terminal problem [46] where multiple stations can see the Access Point but not each other. Unfortunately, in the same scenario where there is only one node, it introduces unnecessary traffic to the network, slowing everything down. Finally, it is important to note the latency is amplified further when more contention is present (*e.g.,* more frames to be transmitted in either direction).

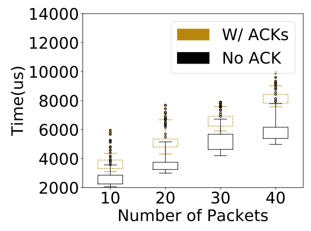
Exploiting the timing channel. To demonstrate the timing channel, we create a probing strategy to measure the delay effects. As we can see in Fig. 3a, we simulate an off-path TCP attack where the attacker sends a spoofed probing packet, along with a pre-probe query and post-probe query to measure the RTT before and after. If the spoofed packet does not trigger an ACK on the client, *e.g.,* because the guessed sequence number is in-window (left half of the figure), then the post-probe query arrives at the client faster and gets back sooner (smaller RTT). On the other hand, if the spoofed packet triggers an ACK on the client, *e.g.,* because the guessed sequence number is out-of-window (right half of the figure), then the post-probe query experiences contention with the ACK from



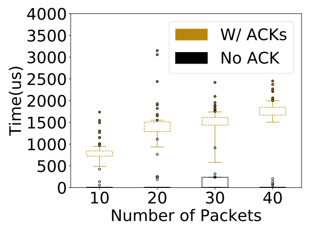
(a) RTT measurement of macOS using 5GHz network of a Huawei router



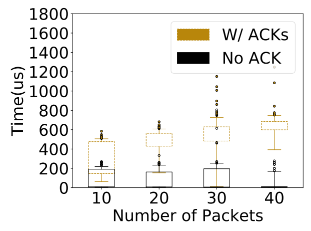
(b) RTT measurement of Linux using 5GHz network of a Linksys router



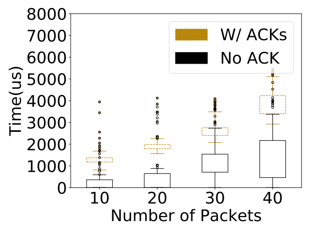
(c) RTT measurement of macOS using 2.4GHz network of a Xiaomi router



(d) Gap measurement of macOS using 5GHz network of a Huawei router



(e) Gap measurement of Linux using 5GHz network of a Linksys router



(f) Gap measurement of macOS using 2.4GHz network of a Xiaomi router

Figure 4: Selective measurement of wireless connections in a local setup. X axis is the number of probing packets that attackers send per test. The box extends from the lower to upper quartile values of the data. And the whiskers extend from the box to show the range of the data at specific percentiles (i.e. [0, 90]). Beyond the whiskers, data are considered outliers, plotted as individual points.

the client, and therefore prolongs the measured RTT. In addition to the RTT difference ($RTT\_2 > RTT\_1$), we can also measure the gap between the replies of the first query and the second, which should capture the delay effects similarly.

In Fig. 3b, we also illustrate the amplifiable nature of the timing channel where the attacker sends two spoofed probing packets, causing more contention which delays post-probe query even further.

In summary, this side channel allows an attacker to determine if the spoofed probing packets have triggered any response or not, coincidentally achieving the same purpose as the global IPID counter on Windows (which is no longer available). In contrast, Wi-Fi contention is here to stay.

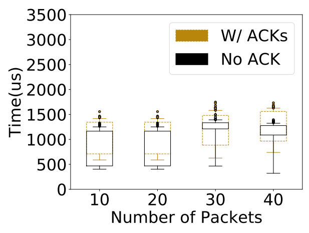
Empirical testing. So far we only conceptually analyzed the side channel and its effects. We now conduct a controlled local experiment to understand its real-world implications. Following the same topology in Fig. 7, we created a total of 16 different setups to make sure that the side channel exists in various generations of technologies and products. We used 4 different wireless routers (from Linksys, Huawei, Xiaomi, and Gee): all latest generations that support 802.11ac and 802.11b/g/n. We used two different machines as clients: an early-2017 Macbook and a mid-2017 Dell Desktop. Finally, we varied the frequency of the router between 2.4GHz and 5GHz so that both 802.11n and 802.11ac were tested (802.11ac is used for 5GHz only).

The measurements are conducted in a single-family house where we have relatively little wireless interference (with at most 4 potential users at home). Due to space constraint, we present 6 representative results of the measurement in Fig. 4. Each plot with a box and whiskers presents the data measured with 100 runs. On average, we can see that the timing difference for RTT is about 1 to 3ms when the number of probing packets is 30 or more. Although differences exist among those setups, the timing side channel is clear and measurable(see §5.4). Later in §6, we also evaluate its robustness to noise.

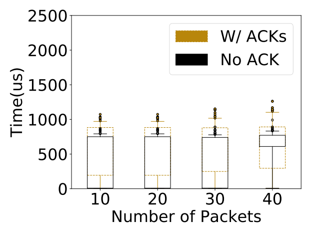
Half-duplex vs. Full-duplex To better understand that the significant part of the RTT difference is due to the half-duplex nature of wireless rather than the processing time to generate an ACK response on the client, we also conducted an experiment with the setup where both the victim and attacker machine connect to a Huawei router via ethernet. As depicted in Fig. 5, the timing side channel is no longer visible and amplifiable (note the heavily overlapped boxes), because of two reasons: (1) Now that downlink and uplink can transmit at the same time, there is simply no contention regardless of how many packets are transmitted. (2) Packets belonging to different sockets can be processed simultaneously on different CPU cores (by OS design), allowing the post-probe query to be processed in parallel to probes. Even if the probes trigger ACKs, they still consume resources (CPU, memory) that are mostly isolated from the post-probe query. The experiment demonstrates that contention caused by half-duplex is the root cause of the timing channel.

4 Attack Overview

In this section, we show how such an inherent side channel can be leveraged in our off-path TCP attack.



(a) RTT measurement of macOS



(b) Gap measurement of macOS

Figure 5: Measurement of wired connections in a local setup

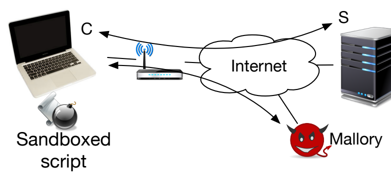


Figure 6: Specific threat model targeting a wireless client

• **Threat model**. Obviously, since the side channel is inherent in Wi-Fi, the threat model requires either the client or server connected through Wi-Fi. As it stands, we do not consider servers here as most of them do not use Wi-Fi (see §7 for a special case of IoT devices). This paper therefore focuses on the threat model as depicted in Fig. 6 where a user is lured into visiting a malicious website first. Subsequently, a sandboxed malicious script (by convention [25, 27], we call them puppets) initiates a connection to the attacker (who is not necessarily close to the victim) to circumvent the reachability problem caused by NAT or firewall commonly found on wirelessly-connected clients. The off-path attacker can then take measurements of RTTs from outside and conduct the side channel attack. Based on this threat model, we consider a number of related attack goals:

(1) inferring the presence of a connection from the client to a server (connection inference);

(2) counting the number of bytes exchanged on the connection, or forcefully terminating the connection (sequence/ACK number inference);

(3) injecting malicious payload into a connection (ACK number inference).

For attack goal (1) and (2), the attack can be targeted at any connection from the client, not necessarily just those that are puppet-initiated. For (3), although not strictly required, it is generally assumed that a puppet-initiated connection is targeted (as shown in prior side channel attacks [25, 27]) because the attacker controls the timing of the connection/request, greatly simplifying the attack.

Overall procedure. Attack goal (1) and (2) are generally straightforward. For (3), in this paper, without loss of generality, we focus on the "web cache poisoning" attack (which is the most powerful among a few other web attacks described in [25, 27]). Assuming a puppet-initiated connection is targeted, the attack can choose to poison any unencrypted target website at any time. It relies on the basic design principle that browsers reuse TCP connections for requests sent to the same server IP address. This means that the puppet in the malicious website can create a single persistent connection to a target domain by repeatedly including HTML elements (*e.g.,* images). The off-path attacker can then conduct the side channel attack to infer the port number and sequence numbers used in the target connection. Afterwards, the puppet can embed a target web object in the page, *e.g.,*

```
<iframe src = "www.bank.com/index.html" />
```

This triggers an HTTP request over the same old TCP connection; the off-path attacker can now simply inject a fake HTTP response that will be cached for arbitrarily long, because the HTTP response header can ask the browser not to re-check the freshness of the object, leading to a persistent cache poisoning[5]. If an attacker caches a commonly used malicious third-party javascript (*e.g.,* jQuery), it can impact a large number of websites.

In the remainder of this section we describe the three different attacks that progressively build on top of each other, and detail strategies for all three major operating systems.

• **Leveraging the TCP packet validation logic**. As mentioned in §2.2, the latest RFC standards specify the packet validation behavior, which consists of *connection (four-tuple) identification*, *sequence number check* and *ACK number check*. In each check, depending on the validity of the incoming packet, a response will be generated, or not. **This is exactly what the Wi-Fi timing channel allows an off-path attacker to observe — whether spoofed packets have triggered responses or not.** Similar to the Windows global IPID side channel that provides the same feedback (but is now eliminated), prior attacks also take advantage of the TCP packet validation logic [1, 25]. However, there are two issues to consider. First, clients connected through Wi-Fi are almost always behind NAT and/or firewall (the wireless router itself often acts as NAT). Therefore, the packet validation logic may change slightly. Second, it is unclear whether the operating systems will follow the standard faithfully.

For the first problem, NAT and firewall primarily change the behavior of connection identification. If an incoming packet does not match any ongoing connec-

[5]HTTP response header can specify a "max-age", indicating that the response is to be considered stale after X seconds where X can be as large as 2^{31} or 68 years (see RFC7234)

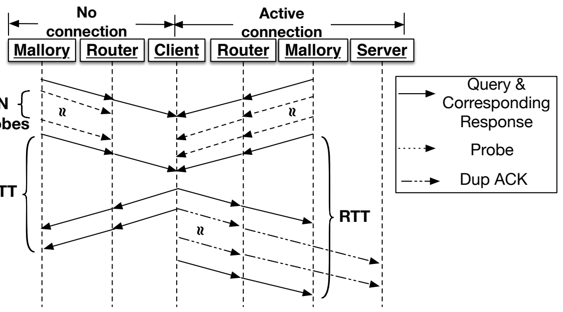
tion, NAT and firewall will simply drop the packet, preventing the client from even observing it; if an incoming packet matches an ongoing connection, the packet is let through and handled as usual. This actually simplifies the connection inference, as the attacker can simply choose to send spoofed packets that always trigger responses (*e.g.,* incoming SYN packets); if there is no response, it must be the case that no connection exists and packet is dropped by a NAT.

For the second problem of real operating system implementations, we survey the latest Linux, macOS, and Windows in terms of their packet validation logic. Our methodology is to inspect the kernel source code of Linux and macOS [11] as they are readily available. We then experimentally verify our understanding of them. Finally, we apply the same test program to measure the behavior of Windows. We summarize our findings in Table 2.

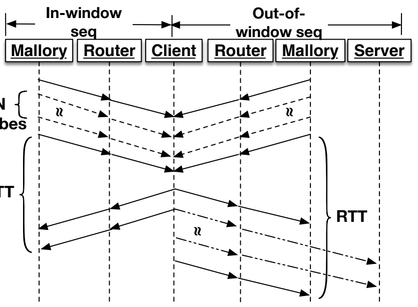
The result is, for the most part, consistent with the standard (except Windows which we talk about later). Linux is the one that most closely follows the standard (also observed previously in [18]). It has implemented the challenge ACKs and the rate limit as suggested by RFC 5961. MacOS is similar to Linux except that it does not implement rate limit and is in general weaker in its validation logic. For instance, even if an incoming packet has no flag bit set, it still checks the sequence number of the packet instead of dropping it without any processing. Based on the concrete testing results, we conclude that all three operating systems have packet validation logic that can be exploited via the Wi-Fi timing channel. We describe how to leverage their specifics to conduct the attack:

Connection (Port Number) Inference. This attack breaches the user privacy because knowing the websites a user visits often reveals a user's medical condition and sexual orientation [36]. As with previous off-path TCP exploits [25, 18], the first step is to infer whether an ongoing connection with a particular target (server IP and server port are given) exists. We know that NAT drops incoming packets that do not match any ongoing connections. All we need to make sure is that all operating systems do generate outgoing ACKs otherwise. Indeed, from the table, an incoming ACK matching an ongoing connection with an out-of-window sequence number is guaranteed to trigger an ACK on all operating systems (row no. 1, 10, and 17)). Fig. 7a depicts the sequence of packets that an off-path attacker can send to differentiate between the cases of (i) the presence or (ii) the absence of an ongoing connection. In both cases, the attacker sends the same sequence of packets, leveraging the probing strategy described in §3 to measure the delay effects.

Sequence Number Inference Assuming the attacker has already identified the four-tuple connection, the off-



(a) Connection (four-tuple) test



(b) Sequence number test

Figure 7: Infer port and sequence number by exploiting the timing side channel. Note that these diagrams are simplified for clearness. In reality, packets belonging to different sockets can be processed simultaneously, and uplink and downlink should have equal access to the wireless channel rather than uplink waiting for downlink.

path attacker now needs to guess a valid sequence number. By continuously tracking how the sequence number progresses, the attacker can effectively count the number of bytes received by the client (and the reverse direction can be monitored similarly through ACK number inference). We label the sequence number inference opportunities in Table 2 by combining two rows with different outcomes (w/ or w/o responses) when the same sequence of packets are processed. For Linux, if 10 incoming ACK packets with just one-byte payload are received, depending on their sequence numbers, 10 responses are triggered (out-of-window), or at most 1 (in-window) due to rate limiting (row no. 1, 2, and 3). For macOS, if an incoming packet with no flags is received, a response is triggered for the out-of-window case; otherwise no response is triggered (row no. 10 and 11). Interestingly, if the ACK flag is on, macOS only generates ACKs half of the time (row no. 12 and 13). Windows is similar and requires only the regular ACK packets (row no. 17 and 18); SYN packets can do the trick as well (row no. 17 and 19). Fig. 7b demonstrates the sequence of packets that an attacker can send to distinguish between the cases of in-window and out-of-window sequence number.

ACK Number Inference Finally, knowing the four

No.	OS	FLAG	SEQ	ACK	PAYLOAD	#Responses	Operation
1	Linux	$ACK \mid SYN \mid RST$	Out-of-window	Any	1	10	SEQ inference
2	Linux	$ACK \mid SYN \mid RST^+$	In-window	< SND.UNA - MAX.SND.WND	Any	1^*	
3	Linux	$ACK \mid SYN \mid RST^+$	In-window	> SND.MAX$^\alpha$	Any	0	
4	Linux	$ACK \mid SYN \mid RST$	Out-of-window	Any	0	1^*	
5	Linux	ACK	In-window	In-window	1	10^-	
6	Linux	ACK	In-window	In-window	0	0	
7	Linux	ACK	RCV.NXT$^\alpha - 1$	< SND.UNA - MAX.SND.WND	1	1^*	ACK inference
8	Linux	ACK	RCV.NXT-1	> SND.MAX	1	0	
9	Linux	ACK	RCV.NXT-1	In-window	1	10	
10	MacOS	$None \mid ACK$	Out-of-window	Any	Any	10	SEQ inference
11	MacOS	$None$	In-window	Out-of-window	Any	0	
12	MacOS	ACK	In-window	< SND.UNA	0	0	ACK inference
13	MacOS	ACK	In-window	> SND.MAX	Any	10	
14	MacOS	RST	!= RCV.NXT	Any	Any	0	
15	MacOS	$SYN \mid FIN$	Any	Any	Any	10	
16	MacOS	ACK	In-window	< SND.UNA	1	10	
17	Windows	$ACK \mid FIN \mid SYN$	Out-of-window	Any	Any	10	SEQ inference
18	Windows	$ACK \mid FIN$	In-window	Out-of-window	Any	0	
19	Windows	$SYN \mid RST$	In-window	Out-of-window	Any	1	
20	Windows	RST	Out-of-window	Out-of-window	Any	0	
21	Windows	ACK	RCV.NXT-1	Any	1	10	
22	Windows	ACK	In-window	SND.NXT†	1	10^-	Idle connection
23	Windows	ACK	In-window	!= SND.NXT†	Any	0	
24	Windows	ACK	In-window	In-window	1	10^-	Busy connection
25	Windows	ACK	In-window	In-window	0	0	

*: Due to rate limit in Linux, we can get at most 1 response per half a second.
$^+$: The sequence number should be in window but not equal to the next expected number, otherwise the connection is reset.
$^-$: Although the client replies to such packets, it would also cause de-synchronization leading to the victim connection to be closed during the keep-alive procedure, if the SACK option enables.
†: Typically, ACK number window refers to the range `[SND.UNA-MAX.SND.WND, SND.NXT]`, but Windows deploys a more stringent check if the connection is idle, requiring a valid ACK to equal `SND.NXT`.
$^\alpha$: `RCV.NXT` = next sequence number expected on an incoming segments, and is the left or lower edge of the receive window; `SND.MAX` = latest unacknowledged sequence number

Table 2: Behaviors on different OSes when processing 10 identical packets

tuples and the expected sequence number, the attacker now needs to learn the correct ACK number to successfully inject malicious payload. According to the standard behavior earlier in §2.2, an attacker can infer whether a guessed ACK number is in-window or not by sending a pure ACK (no payload) assuming its sequence number is already in-window. If its ACK number is out-of-window, a response is triggered and otherwise no response. Surprisingly, from our analysis and experiments, we conclude that no operating system is fully compliant with the standard. Their own variants have often allowed simpler strategies to conduct the ACK number inference.

Linux. As shown in Table 2, instead of always triggering an ACK packet for out-of-window ACK numbers, when the ACK number is too old (smaller than `SND.UNA - MAX.SND.WND`), Linux responds with an ACK (with rate limit); when the ACK number is too new (larger than `SND.NXT`), Linux incorrectly drops the packet without any reply (row no. 2 and 3). Had there been no rate limit, an attacker can infer the correct ACK number via binary search. With rate limit, however, one response versus zero cannot create significant enough of a timing channel. In addition, if a packet with in-window ACK number has no payload, Linux also ignores the packet with no response (row no. 6), which leaves no opportunity to differentiate the in-window and out-of-window cases (result similar to row no. 2 and 3). However, it does correctly handle packets with payload; a response is triggered only when the ACK number is in window (row no. 5). The issue is that when an ACK number is inferred, the client buffers the payload in its receive window, which is undesirable for two reasons: (1) it may cause future server's responses to be corrupted; (2) if selective ACK (SACK) is enabled, the client selectively acknowledges the data which has not actually been sent by the server, causing the server to ignore future packets from the client, effectively de-synchronizing the client and server. Interestingly, Linux has a special edge case that allows us to infer ACK number without the hassle. According to the specification, if the sequence number of an incoming packet is equal to `RCV.NXT-1` (indicating a keep-alive message), it should trigger an ACK. Interestingly, the specification has an ambiguity. RFC 1122 [43] specifies only the valid sequence number of a keep-alive packet, but not the ACK number. Based on the source code, Linux does not actually handle keep-alive explicitly. Instead, it simply treats such a packet (with one-byte payload and `end_seq = RCV.NXT`) as in-window, and decides how to respond based on its standard ACK number check. Therefore, in-window ACK numbers with the specific sequence number (*i.e.,* `RCV.NXT-1`) still trigger responses (row no. 9) and yet no actual data are buffered at the client, while out-of-window ACK numbers can trigger at most one reply (line 7 and 8 in Table 2).

MacOS. Based on the source code and experiments, macOS explicitly handles keep-alive packets and always responds with an ACK regardless of the ACK number so the strategy against Linux does not apply to macOS. On the other hand, macOS has its own implementation of ACK number validation which correctly responds to packets with ACK numbers that are too new (row no. 13). Interestingly, it chooses not to reply to packets with ACK numbers that are too old when there is no payload (row no. 12). The implementation of macOS is likely to be misled by the old statement in RFC 793 that states packets with ACK numbers smaller than `SND.UNA` can be ignored, which is reinterpreted in RFC 5961 (quote): "All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back", where the "above condition" is the acceptable window of [*SND.UNA - MAX.SND.WND, SND.NXT*]. In summary, this non-compliant behavior of macOS allows an attacker to infer if a guessed ACK number is too large or too small, resulting in a binary search.

Windows. Windows is for the most part similar to Linux on the ACK number validation, except that it has made one subtle customization. Initially, we were surprised to find that an incoming data packet with an in-window sequence number is always silently dropped unless the ACK number is equal to `SND.UNA` or `SND.NXT` (the connection is idle during our initial experiments so the two numbers are equal). This implementation is not conformant to the standard at all. Recall the standard says that the acceptable ACK number range is defined to be `[SND.UNA - MAX.SND.WND, SND.NXT]` in RFC 5961 and both Linux and macOS follow the standard. In fact, we thought the implementation was completely wrong because it may drop legitimate data packets in cases like out-of-order packet arrivals. We then realize that it appears to be a reasonable decision, especially when the connection is idle. Indeed, if there are no outstanding data to send, it is safe to require the peer to acknowledge one and only one ACK number. However, as soon as there are outstanding data, it should enlarge the acceptable ACK number range. We experimentally confirmed that this is exactly what Windows does. In summary, the behavior of Windows still allows ACK number inference when it has outstanding data during the inference. This makes our attack in §5.3 more complicated but still possible by taking advantage of the behaviors in row no. 18 and 24.

5 Implementation

Now that we know the Wi-Fi timing side channel applies universally to all operating systems, we want to test them in real-world attack scenarios.

5.1 Connection (Four-tuple) Inference

General method. The general probing strategy is already discussed in §4. In our implementation, we conservatively test one port every round with 30 repeated packets, followed by a post-probe query to measure RTT. When a guessed port number is correct, we see a substantial increase in the measured RTT. If the goal is to infer the presence of any arbitrary connection initiated by the client, then a bruteforce strategy is all that can be done. However, if the attacker is attempting to conduct web cache poisoning attack later on, it is possible to target a connection initiated by the puppet itself [25], which opens up an additional optimization below taking advantage of the ephemeral port selection algorithm employed by different OSes.

Windows and macOS. They use a global and sequential port allocation strategy to select ephemeral port number for their TCP connections. This means that the attacker can deduce the next port number to be used once it observes the initial connection to the malicious web server. This eliminates the need of port number inference completely.

Linux. It uses the Simple Hash-Based Port Selection (SHPS) [27] where there is an independent local port number space for each remote IP and port pair. This means that the local port number observed from the connection to the malicious web server can no longer predict the next local port number for the connection to a different target server which the attacker does not control. To avoid bruteforcing all possible port numbers, we develop an optimized strategy based on the observation that local port numbers allocated for the same remote server and port pair are sequential; therefore, the puppet can potentially create *n* connections to the target server and only needs to test the port number every *n* increments.

At this point, we can conduct the side channel attack on the connection of which we guessed the correct port number. Also, by carefully scheduling those *n* requests we are guaranteed that a future request will use the connection with the smallest port number as opposed to the others closed later.

NAT. In our experience with Wi-Fi routers, we find that they typically are port preserving. So we do not have to worry about the external port being translated and become unpredictable. This is based on our testing of 4 different home routers and the university network. However, if non-port-preserving NAT are indeed used for Wi-Fi, then the attacker can either fall back to the brute-force approach, or apply the optimized solutions proposed in [27] (which has its own benefits and caveats).

Multiple IP addresses from a domain. This essentially requires the attacker to double or triple the effort of port number inference. For Windows and macOS, this is not much more effort. However, for Linux it does require some more time.

5.2 Sequence Number Inference

General method As shown in table 2, we're able to distinguish in-window sequence number from out-of-window one by leveraging timing side channel to tell whether there are corresponding responses. As soon as we get an in-window sequence number, we further narrow down the sequence number space to a single value `RCV.NXT` by conducting a binary search. This is similar to prior work [39, 18]. Similar to connection inference, if the attacker is attempting to conduct web cache poisoning attack against a connection initiated by the puppet itself [25], an additional optimization is possible.

Optimization: Increase window size. To substantially decrease the number of iterations of guesses, one straightforward approach is to drastically enlarge the client's receive window. To this end, the puppet can request excessive amounts of large objects. Upon the receipt of enough full segments, the receiver would significantly increase its receive window size according to TCP flow control. In our experiments, we found that the window size could be typically scaled up to around $x = 500,000$, in striking contrast to the original size (*e.g.,* 65,535). It's worth noting that the window size can never be shrunk once it is enlarged, according to RFC793 [4]. Similarly, by uploading data, the ACK window (*i.e.,* the peer's sequence window) can be extended, though it's usually much smaller than the maximum sequence window size that we can achieve.

5.3 TCP Hijacking

We assume in this section that the attacker is attempting to poison the web cache through hijacking the puppet-initiated connection, which enables the attack to be more efficient. In principle, the attacker can hijack any connection initiated by the client; it is simply more difficult to control the timing and predict what fake response to inject.

Since all three systems do not comply with the specifications in terms of ACK validation, we have to cope with each variant differently:

MacOS incorrectly interpreted the standard, allowing us to perform a binary search (see §4). Once the expected ACK number is inferred, we perform a desynchronization attack [18] to avoid a race condition where the response is sent back by the server first. Then, as soon as the puppet requests for the target object, it informs the attacker to send a spoofed response, which is accepted.

Linux It's feasible to exploit the timing side channel to infer ACK number, though the valid ACK window size is much smaller compared to the receive window size, resulting in longer inference time. One alternative approach is to conduct blind data injection without knowing the exact value of the expected ACK number. Our observation is that by now we've known the exact sequence number and the size of any object that the client retrieves (see §5.4), we are capable of predicting a future expected sequence number after N objects are retrieved. The attack then goes as follows: (1) *Desynchronization.* The puppet keeps requesting an object, while the attacker sends a number of spoofed packets with the same in-window sequence number that matches a future `RCV.NXT`, bruteforcing the ACK numbers (which is much faster than side channel attack as there is no wait for any feedback). When the last valid response comes back advancing the sequence number to the value we anticipate, suddenly the attacker-injected response will be appended and forwarded together to the browser (and yet the browser always has only one pending request). Chrome will close the connection, stopping the attack; in contrast, Firefox will simply accept the first response, ignoring the second one, resulting in desynchronization between the client and server (*i.e.,* the client believes it has received more data than the server has actually sent). (2) *Blind data injection.* Now the puppet will switch the target web object to the one we want to attack (*i.e.,* homepage of a banking website). The attacker now has enough time to send a valid response. Since the attacker knows the next expected sequence number, it only needs to again bruteforce all possible ACK numbers. Note that this strategy requires two rounds of bruteforcing of every possible ACK number, and each round takes only a couple of seconds as there is no waiting. In contrast, a side channel attack would take much longer (minutes) because every guessed ACK number takes 30 packets, and the timing measurement needs to be collected before the next guess can be made.

Windows As we mentioned in §4, to prevent the valid ACK window size being one-byte only, the client has to keep sending requests to make sure there are always outstanding data, which complicates our attacks because the attacker has to synchronize the next expected sequence number. Besides, a large amount of traffic also introduces noise to the timing side channel. Moreover, the blind data injection we utilize on Linux does not apply to the same version of Firefox on Windows according to our tests; it immediately drops the connection when it receives two responses for only one pending request. We therefore devise a new strategy that exploits the TCP behavior of handling overlapping data and the browser behavior of handling corrupted HTTP responses. If a new incoming TCP data packet has an overlapping sequence number range with some previously buffered data, we find that old data are always preferred in Windows whereas new data are preferred in Linux (this observation is consistent with prior studies [38]). In other words, attacker-injected data buffered on a Windows host can corrupt a real HTTP response from the server. Given the insight, we present the exploit in two steps which are illustrated in Fig. 8: (1) *Inject.* The puppet continuously requests scripts from the server, while the attacker sends $\frac{2^{32}}{|wnd|}$ spoofed packets with a deliberate in-window sequence number that matches a future `RCV.NXT` plus a small `offset`, where `wnd` denotes the size of the acceptable ACK window. The i^{th} packet has a guessed ACK number $\texttt{i}\cdot|\texttt{wnd}|$, and contains payload as:

`websocket.send(`$i \cdot |wnd|$`)`

Hence, exactly one of these packets contains a valid ACK number and will be buffered. We intentionally construct the overlap such that the HTTP header of the real response will become corrupted. Interestingly, the browser would still try to interpret the corrupted response where it simply ignores corrupted header and accepts the next header (injected by the attacker) along with the remaining attack payload. When the browser executes the injected script, it will send the guessed ACK number via *websocket*, providing a valid in-window ACK number. (2) *Exploit.* Since the client has accepted the extra spoofed payload, advancing its expected sequence number, the client and server are effectively already desynchronized. The attacker can now simply send a spoofed response (knowing both the expected sequence number and a valid ACK number). Alternatively, if we only want to perform a one-time injection, simply replacing the payload in the first step with a malicious script is sufficient. Note that the attack strategy against Windows is even more efficient than the one for Linux because only one round of bruteforcing of ACK numbers is needed.

Furthermore, there exists an even more general alternative strategy to the *inject* step against Windows that does not depend on browser behaviors at all. Specifically, as the first few bytes of HTTP responses are predictable (*i.e.,* HTTP), instead of corrupting the real response, one overwrite the header and the body to form a legitimate but malicious response. A browser in this case will be completely oblivious to the existence of injection. This demonstrates that once sequence number is leaked, there exist various ways to inject data into browsers efficiently, without conducting the much slower timing-channel-based ACK number inference.

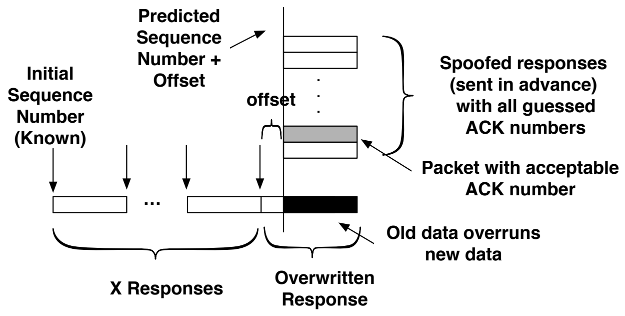

Predicted Sequence Number + Offset
Initial Sequence Number (Known)
offset
Spoofed responses (sent in advance) with all guessed ACK numbers
Packet with acceptable ACK number
Old data overruns new data
X Responses
Overwritten Response

Figure 8: HTTP injection by exploit tolerant browsers

5.4 Other Challenges

Dealing with noise by setting a proper threshold. Latency may vary under different network conditions, thus it is a bad idea to manually set a threshold to differentiate a quiet probe round (without triggering ACKs) versus a responsive probe round (triggering ACKs). In our implementation, we devise a simple procedure that automatically sets a threshold based on a preliminary round of test probes prior to launching the actual attack. Since we have full control of the connection established between the client and attacker, we can send non-spoofed packets to measure RTTs for quiet probe rounds and responsive probe rounds. After we collect data for both cases, we sort the data and set a threshold such that 80% of the responsive round measurements will be above the threshold. The threshold is a trade-off between efficiency and effectiveness. Since most of the rounds we're testing do trigger ACKs (so larger RTTs should be observed), setting a lower threshold will ensure that we correctly classify such cases to avoid double checking the results. However, a threshold too low runs the risk of misclassifying a quiet round into a responsive round, missing the correct guess altogether; this forces us to repeat the whole search process. Finally, we ignore cases where abnormally large RTT values are perceived (*e.g.,* from network noise), if it is out of the range of three times the standard deviations.

Dealing with noise by error recovery. Even with a properly selected threshold, we may still end up with incorrect inferences. We cope with this challenge by embedding extensive error recovery mechanisms into the inference process, such as relative comparisons and double checking. We assume that network jitter/noise does not vary much during the short time interval of testing a few rounds (a common assumption in the networking literature [26]). In the case of sequence number inference as an example, once a sequence number is believed in-window, we further try to narrow down the space to a single value `RCV.NXT` by binary search. During the procedure, we also simultaneously measure additional RTTs (using out-of-window sequence numbers) and their relative difference to the RTTs (using in-window sequence numbers). If the comparison results are not consistent, we can deduce that we made a mistake earlier and will rollback. As for false negatives where an in-window number is believed out-of-window, there is no simple way to detect them but repeating the whole process till the program finally finds out the correct number or fails due to timeout.

Pipeline In order to significantly reduce the time the attack costs, instead of simply probing a single SEQ/ACK number at a time, we also use a pipelined process aiming at maximizing network utilization by scheduling probing packets for multiple targets at appropriate times. However, due to the fact that packet loss may happen from time to time, we suspend the procedure every few tests to wait until we get all the results or restart in a fixed time interval.

Moving SEQ/ACK window and unknown window size Since the victim connection is controlled by the puppet, it's idle most of the time unless the puppet triggers a request. Therefore, the attacker can be fully aware of when the SEQ/ACK window is moving. Besides, regarding the unknown window size to an off-path attacker, our strategy is to initially choose a relatively large window size θ and then half it afterwards. So $\frac{\theta}{2^{i-1}}$ will be the window size we use in i^{th} iteration. Note that we do not test an exact number that has been tested in previous iterations to avoid redundancy.

Detecting the size of any object So far, we have assumed that we are aware of the size of the response sent from the server to the client so that we can predict where to insert the forged payload. This is in fact not difficult to achieve because once we know the next expected sequence number, we can ask the puppet to request the object and then infer the new expected sequence number; the increment is exactly the size of the response.

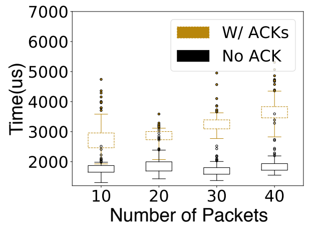
6 Evaluations

Experimental Setup Our network topology is the same as in §3. The attack machine is an Ubuntu 14.04 host in our lab. We tested those attacks against three different operating systems, including macOS 10.13, Linux 4.14.0, and Windows 10 Pro version 1709 (they are also the same versions used to study the behaviors of TCP stacks shown in Table 2). We empirically evaluated different techniques with Chrome 64.0 and Firefox 58.0.1. When we evaluated the attack for the 'Remote Attacker' scenario, the experiments were performed in the same house as mentioned in §3 with at most 4 users, and RTTs between the client and attacker were over 20ms. The bandwidth we utilized in the remote and local experiments are approximately 1000pkts/s and 4000pkts/s respectively (or $\sim$ 0.5Mbps and $\sim$ 2Mbps), which we believe are moderate and comparable to prior work [18].

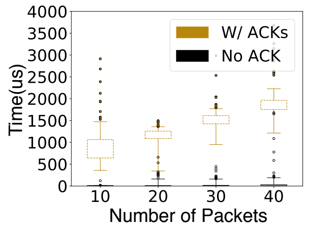
Noise Resilience of Timing Side Channel Using the same experimental setups as in §3, we introduce two different types of noise to evaluate the resilience of the Wi-Fi timing side channel. First, for the 5GHz network, the malicious webpage contains a Youtube video, which would be automatically played while timing measurements are performed. Second, as 2.4GHz networks tend to influence each other, we have also conducted the measurement in the lab where there were 43 accessible Wi-Fi in total, 22 of which were 2.4GHz network and 6 used the same channel that our test router used; there were also more than 10 students actively using the network. As depicted in Fig. 9, the timing channel does encounter additional noise but RTTs are still visibly different.

Evaluation of Local Attacks Our victim webpage can be any page transmitted over HTTP. Although Google Chrome marks some non-HTTPS sites as "not secure", we still found some sensitive bank websites (e.g., www.icbc.com.cn) that haven't deployed HTTPS on all of its pages, rendering them vulnerable to our attack. Typically, while allowing seemingly non-sensitive pages (*e.g.,* homepage) transmitted over HTTP, websites would restrict sensitive pages (*e.g.,* login pages) to HTTPS, presumably because of their concern of both performance and security. Consequently, an adversary who successfully hijacked the homepage could have injected a phishing login component already. Furthermore, even if HTTPS is deployed on all pages, attackers could still mount the attack, as long as HTTP Strict Transport Security (HSTS) is absent; this is because the initial request to the website will still use HTTP and it is the server that subsequently redirects the browser to its HTTPS site. One representative example is the news website 'www.cnn.com' which uses HTTPS but unfortunately not HSTS. When a user tries to access its homepage, an initial request is submitted via HTTP for which an adversary can inject a fake reply, preventing the legitimate response from redirecting to HTTPS.

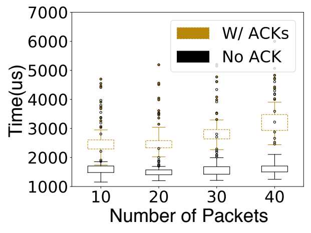
Next we report the attack success rate and the average time to succeed. Depending on our target OS, we leverage different strategies described in §5.3 along with the timing side channel, and present the results in table 3. As



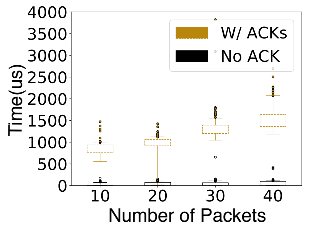
(a) RTT measurement for 5GHz of a Huawei router



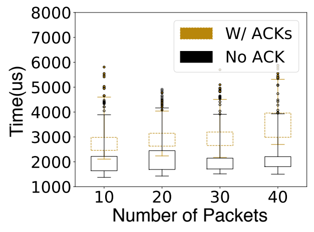
(b) Gap measurement for 5GHz of a Huawei router



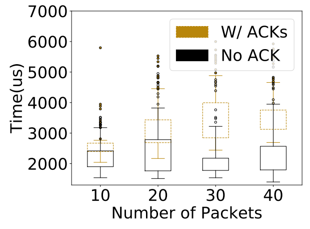
(c) RTT measurement for 2.4GHz of a Linksys router



(d) Gap measurement for 2.4GHz of a Linksys router



(e) RTT measurement for 2.4GHz of a Linksys router



(f) RTT measurement for 2.4GHz of a Huawei router

The upper four figures are measured while playing a youtube video, and the lower two are under wireless interference.

Figure 9: Measurement of MacOS with additional interference in a local setup

it illustrates, three most popular operating systems are all vulnerable to the attack if they connect to the Internet via a wireless router. With respect to the random-increment port selection strategy utilized by Linux, attacks against Linux take around 1 more minute on average to infer the port number. Some optimizations discussed in [28] could be applied to significantly reduce the time of port inference. We demo some of these attacks on our project website [3].

Evaluation of Remote Attacks To further demonstrate the practicality of the attack, we report results under a "remote attacker" scenario described earlier (the RTT between the outside attacker and victim is over 20ms). First, we conducted the same measurements as in §3 to ascertain the timing side channel is not eliminated due to network jitter. Fig. 10 presents the results of measurements at two different locations in the same city. Though there is more overlap between the two boxes compared to the local setup, the signal is clearly present. They can be

OS	Browser	Success Rate	Avg time cost(s)	Technique(s)
Linux	Chrome Firefox	10/10	188.80	Timing Side Channel
MacOS	Chrome Firefox	10/10	48.91	Timing Side Channel
Windows	Chrome Firefox	10/10	43.42	Timing Side Channel & Direct Page Read
Linux	Firefox	9/10	103.53	Timing Side Channel & Blind Data Injection

Table 3: Summary of attacks in a local setup

distinguished with modest false negatives (*i.e.,* missing in-window numbers) and false positives (*i.e.,* misclassifying an out-of-window number), both of which could be further reduced by increasing the number of probing packets per test and more rounds of double-checking.

Next, to complete a realistic attack, we implemented the web cache poisoning attack against MacOS with aforementioned optimizations. Table 4 enumerates the 10 test results along with the number of false negatives produced during each experiment. It's worth noting that we never encountered the case where the attack procedure mistakenly reports a success due to error recovery and double checking. Besides longer RTTs compared to that of a local setup, the significant time cost is attributed to the following factors: (1) Regarding sequence number inference upon MacOS, though an attacker can send probing packets without any flags as shown in table 2, we found those packets are likely to be discarded in a real-world network environment. To cope with it, we send probing packets with ACK bit set and guess two acknowledgement numbers (*e.g.,* 0 and 2G) for every guessed sequence number, effectively doubling the number of packets sent. (2) Traversing through the entire sequence number space already takes roughly 5 minutes, if we happen to miss the correct sequence number (false negative) even once, we need to repeat the search process[6]. Nevertheless, since there is only one 'critical' test (*i.e.,* one correct sequence number) in each iteration, the chance of missing it is quite small. We can further reduce this chance by tuning the RTT threshold parameter, which we leave as a future exercise. (3) The time cost varies substantially due to the large search space of the sequence number. Specifically, while the attack attempts to explore every possible sequence number from 0 to 2^{32} per window, the procedure stops earlier if a correct sequence number happens to be small.

[6]In practice, we consider attacks over 10 minutes to be impractical, thus attacks halt after two iterations of failure

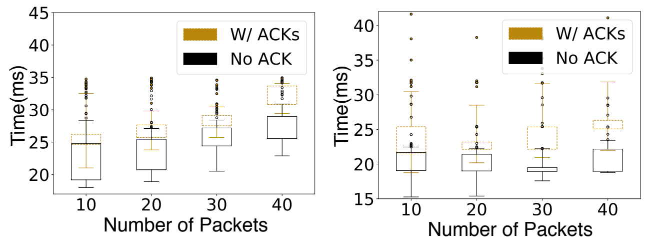


Figure 10: RTT measurement of macOS using 5GHz network of a Xiaomi router at two different locations with RTTs over 20ms

Result	Time cost (s)	#FN	Result	Time cost (s)	#FN
success	25.66	0	success	23.08	0
success	286.31	0	success	580.32	1
success	549.15	1	success	195.03	0
success	335.10	0	success	227.43	0
failure	634.03	2	success	185.74	0

FN: False Negative (*i.e.,* Missing correct SEQ number)

Table 4: 10 trials of remote attacks against macOS

7 Discussion

As discussed in §3, the timing side channel results from the half-duplex nature of wireless networks. It is further magnified due to the collision and backoff inherent in wireless protocols. As we demonstrated, a full-duplex system does not exhibit any timing channel (see §3) as no collision will occur when uplink and downlink traffic happen at the same time. Finally, as confirmed in our test routers, modern wireless routers all support CSMA/CA and RTS/CTS as it is part of the 802.11 standards [31], and the principle is unlikely to change any time soon.

Although we only discuss the threat model where connections originated from a victim client are targeted, the attack actually also applies to connections originated from *other clients connected through the same wireless router*. This is because all these clients (*e.g.,* behind the same NAT) share the same collision domain and therefore suffer from the same timing channel. Responses triggered on *any client* by probing packets will effectively delay the post-probe query. In this case, the victim connection (opened through puppet) simply opens up opportunities for an off-path attacker to measure collision. In addition, we can expand the threat model to consider servers that are wirelessly connected, *e.g.,* IoT devices. It has been shown that millions of IoT devices are reachable through public IP addresses and open ports [14]. In such cases, a completely off-path attack can be launched against a connection on such IoT devices, *e.g.,* counting bytes exchanged on the connection, terminating its connection with another host, injecting malicious command on an ongoing telnet connection (similar to the capability described in [18]).

8 Defenses

After we discovered the time side channel issue, we have disclosed it to the working group in February 2018. They have quickly acknowledged this weakness and became highly engaged in discussion of the matter. However, due to the expected challenges in changing the half-duplex design, we are yet to see an appropriate solution at the 802.11 level. Therefore, the immediate mitigations are expected to be at higher levels. We've also disclosed it to vendors of the routers that we tested, among whom only one replied and actively discussed it with us. Though the company employees acknowledged this weakness, they decided to submit this security issue to Wi-Fi Alliance, hoping that this would be fixed in the protocol standard. In the reminder of this section, mitigations/patches at different layers are offered and thoroughly discussed.

Defenses in Wi-Fi technology. Unlike the previous software-induced side channels, the timing channel introduced by Wi-Fi is inherently difficult to eliminate or mitigate (just as the recent meltdown and spectre vulnerability in CPUs [35, 34]). One straightforward defense would be to make the Wi-Fi channel full-duplex. For instance, with frequency-division duplexing, different frequency sub-bands can be used for uplink and downlink traffic. However, this can potentially introduce low bandwidth utilization as separate dedicated sub-bands have to be pre-allocated (and real-world Internet traffic volume is not symmetric). Even though IEEE 802.11ax working group has been considering the possibility of supporting in-band full-duplex communication [2], research still needs to be done to make sure the real-world challenges such as backward compatibility are carefully considered and addressed [12, 30]. At this point though, it is unclear when the technology will be widely deployed in practice, according to our conversation with the 802.11 working group.

Defenses in the TCP stacks. As described in §2.2, the packet validation logic of the latest TCP specification inherently treats valid and invalid incoming packets differently, in terms of whether a response should be generated. One solution is to revisit the specification and look for alternatives. A good hint is that all three modern operating systems implement the ACK number validation differently, yet they have co-existed without any major issues for a long time now. This leaves some flexibility in the ACK number validation logic. Ideally, no matter what ACK number an incoming packet has, it should either consistently respond or never respond. Assuming an incoming packet already has a valid sequence number, the only constraints we have here are:

(1) if it is a data packet and its ACK number is also in-window, a correct TCP receiver should always respond with an ACK (or delayed ACK); (2) when a pure ACK with sequence number in-window and ACK number in-window arrives, there should be no response (otherwise, an ACK war [6] may be triggered).

In the remaining cases: (3) a data packet with out-of-window ACK number; (4) a pure ACK with out-of-window ACK number, their responses appear to be flexible in practice — see row no. 2, 3, 13, 16, and 18 in Table 2) for the data packet case and row no. 2, 3, 12, 13, and 18 for the pure ACK case. Therefore, assuming an incoming packet already has an in-window sequence number, we can always force a response for a data packet, and no response for pure an ACK packet regardless of their ACK numbers. We plan to validate this idea by formally model checking the proposed changes together with legacy behaviors for the absence of ACK war.

With regards to sequence number validation, we hypothesize that the responses of receiving packets with valid and invalid sequence numbers can also be consistent. However its implications must be evaluated more carefully. A good strategy to consider is to rate limit ACK responses generated for various types of incoming packets. Even if inconsistent, this would allow the differences in responses (e.g., one response vs. zero) to be small enough and impossible to measure. The same rate limiting idea applies to connection identification, where packets are likely dropped by NAT or firewall if no connection is present and some response will be triggered if there is an active connection.

Defenses in Application layer Clearly, HSTS and HTTPS will help ward off most serious web attacks such as the web cache poisoning attack. Other TCP-level attacks (*e.g.,* inferring presence of connection [18], byte counting [20], connection reset [18]) could still be mounted by exploiting the vulnerability. HSTS and HTTPS can prevent only web cache poisoning attack (application-layer attacks) but not the TCP-level attacks.

Some versions of our attack also exploit features of browser implementations, and thus we believe some mitigations can be made in the browser (*i.e.,* make parsing of responses stricter) to complicate the ACK number inference step. The idea is that whenever the browser observes anything abnormal regarding the responses, *e.g.,* malformed or longer than expected, it should immediately drop the connection and restart. A small tradeoff is that this may break some backward compatibility with non-standard-conforming web servers. In terms of its effectiveness in stopping web cache poisoning attacks, it really only helps Linux as the attack now needs to fallback to a much slower version of the ACK number inference (likely tripling the time for a complete attack). Regarding Windows, although it also defeats our first strategy to infer ACK number by creating a malformed response, our alternative strategy is unaffected. MacOS's TCP stack implementation is so vulnerable that we will always favor the binary search on the ACK number to exploiting any browser-specific weakness. Finally, connection inference (privacy breach) and sequence number inference (byte counting and reset) attacks remain potent as they only rely on the TCP stack.

For the purpose of supporting further research to reproduce and mitigate the attack, we open sourced our implementation of the attack against different OSes, now publicly available at [5].

9 Related Work

We have described the most relevant work of various off-path TCP attacks in §2.3. In this section, we discuss a different set of related works.

Other off-path side channels. Besides the TCP sequence number, it has been shown that other types of information can be inferred by a blind off-path attacker. [24, 33, 23, 48, 13, 49, 26, 41, 37]. Most of these side channels do not in themselves allow serious attacks. However, much of the research translates to measurement tools that can be useful. For example, Knockel *et al.* [33] demonstrate the use of a new per-destination IPID side channel that can leak the number of packets sent between two arbitrary hosts on several major operating systems. Alexander *et al.* [13] can infer the RTT between two arbitrary hosts through the shared SYN backlog. Qian *et al.* [41] used global IPID side channel to measure directional port blocking. More recently, the Augur system [37] used the same IPID side channel to measure Internet censorship and connectivity disruption. The same side channel has also been used to count how many hosts are behind a NAT [15] and other applications [21].

Side channel discovery and defenses. Typically, when a specific type of vulnerability becomes known, there are many strategies to discover more concrete instances of them. For instance, static taint analysis has been applied to look for TCP packet counter side channels [19]. The problem is modeled as an information flow problem where the secret is the current sequence number, and the sink is the set of packet counters that report aggregated statistics to user space programs. If the secret sequence number can leak to the sink, then it is flagged as a potential side channel. There may be false positives (due to the over-approximation of the static analysis) but should not have false negatives by design. In the case of CPU cache side channels, symbolic execution has been applied to track the precise cache state over execution traces [47]. If the cache states can be different at any point in the trace with different secret inputs, the program is flagged to have leakage. Since the analysis is applied over concrete execution traces, the approach has no false positives (but may have false negatives). Unfortunately, the Wi-Fi side channel is not a software artifact and therefore cannot be discovered unless it is explicitly modeled and analyzed.

In terms of side channel defenses, there are various standard strategies such as perturbing the channel by injecting noise [7, 22, 45], and isolating the resources altogether [8, 32]. Unfortunately for Wi-Fi, these standard techniques would mean introducing wireless latency (which hurts performance), or making the channel full-duplex which we discussed earlier to be challenging as well.

10 Conclusions

To conclude, we have discovered a subtle yet fundamental side channel inherent in all generations of Wi-Fi or IEEE 802.11 technology because they are half-duplex. Furthermore, we show the timing channel is reliable and amplifiable, and also implement a real off-path TCP exploit in practice, allowing the attackers to inject data into a TCP connection and force the browser to cache malicious objects. Our study reveals that this novel attack affects all three most popular operating systems: macOS, Windows, and Linux. We provide a thorough analysis and evaluation of the proposed attack under different router/network/OS/browser combinations. Finally, we propose possible defenses against this attack.

Acknowledgement

We wish to thank Zakir Durumeric (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. Many thanks to Prof. Srikanth V. Krishnamurthy, Mart Molle, and Zhuo Lv who educated us on the fundamentals of wireless networks. We are also fortunate to work with the IEEE 802.11 working group, specifically Dorothy Stanley, Daniel Harkins, Jouni Malinen, to discuss the side channel and are grateful for their insights. This work was supported by the National Science Foundation under Grant No.1464410, 1652954, and 1646641.

References

[1] Blind TCP/IP Hijacking is Still Alive. `http://phrack.org/issues/64/13.html`.

[2] In-band Full Duplex Radios and System Performance. `https://mentor.ieee.org/802.11/dcn/15/11-15-0043-01-00ax-in-band-full-duplex-radios-and-system-performance.pdf`.

[3] Off-path tcp exploit: Demos of web cache poisoning attacks. `https://sites.google.com/view/tcp-off-path-exploits/`.

[4] RFC 793 - Transmission Control Protocol. `http://tools.ietf.org/html/rfc793`.

[5] TCP Exploit. `https://github.com/seclab-ucr/tcp_exploit`.

[6] [tcpm] mitigating TCP ACK loop ("ACK storm") DoS attacks. `https://www.ietf.org/mail-archive/web/tcpm/current/msg09450.html`.

[7] [patch net] linux tcp flaw lets 'anyone' hijack internet traffic, 2016.

[8] [patch net] tcp: enable per-socket rate limiting of all 'challenge acks', 2016.

[9] About the security content of macos high sierra 10.13, 2017.

[10] About the security content of macos high sierra 10.13.1, security update 2017-001 sierra, and security update 2017-004 el capitan, 2017.

[11] The darwin kernel. `https://github.com/apple/darwin-xnu`, 2017.

[12] AIJAZ, A., AND KULKARNI, P. Protocol design for enabling full-duplex operation in next-generation ieee 802.11 wlans. *IEEE Systems Journal PP*, 99 (2017), 1–12.

[13] ALEXANDER, G., AND CRANDALL, J. R. Off-Path Round Trip Time Measurement via TCP/IP Side Channels. In *INFOCOM* (2015).

[14] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., KUMAR, D., LEVER, C., MA, Z., MASON, J., MENSCHER, D., SEAMAN, C., SULLIVAN, N., THOMAS, K., AND ZHOU, Y. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1093–1110.

[15] BELLOVIN, S. M. A Technique for Counting Natted Hosts. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment* (2002).

[16] BENSAOU, B., WANG, Y., AND KO, C. C. Fair medium access in 802.11 based wireless ad-hoc networks. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing* (2000), IEEE Press, pp. 99–106.

[17] CALI, F., CONTI, M., AND GREGORI, E. Ieee 802.11 protocol: design and performance evaluation of an adaptive backoff mechanism. *IEEE journal on selected areas in communications 18*, 9 (2000), 1774–1786.

[18] CAO, Y., QIAN, Z., WANG, Z., DAO, T., KRISHNAMURTHY, S. V., AND MARVEL, L. M. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)* (2016).

[19] CHEN, Q. A., QIAN, Z., JIA, Y. J., SHAO, Y., AND MAO, Z. M. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *CCS* (2015).

[20] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy* (2010).

[21] CHEN, W., HUANG, Y., RIBEIRO, B. F., SUH, K., ZHANG, H., DE SOUZA E SILVA, E., KUROSE, J., AND TOWSLEY, D. Exploiting the ipid field to infer network path and end-system characteristics. In *Proceedings of the 6th International Conference on Passive and Active Network Measurement (PAM)* (2005).

[22] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[23] Ensafi, R., Knockel, J., Alexander, G., and Crandall, J. R. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *PAM* (2014).

[24] Ensafi, R., Park, J. C., Kapur, D., and Crandall, J. R. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks using Model Checking. In *USENIX Security* (2010).

[25] Gilad, Y., and Herzberg, A. Off-Path Attacking the Web. In *USENIX WOOT* (2012).

[26] Gilad, Y., and Herzberg, A. Spying in the Dark: TCP and Tor Traffic Analysis. In *PETS* (2012).

[27] Gilad, Y., and Herzberg, A. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW* (2013).

[28] Gilad, Y., and Herzberg, A. Off-path tcp injection attacks. *ACM Transactions on Information and System Security (TISSEC) 16*, 4 (2014), 13.

[29] Goguen, J. A., and Meseguer, J. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy* (1982).

[30] IEEE 802.11-13/1421r1. STR radios and STR media access.

[31] IEEE 802.11 Working Group of the LAN/MAN Standards Committee of the IEEE Computer Society. 802.11-2016 - IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

[32] Kim, T., Peinado, M., and Mainar-Ruiz, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012).

[33] Knockel, J., and Crandall, J. R. Counting Packets Sent Between Arbitrary Internet Hosts. In *FOCI* (2014).

[34] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (January 2018).

[35] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown. *ArXiv e-prints* (January 2018).

[36] Miller, B., Huang, L., Joseph, A. D., and Tygar, J. D. I know why you went to the clinic: Risks and realization of https traffic analysis. In *Privacy Enhancing Technologies* (2014).

[37] Pearce, P., Ensafi, R., Li, F., Feamster, N., and Paxson, V. Augur: Internet-wide detection of connectivity disruptions. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 427–443.

[38] Ptacek, T. H., and Newsham, T. N. Insertion, Envasion, and Denial of Service: Eluding Network Intrusion Detection. Tech. rep., 1998.

[39] Qian, Z., and Mao, Z. M. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *IEEE Symposium on Security and Privacy* (2012).

[40] Qian, Z., Mao, Z. M., and Xie, Y. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *CCS* (2012).

[41] Qian, Z., Mao, Z. M., Xie, Y., and Yu, F. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy* (2010).

[42] Quach, A., Wang, Z., and Qian, Z. Investigation of the 2016 linux tcp stack vulnerability at scale. In *Proc. ACM SIGMETRICS* (2017).

[43] R. Braden, Ed. Requirements for Internet Hosts - Communication Layers. rfc 1122, 1989.

[44] Ramaiah, Anantha and Stewart, R and Dalal, Mitesh. Improving TCP's Robustness to Blind In-Window Attacks. rfc5961, 2010.

[45] Shan, Z., Neamtiu, I., Qian, Z., and Torrieri, D. Proactive restart as cyber maneuver for android. In *MILCOM 2015 - 2015 IEEE Military Communications Conference* (2015).

[46] Tobagi, F., and Kleinrock, L. Packet switching in radio channels: part ii–the hidden terminal problem in carrier sense multiple-access and the busy-tone solution. *IEEE Transactions on communications 23*, 12 (1975), 1417–1433.

[47] Wang, S., Wang, P., Liu, X., Zhang, D., and Wu, D. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 235–252.

[48] Zhang, X., Knockel, J., and Crandall, J. R. Original SYN: Finding Machines Hidden Behind Firewalls. In *INFOCOM* (2015).

[49] Zhang, X., Knockel, J., and Crandall, J. R. High fidelity off-path round-trip time measurement via tcp/ip side channels with duplicate syns. In *2016 IEEE Global Communications Conference (GLOBECOM)* (Dec 2016), pp. 1–6.

[50] Zhou, Z., Qian, Z., Reiter, M. K., and Zhang, Y. Static evaluation of noninterference using approximate model counting. In *Proc. of IEEE Security and Privacy* (2018).