UNIVERSITY OF CALIFORNIA
RIVERSIDE

Identification and Mitigation of Information Leakage Caused by Side Channel Vulnerabilities in Network Stack

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yue Cao

December 2019

Dissertation Committee:

Professor Srikanth V. Krishnamurthy, Co-Chairperson
Professor Zhiyun Qian, Co-Chairperson
Professor Chengyu Song
Professor Mohsen Lesani

The Dissertation of Yue Cao is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

Throughout the odyssey to achieve my Ph.d degree, I'm lucky to have so many excellent and wonderful people with me and I want to thank all of them.

First and foremost, I want to show my sincere gratitude to my dear advisors, Prof. Srikanth Krishnamurthy and Prof. Zhiyun Qian, for their excellent guidance with patience, continous support and faithful belief on me. Without their help, I would not have been here.

I also want to thank my rest committee members, Prof. Chengyu Song and Prof. Mohsen Lesani, for their brilliant comments and suggestions. Especially, I would like to highly thank Prof. Chengyu Song for his help and support in my last project.

Besides, I would like to appreciate all of my co-authors. I would like to shout out loudly to thank Dr. Ahmed Atya, Dr. Shailendra Singh, Dr. Tuan Dao and Mr. Zhongjie Wang, for your great contributions in our papers.

I'm also fortunate to have great colleagues and friends in my life. Chunqi Li, Jun Chen, Tianshu Wei, Boweng Zheng, Shukui Zhang, Kittipat Apicharttrisorn, Joobin Gharibshah, Amir Feghahati, Ahmad Darki, Shailendra Singh, Zhongjie Wang, Zheng Zhang, Pengxiong Zhu, Weiteng Chen, Hang Zhang, Yizhuo Zhai, Shasha Li, Indrajeet Singh, Tuan Dao, Ahmed Atya, Azeem Aqil, Xing Zheng, Linchao Liao, etc.

Last but also the most important, I really want to show my greatest gratitude to my family, my parents, my sister, my grandparents and my fiancee, Mo. Words cannot simply express my love to you.

Published Works. Chapter 2 was published at IEEE INFOCOM 2018 and is under submission in Transaction on Networking, Chapter 3 was published at USENIX Security 2016 and Transaction

on Networking 2018, Chapter 4 was recently accepted and is going to appear at ACM CCS 2019.

To my family, and my love Mo, for all the faith and support.

ABSTRACT OF THE DISSERTATION

Identification and Mitigation of Information Leakage Caused by Side Channel Vulnerabilities in Network Stack

by

Yue Cao

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2019
Professor Srikanth V. Krishnamurthy, Co-Chairperson
Professor Zhiyun Qian, Co-Chairperson

Keeping users' sensitive information secure and private in today's network is challenging. Network nowadays are subject to a wide variety of attacks, such as eavesdropping, identity spoofing, denial of service, etc. What is worse, encrypting sensitive data is often not enough in light of advanced threats such as side channel attacks, which enable malicious attackers to infer sensitive data from "insignificant" network information unexpectedly. For this purpose, we propose series of techniques to prevent such information leakage at different layers in network stack, and raise awareness of its severity. In our first work, we propose a practical physical (PHY) layer security framework FOG, for effective packet header obfuscation using MIMO, to prevent eavesdroppers from receiving any packet headers to profile users. Secondly, we identify and fix a subtle yet serious pure off-path side channel vulnerability (CVE-2016-5696) introduced in both TCP specification and its implementation in Linux kernel. This vulnerability allows malicious attackers to indicate arbitrary TCP connection's state, reset the connection or even further hijack the connection. Motivated by the fact that most previous TCP side channel vulnerabilities are manually identified, in our last

work, we propose a principled TCP side channel vulnerability discovery solution based on model checking and program analysis. It automatically identifies 12 new side channel vulnerabilities (and 3 old ones) from TCP implementation in Linux and FreeBSD kernel code. The ultimate goal of my research is to help guide the future design and implementation of network stacks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As more services are provided online, an increasing number of users' sensitive information (such as password, social security number, bank account) are stored and transmitted on the Internet. However, keeping such sensitive information secure and private in today's network is challenging. As networks are groups of large and complicated systems that are interconnected with each other, they are subject to a wide variety of attacks, such as eavesdropping, identity spoofing, password-based attacks, denial of service, etc. By utilizing one or more attack vectors, malicious entities are able to steal users' private information for their own benefits (e.g., financial profit).

Traditionally, encryption is used to protect sensitive information from eavesdroppers on the Internet. However, it is often not enough in light of advanced threats such as side channel attacks. Because of encryption's significant overhead, it's common that only sensitive information is encrypted, and the rest "insignificant" information left in plaintext. Besides, encryption does not always hide everything (for example, eavesdroppers can still find the amount of data exchanged on an encrypted conversation). In side channel attacks, such "insignificant" information can be utilized

by an attacker to infer sensitive and valuable information, regardless whether sensitive information is encrypted or not. As has been demonstrated in the past, the auto suggestion feature in search engines introduced a side channel vulneraiblity, which allows the attacker to infer keywords searched by the users, by simply looking at the sizes of encrypted packets exchanged with the search server [92]. The reason why the side channel vulnerabilities are difficult to deal with is that oftentimes seemingly insignificant information could leak sensitive and valuable information unexpectedly. Furthermore, most of our systems and networks by design do not take into account the side channel threats, as there are other important issues to worry about. Unfortunately, side channel vulnerabilities are therefore ubiquitous.

In order to make progress towards solving this problem, my Ph.d research focuses on identifying and mitigating such serious information leakage caused by side channel attack at different layers in network stacks. In Chapter 2, we propose a MIMO based physical layer security framework to prevent eavesdropper from decoding packet headers in WiFi network, which can potentially leak users' profile information. In Chapter 3 and 4, we firstly identify a subtle yet serious TCP side channel vulnerabilities (CVE-2016-5696) manually and then propose an automatic solution to identify side channel vulnerabilities in TCP implementation from OS kernels. The ultimate goal is to help guide the future design and implementation of network stacks to be free of side channels.

1.1 A Framework for MIMO-based Packet Header Obfuscation

Eavesdroppers can exploit exposed packet headers towards attacks that profile clients and their data flows. In my first work (Chapter 2), we propose FOG, a framework for effective full

and partial header blinding using MIMO, to thwart eavesdroppers. FOG effectively tracks header bits as they traverse physical (PHY) layer sub-systems that perform functions like scrambling and interleaving. It combines multiple blinding signals for more effective and less predictable obfuscation, as compared to using a fixed blinding signal. We implement FOG on the WARP platform and demonstrate via extensive experiments that it yields better obfuscation than prior schemes that deploy full packet blinding. It causes a bit error rate (BER) of > 40 % at an eavesdropper if two blinding streams are sent during header transmissions. Furthermore, even with full header blinding, FOG incurs a very small throughput hit of ≈ 5 % with one blinding stream (and 9 % with two streams). Full packet blinding incurs much higher throughput hits (25 % with one stream and 50 % with two streams).

1.2 Off-Path TCP Exploits of the Challenge ACK Global Rate Limit

In this work (Chapter3), we report a subtle yet serious side channel vulnerability (CVE-2016-5696) introduced in a recent TCP specification. The specification is faithfully implemented in Linux kernel version 3.6 (from 2012) and beyond, and affects a wide range of devices and hosts. In a nutshell, the vulnerability allows a blind off-path attacker to infer if any two arbitrary hosts on the Internet are communicating using a TCP connection. Further, if the connection is present, such an off-path attacker can also infer the TCP sequence numbers in use, from both sides of the connection; this in turn allows the attacker to cause connection termination and perform data injection attacks. We illustrate how the attack can be leveraged to disrupt or degrade the privacy guarantees of an anonymity network such as Tor, and perform web connection hijacking. Through extensive experiments, we show that the attack is fast and reliable. On average, it takes about 40 to 60 sec-

onds to finish and the success rate is 88% to 97%. Finally, we propose changes to both the TCP specification and implementation to eliminate the root cause of the problem.

1.3 Principled Unearthing of TCP Side Channel Vulnerabilities

Recent work (including our second word) has showcased the presence of subtle TCP side channels in modern operating systems that can be exploited by off-path adversaries to launch pernicious attacks such as hijacking a connection. Unfortunately, most work to date is on the manual discovery of such side-channels, and patching them subsequently. In the third work (Chapter 4) we ask "Can we develop a principled approach that can lead to the automated discovery of such hard-to-find TCP side-channels?" We identify that the crux of why such side-channels exist is the violation of the non-interference property between simultaneous TCP connections i.e., there exist cases wherein a change in state of one connection implicitly leaks some information to a different connection (controlled possibly by an attacker). To find such non-interference property violations, we argue that model-checking is a natural fit. However, because of limitations with regards to its scalability, there exist many challenges in using model checking. Specifically, these challenges relate to (a) making the TCP code base self-contained and amenable to model checking and (b) limiting the search space of model checking and yet achieving reasonable levels of code coverage. We develop a tool that we call *SCENT* (for Side Channel Excavation Tool) that addresses these challenges in a mostly automated way. At the heart of *SCENT* is an automated downscaling component that transforms the TCP code base in a consistent way to achieve both a reduction in the state space complexity encountered by the model checker and the number and types of inputs needed for verification. Our extensive evaluations show that *SCENT* leads to the discovery of 12 new

side channel vulnerabilities in the Linux and FreeBSD kernels. In particular, a real world validation with one class of vulnerabilities shows that an off-path attacker is able to infer whether two arbitrary hosts are communicating with each other, within slightly more than 1 minute, on average.

1.4 Dissertation Organization

This dissertation is structured as following. In Chapter 2, we will describe a practical physical (PHY) layer security framework FOG, for effective packet header obfuscation using MIMO. In Chapter 3, we identify and fix a subtle yet serious pure off-path side channel vulnerability introduced in TCP layer. In Chapter 4, we further propose a principled TCP side channel vulnerability discovery solution based on model checking and program analysis on TCP implementation in Linux and FreeBSD kernel code. Chapter 5 summarizes all my works and concluds this dissertation.

Chapter 2

A Framework for MIMO-based Packet Header Obfuscation

Wireless links are susceptible to eavesdropping attacks. Recent standards like 802.11i and 802.11w propose to use link layer encryption, but still expose MAC layer addresses of both APs (access points) and clients. By just capturing the MAC addresses an attacker can analyze traffic to figure out how much data is being transferred between the AP and particular clients. This information can be used to perform various potent attacks. For instance, user input such as keystrokes to web applications can be inferred [92]. Jamming attacks can also target clients that are receiving heavy volumes of traffic [76].

In practice the problem is worse since 802.11w is not used in most scenarios (e.g., in airports or coffee shops). Thus, by intercepting control frames (e.g., ACKs), an attacker can now infer the amount of traffic exchanged with a particular server (in the wired network) since it can access the source and destination IP addresses, port numbers etc. One could conceivably encrypt

the entire packet including the MAC layer header to prevent traffic analysis. However, as we discuss later this is hard to do in practice.

Use of MIMO to cope with eavesdropping: An alternative approach for thwarting eavesdroppers, is the use of an antenna array or MIMO (multiple-input multiple-output) for obfuscating data streams. As proposed in [12], a subset of the transmitter's degrees of freedom can be used to transmit *blinding* streams that interfere with a legitimate stream in the area around the transmission (and thus foil eavesdropping), except at the intended recipient.

Fundamental challenge and tradeoff: MIMO-based obfuscation does not guarantee that an attacker will be unable to decipher eavesdropped information. This is because while a blinding signal will effectively disturb the legitimate signals in most locations, there could be places where it does not. One way to cope with this is to use a plurality of blinding streams. However, as the number of blinding streams is increased for better obfuscation (as in [12]), the achievable throughput on the MIMO link decreases (since fewer legitimate streams can now be transmitted). In fact, even the use of a single continuous blinding stream decreases the throughput by $\frac{1}{N}$ where, N is the number of antennas at the sender.

Our observation: Given that most sensitive traffic is sent using TLS/SSL or in some cases link layer encryption, it makes sense to use MIMO to blind only the headers [1](instead of using a continuous blinding stream as in [12]). This can drastically reduce the performance penalties that are experienced with traditional MIMO-based obfuscation (e.g., [12]).

Challenges in realizing header blinding in practice: First, due to PHY layer functions such as scrambling and interleaving, symbols that correspond to the header bits are dispersed

[1]Headers here represent all headers (MAC, IP or TCP) that need to be obfuscated; the particular headers are chosen based on the type of encyrption used (e.g., 802.11i vs SSH).

throughout a packet to be transmitted. Furthermore, convolution codes (commonly used) jointly encode header and payload bits. Thus, it is inevitable that symbols associated with the header are intertwined with some payload bits that will need to be blinded out as well. Correctly identifying the symbols to blind out is a challenge that has to be addressed.

Second, the header bits in the multiple streams that are to be transmitted simultaneously are not aligned. Thus, if the header bits in any stream are to be blinded, the transmissions of symbols from one or more of the other streams will need to be temporarily *suspended* during that time to allow for a blinding signal(s) to be transmitted instead. This leads to challenges relating to scheduling the blinding signals and providing information about suspensions to legitimate receivers.

Security issues with MIMO-based obfuscation: Recently, it was shown that if an eavesdropper knows some of the transmitted bits (e.g., MAC address of the AP), it can filter received data to extract the blinding signal [90]. Knowing the blinding signal allows the decoding of the remainder of the protected information. These attacks are called *plaintext attacks against physical layer security*. This is of concern since broadcast packets (e.g., for association) are sent in the open, and thus, reveal the AP's MAC address.

Contributions: In this work, we build a practical protocol framework, *FOG*, for effectively performing header/partial header blinding to protect wireless transmissions from eavesdroppers. As discussed above, header blinding significantly improves throughput over traditional full packet blinding. To identify the header symbols, *FOG* uses tainting to track the header bits as they traverse the physical layer system (scrambler, convolutional coder and interleaver).

Instead of using a fixed stream with a single blinding signal, *FOG* schedules different blinding signals at fine-grained intervals, achieving increased randomness. This makes it almost

impossible for an eavesdropper to find locations where the combined effects of the blinding signals are ineffective. This also dramatically raises the bar against *plaintext attacks against physical layer security*; since the blinding signals change dynamically over small time intervals, it is computationally hard for an attacker to launch such an attack.

In summary, our contributions are as follows:

- We design *FOG* for effectively obfuscating headers in wireless packet streams. *FOG* applies MIMO-based blinding that can be used in conjunction with encryption to effectively thwart eavesdroppers from performing traffic analysis on wireless links. *FOG* works with TLS, IPsec, or link layer encryption, obfuscating any headers transmitted in the open.
- We implement *FOG* on our WARP radio testbed [7]. We perform extensive experiments to show that *FOG* provides even better obfuscation than traditional full packet blinding (as in [12]). With two blinding streams, the fraction of locations where an eavesdropper experiences a 40 % bit error rate improves from almost 0 % to 100 %. Further, the throughput hit with *FOG* is much lower than with full packet blinding. Specifically, even with full header blinding it is just 5 % as compared to 25 % with the latter with one blinding stream. It is even lower if partial header blinding is used.

2.1 Background

In this section, we describe how blinding (also called nulling) is achieved with multi-user MIMO (MU-MIMO). We consider the popular practical low complexity zero-forcing beamforming (ZFBF) [99]; ZFBF completely removes the interference among the MU-MIMO transmissions.

Notation: N is the number of transmit antennas at the AP. M is the number of concurrently served receivers (considered single antenna for ease of discussion but can be MIMO equipped). The row vector h_m is a $1 \times N$ vector, representing the channel state with respect to user m. Each element of h corresponds to the complex gain from one transmit antenna to the user. The matrix $H = [h_1; h_2; ..; h_M]$ is the channel matrix of dimension $M \times N$. The column vector w_m is an $N \times 1$ beamforming weight vector for user m. Each element of w corresponds to the complex exponential weighting used by each antenna during transmission to that user. The matrix $W = [w_1, w_2...w_m]$ is the weight matrix of dimension $N \times M$. If A streams are used for blinding, $N - A$ streams are available for legitimate communications.

Zero Forcing Beam-forming (ZFBF): ZFBF enables a transmitter (AP) to construct multiple spatial streams and transmit them to multiple users in parallel. The channel state information (CSI), h_k, is obtained for each user k, using pilots and a corresponding w_k is computed. A composite data stream (consisting of streams destined for different users) is then multiplied by W and transmitted using the antenna array. ZFBF selects weights that cause zero inter-user interference. In [110], it is shown that the optimal W, satisfying the zero inter-stream interference condition, is the pseudo-inverse of H, i.e., $W = H^{\dagger} = H^{*}(HH^{*})^{-1}$. The above matrix multiplication implicitly requires that the maximum number of concurrent spatial streams M, to be less than or equal to the number of transmit antennas, N. The CSI from each user (the h vector), can be fed back via the RTS/CTS in compliance with 802.11ac.

Blinding Process: Blinding signals must be orthogonal to the intended signals (so that they are not affected) and are transmitted concurrently with the intended receivers' signals by the ZFBF enabled transmitter. Together, the intended and the blinding streams should not be $> N$.

For the blinding streams, $\hat{h}$ vectors that are orthogonal to the intended receivers' h vectors are computed. The precoding weight matrix W is then constructed by considering the blinding streams also as streams that are to be transmitted. To create the h vectors for a blinding stream, the Gram-Schmidt orthogonalization process is used (as in [12]). Let us assume that the first $N - A$ streams are designated to intended receivers, without loss of generality. The CSI $h_1,\ h_2, \ldots h_{N-A}$, are obtained and a matrix $\tilde{H}$ is created in the same way as H was created with traditional ZFBF. Here, we first create an Identity matrix I of size $N \times N$ and truncate it to a matrix $\hat{I}$ of size $(N - A) \times N$. $\hat{I}$ is now concatenated with the $\tilde{H}$ matrix to create a preliminary H matrix. This preliminary H matrix, is passed to the Gram-Schmidt process and the matrix, $\hat{H}$ is constructed, as follows:

$$\hat{h_k} = \begin{cases} h_k & \text{if } k = 1 \\ h_k - \sum_{j=1}^{k-1} \frac{\langle h_k, \hat{h_j} \rangle}{\|\hat{h_j}\|^2} \hat{h_j} & \text{if } k > 1 \end{cases}$$

where, k is the index of a vector and $< h_k, \hat{h_j} >$ is the dot product of h_k and $\hat{h_j}$. A corresponding precoding matrix $\hat{W}$ is then computed. The intended streams along with blinding streams (could be any random stream of bits), are then weighted using $\hat{W}$ and transmitted. Note that the process is similar for other combinations of legitimate streams (when they are not the first $N - A$ streams).

2.2 Motivation and System Model

2.2.1 Motivating study

To understand the extent to which information is exposed, we perform a measurement study where we collect WiFi packets in a nearby public coffee shop for 30 minutes using existing software viz., WireShark on Ubuntu 14.10 with Linux 3.16 (we checked with our office of research integrity and an IRB was not required since we do not collect data that reveals user identities and

Total Packets	1809019
Number of Destination IPs	1900
Number of http Requests	2000
Number of Flows	11484
Number of Services	65536

Figure 2.1: Summary of captured packets

behaviors). We capture approximately 1.8 million packets, and identify more than 1900 unique destination IP addresses that clients were communicating with. We also identify 2000 flows between different devices and open destination IP addresses. We also identified more than 65k services running on different ports on different devices (e.g., FTP on port 21 or SMTP on port 25).

What is clear from our results (summarized in Figure 2.1) is that even an unsophisticated attacker can gather a lot of information since control and management packets are often transmitted unencrypted (even if link layer encryption is used for data packets). It has been previously shown that users can be identified and linked with confidential information such as their location history using such management information that is sent in the clear [47, 72]. These attacks by third parties only require observing low level identifiers (such as addresses and network names in transmissions) that map readily to high-level identifiers (such as identities). An attacker, by observing the sequence of packets within a session, can tease out sensitive information about their contents. For example, a distinct pattern of packet sizes and timings (how often do a client and server communicate) is sometimes sufficient to identify the keystrokes [92, 93], the web pages viewed [96], the videos watched [89], the languages spoken [105], and the applications used [106] by the user.[2]

In the best case, even with link layer encryption of all unicast packets (including control packets), the MAC addresses and CCMP (Counter Mode Block Cipher Chaining Message Authenti-

[2]Some of the attack examples are not specific to wireless netowrks. However, wireless transmissions make it easier to capture and perform traffic analysis.

cation Code Protocol) header of clients are exposed. One could thus analyze the traffic (e.g., packet sizes, download patterns) that a client receives. Moreover, some information leaked in the MAC header can be used to infer information about the client [92].

Encrypting headers is hard: It is hard with current 802.11 standards to fully encrypt the packet headers (knowing no header is in cleartext). Most importantly, CCMP encryption uses the AES cipher as its building block. AES uses counter values which the receiver derives from the CCMP header to perform decryption. If there are packet losses, it cannot pre-determine the contents in forthcoming headers to pre-compute the encrypted header values. Thus, the CCMP header has to be sent as cleartext. However, this header can form the basis for side-channel attacks (an example appears in [92]). In addition, sending the CCMP header as cleartext can lead to what is called a "time memory trade-off pre-computation attack," which provides a shortcut for a brute force attack to get the encryption key (details in [57]). To encrypt the CCMP header, one needs another CCMP header in cleartext (therefore a chicken-and-egg problem). Thus, we argue that blinding (at least the MAC) headers with MIMO would be an effective countermeasure.

Prior work: In [48], *SlyFi*, a system that tries to overcome the problem of encrypting everything including headers is proposed. Each receiver tries to pre-compute the encrypted headers and store them in a hash table. When packets are received, the hash of the header is computed and checked against table entries. These hash table checks may involve higher layer operations. While this might not degrade throughput, it could increase energy drain in wireless clients. We consider *FOG* to be an alternative PHY layer solution based on MIMO. Note that both *SlyFi* and *FOG* require changes to both the APs and mobile clients (discussed later for *FOG*).

MAC Layer randomization is an alternative? One might argue that randomization of packets' MAC addresses may sufficiently protect users from eavesdroppers and packet blinding (or *FOG*) is not needed. However, it has been shown that MAC layer address randomization is insufficient to protect user privacy; other fields (e.g., the CCMP headers) need to be obfuscated in addition as shown in [101].

2.2.2 System model and assumptions

Association: All broadcast frames or packets are sent using omnidirectional transmissions [38] to achieve maximum coverage. Clients listen to the beacons (from APs) to identify the APs within range. Alternatively, they can broadcast probe request frames to find reachable active wireless networks. Subsequently, authentication request and response frames are exchanged between a client and its chosen AP. Authentication could be open authentication but we use a more secure authentication scheme as described in the IEEE 802.11w standard. After authentication, association request and response frames are exchanged between the client and the AP. At this time, a secret key is established between the AP and the client using methods described in the 802.11w standard [107].

CSI Exchange: For ZFBF to work, each client needs to exchange channel state information (CSI) with its AP. The CSI is exchanged via the channel sounding process described in the 802.11ac standard [38]. This process begins when the beamformer (AP) transmits a Null Data Packet (NDP) announcement frame (a control frame). A multi-user NDP announcement frame includes multiple client information records, one for each client (or beamformee). Each record is encrypted using the secret key shared with the associated specific client. When an NDP announcement is sent to multiple receivers, the receiver MAC address is the broadcast address. Following

the transmission of the NDP announcement frame, the AP broadcasts an NDP frame (see [38] for details). Each client then uses specific training fields in the NDP frame to compute its CSI matrix. It then sends the CSI matrix using what is called the "Compressed Beamforming Action frame" to the AP. This Compressed Beamforming Action frame is protected by encryption, using the secret key that is shared by the client with the AP.

What is protected and what is not: *FOG* uses 802.11w to encrypt management frames like NDP. Link layer encryption could be used (optionally) to protect data frames; in such cases *FOG* uses blinding to only protect the MAC and CCMP headers. If only higher layer encryption (e.g., SSH) is used, *FOG* blinds out the relevant higher layer headers as well, during beamformed packet transmissions.

Beacon frames and the association requests are not protected and can reveal MAC addresses of the AP and clients. Thus, *FOG* does not try to hide the identities of the clients that are connected to an AP. Its primary objective is to deter eavesdropping attackers from performing traffic analysis attacks to profile the associated clients.

Each frame in 802.11ac (or similar systems) contains known preambles. These are transmitted without any beamforming to allow devices to synchronize and identify the beginning of a packet [38]. These preambles cannot be encrypted or blinded since at this point, the sender and receiver are yet to synchronize with each other; *FOG* does not hide preamble transmissions (they do not yield receiver specific information to an eavesdropper).

Downlink versus Uplink: *FOG* is a framework that tries to hide link layer transmission patterns to prevent traffic analysis. As such, it is agnostic to whether the transmissions are on the downlink or on the uplink as long as the transmitter is equipped with MIMO. Today, most

commodity APs are MIMO based. More recently smartphones and smaller client devices are also MIMO equipped and are 802.11 ac capable (e.g., iPhone6 [4]). If a client is not MIMO based, it could encrypt everything (including the MAC header) on the uplink. The AP would then decrypt all received packets prior to processing. Since the AP is typically connected to a power outlet, this process should not adversely impact communications except for slight increases in delay. This approach however, is infeasible on the downlink; if the AP encrypts the MAC header, each client will need to decrypt all received packets or use a system like *SlyFi* [48], regardless of whether the packet is meant for it; this could cause high energy drain.

Interference: We assume that interference effects are effectively handled by (a) use of multiple channels (b) carrier sensing or scheduling. Thus, in any contention domain, we assume that only one MIMO transmitter is active at any given time. This is true of current deployments of 802.11n or 802.11ac.

Impact of mobility: CSI needs to be exchanged between the transmitter and the receiver periodically. In dynamic (mobile) settings, the periodicity of this exchange will be high in order to effectively beamform transmissions. This problem however, is inherent to MU-MIMO communications and is not a limitation of *FOG*.

2.3 Attacker Model

We assume a passive eavesdropper equipped with MIMO. As the number of antennas at the eavesdropper increases, it is more likely that it will successfully decode information. However, in practice, since the distance between any two antennas has to be of the order of the wavelength used for transmission [99], the form factor of an eavesdropper's device will increase with the num-

ber of antennas. To reflect a practical scenario (and due to limitations with our testbed), in our evaluations, we limit the number of antennas at the eavesdropper to be no more than the number of antennas at the transmitter. However, *FOG* will still be effective with a reasonable increase in the number of antennas at the eavesdropper, beyond this. One can cope with a more potent eavesdropper by increasing the number of blinding streams sent with legitimate streams, as multiple transmitters can cooperate to use their antennas jointly to increase the degree of freedom [87] [54]. Therefore, we agree that the assumption that a transmitter can always have equal or more number of antennas than an eavesdropper is reasonable. In the worst case scenario wherein the attacker has more antennas than the transmitter, the performance of any approach that adds noise (like FOG or STROBE [12]) will degrade but still provides benefits.

We assume that the eavesdropper's mobility is not fast enough for its position to change within one packet transmission time. The assumption typically holds true since this transmission time in 802.11ac is of the order of μs.

The eavesdropper knows the protocol in use (802.11n/ac). With MIMO, it can use selective diversity [99] to reduce its BER[3]. It cannot decode the CSI packets since these are encrypted. When the payload is encrypted, its goal is to decode the header (TCP, IP and MAC) bits. If link layer encryption is used, then its goal is to determine the MAC addresses in the transmitted packets. We assume that the eavesdropper will use the information obtained to either perform traffic analysis, spoof MAC or IP addresses etc., and do not explicitly address its role after these bits are obtained.

The eavesdropper can also try to carry out more sophisticated attacks such as those described in [98] and [90]. It knows some transmitted bits (e.g., the AP MAC address) and can then

[3]Our experiments with two signal combining techniques, viz., equal gain combining [99] and selective diversity [99] showed that the latter did better.

apply a standard filter to try to determine the blinding signals. The attacker then uses an iteratively adjustable filter with this knowledge, to extract other information that cannot be accessed by filtering out the blinding signal. Finally, we assume that the attacker does not launch attacks to disrupt the transmissions (e.g., jamming attacks).

2.4 Obfuscation with *FOG*

In this section, we describe the modules of *FOG* for effective header/partial header blinding. To begin with we provide an overview of (a) full packet blinding which can be considered as a baseline case for comparing the resource efficiency with *FOG*, *and* (b) header blinding and (c) partial header blinding with *FOG*. Subsequently, we describe how *FOG* overcomes the previously mentioned practical challenges to realize (b) and (c).

2.4.1 An overview of how MIMO may be used for obfuscation

Packet Blinding: Packet blinding is the baseline scheme where the transceiver blinds each packet belonging to a stream in its entirety, i.e., both the application layer payload and headers (transport or TCP, network or IP, and MAC headers) are blinded. Towards this, one or more blinding signals are transmitted during each legitimate packet transmission. Packet blinding is primarily used when the payload is not encrypted; examples of such cases include the transmission of DNS packets. Note that depending on the objective, these packet transmissions may also need to be secured (in such a case, full packet heading can be integrated within *FOG* as discussed later). In traditional full packet blinding [12], the blinding signals are fixed during each (joint) packet transmission. Based on a perceived threat, the number of signals used for blinding can be varied.

Throughput overhead: The overhead of using packet blinding can be characterized as follows. For ease of discussion consider an example with a four transmit antenna ($N = 4$) sender. If one blinding signal is generated, the overall throughput is reduced by $1/4$ (it could be higher because of processing penalties but we ignore these here). It is easy to see now that the reduction in throughput (overhead), $\mathcal{O}_m$, resulting from using m blinding signals is roughly $\mathcal{O}_m \approx \frac{m}{N} \times \tau_N$, where τ_N is the throughput if all the N transmissions carry legitimate data (no blinding).

Header blinding: Header blinding is used when the payload is encrypted[4] (e.g., SSH). Blinding signals are transmitted only during the transmission of header bits. While this can potentially improve the throughput efficiency, achieving header blinding in practice comes with a set of challenges; we articulate these below while discussing what needs to be done, but explain how we deal with them in future subsections.

At the physical layer, where *FOG* is implemented, the header size is not known a priori. Only known is the packet size in bits. The challenge is to figure out how many bits in front of the payload belong to the header. Next, the bits are scrambled to randomize the bits, so that long binary strings of zeros or ones are eliminated [99]. Subsequently, Forward Error Correction (FEC) is applied to generate *coded bits*. To provide robustness to bursty errors, the coded bits are interleaved. The resulting bit stream is then mapped onto symbols (modulation) and transmitted using OFDM (with 802.11). As one can readily visualize, the header bits (after the above processes) get mixed intricately with the payload bits. Thus, we need to determine the symbols that correspond to the header bits (track these bits) to perform effective header blinding.

[4]The same principles apply when the higher layer headers are also encrypted; in such cases, only the MAC header will be blinded.

In order to transmit A blinding signals during a header transmission, A legitimate streams will have to be temporarily suspended. In the simplest case, a fixed set of A legitimate signals can be temporarily suspended for this period. However, we propose to *switch* across the legitimate signals that are suspended during the process; from among the $N-A$ streams, different groups of A streams are chosen (could be randomly or in accordance to some policy) to be suspended for different parts of the header bits of any legitimate stream that is being transmitted. Correspondingly, the blinding signals temporally change with such a process (we refer to this process simply as switching for ease of discussion). Switching helps in improving fairness across legitimate streams (the suspensions are distributed across the legitimate streams). Further, it helps in situations where the eavesdropper is at a location where a certain set of blinding signals are not effective. However, temporary suspensions of legitimate streams will need to be indicated to the receivers for the purposes of synchronization.

In a nutshell, *FOG* contains (a) a *header tracking* process that first estimates the header size and then keeps track of the header bits at each stage of physical layer processing (scrambling, FEC, interleaving, modulation) prior to blinding and, (b) a *scheduling* functional process to switch between the suspensions of legitimate signals (and invoking appropriate blinding signals) and indicating these schedules to the receivers.

Throughput Overhead: If the ratio of the header bits to the total number of bits in a packet is γ, it is easy to see that in the best case (where the header bits are perfectly identified and blinded), the throughput penalty is $\mathcal{O}''_m \approx \gamma \times \frac{m}{N}\tau_N$. In reality, the overhead can be higher. As discussed above, legitimate transmissions have to be suspended when transmitting blinding signals in parallel with header bits, and the receivers of those transmissions will need to be made aware of such suspensions. We quantify this overhead later in our evaluations.

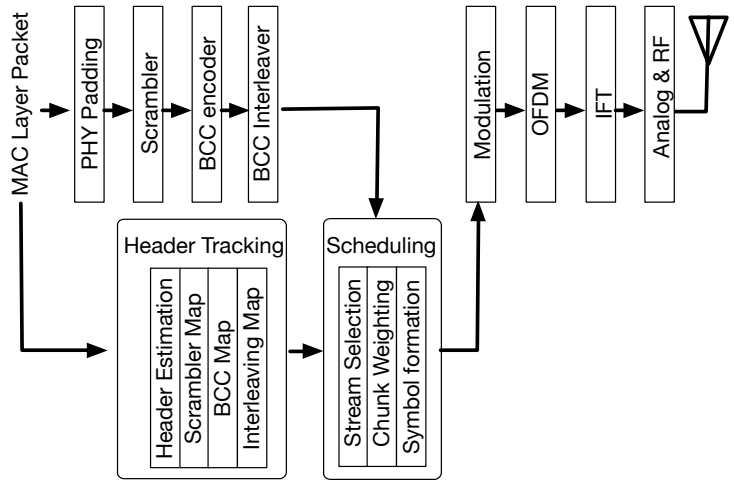


Figure 2.2: Modified 802.11n/ac Architecture

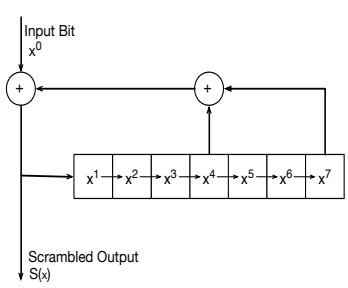
Partial Header blinding: Partial header blinding only blinds the part of the header bits, say a fraction κ, to decrease the overhead incurred with blinding. Except for the issue of choosing which bits of the header to blind out, the process is identical (and the challenges too) to that of header blinding.

Throughput overhead: The overhead of transmitting the blinding signal over a part of the header, using m antennas, is $\mathcal{O}'''_m \approx \kappa \times \gamma \times \frac{m}{N}\tau_N$. While partial header blinding offers lowered overhead, an attacker may be able to retrieve a corrupted packet, and either directly obtain or guess parts of the header bits. For example, simply blinding two header bits, will allow the eavesdropper to reconstruct the packet with four guesses (for the blinded bits). Thus, we need to ensure that a sufficient level of obfuscation is achieved.

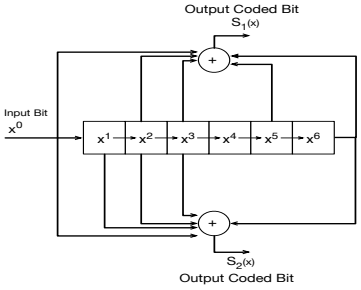
Figure 2.2 provides an overview of the key functional processes used in *FOG*, viz., header tracking and scheduling. We first describe both header tracking and scheduling in the context of full header blinding; subsequently we describe how additional challenges are handled with partial header blinding.

Protocol	Packet Size $\mu(\sigma)$	Header Size $\mu(\sigma)$
FTP Control	98(52)	54(0)
FTP Data	1482(32)	54(0)
HTTP	730(504)	60 (6)
HTTPS	560 (954)	66(0)
DNS	75(245)	42(0)

(a)



(b)



(c)

Figure 2.3: (a) Average packet/header size in bytes, μ, and the standard deviation, σ, for various application layer protocols (b) Data scrambler in 802.11ac (c) Block diagram view of convolutional coding with shift registers for BPSK, code rate =1/2

2.4.2 Header tracking

Header tracking consists of four steps; (1) header estimation (2) scrambler mapping (3) binary convolution code (BCC) mapping and (4) interleaver mapping.

Header estimation: First, *FOG* needs an estimate of the header size in bits (possibly the transport, network and MAC header bits combined). Without any interactions between physical and upper layers, this step is not trivial. It requires packet inspection at the physical layer which is complicated and not easy to do (violates the layering). Thus, we conduct an experimental study to estimate the average size of the packet header for different application layer protocol packets (e.g., HTTP(S), FTP). We collect traces using WireShark [71] at different locations of our campus for this purpose (again, we were told that no IRB was needed) and examine $\approx$ 150,000 packets in total. We find that the average size of the header is about 54 bytes in total. One could use the tabulated results presented in Figure 2.3a to approximate the size of the header given a certain packet size (we follow this approach). On a conservative basis, one could simply use a header size of 70, which is higher

than the header size in all the packets from our measurement study; this results in a slightly higher overhead but ensures that all header bits are blinded.

Next, we describe how we track the header during the scrambling, FEC and interleaving processes. We provide examples of scrambling and encoding processes to illustrate how header bits are tracked. However, we do not describe either the descrambling or decoding processes at the receiver since header tracking is irrelevant to the receiver (details on these processes can be found in [99]).

Scrambler Mapping: Next, we need to track the header bits after scrambling. The scrambler randomizes the bits in a packet, in order to decrease the probability that long binary strings of 1s or 0s exist [99]. The initial seed of the scrambler is 7 bits long and is included in the PHY layer header (and is thus known to the receiver). The function used for scrambling is also known to the both the transmitter and the receiver. The receiver uses the seed value to descramble and retrieve the original sequence of bits. To track the header bits, we taint the output bits of the scrambler if they are header related bits (note that prior to being input to the scrambler, the header bits are at the beginning of the packet). We create an array of indices that record the new locations of the header bits, after scrambling. For instance, let us assume that there are L header bits. We create an array of length L. Each element of this array points to the location of a unique header bit in the scrambled packet. The memory overhead for maintaining this array for an average size header (54 bytes, i.e. $L = 54 \times 8 = \ 432$ bits) is equal to $432 \times$ the size of an integer, which is around 1.6 KB.

To illustrate, consider an example with 802.11ac [75]. Here, scramble operations are implemented using shift registers as shown in Figure 2.3b. There is a generator polynomial $S(x) = x^7 + x^4 + x^0$, known to both the transmitter and receiver. In Figure 2.3b, x^1, x^2, ..., x^7 refer

to the contents of the register. x^0 corresponds to a bit from the input stream. The initial shift register contents are specified by a 7-bit scrambler seed in PHY layer header [75]. Then, the sum bit $S(x) = x^7 + x^4 + x^0$, where the "+" symbol indicates an XOR operation, is computed. This sum bit is the first bit of the output sequence. The contents of the shift register are now shifted up by a stage as follows: $x^6 \rightarrow x^7$, $x^5 \rightarrow x^6$, ..., $x^1 \rightarrow x^2$. The new sum bit $S(x)$ is placed in the shift register in place of x^1. The procedure is repeated with the next bit of the input stream (a new x^0).

To taint the header bits, we mark all the output bits that are generated by using even a single header bit. Thus, for each bit of a L bit header, we *mark* (taint) a sequence of $L + 7$ bits in the output sequence since the output sum bit $S(x)$ depends on the seven previous bits.

BCC Mapping: The next step is to track the header bits after forward error correction (FEC). We assume the use of Binary Convolution Codes[5] (BCC) for ease of discussion. With these codes, the size of the packet (including the packet header) will be doubled after encoding. Similar to what we did in the previous step, we taint all the output bits related to the (one or more) header bits.

To illustrate, let us consider BCC as implemented in 802.11ac. BCC also uses the shift registers to generate the output bits (Figure 2.3c). These registers are initialized with zeros. Each input bit (x^0) generates two output bits using two generator polynomials. For example for a rate $1/2$ code, the generator polynomials are $S_1(x) = x^6 + x^5 + x^3 + x^2 + x^0$ and $S_2(x) = x^6 + x^3 + x^2 + x^1 + x^0$. The scrambler output is the input to the encoder. These polynomials operate on the contents of the shift register (higher order bits) and a single input bit (the lowest order bit). For illustration, let us assume that the first input bit is 1. Then, two output bits are generated. The first polynomial yields $S_1(x) = x^6 + x^5 + x^3 + x^2 + x^0 = 1$ and the second yields $S_2(x) = x^6 + x^3 + x^2 + x^1 + x^0 = 1$.

[5]Note here that with other FEC codes the tracking (using tainting) will be similar to what is described here.

In other words, for the first input (scrambled) bit, 1, we get two output coded bits 11. After first bit is processed it is inserted into the shift register and the original contents are shifted; thus, the new contents are 000001. The outputs corresponding to the header bits which were tainted during scrambler mapping are identified and recorded (tainted). The recorded output coded bits, include those for which the original header bits are either inputs or part of the shift register. Note that if a different code rate is used (a different MCS is used), the polynomials change.

Interleaver Mapping: The interleaver essentially tries to shuffle the input bits to distribute errors uniformly across the packet; bursty errors are thus eliminated. It consists of an array of R rows and C columns. The input fills the array one row at a time and the output is a read out of the array, one column at a time. The parameters R and C are known to both the transmitter and the receiver; the eavesdropper is also aware of these parameters. Since the input header bits are tainted, it is easy to track them at the output of the interleaver. The tracking array is modified to reflect the new positions of the header bits.

2.4.3 Scheduling

As discussed earlier, when blinding the header bits in a specific stream, some of the other legitimate streams will need to be suspended; the scheduling process determines these suspensions. The schedule is constructed in three steps; (1) stream selection (2) weighting and (3) symbol formation. We first provide an overview of how the schedule is formed and then describe the steps in detail.

Overview: For ease of discussion, we assume that the traffic is saturated (if not and there is available capacity, additional blinding streams can be transmitted to improve the level of ob-

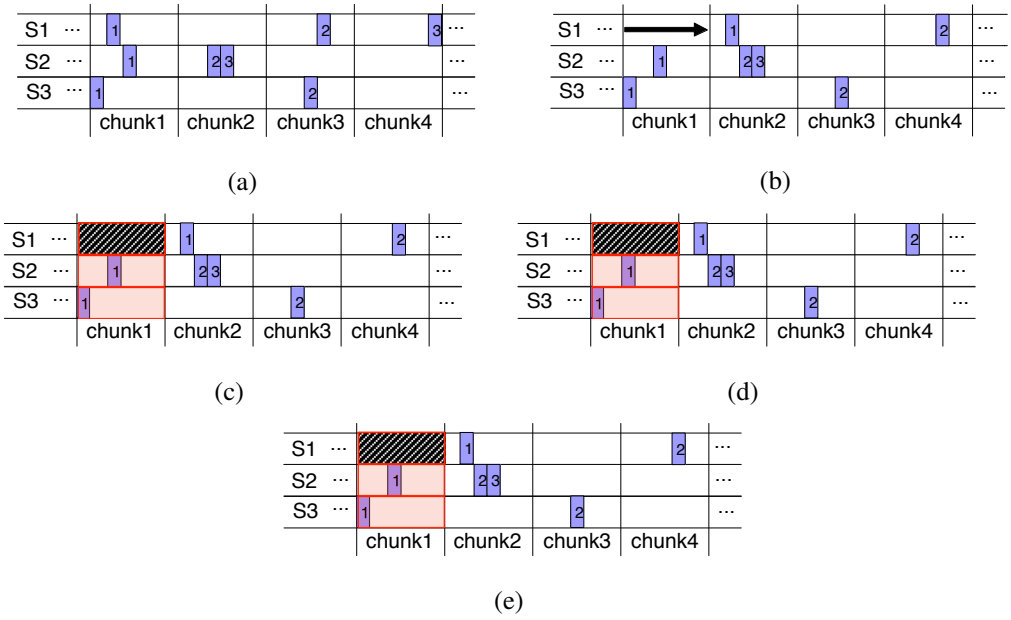


Figure 2.4: An example for scheduling blinding streams during header bit transmissions. The transmitter has 3 antennas. One blinding stream is used. The scheduling sequence is from (a) to (e). (a) shows the original layout of the chunks with the header bits. In the first slot stream S1 is chosen for suspension and thus, the transmission of its chunks are postponed by a slot (see (b)); instead a blinding signal is sent (see (c)). In the next slot (see (d)), stream S3 is chosen for suspension, the corresponding chunks are postponed and a blinding signal is transmitted instead. Finally, stream S2 is chosen for suspension (see (e)).

fuscation). Given this, the transmitter (with N antennas) seeks to simultaneously send N packets to receivers. Towards realizing the scheduling process, the output bits of the interleaver (for each packet) are divided into chunks of bits. Each chunk, c, has a size of L_c bits. Now there are N streams of chunks, where a chunk may or may not contain the header bits. If a chunk on any of the streams contains a header bit(s), then we choose to suspend one or more of the other streams (depending on the level of protection needed) and instead transmit signals associated with the blinding streams. After the streams (both legitimate and blinding) are chosen, the transceiver system generates the ZFBF weights as described in §2.7. The data chunks to be transmitted (including those for blinding) are weighted by the ZFBF beamformer and then OFDM symbols are formed by combining the weighted chunks. In the following, we describe the above steps in detail.

Step 1: Stream Selection: The stream selection process determines on a *per slot* basis, which stream(s) are to be suspended in order to instead send blinding streams (each slot carries a chunk per stream). A queue per stream is created and the indices of all the chunks associated with that stream, that are to be blinded, are recorded. As per a chosen policy, a stream is selected for the subsequent suspension, and a blinding stream is instead generated and sent. In our implementation, we select the stream for suspension randomly as per a uniform distribution. In the long term, this policy ensures fairness between stream suspensions. Given the relatively small number of header bits compared to the packet length, the delays incurred due to stream suspensions and the consequential postponing of chunk transmissions is very small.

To allow the receiver to determine which slots have blinded transmissions, the streams are indexed and the indices of streams are transmitted to the receiver, using ZFBF, prior to the actual data transmission. With knowledge of the suspsensions, the receiver can easily reconstruct the

packet from the data received. Transmitting information with regards to the suspended streams constitutes a significant part of the (small) overhead with our approach (this information is represented using a bitmap as described in §2.5).

Figure 2.4 depicts an example for how blinding signals are scheduled along with chunks that contain header related bits for a transmitter with $N = 3$, and $A = 1$. In the example, as shown in Figure 2.4 (a) three chunks with header related bits are to be scheduled on stream one, and two chunks with header bits are to be scheduled on streams two and three. As shown in Figure 2.4 (b) the first stream is suspended first; its chunk is postponed and the bits from a blinding stream are instead transmitted (shaded chunk). All the following data chunks for S1 are delayed by one slot as a consequence. The process is easy to follow in sub figures (c), (d) and (e).

Choice of chunk size: The chunk size defines the required precision for blinding decisions. If $L_c = 1$, then a decision on whether to blind or not is made with respect to each bit. This results in the minimum blinding time and overhead (only the chosen header bits are blinded). However, the legitimate receivers will need to know of suspensions at the "bit level". In addition, switching across modes (legitimate transmission versus blinding) and the computation of ZFBF weights will need to be done every bit and this can be prohibitive. As the chunk size increases, the granularity of blinding becomes coarser (more non-header bits are blinded) and thus, the throughput penalty increases. However, the complexity of the information made known to receivers (which chunks are suspended) and the switching overhead decreases (due to fewer chunks per packet). In the other extreme case, when the chunk size is equal to the packet size, the scheme reduces to *full packet blinding*. The switching overhead is minimized, and there is no need to inform receivers about any suspensions; however, the throughput penalty is the highest.

W_0	H_0	No Gram-Schmidt Process	h_1	h_2	h_3
W_1	H_1	$Gram - Schmidt(h_2, h_3)$	$\hat{h}_1$	h_2	h_3
W_2	H_2	$Gram - Schmidt(h_1, h_3)$	h_1	$\hat{h}_2$	h_3
W_3	H_3	$Gram - Schmidt(h_1, h_2)$	h_1	h_2	$\hat{h}_3$

(a) $A = 1$

W_0	H_0	No Gram-Schmidt Process	h_1	h_2	h_3
W_1	H_1	$Gram - Schmidt(h_3)$	$\hat{h_1}$	$\hat{h_2}$	h_3
W_2	H_2	$Gram - Schmidt(h_2)$	$\hat{h_1}$	h_2	$\hat{h_3}$
W_3	H_3	$Gram - Schmidt(h_1)$	h_1	$\hat{h_2}$	$\hat{h_3}$

(b) $A = 2$

Table 2.1: A formation of W/H matrices, For $N = 3$.

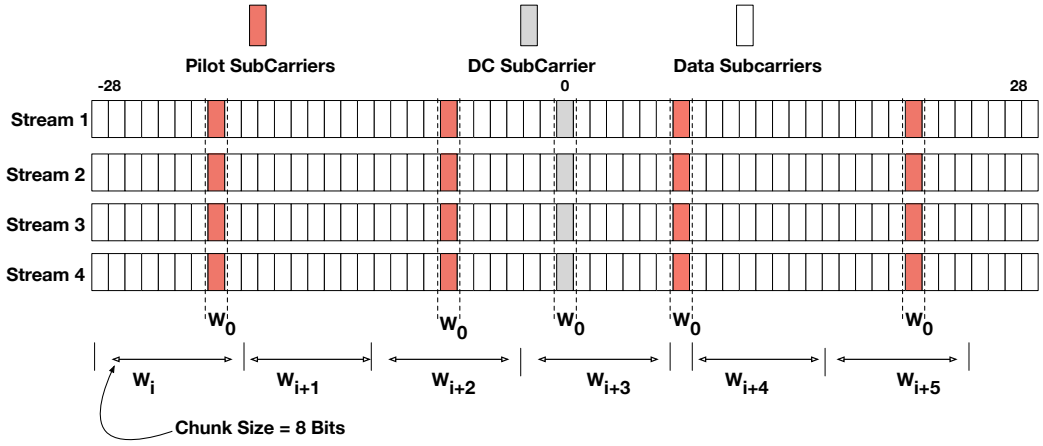


Figure 2.5: Symbol formation : Chunks of fixed sizes are mapped onto modulation symbols (e.g., BPSK), weighted differently, and then mapped onto OFDM subcarriers.

Step 2: Chunk Weighting: Weighting is part of the blinding process as described in §2.7. The weight matrix, W is recomputed each time the AP (transmitter) receives CSI feedback from the receivers. For a N-antenna transmitter, there can be a number of different combinations of data and blinding streams according to which transmissions could be performed (as described in Step 1). For each chunk, W depends on the selected data streams and the blinding streams.

The number of different weight matrices to be computed prior to a packet transmission depends on the total number of streams (N) and the number of desired blinding streams (A). For example, with $N = 3$ and $A = 1$, four weight matrices are needed. These are (i) W_0: no streams used for blinding; (ii) W_1: stream one is replaced with a blinding stream (iii) W_2: stream two is replaced with a blinding stream and (iv) W_3: stream three is replaced with a blinding stream. Examples of weight matrix calculation with $A = 1$ and $A = 2$ are shown in Table 2.1. Each data chunk is weighted based on the *blinding decision*. For the example in Figure 2.4, chunks from streams 2 and 3 are being transmitted in the first slot; stream 1 is replaced with a blinding stream. Thus, the weight matrix W_1 is applied to the data during this slot. Similarly, it is easy to see that W_3 will be the weight matrix that is to be used for the chunks to be sent in the subsequent slot. Since different weight matrices are used for different parts of the packet, it becomes much more difficult for an eavesdropper to (a) recover the entire packet at a static location and (b) launch the so called "plaintext attacks against PHY layer security," wherein an attacker tries to recover a single fixed blinding signal that is used for the entire packet duration using known plaintext [90] (discussed later).

Step 3: Symbol formation: After the interleaving stage, multiple data chunks are combined to form OFDM symbols. This symbol formation is based on the 802.11ac standard. We

provide a brief description here of how the division of each packet into chunks and the application of different weights for each chunk period, influences this process. We assume that the total number of OFDM subcarriers available is equal to 56 and only 52 of those subcarriers are used for data (as in 802.11ac [38]). Four subcarriers are used for pilot tones; these are typically used to correct frequency offsets, synchronization etc. The rest are used for guard carriers.

The mapping of the chunks to OFDM symbol data is shown in Figure 2.5. The bits from each chunk scheduled in a slot, are mapped onto symbols in a modulation constellation (e.g., BPSK has two symbols, QPSK has four). The appropriate weights are then applied on the modulation symbols. To illustrate, let us consider the example shown in Figure 2.5, where the chunk size is equal to 8 bits and BPSK modulation is in use. With this modulation, each subcarrier carries one bit. Then, an OFDM symbol (52 subcarriers) can accommodate 6 chunks. In this example, we have 4 data streams and for each data steam the modulation symbols are determined and mapped on to the subcarriers. For each slot (or the chunks scheduled in that slot) the appropriate weight matrix W is applied. In the figure, we see that for each slot, a different weight matrix is used and thus, the set of signals transmitted are different (different desired streams are suspended).

For some chunk sizes, it may not be possible to fill an OFDM symbol with an integral number of chunks; in that case we use filler bits. In our case, the chunk size is equal to 8 bits; then with BPSK, the remaining 4 bits (from the 52 bits) are called filler bits and are randomly generated. The filler bits are discarded after decoding at the receiver.

After mapping, the weighted and modulated symbols from each stream are converted to time domain symbols via FFT and the resulting symbols are transmitted jointly.

2.4.4 Partial Header blinding

Partial header blinding is very similar to full header blinding, except that *FOG* only blinds certain bits (part of the header) corresponding to specific fields that it seeks to hide. Once we know the header size in the header estimation step, we assume the use of the standard 802.11 header format to estimate the positions of the different fields in the header.

Is it enough to choose header bits randomly for partial header blinding?: Randomly choosing and blinding header bits can expose some of the fields to the attacker. To illustrate, let us assume that the total length of the headers (including TCP, IP and MAC) is L bits and n bits are blinded. Let the probability of a specific field of length j bits being fully exposed be p_e; then, the probability of the field at least being partially blinded is $p_j = 1 - p_e$. It is easy to see that p_e can be computed as follows.

$$p_e = \frac{\binom{L-j}{n}}{\binom{L}{n}} \tag{2.1}$$

The numerator corresponds to choosing all the blinded bits from outside the field and the denominator corresponds to the general case where the blinded bits are chosen to be distributed throughout the header. In Figure 2.6, we plot the p_e as a function of n for $L = 400$; this value of L corresponds to approximately the total number of bits in the composite header (including TCP, IP and MAC) of a 802.11 frame. The values chosen for j correspond to possible field sizes for the IP prefix, the IP header and the MAC header (e.g., IP header is 32 bits). From the figure, we see that if only $\approx$ 75 bits are blinded from among the L bits ($\approx$ 20 %), the likelihood of an IP prefix (assuming that the prefix length of 11 is known) being exposed (not blinded) is about 0.15. The likelihood is lower for a larger field such as the MAC header. However, in practice, if the field is

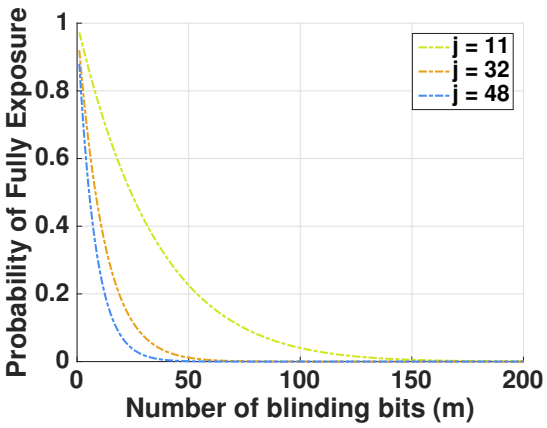


Figure 2.6: The probability of blinding (j bits) given n chosen bits to be blinded when $L = 400$

not fully exposed, but only a few bits are blinded (e.g., say < 4 bits), the attacker can enumerate all possible bits in the field fairly easily. In that case, the likelihood of information leakage further increases. Thus, we seek to ensure that a sufficiently large part of each field is chosen for blinding.

Choosing the header bits for blinding: The presence of optional fields can cause challenges in the estimation of where specific fields lie in the header. To handle this we choose a higher number of bits in the vicinity of the fields we seek to blind, rather than just do so based on the exact size of the field specified by the standard frame format without options. The number of bits to blind with regards to each field, can be chosen depending on the required accuracy. In our experiments, we taint and blind 20% and 50% of the header bits near the approximate locations of the MAC, IP addresses TCP port numbers[6]. We blind bits on both sides of the approximate location to handle both the presence and the absence of optional fields in the header.

For example, if we need to blind the destination IP address we first get to the IP header by skipping the first 36 bytes in the MAC frame to get to the IP datagram in the frame's body. Once

[6]We have experimented with other percentages of blinded bits; the behavioral results were similar to the cases presented. We believe that these values are sufficiently representative of what one might expect with partial header blinding.

we know the beginning of IP frame, we just use the IP header frame format to figure out how many more bits we need to skip to get to the destination IP address. The destination IP address is located after 52 bytes (from this point) and is 4 bytes long. However, this may vary a bit because of optional fields in the IP header. In order to blind the destination IP address of a packet we mark 14 bytes (with 20% blinding, assuming a 70 byte header) from the 45th to the 59th byte. These marked bits are then tracked during the header tracking process as they go through the scrambler, the application of BCC and the interleaver. The scheduling process then generates the blinding stream for the chunks that contain these specific header bits. Bits from the other header fields are chosen for being blinded by (i) similarly tracking the location of these fields in the header and (ii) choosing a range of bits in that space for being blinded.

2.4.5 Effects at receivers and eavesdroppers

Since the blinding streams are orthogonal to the legitimate receivers' streams, these receivers are unaffected and can function as in a traditional MU-MIMO system. The received symbols are first mapped onto coded bits and deinterleaved. These are then input to a decoder that tries to derive the raw bits from the coded bits using the Viterbi algorithm [37] (which is based on dynamic programming). The Viterbi algorithm essentially tries to progressively derive the sequence of raw bits given the coded (interdependent) bits (details can be found in [37]). While the Viterbi algorithm is very effective in correcting isolated errors, it cannot correct long bursty errors. The blinding signals essentially construct a burst of errors at the eavesdropper i.e., the input to the Viterbi decoder first consists of the long sequence of *corrupted* header bits; thus, the decoding will fail.

One could ask if it is possible to reverse engineer the decoding and scrambling by knowing the shift register structures (recall §2.4.2) and retrieve the header bits. To see why this is impossi-

ble let us examine what happens with the shift register corresponding to the scrambler. Similar arguments hold with regards to the BCC encoder, but they are a bit more involved due to the more complex structure of the shift register; we omit a discussion due to space limitations. Assume that initially the scrambler holds the seed values S_1 to S_7 and that these are known to the eavesdropper. The input bit sequence is denoted by I_1, I_2 and so on. It is easy to verify that the first four output bits are $O_i = S_{7-i+1} + S_{4-i+1} + I_i$ for $i \leq 4$. Since these O_i values are either erased or corrupted, the attacker has to guess them. This means that he has to guess the values of I_i and for each of these there are two possible values. For $4 < i < 8$ it is easy to verify that $O_i = S_{7-i+1} + O_{i-4} + I_i$. Since these are again blinded, and because the eavesdropper does not know O_{i-4}, he has to again guess the values of I_i. One can follow through the expressions for $i \geq 8$, and it is easy to see that the attacker can at best only guess the value of a blinded bit. Thus, the number of possible combinations that have to be considered to infer the header bits will be $O(2^H)$. If $H = 400$ bits, with a brute force approach he will have to try 2^{400} combinations!

2.4.6 Robustness to plaintext attacks against PHY layer security

Plaintext attacks against PHY layer security [90] try to reduce the effectiveness of a blinding signal. The attack depends on some part of the header being known. The attacker constructs a filter that estimates the weighting matrix W that is applied. Traditionally since a fixed blinding stream is generated, an attacker can slowly adjust the coefficients of a filter to estimate W and decode with less than a 10% bit error rate [90].

Unlike previous schemes, *FOG* varies the blinding signal depending on which stream is suspended as discussed earlier. Let the transmitter possess N antennas. For simplicity, assume that only one data stream is suspended and a blinding stream inserted in its place. The total number

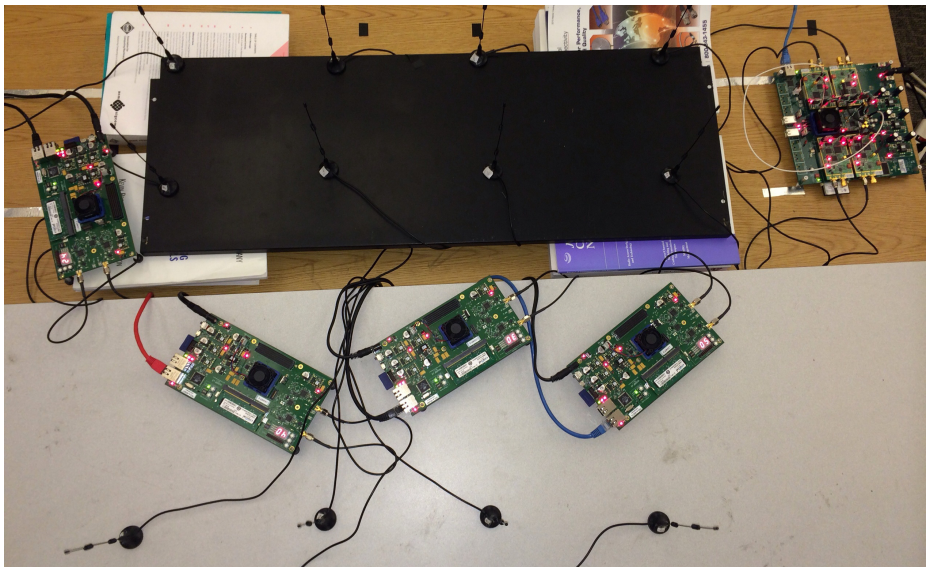


Figure 2.7: An illustrative scenario with our testbed.

of weights used will then be $N + 1$ (N possibilities for the cases where each specific stream is suspended and one for the case where none are suspended). Assume that the attacker knows that the header bits are being blinded and only one blinding stream is used. However, for each header bit, it is unware of which weighting matrix W_j is used, where $j \in \{1, 2, \ldots, N\}$ (assuming that W_0 is used in the case without blinding). If the attacker is able to determine the weights using a known field (e.g., the AP's MAC address), it will still have to consider all possible combinations of these weights for the remainder of the header. If there are H such header bits, each bit can be encoded with *any* of the N weights, and thus, there are N^H possibilities to consider. For a 50 byte header, if the transmitter has 4 antennas, this corresponds to 4^{400} possibilities (exponentially large). The process becomes more difficult if the attacker is unaware of how many streams are used for blinding. For example, if the transmitter hops randomly between 1 and 2 blinding streams, the likelihood of decoding will further decrease.

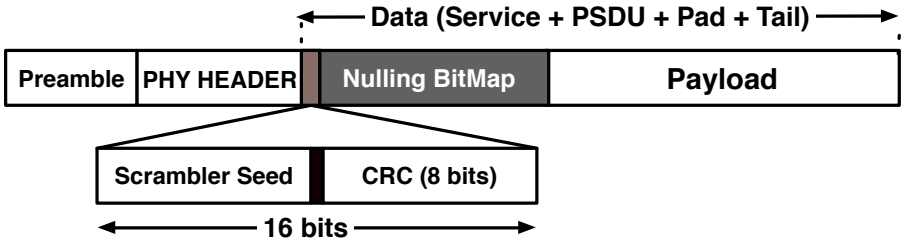


Figure 2.8: Packet format after adding the bitmap.

2.5 Testbed and Implementation

Testbed: Our experiments are on our WARPlab testbed [7] with Versions 1 and 3 (V1 and V3) WARP boards. We implement the transmitter and the eavesdropper on the V1 boards since they can be used for MIMO communications with four antennas. We use the V3 boards for the receiver nodes (single antenna). The transceivers were at fixed locations but the eavesdropper was moved to different locations. Our WARPlab nodes in one scenario are shown in Figure 2.7.

Topology: The default topology consists of one transmitter, four receivers and one eavesdropper. We form a 6×5 grid (60 in. $\times$ 50 in.); the transmitter is placed in the middle of the grid and the receivers at the corners. The eavesdropper is placed at 25 different locations at grid intersections. The packet sizes we use, are 240 and 1514 bytes unless otherwise stated. Note that the maximum distance between the transmitter and the receivers is governed by the hardware/power limitations of WARP boards. With higher powers, larger distances can be covered and our results will apply.

Implementation: The blinding schemes are implemented as a thin sublayer within the physical layer. Our implementation consists of three modules; (i) the header tracking module, (ii) the scheduling module and (iii) the communication module. The first two modules are described in §2.4. The communication module (COMM) is responsible for exchanging the CSI information

between the transmitter and receivers and for performing transmissions. We implement the explicit feedback scheme used for beamforming as described in the 802.11ac standard [75]. First, the transmitter periodically broadcasts a request for the CSI. Then, the COMM module at each receiver encrypts and transmits the CSI; the transmitter and each receiver share an a priori loaded secret key and use AES for encryption [1]. At the transmitter, the CSIs are received and decrypted by COMM. The H and W matrices are constructed based on this CSI.

We also implement the scrambler, the BCC encoder and interleaver from the IEEE 802.11ac specification [75]. After scheduling, COMM constructs the PHY frame as shown in Figure 2.8. A bitmap specifying which transmissions are suspended during the blinding process, is included after the service bits (Scrambler Seed, Reserved bit and CRC), in the frame. The bitmap represents each chunk in the packet by a bit; an entry of '0' indicates a suspension, and '1,' otherwise. The bitmap is also encrypted using AES. Note that the Cyclic Redundancy Check (CRC) is calculated over the modified payload which contains this bitmap. If the bitmap is corrupted, the packet is retransmitted.

2.6 Evaluations

We extensively evaluate *FOG* experimentally. For comparison we also implement a baseline case where the entire packet is blinded (full packet blinding) with a single fixed blinding signal as in [12].

Notation: We refer to blinding the entire packet as *PB* (for Full "Packet Blinding"). If A blinding streams are used we refer to it by PBA; e.g., $PB2$ denotes full packet blinding with two blinding streams ($A = 2$). Similarly the notation FA refers to *FOG* with A blinding streams used

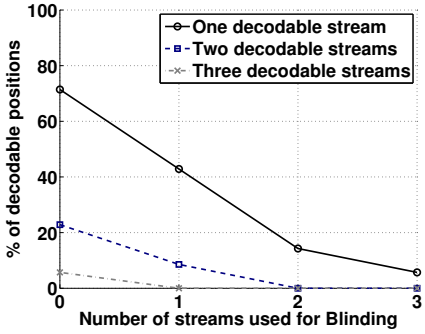


Figure 2.9: The effect of varying the number of streams used for blinding on security.

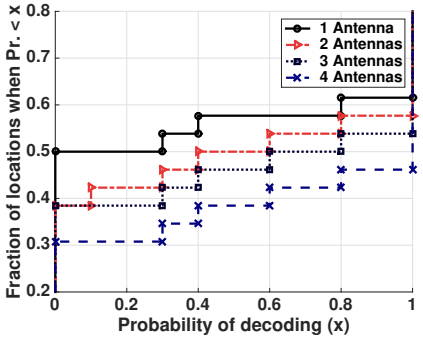


Figure 2.10: Variation in prob. of decoding by eavesdropper equipped with one or more antennas.

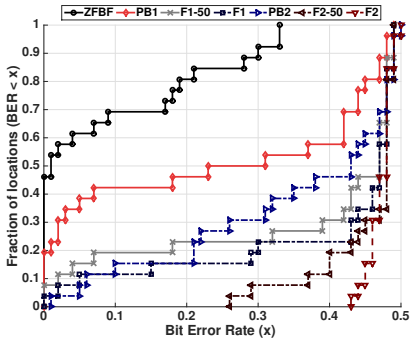


Figure 2.11: Bit error rate as perceived by the eavesdropper for different blinding schemes.

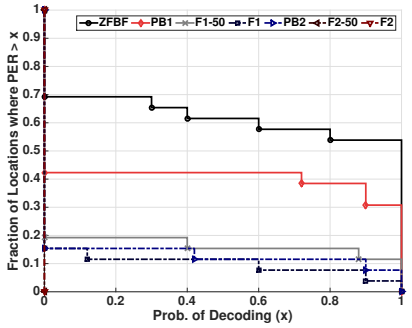


Figure 2.12: Packet error rate as precevied by the eavesdropper for different blinding schemes.

for header protection (e.g., $F2$ refers to the case where two blinding streams are used to protect header bits). The case without blinding is denoted by ZFBF.

The effect of blinding on the eavesdropper: First we perform experiments to determine the effectiveness of blinding on eavesdropping. We use PB; *FOG* is not used. $N = 4$ and A is varied. When $A = 0$, no blinding streams are transmitted i.e., four legitimate data streams are sent. Figure 2.9 plots the percentage of positions where the eavesdropper (with 4 antennas) perceives a

bit error rate (BER) of less than or equal to 0.1, with respect to decoding one or more data streams. Here (and henceforth), we assume conservatively, that an eavesdropper can somehow recover (either by using additional antennas compared to what is shown in the results or by a brute force search) a packet with this BER (of 0.1); in reality, a much more stringent requirement on BER (lower) will be necessary for successful recovery.

Without blinding, the eavesdropper can decode at least one data stream in approximately 72% of the locations. This percentage goes down to 5% when three blinding streams are used. This dramatic improvement is due to the increase in power allocated to the blinding streams, which significantly hurts the eavesdropper in decoding the legitimate data. Figure 2.10 shows the fraction of locations where the eavesdropper can decode with a probability lower than a certain threshold (specified on the x-axis) in the absence of any blinding streams. We see that the probability of the eavesdropper decoding a data stream, increases significantly if the eavesdropper uses multiple antennas for reception. By using four antennas, the percentage of the locations where the eavesdropper can decode more than 80% of the time, increases from 40% to 55%, on average as compared to the case where it has a single antenna. This is because the probability that the received signal is either distorted or destroyed completely on all antennas is much lower than it being so on a single antenna.

Efficacy of *FOG*: Next, we evaluate *FOG* in terms of both bit and packet error rates (BER and PER) seen at the eavesdropper with 4 antennas. Note that if a packet is correctly decoded (CRC passes), only the header bits are exposed (payload is encrypted). Figure 2.11 shows the fraction of eavesdropper locations where the BER observed was less than what is shown on the abscissa (CDF). Without any blinding, in 50% of the locations the BER at the eavesdropper is $\leq 1\%$. By using one or two streams for PB, this BER increases to 10% and 30%, respectively, in 40 % of the locations

(even with convolutional coding). This is because the decodability at the eavesdropper is hurt by the blinding signal(s).

Interestingly, we observe in Figure 2.11 that *FOG* provides better obfuscation than PB. This is due to different blinding signals being used during the transmission of different parts of the header, for obfuscation. Thus, at an eavesdropper location where a single (static) blinding stream is ineffective, this combination of blinding signals provides effective obfuscation (causes higher BERs). In some cases, FOG *with one blinding stream, even outperforms the PB with two streams!* For example, with $F1$,almost all locations considered experience a BER of 40 % or higher; however with $PB2$, the BER experienced is 10 % in approximately 18 % of the considered locations.

In Figure 2.12, we plot the fraction of locations where, the ability of the eavesdropper to recover packets (1-PER) is greater than the probability of decodability shown on the x-axis (complementary CDF). We see that without blinding, the probability of recovery is significant in over 30 % of the locations. With $F2$, the probability of recovering packets is 0 % in 100 % of the locations (all the points are at the origin in the figure). With $F1$, the probability of recovery is less than 10 % in over 90 % of the locations. This again, showcases the effectiveness of *FOG*.

Eavesdropper's ability to decode headers: Next, we run a micro benchmark where, the attacker collects all retrievable packets (decodable or corrupted). We compare a sample set of fields in these packets with ground truth. If there is a match, we assume that the attacker has access to that field. Figure 2.13a shows that without blinding, the eavesdropper can retrieve many of the header fields in 30 to 35 % of the considered locations. With blinding, the eavesdropper can only retrieve a particular header field in at most 8% of the locations. As in earlier results, increasing the number of blinding streams protects better and *FOG* is more effective than PB.

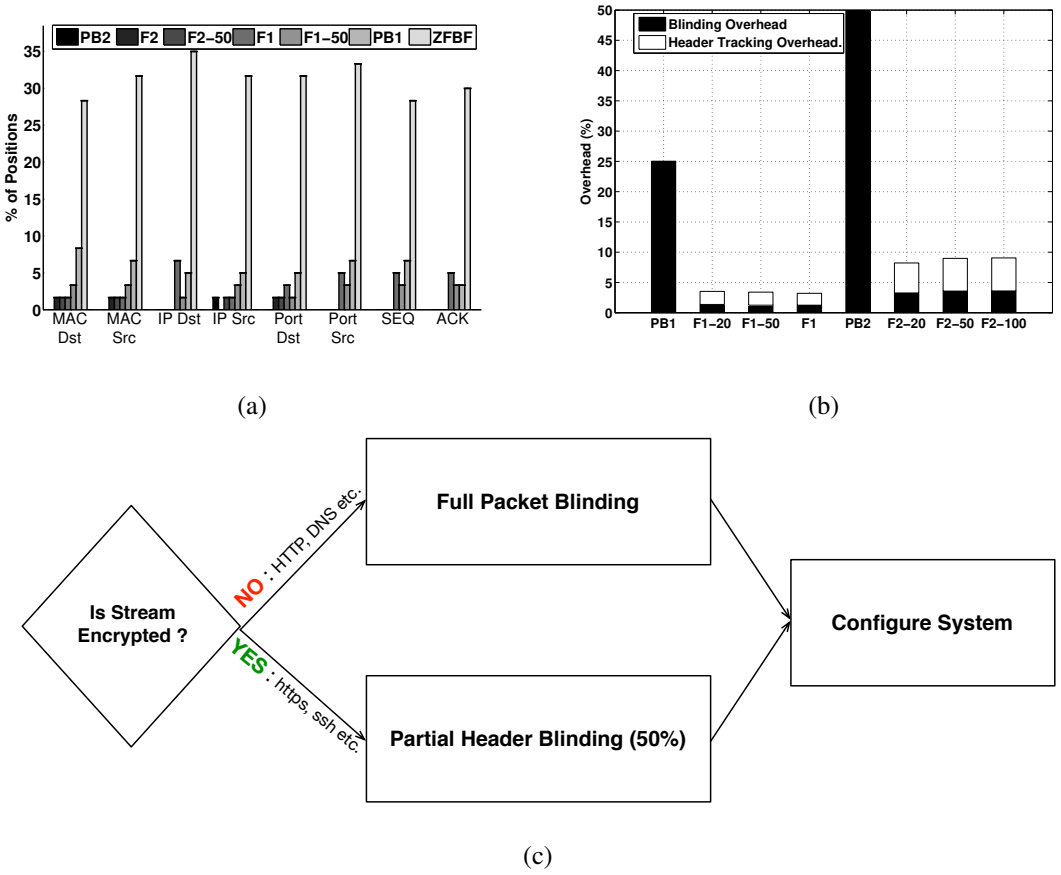
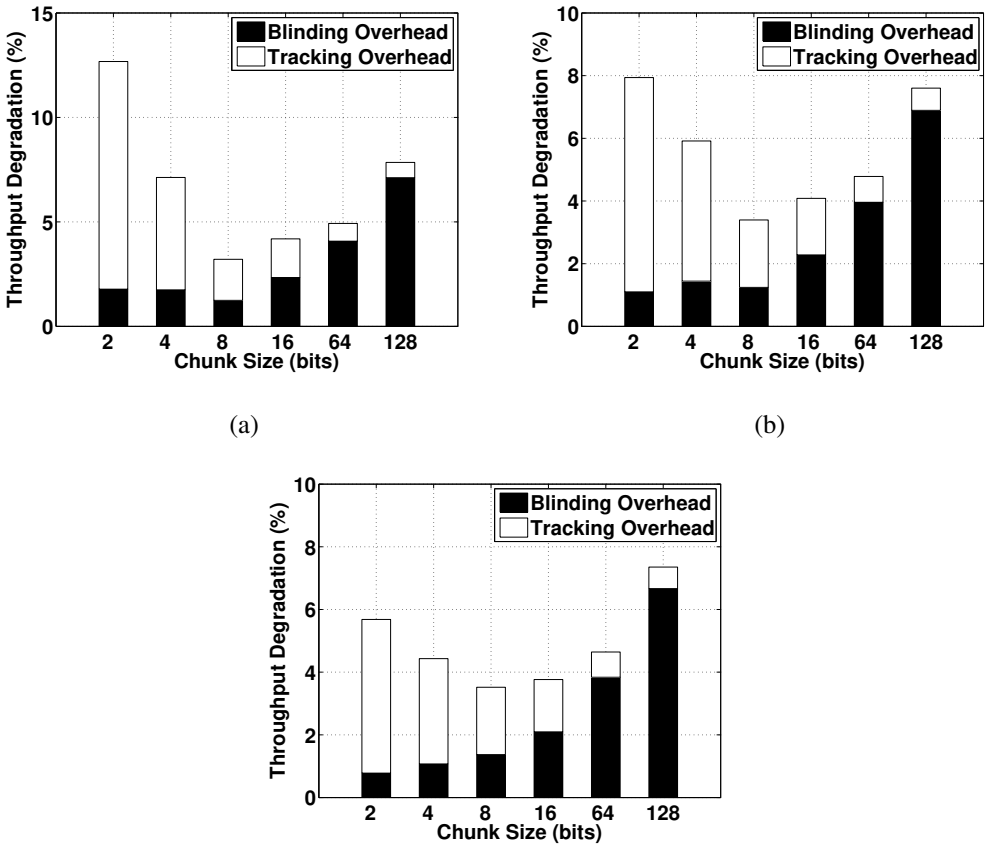


Figure 2.13: (a) Probability of decoding of specific header fields with different blinding schemes. (b) Overheads with PB and *FOG*. (3) A holistic system that adaptively invokes different blinding functions.

Throughput performance: Next, we quantify the performance of *FOG* in terms of the degradation in the average achieved saturation throughput and compare it with that of PB. Figure 2.13b plots the degradation in throughput with one blinding stream. With *FOG*, we use chunk sizes of 8 bits in these experiments. As one might expect, there is a drastic reduction in the throughput degradation with *FOG*. $PB1$ essentially decreases the number of legitimate streams by 1 (from 4) and thus, the throughput hit is about 25 %. With any variant of *FOG*, we see that the throughput hit drops to 5 % or less.

The throughput penalties with *FOG* can be divided into two parts as shown in Figure 2.13b. Specifically, there is a throughput hit because of blinding of the chunks that contain header bits (and some non-header bits as collateral); this is referred to as *blinding overhead*. In addition, there is the overhead of adding information with regards to when a stream is suspended in order for the transmitter to send a blinding signal. This overhead contributes to an overall throughput degradation (longer packets) and is referred to as *tracking overhead*.

Effect of chunk size on throughput: In Figure 2.14 and Figure 2.15, we show the effect of varying the chunk size on the throughput degradation. As discussed above, there is a throughput degradation due to (a) blinding, and (b) the additional overhead of suspension tracking (tracking overhead). As the chunk size in bits decreases, the tracking overhead increases. This is because there are more chunks and one needs to indicate to the receivers which chunks are being suspended. However, the blinding overhead itself decreases. As the chunk size increases we see the opposite effect; the blinding overhead increases since we are blinding larger chunks (and thus, more unnecessary bits) but the tracking overhead decreases. With one blinding stream, a chunk size of 8 bits seems to be the optimal. If two blinding streams are used, we observe that a chunk size of 16 bits is better since the tracking overhead penalty dominates. We use a chunk size of 8 as the default setting in our implementation given that the difference between the throughput degradations with the two chunk sizes are comparable. Interestingly note that (because of reasons discussed earlier), the throughput reduction with the partial header blinding approaches with 50 % and 20 % of the header blinded, are similar with a chunk size of 8. Thus, given its better obfuscation ability, it is better to use 50 % partial header blinding.



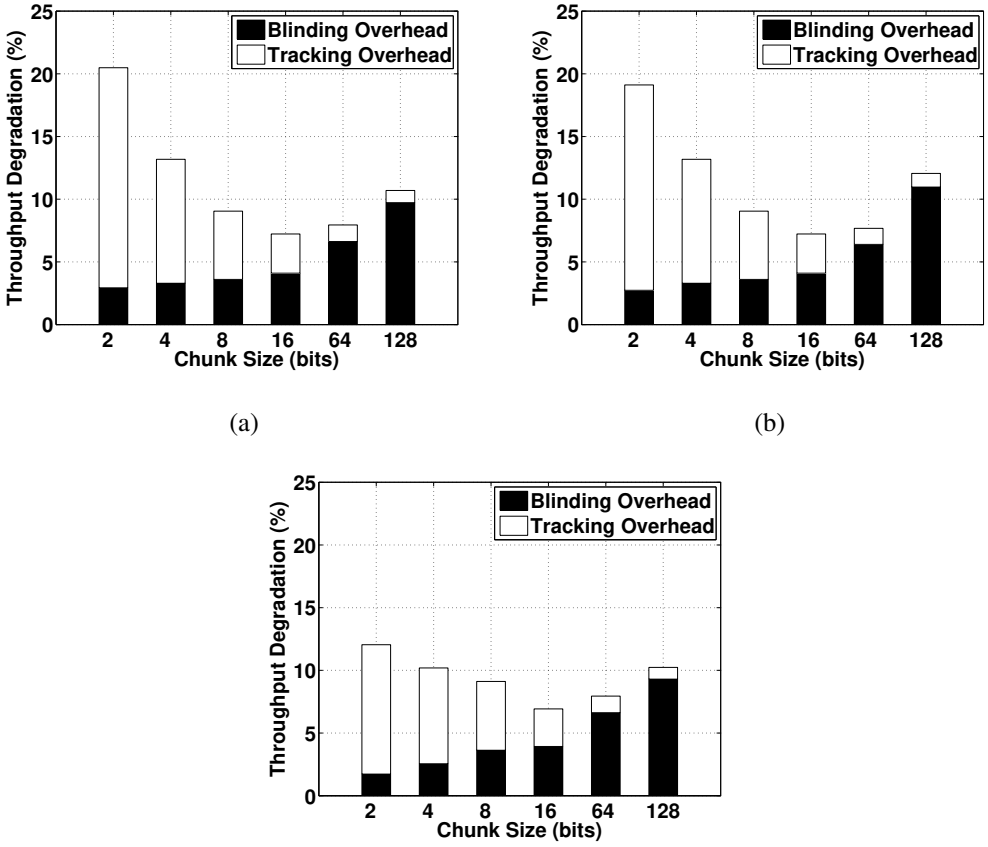
(a) (b)

(c)

Figure 2.14: Overhead comparison for different chunk sizes when one blinding stream is used. (a) Full header blinding is used. (b) 50% partial header blinding is used. (c) 20% partial header blinding is used.

Version	Header	After Convolution	Number of blinded bits	% of pkts blinded
F1	432 bits	876 coded bits	936 coded bits	3.86%
F2	432 bits	876 coded bits	952 coded bits	3.93%

Table 2.2: Number of bits blinded using *FOG*



(a)

(b)

(c)

Figure 2.15: Overhead comparison for different chunk sizes when two blinding stream is used. (a) Full header blinding is used. (b) 50% partial header blinding is used. (c) 20% partial header blinding is used.

Table 2.2 captures the number of "extra" bits that are blinded with *FOG* with a header of size 432 bits (packet size is 1514 bytes). After convolution coding the number of bits get doubled (coded bits $\approx$ twice the original bits). The reported results only consider the extra bits that are blinded from one data stream (other streams may also be blinded as a consequence of this, but we consider that as collateral i.e., an additional benefit). We see that the overall "fraction" of the packet that is blinded is less than (3.86% in the F1 case) with *FOG*. In the case of two blinding streams, a slight increase in the number of coded bits is observed. This is because, with two blinding streams, the likelihood of a stream being suspended increases. This reduces the possiblity that chunks from different streams have common header bits (recall Figure 2.4). Thus, there is a slight increase in the number of slots that carry blinding signals and thus, a slight increase in overhead.

2.7 Related Work

There is information theoretic work on coping with eavesdropping (e.g.,[68, 67, 60, 45, 97]). Unlike these efforts, we address the practical challenges in MIMO-based blinding.

Physical layer blinding attacks: In [98] and [90], a PHY layer passive attack that resembles known plaintext attacks in cryptography is identified. The attacker guesses part of the data transmitted by a sender (e.g., MAC header bits). Then, iteratively, it filters the received data and compares the known parts of the original data with the derived data (filter output) until the difference between the filter output and the expected original data is minimized. The work in [116] improves this attack capability via techniques to minimize the amount of the data required for the attack to be successful. However, the proposed attacks implicitly assume that the header information is transmitted at the beginning of the packet and known to the attacker. However, 802.11n/ac's scrambling and

interleaving processes distribute the header bits across the entire packet. More importantly, these attacks implicitly assume that a single blinding signal is used by the transmitter (as in traditional full packet blinding). Since a combination of blinding signals are used in *FOG*, these attacks are much harder if not impossible.

Friendly Jamming: With friendly jamming (e.g., [91, 86]), the idea is to generate jamming signals and control the decodability of these signals by sharing secret keys between the legitimate transmitter and receiver. The jamming signal is not decodable by any other receiver (eavesdropper) without the key. Friendly jamming, typically carried out by other nodes, increases the interference and thus, could reduce the overall network capacity. Furthermore, it requires co-ordination across nodes. Traditional Packet Blinding [12] essentially performs friendly jamming to mitigate eavesdropping, and does not require coordination across nodes. While both *FOG* and friendly jamming consume overhead, as shown in §2.6, the overheads with *FOG* are much smaller.

Other work: There is other work on using MIMO for deriving other security properties (e.g., [109, 112]). These papers however, do not target the obfuscation of traffic.

2.8 Discussion

Building an adaptive system: In this chapter, our primary goal was to showcase different approaches to obfuscate wireless packet streams using an antenna array. We show that if the payload is encrypted, header blinding or partial header blinding with 50 % of the header blinded, provide effective ways to thwart an eavesdropper with very low throughput penalties. In practice, we would envision a holistic system to function as shown in Figure 2.13c. Realization of this system however, has some challenges. Specifically, there could be a combination of streams, some with encrypted

payloads and others which are not encrypted. Here, one could potentially adopt a conservative approach and fully obfuscate all payloads (full packet blinding). A combination of sequentially transmitted blinding signals could be used similar to what was done with *FOG* for header blinding, to overcome "plaintext attacks against physical layer security". However, with such an approach, the throughput gains that we discussed earlier will not be achievable. Alternatively, one could group streams into two categories, (a) those with encrypted payloads and, (b) those with unencrypted payloads. Each stream will need to be transmitted with other streams within its group. In such a system, the throughput penalties are reduced for those streams belonging to group (b). Realization of such a holistic system is beyond the scope of this chapter and will be considered in the future.

Other Scheduling approaches: In our design of header blinding, we choose to suspend the different legitimate (desired) packet data streams and instead transmit a blinding signal(s) randomly as per a uniform distribution. This was done primarily with long term fairness in mind. To further provide fairness in the short term, one could perturb the scheduling to choose streams that have not been selected in the near past. One could also randomly vary the number of blinding streams (considered fixed in this work). We will consider these possibilities in future work.

Transceiver modifications: Both the transmitter and the receiver need to be *FOG*-aware (unlike with full packet blinding). Specifically, the bitmap is required for synchronizing the transceiver pair with regards to blinding signals. We believe this is worthwhile given (a) the drastic improvements in throughput compared to full packet blinding and (b) the resilience to plaintext attacks against PHY layer security.

The impact of higher modulation schemes: In all our experiments, we employ BPSK (binary phase shift keying) as the modulation scheme. If we use other higher modulation schemes,

the error rates will increase for both the attacker and the legitimate receiver. However, we believe that the behavioral results will be similar to what we have presented in this chapter. Further validation of this is left to future work.

Different MCSs may be used for different streams: In 802.11ac, different modulation and coding schemes (MCSs) can be used for different streams. In order to make our system compatible with this design, we can require each chunk, c, to have a size of L_c modulation symbols, instead of bits. Each symbol represents different number of bits, based on which modulation is used. For example, BPSK results in the mapping of one bit per symbol, while QPSK maps two bits per symbol. In this case, chunks are still aligned, and thus, the weight matrix W is derived in the same way as discussed in this chapter. The throughput of the system may potentially be degraded for high rate streams. However, we can reduce degradation by adjusting the chunk size, or modifying scheduling such algorithm so that high-rate streams have lower the probability of being suspended compared to low-rate streams. Finally, one can try to bundle transmissions of the same (or similar) rates into the same joint transmission using the multiple antennas; this way any throughput hit will be minimal.

***FOG* does not need to obfuscate packet size:** Even though the frame size is announced in the "PHY" layer header, *FOG* does not try to obfuscate packet size. Packet size alone is hard to use for the purpose of realizing the leakage of sensitive information. Without MAC or higher layer address information, an attacker cannot distinguish sequences of packets from different clients. In addition (although not considered), our work can potentially obfuscate packet sizes, by padding packet payloads.

Chapter 3

Off-Path TCP Exploits of the Challenge ACK Global Rate Limit

TCP and networking stacks have recently been shown to leak various types of information via side channels, to a blind off-path attacker [78, 40, 33, 61, 32, 113, 10]. However, it is generally believed that an adversary cannot easily know whether any two arbitrary hosts on the Internet are communicating using a TCP connection without being on the communication path. It is further believed that such an off-path attacker cannot tamper with or terminate a connection between such arbitrary hosts. In this work, we challenge this belief and demonstrate that it can be broken due to a subtle yet serious side channel vulnerability introduced in the latest TCP specification.

The two most relevant research efforts are the following: 1) In 2012, Qian *et al.*, framed the so called "TCP sequence number inference attack", which can be launched by an off-path attacker [78, 80]. However, the attack requires a piece of unprivileged malware to be running on the client to assist the off-path attacker; this greatly limits the scope of the attack. 2) In 2014,

Knockel *et al.*, identified a side channel that allows an off-path attacker to count the packets sent between two arbitrary hosts [61]. The limitation is that the proposed attack requires on average, an hour of preparation time and works at the IP layer only (cannot count how many packets are sent over a specific TCP connection).

In this work, we discover a much more powerful off-path attack that can quickly 1) test whether any two arbitrary hosts on the Internet are communicating using one or more TCP connections (and discover the port numbers associated with such connections); 2) perform TCP sequence number inference which allows the attacker to subsequently, forcibly terminate the connection or inject a malicious payload into the connection. We emphasize that the attack can be carried out by a purely off-path attacker without running malicious code on the communicating client or server. This can have serious implications on the security and privacy of the Internet at large.

The root cause of the vulnerability is the introduction of the *challenge ACK* responses [88] and the global rate limit imposed on certain TCP control packets. The feature is outlined in RFC 5961, which is implemented faithfully in Linux kernel version 3.6 from late 2012. At a very high level, the vulnerability allows an attacker to create contention on a shared resource, i.e., the global rate limit counter on the target system by sending spoofed packets. The attacker can then subsequently observe the effect on the counter changes, measurable through probing packets.

Through extensive experimentation, we demonstrate that the attack is extremely effective and reliable. Given any two arbitrary hosts, it takes only 10 seconds to successfully infer whether they are communicating. If there is a connection, subsequently, it takes also only tens of seconds to infer the TCP sequence numbers used on the connection. To demonstrate the impact, we perform case studies on a wide range of applications.

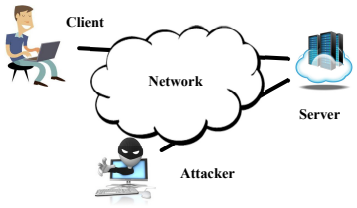


Figure 3.1: Threat model 1

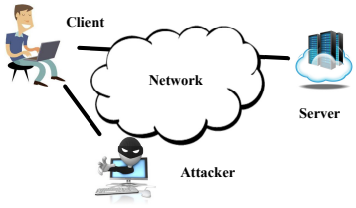


Figure 3.2: Alternative threat model

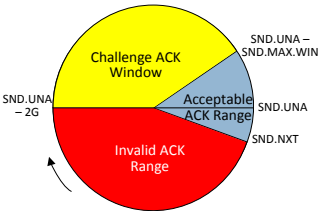


Figure 3.3: ACK window illustration

The contributions of the work are the following:

- We discover and report a serious vulnerability unintentionally introduced in the latest TCP specification which is subsequently implemented in the latest Linux kernel.
- We design and implement a powerful attack exploiting the vulnerability to infer 1) whether two hosts are communicating using a TCP connection; 2) the TCP sequence number currently associated with both sides of the connection.
- We provide a thorough analysis and evaluation of the proposed attack. We present case studies to illustrate the attack impact.
- We identify the root cause of the subtle vulnerability and discuss how it can be prevented in the future. We propose changes to the kernel implementation to eliminate or mitigate the side channel.

3.1 Background

Security was not the primary concern in the design of TCP. There have been many security patches over the years at both the specification and implementation level. Interestingly, most new specifications are well thought out and typically improve security. Unfortunately, as we discover,

one of the most recent specifications intended to improve security creates an even more serious vulnerability.

In this section, we first present the threat model that is being addressed in RFC 5961 and how the new specification is supposed to protect against blind in-window attacks. In the next section, we will show that how this specification introduces a new vulnerability.

Threat model: As illustrated in Figure 3.1, a realistic threat model for TCP is off-path attacks. There are three hosts involved: a victim client, a victim server and an off-path attacker. Any machine might act as the attacker in this model as long as its ISP allows the off-path attacker to send packets to the server with the spoofed IP address of the victim client. Alternatively, as shown in Figure 3.2, the off-path attacker is able to send packets to the client with the spoofed IP address of the victim server.

Blind in-window attacks: Under the above threat models, the most common attacks considered are "blind in-window attacks" where an off-path attacker sends spoofed TCP packets with guessed sequence numbers in an attempt to achieve DoS or data injection attacks. To succeed in such an attack, it is necessary to first know the target 4-tuple <src IP, dst IP, src port, dst port> of an ongoing TCP connection between a client and a server[1]. Once the correct 4-tuple is known, if the guessed sequence number of the spoofed packet happens to fall in the receive window, (called an in-window sequence number), one can in fact reset or inject *acceptable* malicious data into the connection. To be more precise, an in-window sequence number is one that satisfies the following condition, ($RCV.NXT \leq SEG.SEQ \leq RCV.NXT + RCV.WND$), where $SEG.SEQ$ is the guessed sequence number, $RCV.NXT$ and $RCV.WND$ are the sequence number of the next byte that the receiver expects to receive, and the receive window size, respectively. To carry out a blind

[1]This can be achieved, among other methods, through brute-force attempts.

attack, one typically needs to blast the entire sequence number space by sending a large sequence of spoofed packets. In this sequence, the sequence number of a packet is larger than that of its predecessor by a window size.

To defend against such attacks, RFC 5961 proposes several modifications on how TCP should process incoming packets, We highlight only the necessary details below.

3.1.1 Mitigating the Blind Reset Attack using the SYN Bit

An attacker might tear down an existing TCP connection by injecting SYN packets (TCP packets in which the SYN flag is set). This is because a valid SYN packet will cause the receiver to believe that the sender has restarted and thus, the connection should be reset.

In the former (pre-RFC 5961) Linux kernel versions, an incoming SYN packet is processed as follows:

- If the sequence number is outside the valid receive window, the receiver will send an ACK back to sender.
- If the sequence number is in-window, the receiver will reset this connection.

It is obvious that the attacker only needs a single SYN packet with an in-window sequence number to reset an ongoing TCP connection. Instead, RFC 5961 proposes modifications in processing the SYN packets as follows:

- If a receiver sees an incoming SYN packet, regardless of the sequence number, it sends back an ACK (referred to as a challenge ACK) to the sender to confirm the loss of the previous connection.
- If the packet is indeed initiated from the legitimate remote peer, it must have truly lost the previous connection and is now attempting to initiate a new one. Upon receiving the challenge

ACK, the remote peer will send a RST packet with the *correct* sequence number (derived from the ACK field of the challenge ACK packet) to prove that the previous connection is indeed terminated.

Hence, if the SYN packet is a spoofed one, it can no longer terminate a connection with an in-window sequence number.

3.1.2 Mitigating the Blind Reset Attack using the RST Bit

An attacker might also tear down the connection by injecting RST packets (TCP packets in which the RST flag is set) into an ongoing TCP connection.

In pre-RFC 5961 Linux kernels, just like in the SYN packet case, a RST packet can terminate a connection successfully as long as its sequence number is in-window. RFC 5961 suggests the following changes:

- If the sequence number is outside the valid receive window, the receiver simply drops the packet. No modifications are proposed for this case.
- If the sequence number *exactly* matches the next expected sequence number ($RCV.NXT$), the connection is reset.
- If the sequence number is in-window but does not exactly match $RCV.NXT$, the receiver must send a challenge ACK packet to the sender, and drop the unacceptable RST packet.

In the final case, if the sender is legitimate, it sends back a RST packet with the correct sequence number (derived from the ACK number in the challenge ACK) to reset the connection. On the other hand, if the RST is spoofed, the challenge ACK packet will not be observable by the off-path attacker. Therefore, the attacker needs to be extremely lucky to be able to succeed — only one out of 2^{32} sequence numbers will be accepted.

3.1.3 Mitigating the Blind Data Injection

An attacker might corrupt the contents of a transmission by injecting spoofed DATA packets. When a packet arrives, the receiver first checks the sequence number to make sure it is in-window; in addition, the ACK number will be checked. Pre-RFC 5961, the ACK number is considered valid as long as it falls in the wide range of [$SND.UNA - (2^{31} - 1)$, $SND.NXT$]; this is effectively half of the ACK number space. Here, $SND.UNA$ is the sequence number of the first unacknowledged byte. $SND.NXT$ is the sequence number of the next byte about to be sent.

RFC 5961 suggests a much smaller valid ACK number range of [$SND.UNA - MAX.SND.WND$, $SND.NXT$], where $MAX.SND.WND$ is the maximum window size the receiver has ever seen from its peer. This is illustrated in Figure 3.3. The reasoning is that the only valid ACK numbers are those that are (i) not too old (bytes that are recently sent) and (ii) not too new (receiver cannot ACK bytes that are yet to be sent). The remaining ACK values will be in the range of [$SND.UNA - (2^{31} - 1)$, $SND.UNA - MAX.SND.WND$), denoted as the *challenge ACK window*. Even though ACK numbers inside this window are still considered invalid, the specification requires the receiver to generate outgoing challenge ACKs in response to packets with such ACK numbers. Overall, this more stringent ACK number check does not eliminate, but helps dramatically reduce the probability that invalid data is successfully injected. Specifically, if the $MAX.SND.WND$ is small (which is typically the case for most connections), then the acceptable ACK window will be much smaller than the half of the ACK number space (as illustrated in Figure 3.3).

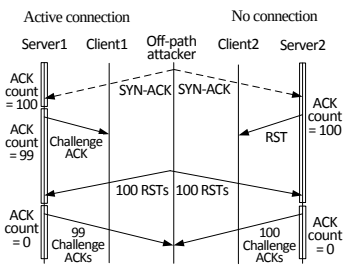


Figure 3.4: Connection (four-tuple) test

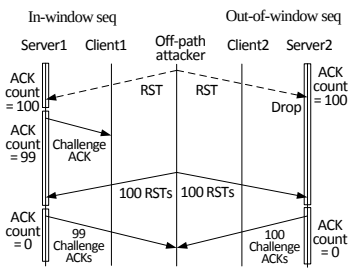


Figure 3.5: Sequence number test

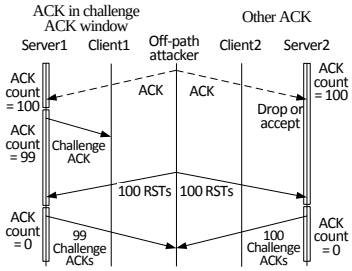


Figure 3.6: ACK number test

3.1.4 ACK Throttling

In general, as explained earlier, RFC 5961 enforces a much stricter check on incoming TCP packets; for example, it requires the RST packets to have an exact sequence number to actually reset the connection, whereas a "good enough" in-window value only triggers a challenge ACK. In order to reduce the number of challenge ACK packets that waste CPU and bandwidth resources, 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 (say, 1 second). The RFC clearly states "An implementation SHOULD include an ACK throttling mechanism to be conservative." Therefore, the Linux kernel has faithfully implemented this feature by storing the challenge ACK counter in a global variable shared by *all* TCP connections. This approach, unfortunately, creates an undesirable side channel, as will be elaborated. We emphasize that the RFC states that ACK throttling applies to only *challenge ACKs* and not to regular ACKs. This means that the challenge ACK counter is unlikely to be affected by legitimate ACK traffic as the conditions that trigger challenge ACKs are all considered rare or due to attacks.

3.2 Vulnerability Overview

The Linux kernel first implemented all the features suggested in RFC 5961, in version 3.6 in September 2012. The changes were backported to certain prior distributions as well. Faithfully following description in RFC 5961, the ACK throttling feature is specifically implemented as follows: a global system variable `sysctl_tcp_challenge_ack_limit` was introduced to control the maximum number of challenge ACKs generated per second. It is set to 100 by default. As this limit is shared across all connections (possibly including the connections established with the attacker), the shared state can be exploited as a side channel.

Assuming we follow the threat model in Figure 4.1, the basic idea is to repeat the following steps: 1) send spoofed packets to the connection under test (with a specific four-tuple), 2) create contention on the global challenge ACK rate limit, i.e., by creating a regular connection from the attacker to the server and intentionally triggering the maximum allowed challenge ACKs per second, and 3) count the actual number of challenge ACKs received on that connection. If this number is less than the system limit, some challenge ACKs must have been sent over the connection under test, as responses to the spoofed packets.

Depending on the types of spoofed packets sent in step 1, the off-path attacker can infer 1) if a connection specified by its four-tuple exists; 2) the next expected sequence number ($RCV.NXT$) on the server (or client); 3) the next expected ACK number ($SND.UNA$) on the server (or client). It is intriguing to realize that the three information leakages are enabled by the three (and only three) conditions that trigger challenge ACKs as described in §3.1.1, §3.1.2, and §3.1.3, respectively.

We elaborate below, the intuition on how the inference can be done in each case.

Connection (four-tuple) inference. Figure 3.4 shows 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. Dashed lines represent packets with spoofed IP addresses. In the figure, the initial SYN-ACK packet is spoofed so that it appears to come from the client. The counter for the number of challenge ACKs that can be issued (100 initially) is tracked and depicted on the timeline of the server.

The hope is that the initial spoofed SYN-ACK packet will hit a correct four-tuple that corresponds to an active connection between the client and the server. In such a case (the left of Figure 3.4) the server will reply with a challenge ACK[2] (in accordance with the countermeasure proposed to defend against blind SYN packet injection as described in §3.1.1). At the same time, this will reduce the global challenge ACK count from 100 to 99. In the case where the spoofed SYN-ACK does not hit a correct four-tuple (on the right of the figure), the server will simply reply with a RST back to the corresponding client (as per TCP standards).

The attacker will then send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count (this behavior is described in §3.1.2). In the *active connection* case, since the challenge ACK count is 99, the attacker can now observe only 99 challenge ACKs. In the *no connection* case, the attacker can observe 100. The difference in the number of challenge ACKs effectively leaks the information about whether a tested four-tuple corresponds to an active connection or not.

Sequence number inference. Assuming the attacker has already identified a four-tuple that corresponds to an active connection between the client and server, the off-path attacker now needs to guess a valid sequence number that is considered acceptable by the server. Figure 3.5 shows the sequence of packets that an attacker can send to distinguish between the cases of (i) *in-window*

[2]The effect is the same as sending a spoofed SYN. However, sending a SYN-ACK is generally more stealthy.

and (ii) *out-of-window sequence number*. In the first case where the spoofed RST packet has an in-window sequence number (but not the next expected sequence number), as per the countermeasure proposed to defend against blind RST packet injection as described in §3.1.2, a challenge ACK is triggered and this reduces the global challenge ACK count from 100 to 99. In the second case where the sequence number falls outside of the window, no challenge ACK will be generated (the global challenge ACK count remains at 100).

Similar to connection inference, the attacker will now send 100 non-spoofed in-window RST packets to exhaust the challenge ACK count. Once again, based on how many challenge ACKs are received, the attacker can tell if the guessed sequence number in the spoofed RST, is in-window or out-of-window.

ACK number inference. After an in-window sequence number of an active connection is identified, the attacker now will need to guess a valid ACK number that is considered acceptable by the server. Figure 3.6 shows the sequence of packets that an attacker can send to differentiate the cases of (i) *ACKs in challenge ACK window* and (ii) *other ACK numbers*. In the first case where the spoofed ACK packet has an ACK number in challenge ACK window (but with an in-window sequence number), the server will reply with a challenge ACK, in accordance with the countermeasure proposed to defend against blind data packet injection (as described in §3.1.3). Following the same procedure as before, an attacker can infer if the guessed ACK number falls in the challenge ACK window. As will be described in §3.4.2, this helps the attacker to eventually identify the $SND.NXT$ on the server.

It is worth noting that once both the sequence number and ACK number acceptable by the server are inferred, an attacker can determine the sequence number and the ACK number acceptable

by the client as well. This is because the $RCV.NXT$ and $SND.NXT$ on the server are basically equivalent to $SND.NXT$ and $RCV.NXT$ on the client [85, 49]. In practice, if the victim connection has ongoing traffic, the inferred sequence and ACK number may shift as the attack is in progress. We discuss such cases in §3.5.

An alternative approach for sequence number inference. In some cases a large number of RST packets observed in a short period time may be considered abnormal. Firewalls may even rate limit RST packets on a per-connection basis. In order to alleviate this, one can in fact replace RST packets with ACK packets, which are likely to stay under the radar. As shown in Figure 3.3, a challenge ACK will be sent when ACK number is in *challenge ACK window* while sequence number is in-window. Since the *challenge ACK window* space is at least $1/4$ of the entire 4G of the ACK number space, one can send 4 packets with ACK numbers 0, 1G, 2G, and 3G respectively and at least one packet will trigger a challenge ACK if the guessed sequence number is in-window. To understand why the *challenge ACK window* is at least this large, we first point out that the maximum receive window size is 1G with the TCP window scaling option (RFC 7323), which means that SND.MAX.WIN cannot be larger than 1G. Therefore, according to definition of the challenge ACK window described in §3.1.3, it is at least 1G as well. Given this, every spoofed RST packet sent earlier for sequence number inference is replaced by four ACK packets, which is less efficient but still effective. We have implemented and tested this alternative approach for sequence number inference. However, to simplify the description, we assume the use the original sequence number inference with RST packets in the subsequent sections.

3.3 Off-Path Connection Reset Attack

In the previous section, we illustrate how the global challenge ACK rate limit can theoretically leak information about an ongoing connection to an off-path attacker. In this section, we focus on how to construct a practical off-path connection reset attack that succeeds when a spoofed RST arrives with a matching sequence number of $RCV.NXT$. This requires an attacker to successfully carry out both *connection (four-tuple) inference* and *sequence number inference*. As will be discussed, to construct a realistic attack, several practical challenges need to be overcome. We assume the threat model to be the one in Figure 3.1 throughout the section, but the attack works with the alternative threat model (Figure 3.2) as well.

Goals and constraints. The main goal of the attack is to *quickly* and *reliably* conduct the sequence number inference and use it to reset an ongoing connection. The faster the attack succeeds, the more potent the DoS effect will be. However, the extent of the effect is subject to two practical constraining factors: (i) The bandwidth may be limited between the attacker and the victim (either server or client). (ii) Packet loss may occur between the attacker and victim, especially when they are far away. In this section, we focus only on designing fast probing schemes with given bandwidth constraints and leave the strategy to deal with packet loss to §3.5.

3.3.1 Time Synchronization

Challenge: As mentioned in §3.2, the challenge ACK rate limit is on a per second basis. In other words, the counter for the number of challenge ACK packets that can be issued, gets reset each second. Therefore, it is critical that in each cycle, all the spoofed and non-spoofed packets sent from the attacker arrive within the same 1-second interval, at the server.

One naive solution is that the attacker sends all those packets in a very short period (say, 10 ms), to ensure that the likelihood that they arrive within the same 1-second interval is high. Unfortunately, in practice, this solution does not work well since (i) many factors influence packet delays and thus, the gaps between packet arrival times at the receiver, might be much larger than the gaps in their transmission times, (ii) such bursts of traffic are likely going to experience congestion and packet loss. Thus, it is best for the attacker to synchronize with the clock on the server, so that the attacker can spread the traffic over the 1-second interval, without worrying that some packet arrivals may cross the boundary between two 1-second intervals.

The most common way to synchronize time between two machines is using the Network Time Protocol (NTP). But in practice, the attacker does not know if the server uses NTP, or to what NTP server it connects to; thus, it is not a reliable solution.

Solution: We propose a time synchronization strategy based on the very side channel introduced by the challenge ACK rate limit. The idea is to send more than 200 in-window RST packets spread out evenly in one second and check if we can see more than 100 challenge ACKs; if so, this indicates that we have crossed the boundary between two one second intervals (and have therefore not synced with the server yet). We then adjust the timing for next round of probing (shift it just enough) until we receive exactly the 100 challenge ACKs; in this case, we have succeeded in synchronizing with the server clock.

The reason we choose 200 packets is two-fold: 1) We are able to trigger at most 200 challenge ACKs no matter how many RST packets we send. These 200 challenge ACKs are triggered only when half of the RST packets arrive before the start of a new 1-second interval and half arrive after. 2) By evenly spreading the 200 packets over a 1-second window, i.e., sending one packet every

5ms, allows us to adjust the timing of the next round probing with the finest granularity. Specifically, we show that the time synchronization can be done in at most three rounds of probing in an ideal case (without packet losses).

Round 1: As described before, the attacker sends 200 in-window RST packets to the server evenly spread out over a 1-second window. The attacker then listens and counts the number of received challenge ACK packets. This value is stored as n_1. Here, the attacker listens for incoming packets for 2 seconds conservatively, before sending any additional packets to make sure a 1-second interval on the server has elapsed. Note that apart from the 200 RST packets, no other packet is sent to the server in this interval. If n_1 equals 100, it means that all 200 RST packets all arrive in the same 1-second interval on the server, thereby indicating that we have already synchronized with the server. Otherwise, it must be true that $n_1 > 100$, and the attacker will proceed to the next round.

Round 2: The attacker waits for 5ms (shifting the start time of the probes by 5ms) and repeats the same process as in the first step. The number of received challenge ACK this time is stored as n_2. If n_2 equals 100, the synchronization is done. Otherwise, the attacker proceeds to round 3.

Round 3: By comparing n_1 with n_2, the attacker can determine the final move to be synchronized. Specifically, we provide the following reasoning to support the decision. Assume that in step 1, x RST packets arrive in the first 1-second interval on the server, and y RST packets arrive in the second 1-second interval; note that $x + y = 200$. Similarly, in step 2, there are $(x - 1)$ and $(y + 1)$ RST packets arrive in the first and second 1-second intervals respectively, since in step 2 the attacker time shifts its probes by a period of 1 sub-interval. Thus, $n_1 = min(x, 100) + min(y, 100)$ and $n_2 = min(x - 1, 100) + min(y + 1, 100)$.

(i) If $n_2 \geq n_1$: Let us assume that $y \geq 100$ and $x \leq 100$; then $n_1 = min(x, 100) + min(y, 100) = x + 100$, and $n_2 = min(x - 1, 100) + min(y + 1, 100) = (x - 1) + 100 < n_1$, which contradicts the assumption that $n_2 \geq n_1$; thus $y < 100$ and $x > 100$. With these conditions, $n_2 = 100 + (y + 1) = 100 + (200 - x + 1)$, or $(x - 1) = 300 - n_2$. In step 2, $(x - 1)$ RST packets arrive in the first 1-second interval on the server; thus, the attacker has to wait for $(x - 1)$ sub-intervals, i.e., $(300 - n_2).\frac{1}{200}$ seconds to synchronize her time interval with the server.

(ii) If $n_2 < n_1$: With the same reasoning, the attacker knows that $x < 100$ and $y > 100$. In this case, $n_2 = (x - 1) + 100$; thus, the attacker has to wait $(n_2 - 100)$ sub-intervals, or $\frac{n_2 - 100}{200}$ seconds to synchronize her time interval with the server.

If no packet loss occurs (which is likely due to the small number of packets sent every second), then the three rounds are enough to complete the synchronization process. To handle the rare event that packet loss may occur, we double check that the synchronization was successful by sending another round of 200 RST packets. If it is inconsistent with the previous round, we start over. As will be discussed later, such cases were almost never seen in our experiments.

3.3.2 Connection (Four-tuple) Inference

After time synchronization, the attacker can successfully launch subsequent attacks by knowing the boundaries between the 1-second intervals. The first step is "four-tuple inference", wherein the attacker determines if a connection is established between the client and the server. As mentioned in §3.1.1, the receiver will send back a challenge ACK (regardless of the sequence number of the packet) when a packet with a SYN flag set, arrives.

In §3.2, we discussed how this behavior can be exploited to determine whether or not a specific four-tuple is currently active. Basically, for each four-tuple in question, the attacker needs

Algorithm 1 Binary search for source port number

1: $left$ = left boundary of the port range
2: $right$ = right boundary of the port range
3: **while** $left < right$ **do**
4: $mid = (left + right)/2$
5: **for** $i = mid$ **to** $right$ **do**
6: Send a spoofed SYN packet with i as the client port number
7: **end for**
8: Send 100 RST packets on the legitimate connection
9: Wait until the end of the 1-second interval, count the number of received challenge ACK packets
10: **if** received ACK packets = 100 **then**
11: $right = mid - 1$
12: **else**
13: $left = mid$
14: **end if**
15: **end while**
16: return $left$; //the correct port value

to send a spoofed SYN-ACK packet (a TCP packet in which the SYN and ACK flags are set) with <srcIP = clientIP, dstIP = serverIP, srcPort = X, dstPort = serverPort>. The above assumes both the client and server IP addresses are known. In addition, the server port is assumed to be publicly known according to its service type. Therefore, the only unknown is the source port the client uses. The maximum possible port range is 2^{16} = 65536, and the default range on Linux is only from 32768 to 61000.

A naive approach is to test each port number at a time per second, as depicted in Figure 3.4, which, in the worst case, requires hours to complete. Therefore, a practical attack requires the attacker to test several port numbers in a second. Let us denote the maximum number of spoofed

packets that can be sent in one second by n (constrained by network bandwidth). If n is large, one can search for the port number using a binary-search-like algorithm, the pseudo-code of which is shown in Algorithm 1. Specifically, assuming n is larger than 32767, in the first round the attacker can test the port range from 32768 to 65535 (the most likely half) in a 1-second interval. If the actual port number falls in the range, then the challenge ACK observed by the attacker at the end of the interval will be 99 (one goes to the victim). If the actual port number *does not* fall in this range, the observed number of challenge ACKs will be 100. In either case, the attacker can narrow down the search space by half and proceed to the next round of search.

An even better strategy is to divide the search space into multiple bins and probe them together in the same round. That way, one can eliminate $\frac{n-1}{n}$ of the search space. A similar multi-bin search strategy is used for sequence number inference in (§3.3.3).

In cases where n is smaller than 32767 (due to bandwidth constraints), the best the attacker can do is to simply try as many port numbers as possible in each round. The binary search or multi-bin search can be applied later when the search space becomes small enough.

3.3.3 Sequence Number Inference

As discussed in §3.1.2, the receiver generates a challenge ACK in response to a RST packet that contains an in-window sequence number which does not match exactly the expected value. The related Linux kernel code is shown in Figure 3.7; the `tcp_sequence()` function returns true if the sequence number is in-window, and false if it is out-of-window. In the latter case, the packet will simply be dropped. When the sequence number is in-window and the packet has the

```
if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
...
        goto discard;
}
if (th->rst) {
        if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
                tcp_reset(sk);
        else
                tcp_send_challenge_ack(sk, skb);
        goto discard;
}
```

Figure 3.7: Logic of handling an incoming packet with RST flag in latest Linux kernels

RST flag set, its sequence number is analyzed further. As we can see, the connection is terminated only when the sequence number matches $RCV.NXT$; otherwise, a challenge ACK is sent.

The main difference between port number inference and sequence number inference is that the attacker does not need to check every possible sequence number to trigger a challenge ACK. Therefore, the attacker can divide the sequence number space into blocks whose sizes are equal to the receive window size, and probe with a guessed sequence number in each block to determine which sequence numbers fall in the receive window. Theoretically, an attacker can apply the same binary search algorithm used in connection inference. This process is illustrated in Figure 3.8. In the first round of probing, the attacker can probe the right half of the sequence number space — (2G, 4G). If any of the spoofed RST packets triggers a challenge ACK, the attacker will observe less than 100 challenge ACKs at the end of the 1-second interval. If there are exactly 100 challenge

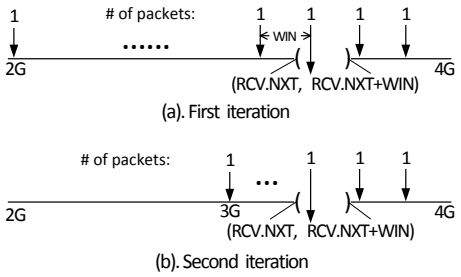


Figure 3.8: Binary search for sequence number illustration

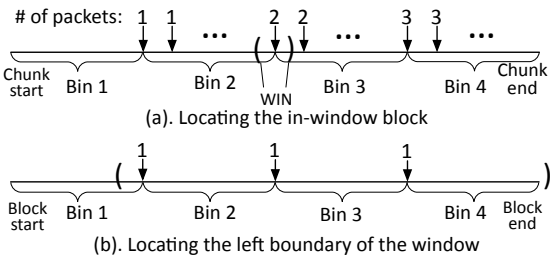


Figure 3.9: Multi-bin search for sequence number illustration

ACKs observed, it indicates that the receive window is on the left side of the search space. In either case, in the second round, the attacker knows "which half the receive window belongs to." Let us say that the receive window is in the right half. The attacker would then divide the search space of (2G, 4G) into (2G, 3G) and (3G, 4G). Similar to the first round, only (3G, 4G) needs to be probed in order to determine the part that contains the receive window. This search will eventually stop after 32 rounds exactly (because the sequence number is 32-bit).

However, in practice, the sequence number search space is significantly larger than port number space. Let us consider a receive window size of 12600. This leaves the attacker 340870 possible blocks to search through. If the attacker were to transmit this many packets in one second, the bandwidth requirement would be around 150Mbps, which is extremely high. Likely, the attacker will have to perform a linear search by attempting to search as many blocks as allowed by bandwidth in one second.

Dealing with unknown window sizes: Ideally, the block size should be determined by the window size of the target connection, i.e., the server's receive window size. In reality, however, an *off-path* attacker cannot observe the window size. If the attacker chooses a smaller window size (compared to the actual window size), the attack will send more packets unnecessarily and take

more time. On the other hand, if the guessed value is larger than the actual value, the attacker might miss the correct window of sequence numbers while traversing consecutive blocks. Thus, there is an inherent trade-off between the success rate and the cost incurred (in terms of time and bandwidth) of the attack. Even if the attacker can come up with a correct receive window size at one particular time, the size can change over time.

Our solution is to use a conservative estimate of the window size as the block size in the beginning and update it later given proper feedback. The conservative window size is determined by the initial window size advertised by the server in the SYN-ACK packet. By surveying Alexa top 100 websites, we find that the average initial receive window size is 26703. This window size is the lower bound as the window typically grows after the connection is established. To observe the initial window size, the attacker simply attempts to establish a valid (non-spoofed) TCP connection with the server. This strategy works because a server typically uses the same initial receive window size for all clients. Such a conservative estimate of window size may force the attacker to send more packets, but it at least will guarantee success. We will also discuss how to update the window size dynamically during the search process.

Next, we elaborate the design of sequence number inference:

- **Step 1 – Identify the approximate sequence number range.** Let us assume that the attacker, in n blocks, can send n spoofed packets per second (n is on the order of thousands in our experiments). We call such n consecutive blocks a chunk. The guessed sequence number is always chosen to be the first sequence number within a block. If at the end of the 1-second interval, the attacker observes 100 challenge ACKs, then the attacker proceeds to the next chunk, i.e., the next n consecutive blocks. If the attacker observes less than 100 challenge ACKs, it indicates that

the receive window is within the chunk that was just probed. The attack can now proceed to step 2. Note that if the number of observed challenge ACKs is less than 99, it indicates that the initially estimated window size (block size) is too small.

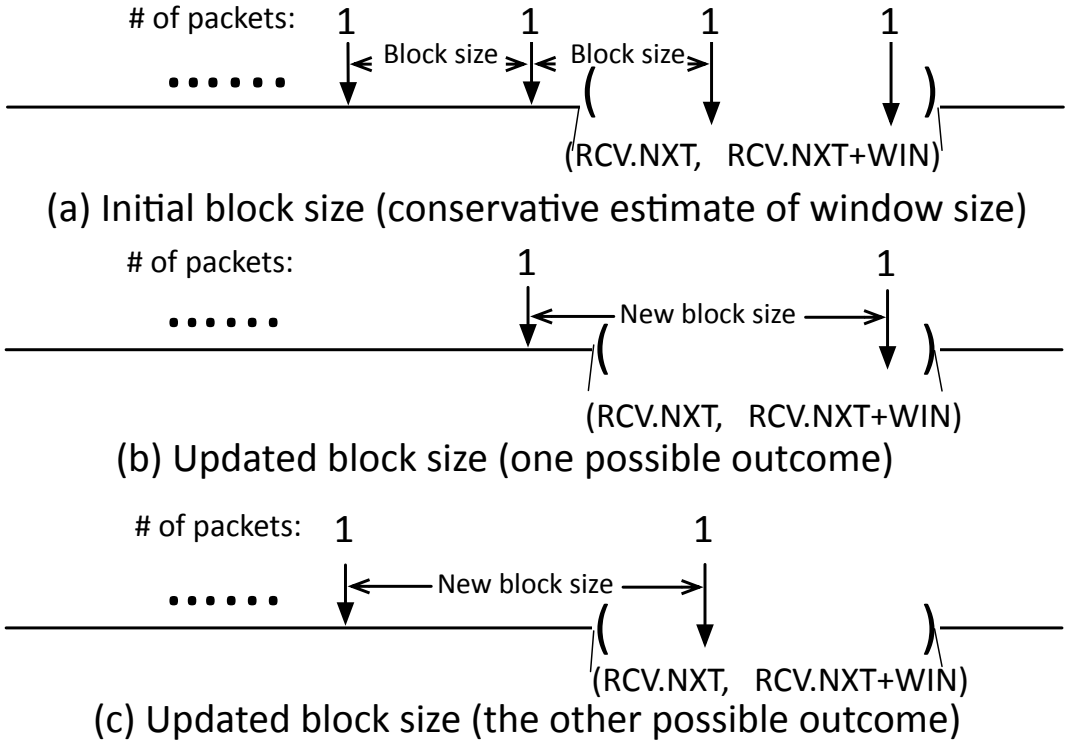


Figure 3.10: Window size estimate and adjustment

For example, as illustrated in Figure 3.10(a), if there are two blocks whose beginning sequence numbers are inside the actual receive window, then the number of observed challenge ACKs will be 98; this indicates that the actual window size should be approximately twice the estimated window size (initial block size). We therefore update the block size to be twice as much in the subsequent search steps. The two possible outcomes are shown in Figure 3.10(b) and Figure 3.10(c).

- **Step 2 – Narrow down the sequence number space to a single block.** From step 1, we know that the receive window is within a chunk. We now further narrow down the search

space to an exact block within the chunk. Note that we have now updated the block size so that there will be one and only one block that can trigger challenge ACKs. To locate the exact block, the same binary search strategy outlined in Figure 3.8 can be used except that the search space now is dramatically reduced after step 1.

The located block has a beginning value which, is an in-window sequence number; therefore, one of the following is true: (i) its beginning value is the correct sequence value; or (ii) the correct sequence value is in its left neighboring block. In the first case, since the sequence number matches the $RCV.NXT$, the spoofed RST packet can already terminate the connection. In the second case, the attacker performs an additional search in the left neighboring block (see Step 3).

- **Step 2 (optimized version) – Identify the correct sequence block using multi-bin search.** With the previous assumption that the attacker can send n spoofed packets per second, with a binary search, the first round requires only $\frac{n}{2}$ packets (as we divide a chunk into two halves initially). The second round requires only $\frac{n}{4}$ packets and so on. As we see, the number of packets sent in each round reduces quickly. This is not an efficient use of the network bandwidth. We show that it is possible to speed up the search process by sending more packets per round (still at most n per round).

The idea is, instead of dividing the search space into two halves in each round, we can divide the space into multiple bins and probe them simultaneously. This is illustrated in Figure 3.9(a) where 4 bins are present in a chunk. Each bin here holds an equal number of blocks. To determine which bin the receive window falls in, the attacker sends a different number of spoofed RST packets in each bin. In the example, he sends 1 RST packet per block in the 2nd bin, 2 RST packets per block in the 3rd bin, and 3 RST packets per block in the 4th bin. Since the receive window can fall

into one and only one of the bins, the attacker can determine which bin it is in, by observing how many challenge ACKs are received at the end of the 1-second interval. If there are 100 challenge ACKs received, it indicates that the receive window is in the 1st bin (since no RST packets were sent in the 1st bin). Receipt of 99 challenge ACKs indicates that the receive window is in the 2nd bin, etc.

Note that the more bins we have, the faster we can narrow down the sequence number space. However, the number of bins chosen for each round is constrained by n. The larger the n, the more the bins that can be created. The number of bins is also capped at 14, given that the number of spoofed packets may already exhaust the 100 challenge ACK counter in one round $(0 + 1 + 2 + ... + 13 = 91)$.

- **Step 3 – Find the correct sequence number using binary search.** Now we are sure that $RCV.NXT$ is within a specific block, we need to locate its exact value. To achieve the goal, another modified binary search strategy is used here. The observation is that the correct sequence number ($RCV.NXT$) is the highest value in the block, such that any spoofed RST packet with a sequence number less than it will not trigger a challenge ACK packet. It is worth noting that we may not realize which value is the correct sequence number until the connection is terminated, as all the probing packets are RST packets.

- **Step 3 (optimized version) – Find the correct sequence number using multi-bin search.** Similar to the previous multi-bin search, the attacker can divide the single block into many small bins and probe them simultaneously. All bins before the left boundary of the receive window ($RCV.NXT$) will not trigger any challenge ACKs; the ones after will. Thus, in this step attacker only sends one spoofed packet per bin and accumulates all the challenge ACKs received

from right to left (See Figure 3.9). If the attacker sees (100-X) challenge ACKs at the end of the 1-second interval, it indicates that X probed bins are after $RCV.NXT$. In Figure 3.9, let us say we divide the block into 4 bins. After probing them, the number of observed challenge ACK will be 97 because 2nd, 3rd, and 4th bins turn out to be after $RCV.NXT$. Note that if the observed challenge ACK is 100, it indicates that the correct sequence number is somewhere inside the 4th bin (but not its beginning value).

Similar to the previous multi-bin search, the number of bins chosen for each round is constrained by n. In addition, the number of bins is always capped at 100, as the spoofed packets may exhaust the limit of 100 challenge ACK count.

The RST off-path TCP attack is successfully launched after the above three steps. The exact number of probing rounds depends on the available bandwidth, and will determine the time it takes to finish the attack. We will evaluate this in §3.6.

3.4 Off-Path Connection Hijacking Attack

In this section, we discuss how an off-path attacker can hijack an ongoing connection and inject spoofed data. The methodology used to inject data into the client or to the server are similar; thus, without loss of generality, we exemplify the attack targeting the server. First, we describe the challenges that the attacker will need to overcome; subsequently, the entire attack process is described in detail.

3.4.1 Challenges and Overview

The attacker will experience obstacles that are similar to those associated with launching an off-path reset attack. In addition, the following additional challenges need to be addressed.

Preventing unwanted connection reset. As described in §3.3.3, the RST packets with in-window sequence numbers are leveraged towards identifying the next expected sequence number on the connection. However, with that process, sending a RST packet with the exact, expected sequence number ($RCV.NXT$) to the server will terminate the TCP connection; this is not the goal of the hijack attack. The challenge is thus, to infer $RCV.NXT$ without causing connection termination.

Identifying both the sequence number and ACK number. In order to trick the server into believing that the injected data is valid, and sent from the server, the attacker needs to know both the correct sequence number ($RCV.NXT$) and the acceptable ACK range on the server side of the connection. The latter is typically a fairly small range as discussed in §3.1.3.

At a high level, our design of the attack consists of the following steps: First, the attacker finds an in-window sequence number on the server using the techniques described in §3.3.3. Based on this, the attacker will be able to guess the range of acceptable ACK values that trigger challenge ACKs. The range of these acceptable values (ACK window) can be used to identify the highest acceptable ACK number, i.e., $SND.NXT$, on the server. We will show next that obtaining this ACK number then allows the attacker to infer the exact expected sequence number on the server without resetting the connection.

3.4.2 Inferring Acceptable ACK Numbers

Assuming an in-window sequence number is already inferred, we now discuss how an attacker can infer the next ACK number, $SND.UNA$, which is expected by the server. As illustrated in Figure 3.3, an incoming data packet is accepted if the ACK number is in the range of [$SND.UNA - MAX.SND.WND$, $SND.NXT$]. If not, the receiver will respond with a challenge ACK packet, if the ACK number is in the range of [$SND.UNA - (2^{31} - 1)$, $SND.UNA - MAX.SND.WND$); this range is called the *challenge ACK window*. It is obvious that $SND.UNA$ can be computed if one can successfully infer the left boundary of the challenge ACK window, $SND.UNA - (2^{31} - 1)$. This in turn can be found using the following approach.

Step 1: Identify the challenge ACK window position. In RFC 1323, by using the window scaling option, the maximum receive window size can be extended from 2^{16} to a maximum of $2^{30} = 1G$. Thus, the $MAX.SND.WND$ cannot be larger than 1G. Accordingly, the *challenge ACK window* size is between 1G and 2G, which is one quarter of the entire ACK space size. Because of this, we divide the entire ACK space into 4 bins and probe each bin to check which bin(s) the *challenge ACK window* falls in. In our implementation, we probe the first value of each bin, i.e. 0, 1G, 2G, 3G. We know for certain that either one or two bins can trigger challenge ACK packets. Therefore, we send different number of packets for each bin to differentiate the resulting cases. One strategy is to send one packet at ACK number 0, two packets at 1G, four packets at 2G, and 8 packets at 3G. For instance, if the number of observed challenge ACKs is 94 (6 missing), we can infer that both ACK number 1G and 2G have triggered challenge ACKs. If the number of observed challenge ACKs is 96 (4 missing), only ACK number 2G has triggered challenge ACKs. We can then easily determine the "left-most" bin whose beginning value falls in challenge ACK window.

Step 2: Find the left boundary of the challenge ACK window Now the problem is, given the bin located in the previous step, we need to identify an ACK number in the left neighboring bin, such that it is the "left-most" value (in the circular sense) that can still trigger challenge ACKs. This is a problem that can be solved in a similar way to the last step of sequence number inference using multi-bin search (§3.3.3).

Finally, when the left boundary of the challenge ACK window ($SND.UNA - (2^{31} - 1)$) is found, an acceptable ACK value ($SND.UNA$) is trivially computed.

3.4.3 Identify the Exact Sequence Number

To locate $RCV.NXT$ without resetting a connection, we leverage the knowledge learned about the various ACK number ranges. The idea is that, instead of sending spoofed RST packets (which may terminate a connection), the attacker can send spoofed *data* packets with ACK numbers that fall in the challenge ACK window and thus, intentionally trigger challenge ACKs (if the sequence number is in-window). Combined with the fact no challenge ACK will be triggered if the guessed sequence number is before $RCV.NXT$ (considered old packet and dropped), $RCV.NXT$ can be located as the "left-most" value that can trigger challenge ACKs. The search process is in fact similar to the last step in sequence number inference except that we now use spoofed data packets.

Now that the attacker knows both the $RCV.NXT$ and $SND.UNA$ on the server, it is trivial to inject legitimate-looking data packets that will be accepted by the server. Further, it is also trivial to inject legitimate-looking data packets to the client because the $RCV.NXT$ on the server is effectively the $SND.UNA$ on the client, and the $SND.UNA$ is the $RCV.NXT$ on the client (assuming no traffic is in flight). In §3.6.2, we will present a case study on how a web service can be hijacked by a completely blind off-path attacker.

3.5 Other Practical Considerations

We have fully implemented the attacks described in §3.3 and §3.4. In §3.6, we will evaluate the effectiveness and efficiency of the attacks extensively. In this section, we outline a few practical considerations that need to be handled.

Detecting and handling packet loss. So far, we have assumed that spoofed connections will not incur packet loss and the challenge ACK side channel has no noise. However, in reality, even if the number of packets sent per second is chosen conservatively (well below bandwidth constraints), there is still no guarantee that packet loss will not occur, and a host may legitimately generate challenge ACKs that are not triggered by the attack. They exhibit the same effect to the attacker — the number of observed challenge ACKs will be smaller than expected. In this chapter, we call them both packet loss for convenience. We address packet loss based on the two following principles: 1) when in doubt, repeat the probes; 2) add redundancy in the probing scheme to proactively detect packet loss.

In the initial step of the sequence number search, if packet loss occurs, the number of observed challenge ACKs may reduce to 99; the attacker thus, may incorrectly conclude that a chunk that contains the receive window is located. This will affect all subsequent search steps. Therefore, every time when a "plausible" chunk is detected, we repeat the probe on the same chunk. The search will proceed to step 2 only when both rounds return exactly 99 challenge ACKs (no more, no less).

In step 2 and step 3 of the sequence number search, we add redundancy to actively detect packet loss so that we repeat only the round of probing that experienced packet loss. The idea is similar to using parity bits. In each round, instead of allowing the number of observed challenge

ACKs to be any value equal to or below 100, we can construct the probing packets such that only odd number of challenge ACKs will be considered a valid outcome. If an even number of challenge ACKs is received, packet loss must have happened. This strategy can be visualized by referring to Figure 3.9(a). Instead of sending 1, 2, or 3 packets per block for each bin, we will send 1, 3, and 5 packets per block for each bin. This means that if the receive window falls in 2nd bin, the number of challenge ACKs will be 99; if the receive window is in 3rd bin, the number of challenge ACKs will be 97, etc.

Both schemes are implemented and shown to be very effective in cases where the network conditions between the attacker and the victim are poor.

Moving receive window and challenge ACK window. So far, we have assumed that the connection is relatively idle, and the window does not change while the inference is in progress. This is likely to be the case in many real world scenarios, especially with long-lived connections. One example is the push notification connections on mobile platforms [3]. They are idle most of the time until a new push notification arrives. Even when a connection is not idle at one point, it is likely to become idle at some point and become more susceptible to the attack. Moreover, the traffic activity will mostly be concentrated on either uplink and downlink, rarely both. Typically, downlink traffic dominates; therefore, the attacker targeting at resetting the connection on the server side will experience less difficulty (client's sequence number increases very slowly). Tor network connections are also candidates as the end-to-end throughput is typically very low.

To support sequence number inference against (slow) moving receive windows, we implement a simple strategy which conducts a brute-force style sequence number guessing. Specifically, once a "left-most" in-window sequence number is inferred (which may become invalid in the next

interval due to the ongoing activities), we send 20,000 RST packets with sequence numbers, with offset 1, 2, ..., 20,000 to the valid sequence number. As will be shown in §3.6.1, for low-activity connections, this strategy works well. We leave the exercise to come up with a strategy to target connections with heavier traffic to future work.

Per-connection rate limit. Since the Linux kernel version 4.0 (released in Apr 2015), in addition to the global challenge ACK rate limit, a per-connection rate limit was introduced. The idea is to reduce the impact of potential ACK loops [8] that may occur if client and server are de-synchronized. Theoretically, the per-connection rate limit provides an isolation between the victim connection and the attacker connection, and the side channel should be eliminated completely. For instance, even if the challenge ACK count limit is reached for the victim connection, it does not affect the limit on the attacker connection at all.

However, interestingly, the per-connection rate limit only applies to SYN packets or packets without any payload. The comment in the Linux kernel states "Data packets without SYNs are not likely part of an ACK loop", hinting that such packets do not need to be governed by the per-connection rate limit. It is evident that the developers assumed a benign scenario instead of an adversarial one. To get around this restriction, we simply send spoofed packets with a single byte of payload. For the spoofed SYN-ACK packets though, it is impossible to bypass the per-connection rate limit. Unfortunately, upon a closer look at the implementation, when a per-connection challenge ACK is sent out, it is also counted towards the global challenge ACK limit. Therefore, it is still possible to infer that the four-tuple of an ongoing connection has been guessed correctly by observing only 99 challenge ACKs at the end of the 1-second interval. In practice, the per-connection rate limit is 1 packet every 0.5 second, which does allow the attacker to proceed with the binary

search approach outlined in §3.3.2. We have verified experimentally that it does work against the latest Linux kernels with per-connection rate limit.

Configurable maximum challenge ACK count. For simplicity, throughout the chapter, we assume the challenge ACK count to be 100, which is the default value. Our test on a variety of Linux operating systems also confirmed the result. However, as proposed in RFC 5961, this value is configurable by a system administrator. According to the specification, the flexibility is provided to allow the tradeoff between resource (bandwidth and CPU) utilization and how fast the system cleans up stale connections. Fortunately, the exact configured value can be inferred quite easily with some simple steps (as long as it is not excessively large). After establishing a legitimate connection to the server, the attacker can send many RST packets, e.g., 1000 packets which is much larger than default value of 100, with in-window sequence values to trigger as many challenge ACKs as possible. The packets are sent in a very short period of time (say, 200 ms[3]) to increase the likelihood that they end up in the same 1-second interval. The attacker then counts the total number of challenge ACKs returned. Finally, the attacker can wait for a short amount of time and repeat the process one more time to verify the number of received challenge ACK packets is the same; that value would be the actual limit set by the server. Note that this is only a one-time effort for each target.

3.6 Evaluations

To showcase the effectiveness of our attacks, we next evaluate them in terms of metrics such as success rate and the time to succeed.

[3] Sending 1000 packets within a 200ms window will rarely cause congestion or packet loss.

3.6.1 Connection Reset Case Studies

There are two sets of experiments reported in this section viz., where (i) we reset an SSH connection and (ii) perform a Tor connection reset.

Experimental setup. For the SSH experiments, we use a Ubuntu 14.04 host on the University of California - Riverside campus as the victim client. The victim SSH server is one of the instances we create on Amazon EC2 in different geographic locations, worldwide. The attack machine is a Ubuntu 14.04 host in our lab. For the Tor experiments, we target the connection between a Tor relay (set up in our campus) and a random peer relay. Our Tor relay is also a Ubuntu 14.04 host and has been running the service for several months. The attack machine is the same host as the one in the SSH experiments.

In both the SSH and Tor experiments, the attacker attempts to reset the connection on the server end by connecting to it and performing the inference attacks. The diversity of servers and the corresponding network paths help test the robustness of the attack. We assume that the 3-tuple <client IP, server IP, and server port> is known. Further, the attack machine is capable of spoofing the IP address of both the victim client and server.

SSH Connection Reset

Summary. We run the reset attack against 8 different Amazon EC2 servers in different geographical locations. They are all micro instances set up for our experiments only. We establish a connection from the victim client to each server, and have the attacker perform the off-path connection reset attack. For each server, we repeat the experiment 10 times and report the average. As shown in Table 3.1, the attack is highly effective: the average success rate is 97% over all runs, with an average time cost of 44.3s. Note that the overall time excludes the time for synchronization

Location	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost (s)
US West 1	10/10	0	0	5000	48.00
US West 2	9/10	1.0	1.91%	5000	58.00
US East	10/10	0	0	5000	32.00
EU German	9/10	0.3	0.67%	5000	48.00
EU Ireland	10/10	0	0	5000	35.20
Asia 1	10/10	0	0	5000	51.00
Asia 2	9/10	1.7	5.34%	5000	36.67
South America	10/10	0	0	5000	45.70

Table 3.1: SSH connection reset results

(recall §3.3.1) as it is a one-time effort for a server and can be done a priori. The bandwidth cost here is 5000 spoofed packets per second, which translates to 4Mbps. Note that the probing scheme has already built in packet loss detection using "parity bits" as described in §3.5. To show that the packet loss detection scheme works, we report the number of rounds and the percentage of rounds on average, when packet loss is detected. For instance, even when packet loss between the attack node and "Asia 2" server is frequent, we still manage to succeed 9 times out of 10.

Failures may still occur since the detection scheme is rudimentary and may fail to detect packet loss. In some cases, the failure can also be the result of the attacker and server becoming out-of-sync due to network delay variance.[4] The success rate can be further improved by adding more redundancy and using better error detection schemes. However, we argue that the current success rate is already good enough to carry out effective DoS attacks.

[4]The failures that we experience are predominantly if not always because of packet loss. However, since we do not have access to routers, middleboxes, or even the end-server, we are unable to determine where and why the packet losses happen (recall that we only control the victim and attack client devices).

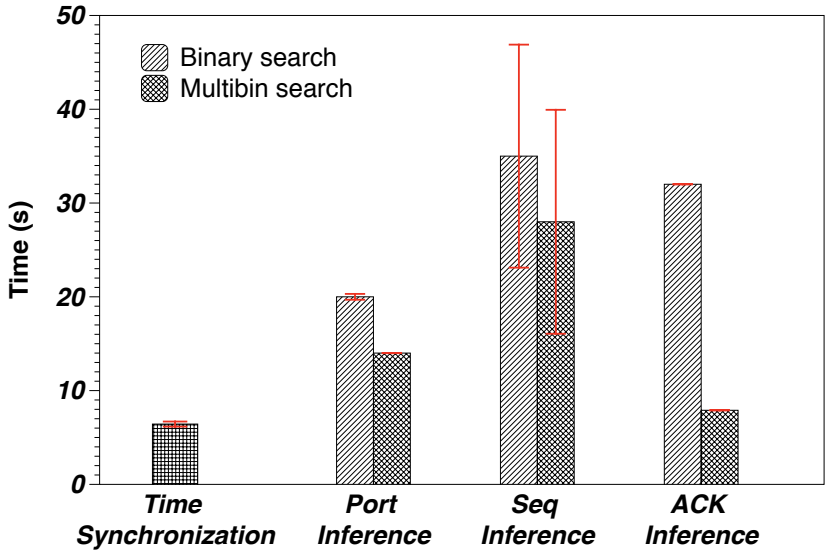


Figure 3.11: Time breakdown

Time breakdown. To understand where the time is spent in our attacks, we conduct another benchmark experiment against one of the SSH servers with both sequence number and ACK number inference. As shown in Figure 3.11, we break down the time spent into time synchronization and the three search phases of port number inference, sequence number inference, and ACK number inference with error bars. We also compare the optimized multi-bin search versus the regular binary search in each phase. Time synchronization takes around 7 seconds (optimization is not applicable). As discussed, it is only a one-time effort and therefore not on the "critical path". We see that with the optimized multi-bin search, the time spent on port search is fairly short (around 14 seconds). The time spent on sequence number search takes the most time due to the fact that the sequence number space is much larger. The time spent on ACK number inference is also fairly short (around 8 seconds) due to the fact that the challenge ACK window is extremely large and easy to locate.

Compared to the results with binary search, we see that the optimized multi-bin search has greatly improved the search speed by more than 30 seconds overall. This is due to the fact that

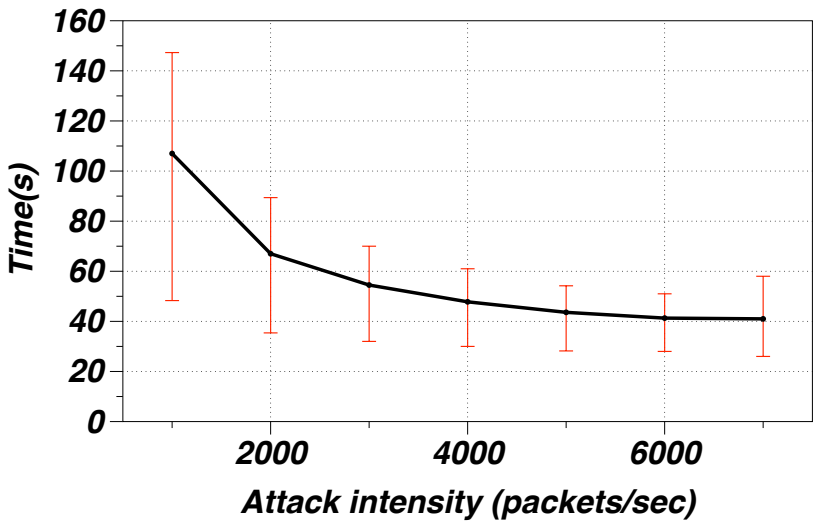


Figure 3.12: Attack intensity impact on time to succeed

binary search significantly under-utilizes the bandwidth resources and significantly increases the number of rounds of probes. The reason why the sequence number search does not benefit as much is because most of the time is spent on the initial linear search of the huge sequence number space. This step cannot be optimized with the multi-bin search.

Attack intensity vs. Time to succeed. Using the same experimental setup as before, we vary the attack intensity, i.e., the number of packets sent per second and show how this affects the time it takes to succeed. As shown in Figure 3.12, we plot the average, min, and max time to successfully conduct sequence number inference only (reset attack), as well as with the ACK number inference added (hijacking attack). Clearly, the higher the attack intensity the faster the attack. When the intensity is only $\approx$ 512 Kbps (1000 packets per second), the time to succeed is over 100 seconds, on average. When the intensity is $\approx$ 4 Mbps, (5000 packets per second), the average time reduces to $\approx$ 50 seconds for hijacking and only 30 seconds for reset. Note that an intensity $>$ 4 Mbps does not substantially improve the time to succeed because we begin to observe more packet losses, which cause additional rounds of probing. Of course, this is experienced on the

specific network environment between the attack host and the server, which could differ elsewhere; if the network conditions are even better, the time to succeed can be further improved.

Tor Connection Reset

Node	Target	Success Rate	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost(s)
62.210.x.x	FR	8/10	1.9	4.58%	4000	46.36
89.163.x.x	DE	9/10	4.0	7.97%	4000	49.08
178.62.x.x	GB	7/10	3.2	4.20%	4000	53.00
198.27.x.x	NA	10/10	0.8	1.45%	4000	59.86
192.150.x.x	NL	8/10	4.1	5.64%	4000	68.03
62.210.x.x	FR	6/10	2.5	5.85%	4000	49.57
89.163.x.x	DE	8/10	1.7	3.06%	4000	52.51
178.62.x.x	GB	8/10	6.0	8.15%	4000	78.35
198.27.x.x	NA	7/10	2.1	3.64%	4000	72.49
192.150.x.x	NL	6/10	5.5	7.14%	4000	79.42

Table 3.2: Tor connection reset results (first half under browsing traffic and second half under file downloading traffic)

To conduct a realistic experiment, we use a Tor relay set up in our campus and have a user using it as an entry relay. The entry relay establishes connections with an arbitrary middle relay (anywhere in the world). For ethical reasons, we do not perform attacks against arbitrary relay nodes that are not connected to our node.

To understand how the attack performs against mostly idle connections, we test it against connections between our own Tor relay and 40 other Tor relays throughout the world. The attack node has to connect to these Tor relays that are far away to perform attacks. In each case, we repeat the reset experiment 10 times. First, we discover that 16 of them do not appear vulnerable to the

side channel attacks, even though they appear to be Linux hosts. We suspect that this is because of certain firewalls that drop our spoofed packets. For the remaining 24 hosts, the average success rate is 88.8% and the average time to succeed is 51.1s. We find these results to be slightly worse than those in the SSH experiments because of higher packet loss rates.

In addition, we pick 5 random relays and simulate background traffic with browsing and file downloading, and conduct the same experiment as above. Here, to deal with moving windows, we use the simple brute-force strategy described in § 3.5. The results are shown in Table 3.2. The average success rate is now down to 77% and the average time to succeed is 60.9s. Upon further inspection, the increased failure rate is exactly due to the moving window problem i.e., it interferes with the sequence number search. Nevertheless, we think the result is acceptable as we have not designed a robust solution specifically for dealing with a moving window (this is left for future work).

In general, we believe that a DoS attack against Tor connections can have a devastating impact on both the availability of the service as a whole and the privacy guarantees that it can provide. The default policy in Tor is that if a connection is down between two relay nodes, say a middle relay and an exit relay, the middle relay will pick a different exit relay to establish the next connection. If an attacker can dictate which connections are down (via reset attacks), then the attacker can potentially force the use of certain exit relays.

3.6.2 TCP Hijacking Case Study

Our attack does not require any assistance from client-side or server-side malware or puppet (which are required in prior studies [80, 40]). Therefore, our target is any long-lived TCP connection that does not use SSL/TLS. There are several attractive targets: video, advertisements,

news, and Internet chat rooms (e.g., IRC). Depending on the implementation, one can envision the following possibilities: 1) the client periodically initiates a request and asks for responses, or 2) the server proactively pushes notification messages. In both cases, our attack can inject malicious messages to the client and induce a variety of classic attacks such as phishing or cross-site scripting.

Here, we pick a news website $www.usatoday.com$ which has a long-lived TCP connection that periodically retrieves news updates every 30 seconds. This gives ample idle time for our sequence number and ACK number inference. The attacker machine and the victim client are Ubuntu 14.04 hosts in our lab (as in the other case studies). Once the numbers are inferred, we perform a de-synchronization attack [9] by sending a spoofed request to the server that will force it to send a response to the client. Since the request was never sent by the client, it will not accept the response as the response packet contains an invalid ACK number (acknowledging data that have not been sent). Later, when the client itself initiates a real request, the server would no longer accept it as the packet is considered to be data with an old sequence number. Now that the client and server become de-synchronized, the attacker no longer needs to worry about a race condition where the response to the victim client is sent back by the server first. During all this, the attacker simply sends spoofed responses periodically every few seconds with ACK numbers properly acknowledging the client's requests. If such spoofed responses arrive before the client sends a request, they will simply be dropped without any adverse effect (because the ACK numbers are acknowledging data that has not been transmitted yet).

We implement the attack end to end, and successfully hijack the connection and inject a phishing registration window to solicit email and passwords at the top of the webpage as shown in Figure 3.13. We repeat the experiment 10 times and summarize our results in Table 3.3. The

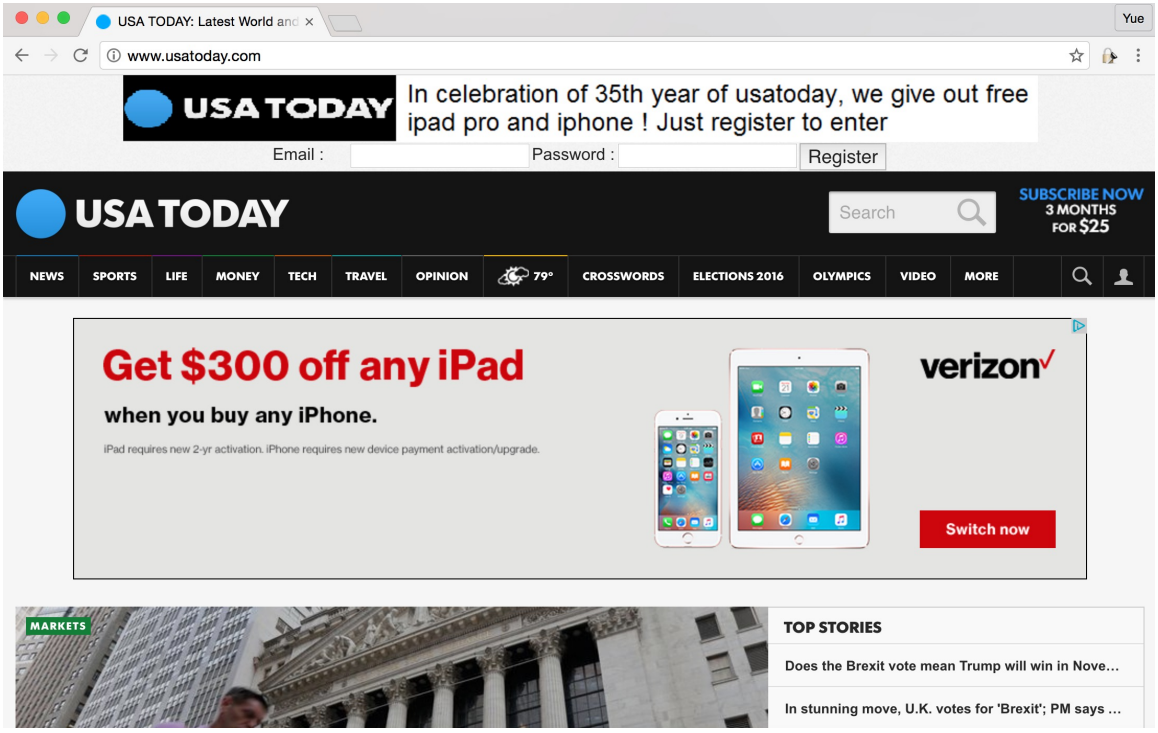


Figure 3.13: USAToday screenshot with phishing registration window

attack first infers sequence and ACK numbers before injecting the malicious payload. Success rate 2 quantifies the rate of inferring the sequence and ACK numbers correctly. However, USAToday occasionally switches the HTTP request from one type to another and therefore the injected payload will not match the request. Success rate 1 quantifies the rate of injecting the response that matches the request, which is strictly lower than success rate 2, but is still reasonable in our experiments. In addition, the time to succeed is longer than in the case of SSH and Tor experiments mostly because of the extra steps of ACK number inference and data injection. Based on prior research efforts (e.g., [77, 83]), a significant number of TCP connections are likely to last for durations more than 100ms; this period is sufficient for carrying out both the reset and hijack attacks.

Success rate 1	Success rate 2	Avg # of rounds with loss	Avg % of rounds with loss	BW (pkts)	Time Cost (s)
7/10	9/10	2.22	3.63%	5000	81.05

Success rate 1 = success rate of injecting the phishing registration window

Success rate 2 = success rate of inferring the correct sequence and ACK number

Table 3.3: USAToday injection results

3.7 Discussion and Defenses

Determining when the HTTP Request Packet was sent by the client: As shown in § 3.6.2, an off-path attacker has no information about when HTTP request packets are sent by the client. Therefore it simply sends one spoofed malicious HTTP response packet to the client periodically. However, repeated attempts at sending such packets can cause suspicion and make it easy for the victim client to detect such packets. Furthermore, the content itself will expose the purpose of attack, and can be used as evidence. Instead, if the attacker is able to determine when the client sends the HTTP request, it could simply inject a single packet with the malicious content immediately afterwards.

Towards achieving this goal, the attacker can simply send a packet with an invalid ACK number (acknowledging data that have not been sent) to the client to probe whether a request has been sent; this will cause the challenge ACK window to slide forward, iff an HTTP request was sent. Specifically, if the packet has an ACK number towards the originally perceived left boundary of the client's challenge ACK window (i.e., $SND.UNA - 2G + 1$), it is expected to trigger a challenge ACK; otherwise, the HTTP request must have been sent which shifted the left boundary of the challenge ACK window on the client. This result could be confirmed with the same side channel after each probe, as described in §3.4.2. Using this approach, the attacker can repeat the probing every few seconds; and subsequently inject the malicious packet only once.

Vulnerabilities in other OSes: We examine if the studied vulnerability exist in the latest Windows and FreeBSD OSes (the latter TCP stack is also used by Mac OS X). In brief, these OSes are not vulnerable to the attack. First of all, neither Windows nor FreeBSD has implemented all three conditions that trigger challenge ACKs according to RFC 5961. More importantly, the ACK throttling is not found for Windows or MAC OS X. Ironically, not implementing the RFC fully, in fact is safer in this case.

Defenses. As highlighted earlier, the root cause of all the attacks described is the side channel associated with the global challenge ACK count. This side channel can leak various types of information about an ongoing TCP connection. In general, as asserted in previous studies [61], network protocols are not designed rigorously to guarantee the non-interference property. Specifically, if there are multiple independent connections, one must ensure that the actions performed on one must not affect those of the other connection. One common case where this occurs is when the protocol uses global variables that are shared across connections (e.g., the challenge ACK count). We believe that a formal verification of both network protocols and implementations can shed light on if and when the non-interference property is violated.

In our study, we discover that the design and implementation of RFC 5961 has actually introduced an information flow that leaks TCP connection state through the shared challenge ACK counter (thus violating the non-interference property), and is highly exploitable. The best defense strategy is to eliminate the side channel (the global challenge ACK count) altogether. One can still enable the per-connection rate limit as long as each connection has a completely separate counter that does not interfere with those of other connections. The downside of this strategy is that if the number of connections in a system increases, the aggregate challenge ACK count can go up without

any bound. There is currently no evidence to suggest that this worst case scenario is likely to ever happen. However, if one is really concerned about wasting resources on sending challenge ACKs, we suggest a second solution which is adding noise to the channel. This is a common defense strategy in mitigating side channel attacks [25, 94]. Specifically, instead of having a fixed global challenge ACK count of 100 in all intervals, we can add random values (either positive or negative) for each interval. This will essentially confuse the attacker during the search process. In fact, even if the attacker repeats the probe many times, the result will always differ over time. To ensure that the added randomness is theoretically sound, one can even apply differential privacy to systematically introduce noise, as was done recently in [108]. We leave the design of the exact scheme to add randomness to future work.

Patch process: On July $5^{th}2016$, we proposed our defense to the Linux community. We point out that our side channel vulnerabilities are subtle; even during the patch process, the side channel vulnerabilities went through several iterations. The first two unofficial patch attempts were discussed in the private Linux security mailing list (`security@kernel.org`) within a day. They essentially try to set a random global challenge ACK count limit that varies between 68 to 131, every second. Interestingly, it turns out that this makes the attack considerably harder; yet the attack is still possible to execute (with substantially increased time). We demonstrate two possible attacks below. The main idea is based on the fact that the remaining ranges of challenge ACK counts, with good and bad guesses, are different. If all probing packets are bad guesses, then the possible range of challenge ACK received by the attacker is $[68, 131]$. Otherwise, the number range would change to $[67, 130]$ (assuming only one probing packet is a good guess). Therefore, as soon as the attacker receives 67 or fewer challenge ACKs from that second, there must have been a good

guess. Similarly, if the attacker receives 131 challenge ACKs, all probing packets are bad guesses. All other numbers between [68 and 130] will be ambiguous and the probing has to be repeated. Alternatively, the attacker can optimize the attack with more redundant packets. Specifically, for every probed sequence number, if the attacker sends 64 such packets repeatedly, in one second, the attacker will observe the remaining challenge ACK count to be $[4, 67]$ if it was a correct guess. This range has no overlap with the range $[68, 131]$ if it was a wrong guess. This attack is estimated to take around 30 mins with using our previous transmission rate (5000 packets per second).

Given our modified attack, another unofficial patch was proposed on July 8^{th} 2016. The main idea behind this patch is to randomize the time window (that governs when the challenge ACK count is reset) from 1s to [0.5s, 1.5s). This creates a time synchronization issue at the attacker's end. Besides, this patch also increases the default value of `tcp_challenge_ack_limit` to 1000. Unfortunately, after modification, our side-channel attack is still able to work. Instead of spreading the packets over the 1s duration, the attacker can send fewer packets in a short period (e.g., 0.2s, or even shorter). It is likely that this short period will fall in a single window. Specifically, if one of the probing packets is a good guess, then the remaining challenge ACK count has to be smaller than 1000. Otherwise, it will be exactly 1000. In the unlikely events where the packets do fall in two consecutive windows, we will be able to observe that the remaining challenge ACK count exceeds 1000.

Considering all modified attacks, the first public Linux kernel patch [6] was published on July 10^{th} 2016 (and later accepted to Linux); the patch essentially sets a random challenge ACK count limit, which varies between 500 to 1500, every second. By using a larger possible range, this patch makes the attack practically hard to execute. On July 14^{th} 2016, a second public Linux kernel

patch [5] was released. We are happy to see that this new patch eliminates the global challenge ACK count and enables "per-connection rate limit" instead, which is exactly what we initially proposed.

3.8 Related Work

Previous work on off-path TCP sequence number inference heavily relies on executing malicious code on the client side [78, 80, 40, 43, 39, 2], either in the form of malware [78, 80] or malicious javascript [40, 43, 39]. They share the same scheme of "guess-then-check" based on some side channels observable by the malicious code on the client side. They include OS packet counters [78, 80, 23], global IPID [40, 2], and HTTP responses [43]. In contrast, our off-path TCP attack eliminates the requirement completely, which makes the attack much more dangerous. The only prior study that shares the same threat model is the one reported by lkm in phrack magazine in 2007 [2]. The authors exploit the well-known global IPID side channel on Windows hosts to perform such attacks. Unfortunately, the IPID side channel is extremely noisy and the attack can take close to 20 minutes to succeed, as reported by the authors. Furthermore, as reported in [40], the success rate of such an attack is very low, unless the attacker has a low latency to the victim (e.g., on the same LAN). In comparison, our newly reported attack finishes much faster and is significantly more reliable.

Besides the TCP sequence number, it has been shown that other types of information can be inferred by an off-path or blind attacker [33, 61, 32, 113, 10, 42]. For instance, Ensafi *et al.* [33] show that, by leveraging the SYN cache and RST rate limit on FreeBSD, one can infer if a port is open on a target host through bouncing scans off of a "zombie" FreeBSD host. Knockel *et al.* [61] 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 with a bootstrapping time of an

hour on average. Alexander *et al.* [10] can infer the RTT between two arbitrary hosts with reasonable accuracy within minutes. Gilad *et al.* [42] are also able to infer if two hosts have established a TCP connection identified by a specific four-tuple, by utilizing the same noisy global IPID side channel. Compared to the newly discovered side channel, it has the following limitations: 1) requires the presence of stateful firewall or NAT which may not be universally present; 2) has a low success rate even when the tests are repeated multiple times (e.g., for more than a minute). Utilizing the new side channel, we can do this much faster.

Many of the side channels can be abused and cause unwanted information leakage. However, in some cases, they can also be used legitimately for network measurements. For instance, the global IPID side channel has been used to infer a network's port blocking policy [81]. The same side channel has also been used to count how many hosts are behind a NAT [15]. In addition, even though commonly considered a vulnerability, ISPs that allow IP spoofing are still prevalent according to the latest reports in 2009 [17] and 2013 [18]. Further, very recently, IP spoofing has also been used in legitimate applications such as reverse traceroute [59], detecting Interdomain Path changes [55], and detecting routing policy violations [36].

Chapter 4

Principled Unearthing of TCP Side Channel Vulnerabilities

TCP side-channels are critical vulnerabilities that can be exploited by adversaries towards launching dangerous attacks. Prior studies have demonstrated that TCP side-channels can be exploited by off-path attackers to perform idle port scans [34], to estimate the round trip time (RTT) of a connection [11], or to infer how many packets were exchanged over a connection [27]. They even allow attackers to hijack connections between a client and a server [79, 80, 41, 27, 21], These side-channels are an artifact of unforeseen code interactions, can arise with the deployment of large code bases, and are subtle and hard to find.

Most of the aforementioned side-channel vulnerabilities are discovered manually by domain experts. While manual analysis has been immensely useful in discovering and patching such subtle vulnerabilities, it requires a significant effort, and is thus not scalable and cannot guarantee the elimination of such vulnerabilities. In this work our goal is to *develop a principled approach to automate the discovery of such hard-to-find TCP side-channel vulnerabilities*.

In principle, TCP side-channels are violations of the *non-interference* property [46] between simultaneous TCP connections, i.e., the existence of one connection can have an observable effect on the other connection(s). Thus, off-path attackers can use their own connections to the server to infer the properties (e.g., sequence number) of a targeted TCP connection between a victim client and the same server. Specifically, an attacker can send spoofed packets with guessed properties to the server. If the guess is correct or close, the spoofed packet will cause a change in the state at the server which in turn, causes changes pertaining to the attacker's own connection.

Based on this observation, we design a tool *SCENT*, to find TCP side-channels in a complex code base with very little manual intervention. At a high level, *SCENT* detects TCP side-channel vulnerabilities by detecting violations of the non-interference property between connections. In particular, it uses two instances of the same server (TCP stack), where the only differences are in the security sensitive properties (e.g., sequence number, acknowledgement number, or port) of an idle (victim) connection. It then sends a set of packets (inputs) to the two servers. If the responses from the servers are different, then *SCENT* has detected a violation of the non-interference property.

While this approach is intuitive, the challenging part is determining what kind of packets to send in order to induce such a violation. Given the large search space of possible combinations of TCP packets, popular dynamic testing techniques like symbolic execution and fuzzing all face efficiency problems. In this work, we resort to bounded model checking [30, 63, 65] to drive an analysis to answer this question. Compared to bounded testing [95, 66] (i.e., blindly enumerating all possible packets up to the bound), bounded model checking enjoys the benefit of state deduplication and is thus, much more efficient (see §4.7 for more details).

Unfortunately, applying model checking to a real-world TCP stack implementation is non-trivial. First, we need to prepare a self-contained model that is amenable for model checking (otherwise the code base is simply too large). Previously, the work by Enasfi et al., [34] has adopted model checking to detect non-interference property violations in the network stack. However, due to the complexity of implementation level code, they had to manually craft a much simplified abstract model for the analysis. Such an approach, while useful in their context of interest (discovering idle port scan techniques), cannot guarantee that subtle TCP side-channels buried in complex implementations like the Linux kernel, will not be (unintentionally) removed during the abstraction. To avoid this problem (high false negatives), we opt to use the unmodified TCP stack implementation for analysis and only abstract away code that is outside the core TCP stack.

The second challenge is state explosion. TCP implementations from real-world kernels contain many variables; if we blindly mark all the variables as *states*, then any change to any variable will be deemed as a new state. However, if a variable is never "shared" between two connections, it cannot leak any information and is thus, not interesting to track. To solve this large state space challenge, we develop a conservative static analysis within *SCENT* to safely reduce the state space.

The last challenge is that bounded model checking has bounded code/state coverage and hence, cannot detect all vulnerabilities. For instance, the TCP side-channel discovered by Cao et al., [21] requires sending 100 packets, which is way beyond the capability of bounded model checking. To solve this problem, we developed a program transformation technique to automatically simplify the model as a way to improve the code coverage. In particular, we observe that many uncovered cases relate to branches that compare an attacker-controllable value with a fixed value (e.g., the global rate limit exploited in [21]), and the problem is that the bounded input space cannot drive

the variable side of the branch to go beyond the fixed threshold. Based on this observation, *SCENT* automatically identifies such branches and downscales the fixed threshold so that both branches can be visited during a subsequent iteration of bounded model checking.

To demonstrate the effectiveness of our approach, we have implemented a prototype of *SCENT* and created two realistic TCP models, one based on the Linux kernel (version 4.8.0) and the other one based on the FreeBSD kernel (version 13.0)[1] We applied *SCENT* on these two models and found 12 new side-channel vulnerabilities. A real world evaluation shows that in particular, with one of the classes of vulnerabilities discovered, an off-path attacker is able to infer whether two arbitrary hosts are communicating with each other, within slightly more than 1 minute on average. The evaluation results also show that our transformation step is critical for finding these side-channels—none of them can be found without the transformation. Besides, we also did not observe any false positives during our evaluation.

Contributions. Our contributions can be summarized as follows:

- We design and implement *SCENT*, a system that finds subtle TCP side-channels by detecting violations of the non-interference property between TCP connections, using model checking as a basis.
- We developed several techniques to automate the process of creating self-contained code amenable for use with an off-the-shelf model checker, from real-world kernels that keep the core TCP implementation intact. We applied these techniques to the Linux and the FreeBSD operating systems and open sourced the extracted models at [**?**]

[1] *SCENT* can be applied to any OS kernel as long as the source code is available. Therefore, *SCENT* can be potentially applied on Windows in its internal environments.

- We developed a code-transformation-based model simplification technique that improves code coverage for bounded model checking.
- We applied *SCENT* to the Linux and the FreeBSD TCP models and found 12 new side-channel vulnerabilities. We open sourced our system and released the complete details of findings at [**?**].

4.1 Background

In this section, we briefly describe the non-interference property and why it is relevant to the problem of interest. Subsequently, since we use model checking as a basic building block, we provide relevant background in brief.

The non-interference property. The non-interference property [46] has been widely used as a requirement to prove that neither explicit nor implicit information leakage can occur in a scenario of interest. Because side-channels are a consequence of information leakage, the non-interference property can be used as a verification condition to ensure that they do not exist. If the property is violated, it indicates the potential presence of an information leak, which can in turn lead to an exploitable side-channel vulnerability. With regards to the context of interest, if this property holds, it implies that a state change on a given connection does not (implicitly or explicitly) become observable in another connection.

Ensafi et al. [34] applied model checking to verify the non-interference property in the TCP/IP stack, towards finding side channel vulnerabilities relating to idle port scans. While they find two port scan vulnerabilities, we point out that they use model checking more like a validation tool instead of a tool to discover these; they heuristically specify the scope of the TCP code (to only consider the specific shared resources across connections) and then manually build the model.

In this work, we seek to perform non-interference analysis in more general attack sce-

narios. Importantly, since a manually abstracted model like that in [34] is very approximate and may miss vulnerabilities that exist in real code, we explore applying model checking to real TCP implementations from commodity kernels.

Software model checking. Model checking [29, 84] exhaustively checks if a given model of a system satisfies a given formal property. If violations are encountered, the model checker outputs counter examples which enable the locatation of where a violation has occurred with relative ease. Model checking methods can be broadly classified into two categories viz., those that use abstraction (e.g., SLAM [14], BLAST [50], Event-Driven Software Verification [51]) and those that are applied directly on implementations (e.g., VeriSoft [44], CMC [70] [69], and Model-Driven Software Verification [53]). Since the former relies on extracting an abstraction from the real code (and thus can result in significant approximation), we use the second category to verify the non-interference property in our work.

Specifically, the basis for our work is a TCP event-driven execution model that we build. Different from previous work relating to the use of model checking with an abstracted state machine of either TCP or the network stack (e.g., [70, 69]), we check for possible violations of the non-interference property in real TCP implementations. Our model is much more complicated since we have to look at verifying a property relating to connection interactions (as discussed later our model contains 4 live TCP connections with 6 different sockets). Furthermore, we need to address challenges relating to making the model self-contained (to ensure that it can be used with an off-the-shelf model checker) and concise (without which the complexity of the code will make it untenable to the model checker). Unfortunately, even just the core TCP stack implementation is too complex for the model checker to exhaustively check all possible states of the code. For this reason, we

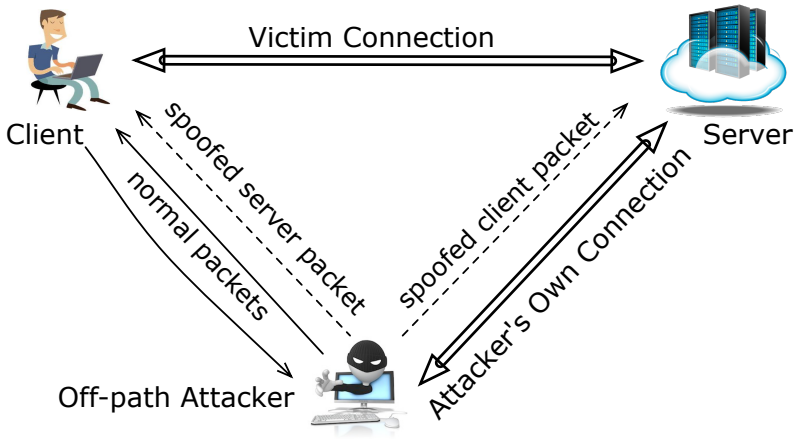


Figure 4.1: Threat model

can only perform bounded model checking and therefore the conclusions (existence or absence of violations) are only applicable to a bounded set of states instead of the entire code base.

4.2 Threat Model

Our threat model is that of an off-path TCP attacker as shown in Figure 4.1. We consider 3 hosts viz., a victim client, a victim server, and an off-path attacker. The attacker can either send packets on its own connection to the server, *or* send spoofed packets with the victim client's IP address or a victim server's IP address. Different from a Man-in-the-Middle (MITM) attack, the off-path attacker can neither eavesdrop nor inject packets into the victim connection. Instead, it attempts to exploit a side-channel vulnerability to infer the state of the victim connection based on the packets sent/received on its own connection. Specifically, it could target the inference of (a) the *port number* of the victim client (the server's port number is usually known), (b) the *sequence (SEQ) number* from the client, and/or (c) the *acknowledgement (ACK) number* expected by the server. By inferring just the port number, the attacker can determine if there is an established victim connection

between the server and the client. With the port number and the SEQ number expected by the server inferred, the attacker can launch a DoS attack by sending a packet with the reset (RST) flag (and correct SEQ number) to terminate the victim's connection. If all the three attributes are inferred, the attacker can hijack the victim connection and inject malicious payloads as shown in [21]. Note that any machine around the world can launch an off-path attack, as long as it is able to send spoofed packets with the victim client's (or server's) IP address.

Previous TCP inference attacks [79, 80, 21, 27] follow a "guess-then-check" strategy. Specifically, during the guess phase, a spoofed packet is sent with a guessed value (for either or a combination of the port number, SEQ number and/or ACK number). A correct guess will be "accepted" by the TCP state machine thus causing it to transit into a state that is different from that due to wrong guesses. During the subsequent check phase, the attacker exploits the side-channel vulnerability to leak the state transition of the victim's connection, which allows the attacker to tell whether the guess is correct or not. Like in these efforts, the focus of this work is on identifying similar "software-induced"[2] side-channels but by using a more principled approach.

An illustrative TCP side-channel vulnerability. To illustrate how an off-path attacker can exploit a side channel vulnerability to determine the state of a victim connection (in terms of port number, SEQ number or ACK number) consider the recent example from [21]. Figure 4.2 captures this example wherein the off-path attacker infers the expected SEQ number of the victim connection to the server.

[2]Other types of side channels, such as timing based ones [27], are out of the scope.

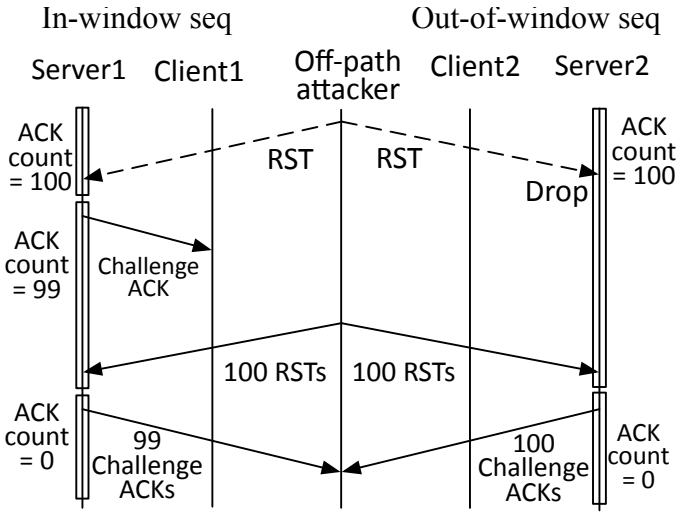


Figure 4.2: An illustrative TCP Side-Channel Vulnerability.

To understand how the attack works, consider two cases. In the first case, the SEQ number guessed by the attacker is within the "receive window" (in-window) of the server while in the second case, the SEQ number is out-of-window. The attacker sends a spoofed RST packet with a guessed SEQ number. If the number is in-window, the server responds to the victim with a "Challenge ACK" packet to ask the client to confirm the RST. Since the victim client did not really send the RST packet, it will simply discard the Challenge ACK packet. To control how many Challenge ACKs can be sent within a time period, the Linux kernel maintains a *global shared counter* (equal to 100 prior to the work in [21]). Thus, when the attacker subsequently sends in-window RST packets on its own connection (one after the other as shown in the bottom part of the figure), it gets back 99 Challenge ACKs; in contrast, if the spoofed RST packet is out-of-window, the attacker will receive 100 Challenge ACKs. This difference/side-channel can then be used to infer whether the guess is correct or not.

What is evident in the above example is that, by observing the number of Challenge ACK responses from the server on its own connection, the attacker can distinguish between two cases

with regards to its spoofed packet viz., whether the SEQ number guessed is within the server's receive window or not. Thus, this is a violation of the non-interference property i.e., the state of the client's connection influences how many Challenge ACKs are received by the off-path attacker.

4.3 *SCENT* Overview

In this section, we provide an overview of our system *SCENT* and its core innovation.

4.3.1 Workflow

Figure 4.3 shows the overall workflow of *SCENT*. Specifically,

- Taking the source code of a commodity OS kernel as input, the Model Generator (§4.4) generates a self-contained model[3] amenable for application of an off-the-shelf model checker and pushes this initial model into a queue.
- The Non-interference Checker (§4.5), at each step, takes one self-contained TCP model from the queue, constructs an attack scenario, and executes bounded model checking to verify the non-interference property between connections.
- If violations are found by the model checker, validated counter-examples are output as the proof-of-concepts for possible TCP side-channel vulnerabilities inside the kernel's TCP stack implementation.
- Finally, to mitigate the limited code coverage of bounded model checking, the Model Transformer (§4.6) automatically generates a new, downscaled model and pushes it into the queue for the next round of analysis.

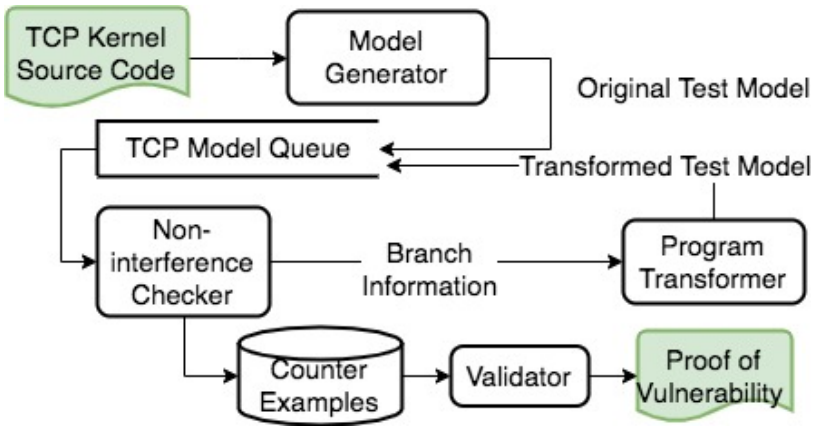


Figure 4.3: Overview of *SCENT*'s workflow.

4.3.2 Automated downscaling

While applying bounded model checking to TCP implementation as a way to find non-interference violations is not entirely new, *SCENT* solves an important and non-trivial problem. In principle one will need to send an extremely large sequence ($\approx \infty$) of packets in order to excavate all possible violations of the non-interference property. Unfortunately, we point out that due to the complexity of commodity kernels' TCP implementations, even a relatively small sequence of TCP packets can lead to an explosion of the state space that cannot be explored by the model checker with limited computation resources (CPU time and/or memory). As a result, the bound we can afford is considerably small (e.g., only 3 packets in our evaluations); otherwise, the model checker will either exhaust memory or take a prohibitively long time to finish. This further translates to limited code coverage and impacts the effectiveness of *SCENT* (i.e., it cannot detect side-channel vulnerabilities in uncovered code). For example, the vulnerability illustrated in Figure 4.2 cannot be detected as triggering it requires sending 100 RST packets. In fact, side-channels are more likely

[3]Note that we only abstract code irrelavent to TCP stack; previous work abstracts the TCP stack itself.

than not, triggered by such uncommon sequences of packets. *SCENT* copes with this scalability issue via a novel technique we call *automated downscaling*.

Our observation is that the TCP code base contains many checks (branches) that compare attacker-controlled variables against either some constant values or variables that remain the same during model checking. Due to the limited input bound, those attacker-controlled variables have limited value ranges. When the attacker-controlled value range does not overlap with the fixed value (cover both sides), only one branch can be covered. However, such linear relationships between an attacker value and a fixed value can be satisfied easily by downscaling the fixed value (i.e., moving it torwards the attacker-controlled value range). More importantly, this transformation will not change the fundamental behavior of the TCP implementation: without downscaling, the relationship can still be satisfied, but simply takes significantly longer inputs and therefore times.

To further elucidate this observation, let us revisit the example from Figure 4.2. The side-channel relies on the global Challenge ACK rate limit (a variable with fixed value 100) and the attacker has to send 100 packets in total (one spoofed and 99 on its own connection in the example), to trigger the information leakage. To find this vulnerability, intuitively, the model checker will have to examine what happens when the TCP code base has received different numbers of packets which have the RST flag set and are in-window (it has to perform 100 such checks). Unfortunately, this is not possible during our bounded model checking because we can only increase the counter from 0 to 3. However, if we were to simply (artificially) change this rate limit to say 2, then we will be able to trigger this vulnerability and observe the difference.

Furthermore, the advantage of this approach is that it also inherently reduces the required input space we need to enumerate. For example, one can reduce the space of possible SEQ numbers

(from 2^{32} to a much smaller value) by downscaling other fixed constants (e.g., the receive window size). This also contributes to a drastic reduction in the time-complexity associated with our analysis.

Practical Realization. To practically realize automated downscaling, we pursue an iterative approach (alluded to in the workflow described in § 4.3.1). This approach is driven by the key insight that there is a tight coupling between the input space (i.e., length of our input packet sequence and the space of the fields in the TCP header such as SEQ number space) and the values to which the limits in the code are to be changed. In the example above, changing the limit to 2 requires the attacker to send a sequence of two packets. If on the other hand, we knew that the attacker had a packet sequence of length 5, the limit could be anywhere from 2 to 5.

Given this, for ease of realization, to begin with, we fix the length of the input packet sequence and the sizes of the header fields in each, but do not modify the TCP code that is input to the model checker. During the model checking phase, we log information relating to what parts of the code (what branches) are not covered because of control statements relating to such limits. We then use concolic execution to establish transformations of such constraints (guided by the constraints imposed on the input packet sequence) that may make such coverage viable (using the program transformer module shown in Figure 4.3). The transformed model is then considered for bounded model checking. We iterate the process until we either (a) do not find any additional transformations that we can perform or (b) we exceed a pre-specified time limit.

4.4 Model Generator

In this section, we describe in detail how we address the challenges in constructing a standalone TCP code base that can be input to the model checker and how we initialize variables to ensure that the model begins with a valid and consistent TCP state.

In principle, one can apply the design principles from [53] to construct a test model, which combines a test-harness and the real kernel code with an initial state. Given this initial state, the test harness would enumerate a sequence of packets as input, and calls the TCP packet reception code to explore the set of reachable states. Here, the state of the model is defined as the union of internal states at a host, and is determined by the values of global variables and heap objects that are reachable by the connection object (i.e., the socket). Unfortunately, applying model checking directly on a kernel code in its entirety, is not practical. This is because model checking has high associated time complexity, and using the entire kernel code base as the model can make the analysis prohibitively costly. More importantly, many of the paths explored by the model checker will have no bearing on what we seek to analyze. Last but not least, there is significant non-determinism in real TCP implementations which will interfere with the model checking.

Therefore, assembling a standalone TCP implementation without any kernel dependencies becomes important for the feasibility of our approach. However, extracting the TCP code from a kernel is challenging given the fact that the TCP code interacts with the rest of the kernel via complex interfaces. We solve this challenge by identifying boundaries where the code can be pruned and manually constructing stub implementations to close the boundaries.

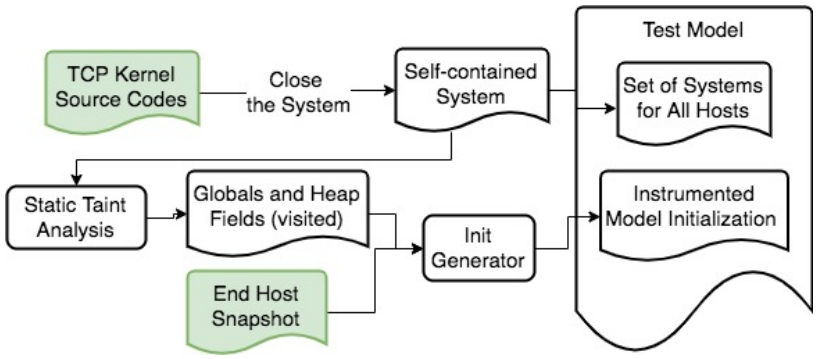


Figure 4.4: Workflow of the Model Generator.

In addition to generate a self-contained TCP code base for the model checker, another challenge is how to properly initialize the model. *SCENT* solves this challenge by automatically extracting correct values from a memory snapshot.

4.4.1 Building a Standalone TCP Model

The high-level guideline for building a standalone TCP code base is that we want to make sure that all the code related to the TCP stack remains exactly the same as in the target kernel, while code not related to the TCP stack should be minimized/abstracted. Following this guideline, we use a simple worklist-based, semi-automated approach to gradually grow the code base until the whole TCP stack is included.

1. We initialize the worklist with the entry function of the TCP layer when a packet is received (e.g., `tcp_v4_rcv` in Linux and `tcp_input` in FreeBSD).

2. We try to remove one function from the worklist. If the worklist is empty, we terminate the process; otherwise we move on to the next step.

3. We check if the current function belongs to the TCP layer (based on our domain knowledge). If so, we copy the whole function to the standalone mode and move on to the next step; otherwise we manually write a stub function to abstract it and go back to Step (2).

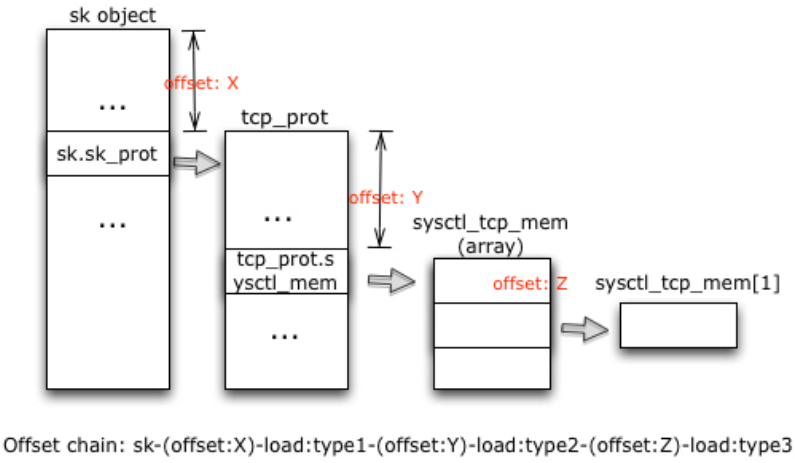


Figure 4.5: Using offset chains to locate the target variables during initialization.

4. We find all the callees of the current function and add them into the worklist and go back to Step (2). For indirect calls, we manually resolve the target based on domain knowledge.

Note that because in our attack scenario (§4.5.2) we keep the victim connection idling, our current model excludes functions relating to sending packets on that connection.

4.4.2 Initializing the Standalone TCP Model

Because our TCP model is built using partial kernel code starting at an entry function, we need to initialize what we call *environmental variables* at this entry point. This is to ensure that the initial state provided to the model checker is correct and consistent with TCP executions. Such environmental variables include the entry function's arguments, global variables, and heap objects that may be accessed or reachable by the code extracted above. We point out that there is no need to initialize local variables or heap objects that are allocated (and initialized) during the execution of the model.

Manual identification of all the variables that have to be initialized is not only an onerous task but is also error-prone. Thus, we develop an automated procedure to initialize them based on a memory snapshot from a running kernel, which is captured when the entry function is invoked, Because our standalone model runs in user-space, values from the snapshot cannot be directly used as they could be pointers. So our method needs to (1) identify all accessible variables and their types, (2) locate each target variable in the snapshot (i.e., determine its address), (3) extract its value according to its type and size. Finally, this will allow us recreate the variables and initialize their values in the model checker.

We achieve these goals via a process that is similar to previous work on recovery of kernel objects from memory snapshots [22]. First, starting from anchor variables (i.e., entry function's arguments and global variables explicitly referred to in the model), we use static taint analysis to recursively identify all accessible/reachable heap and global objects by following pointers. Due to the existence of typecasting, we identify pointers in two ways: (1) based on the variable/field's declaration type and (2) based on the use of the variable/field.

To locate variables inside the memory snapshot, we maintain the point-to relationship between kernel objects in a data structure that we call *offset chain*, which tracks how each variable is derived from an anchor variable and the used type associated with the variable. The offset chain allows us to traverse the snapshot and recover the corresponding variables.

Once we locate a variable inside the snapshot, we extract its "initialization" value based on whether it is a pointer or not. For non-pointer variables, we will directly use its value from the snapshot; for pointer variables, we will allocate the target variable statically in the model checker and assign the target object's address as the initialization value. One particular challenge in this step

is how to decide the size of the variable if its type is an array with unknown size. For example, in Linux, the packet header pointer `skb->head` is a pointer to an unsigned char, which can be used to visit the packet payload with a specific offset (via a value of header field `doff`). As associating size with a pointer is a hard program analysis problem, currently we solve this challenge manually.

Figure 4.5 illustrates this process via an example. In the figure, each offset depicted represents the address offset between the beginning of a heap object and the current field in the object. Given the base address in the snapshot and `offset:X`, our method can obtain the corresponding field value. If the field is a pointer, its value can be further dereferenced in the snapshot to locate the next (new) heap object. Given this new heap object's base address and `offset:Y`, a new field can be located and so forth.

4.5 Non-interference Checker

In this section, we describe how we detect violations of the non-interference property between two TCP connections.

4.5.1 Constructing the attack scenario

To detect violations of the non-interference property between two connections, we craft an attack scenario that is similar to what was captured in the illustrative example (Figure 4.2). The scenario consists of two servers (Server1 and Sever2), two clients (Client1 and Client2), and an attacker (Figure 4.6). Both servers and clients use the same self-contained model from the Model Generator. A connection between Server1 and Client1, and Server2 and Client2 is initialized before testing. The two connections are identical except *a specific secret* relating to the victim connection.

We use the connection between Client1 and Server1 to model the case when the guessed secret is correct, and use the connection between Client2 and Server2 to model the case when the guessed secret is wrong.

Ideally, to find all possible side-channel vulnerabilities, the attacker (test-harness) should exhaustively generate all possible input packet sequences, including both spoofed packets (with the IP address of the victim clients) and packets on its own legitimate connection. Unfortunately, given the unbounded search space, this is simply infeasible. So our test-harness only enumerates all possible input packet sequences up to a bound (i.e., performs bounded model checking). Once the test-harness generates a packet sequence, it sends the same sequence to both servers. Because only the secret attribute is different for the two victim connections, if the packets received from the two servers are different (including the number of packets, the pattern/order of received packets, the contents, etc.), the non-interference property is violated and the secret is leaked. The counter-example (packets being sent from the attacker) is then reported as violations by the model checker.

To reduce the effort of the attacker and to make the model more deterministic, we keep the victim connections "idle" during the model checking (i.e., neither the victim client or the server will actively send packets in our model). By doing so, we can be sure that differences in the received packet sequence are indeed caused by the spoofed packet sequence. If the server and client are actively exchanging packets, it becomes hard to identify a violation (differences may simply be due to those exchanges).

[4] We set the secret as whether the specific port is being used by the victim connection.

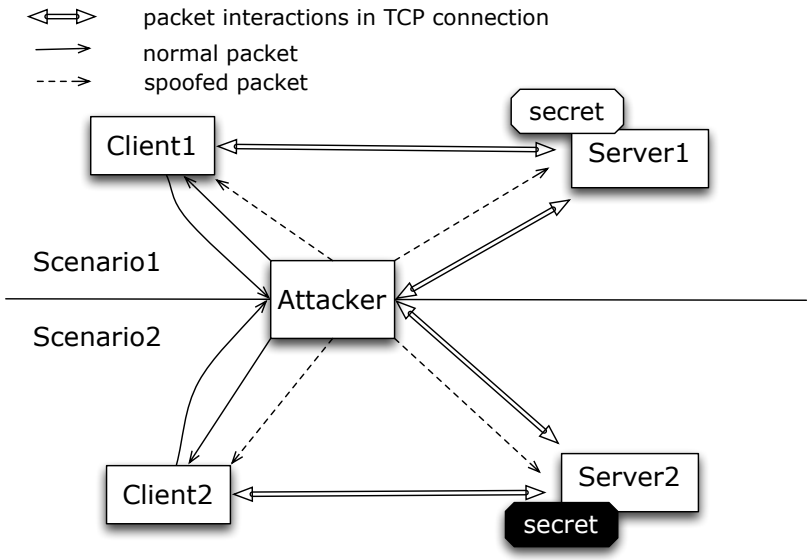


Figure 4.6: Our setup for the scenario relating to non-interference property verification.

Table 4.1: The 6 different secret settings of interest (The initial state captures the victim socket state at the server side)

Initial State	SYN-RECEIVED			ESTABLISHED		
secret	port no.[4]	SEQ	ACK	port no.	SEQ	ACK

4.5.2 Secrets of interest

Our focus in this work is on identifying side-channel vulnerabilities that result in the leakage of three specific secret attributes of a connection viz., *port number*, *SEQ number*, and *ACK number*. While there could be other sensitive information (e.g., that reveals the socket state), we focus on these since they have been shown to be exploitable for DoS or connection hijacking [21]; however, our approach can be used to infer the leakage of other secrets.

To formalize, we require our model checker to verify the following three properties with respect to non-interference. The sequence of packets received by the attacker is identical with respect to the two servers, regardless of :

- The port numbers used in the victim connection;
- The SEQ numbers used in the victim connection; and,
- The ACK numbers appearing in the victim connection.

While previous work only examines if and how these secrets are leaked when the victim connection is in the ESTABLISHED state of TCP, we extend the scope to include cases wherein the victim is in the SYN-RECEIVED state (i.e., during the three way handshake). This is because in this state, if the attacker can acquire the information of interest, it can infer whether the client tried to establish a connection with the server, or even potentially establish a fake connection itself (by sending a spoofed SYN packet with the client's IP address – note that in this case, the SYN-ACK returned by the server to the victim client is simply dropped). The latter attack can be serious in practice, since the attacker if successful, can subsequently inject malicious data on to the server pretending that the data came from the spoofed client's IP address. Thus we have a total of 6 secrets (in the two states combined) listed in Table 4.1.

4.5.3 Bounding the input packet sequence

In this work, we focus on the control bits and the secret of interest (port, SEQ, or ACK number) in the TCP header. All other fields are fixed, and are copied from snapshots from real connections. We exclude the TCP header options in this work since not all systems support these. Table 4.2 captures the bounded input space that the attacker (test harness) in our scenario, generates. For fields that have small value ranges, such as the TCP flags, we enumerate all the possibilities, (except for FIN and congestion related ones). For fields that have larger value ranges, we determine the range as described below.

Recall that in our attacker scenario (Figure 4.6), we use the connection between Client1 and Server1 to model the case when the guessed secret is correct (the guess will automatically be wrong for the connection between Client2 and Server2 because the secrets are different). Therefore, for the field related to a secret of interest, the value range is decided based on the concrete values from the connection between Client1 and Server1. Assume that on Client1's side, the port number, SEQ number, and ACK number are P, S, and A, respectively. We will always set the port number of all generated packets to P, because this allows us to exercise the scenario where we made a correct guess of the port number of the connection between Client1 and Server1, and a wrong guess of the port number of the connection between Client2 and Server2. For the SEQ number and ACK number, because the TCP stack performs a range check, we want to simulate cases where the guessed number is close to, but not equal to the correct number. For this reason, we enumerate the range $[S-2, S+2]$ and $[A-2, A+2]$. Currently, we limit these variables to this range, because with our automated downscaling (of the receive window), the enumerated cases are enough to explore all three of the following cases with respect to those numbers: (a) outside the receive window, (b) exact match, and (c) within the receive window.

We currently limit the range of the payload size to $[0, 2]$. This range allows our model transformer to downscale packet size related checks; yet, it will not significantly increase the input space or the state space.

Finally, we determine the packet length through empirical experiments, i.e., given that the ranges of each packet's fields are fixed, we try to enumerate as many packets as possible until the model checker either exhausts the memory or takes a prohibitively long time to finish. On our evaluation platform, we can only enumerate a maximum packet sequence length of 3.

Table 4.2: Packet fields enumeration ranges. C1 means the corresponding value used by Client1 in our attack scenario. Packet with IP equal to C1 is spoofed packet, while packet with IP equal to Attk is on attacker's own connection.

	Packet Fields									Length of
	IP	SEQ Num	ACK Num	SYN	ACK	RST	PSH	URG	Payload Size	Packet Sequence
Original range	[0, 2^32)	[0, 2^32)	[0, 2^32)	0/1	0/1	0/1	0/1	0/1	[0-1460)	Infinite
Bounded range	C1/Attk	[C1-2, C1+2]	[C1-2, C1+2]	0/1	0/1	0/1	0/1	0/1	[0-2]	3

4.5.4 Deduplication

Since different input sequences can trigger the same vulnerability, the counter-examples can also be duplicated. To find distinct vulnerabilities, we follow a similar approach as semantic crash bucketing [100]: given a counter-example, we use the branch trace recorded for the model transformer (§4.6) to locate the key branch/constraint that leads to the different behavior and "patch" the branch so that the counter-example will no longer yield the different behavior. One can consider this as the opposite process compared to our model transformation process. Then, we run all the counter-examples again. All the other counter-examples that no longer yield the different behavior will be considered to be duplicates.

4.6 Model Transformer

In this section, we describe how we practically realize the vision of automated down-scaling using an iterative approach. We begin with a limited input space and execute bounded model checking. During the process, we log information relating to the code that are not covered but relate to fixed limits (discussed earlier). We then apply concolic executions to determine how these limits must be transformed (flipped), to make the code coverage feasible while adhering to

the inherent constraints imposed by the chosen, limited input space. The transformed model is then re-considered (in the next iteration) and the process is repeated until we do not find any new transformations that can be performed or when we exceed a pre-specified time limit. We describe these steps in detail below.

4.6.1 Identifying target branches

Given the above premise, our first challenge is to locate branches we aim to flip. To do so, we instrument our model so as to trace all the branch instructions and dump their conditions during the model checking phase. Then we parse the trace and look for branch conditions (a relation operation like $<, >$) that have one operand that varies (e.g., a stateful variable), while the other operand is always a fixed value. Next, we check if both the true branch and false branch are visited during model checking; if only one branch is visited, we have found a target branch.

4.6.2 Determining expected values

After identifying the target branch, the next step is to determine the expected value. Given that the other operand imposes a range $[l, h]$, we have two general options: we can either move the fixed value to the other side of the range ($> h$ or $< l$) or move it to the middle of the range. In this work, we choose to move the value to the middle of the range for the following reason. The goal of the non-interference checker is to find a behavior that differs between Server1 and Server2 when handling the same input sequence. One reason such a difference can arise is Server1 and Server2 taking different paths at the same branch. For a target branch, the input sequences we enumerate can only go down one path, with both servers; otherwise we would have observed these cases. So if we move the fixed operand to the other side of the range, the input sequence still can only go down only one path, which is not particularly useful for revealing potential different outputs.

4.6.3 Identifying targets for transformation

The goal of our model transformer is to rewrite the program so that our limited input packet sequence can visit *both* the true and false conditions of a branch. After identifying a target branch, there are multiple ways to rewrite the program to achieve this goal. One way is to replace the relational operation with one that compares the variable operand with a smaller but *fixed constant*. However, this simple approach could introduce inconsistencies when the operand with the fixed value is derived from one or more program variables that are also used in other constraints (e.g., branch conditions). To avoid potential false positives or false negatives introduced by such inconsistencies, we choose to modify the source variables during initialization, instead of directly patching the branch.

There are two general approaches to identify the source variable(s), data-flow (taint) analysis and symbolic execution. Because the source variable(s) could go through a series of operations before being used in the target branch (e.g., `limit+10`), we choose symbolic execution. This approach provides us with a symbolic formula expressing the relationship between the source variable(s) and the value used in the target branch. Therefore, given an expected value to be used in the target branch, we can consult a SMT solver to automatically determine the corresponding value(s) to which the source variable(s) need to be set during initialization. Moreover, symbolic execution also allows us to collect all path constraints prior to reaching the target branch. By adding these constraints while querying for suitable initialization value(s), we can ensure that the new value(s) will not break path constraints leading to the target branch.

In brief, *SCENT* uses concolic execution to determine (a) which variable(s) should be modified during initialization and (b) what is the value(s) to which it must be initialized. Because

we perform concolic execution over a single trace (only to collect the symbolic formula relating to a branch predicate), we point out that we do not have a problem of path explosion. A sketch of the process is as follows:

1. *SCENT* conservatively symbolizes all variables that are related to the system's internal states. (i.e., all the global and heap objects found in §4.4).
2. *SCENT* applies concolic execution wherein one path recorded during model checking is followed to reach a target branch constraint.
3. *SCENT* checks if the operand with a fixed value is symbolic (i.e., derived from internal states). If not, we directly patch the constant and exit; otherwise we move on to the next step.
4. *SCENT* queries a SMT solver for a feasible assignment to the internal states such that (a) the path constraints are satisfied and (b) the operand used in the branch will fall into the range of the variable operand.
5. If the solver can return an assignment, *SCENT* modifies the model initialization procedure to assign the values returned from the SMT solver to the related internal states; otherwise it tries to find another recorded path that can lead to the target branch and goes back to Step (2).

4.7 Evaluations

In this section, we evalute *SCENT* on two OS implementations, viz., Linux 4.8.0 and FreeBSD 13.0, with the goal of answering the following questions:

- **Effectiveness on vulnerability finding**: Can *SCENT* find real TCP side-channel vulnerabilities from these two kernels? (§4.7.2)

Table 4.3: Side-channel vulnerabilities discovered by *SCENT* with different initial secret settings.

OS Kernel	Index-ClassID	Key Constraint that Causes Violations	Different Outputs	6 Secret Settings in Table 4.1						Transfor-mation Required	New
				SYN_Recv			Established				
				port	SEQ	ACK	port	SEQ	ACK		
FreeBSD 13.0	1-A	V_icmp_rates[3].cr.cr_rate <V_icmplim	RST pkt vs NULL	+			+			Y	Y
	2-A		RST pkt vs NULL	#	#	#	#			Y	Y
	3-A		RST pkt vs NULL	+			+			Y	Y
	4-A	V_icmp_rates[4].cr.cr_rate <V_icmplim	RST pkt vs NULL	+			+			Y	Y
	5-B	sch->sch_length >= V_tcp_syncache.bucket_limit	NULL vs RST pkt				+			Y	Y
Linux 4.8.0	6-C	tcp_memory_allocated <sysctl_tcp_mem[2]	ACK pkt vs NULL	#	#	#	#	#	#	Y	Y
	7-C	tcp_memory_allocated <sysctl_tcp_mem[1]	Immediate ACK vs Delayed ACK	#	#	#	#	#	#	Y	Y
	8-C		ACK pkt with different window size	#	#	#	#	#	#	Y	Y
	9-B	inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL	+			+			Y	Y
	10-B	inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL		#	#				Y	Y
	11-B	sk->sk_ack_backlog >sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL	#	#	#	#			Y	Y
	12-D	challenge_count <sysctl_tcp_challenge_ack_limit	ACK pkt vs NULL	#	#	#	#			Y	Y
	13-D		ACK pkt vs NULL	+			+			Y	N
	14-D		ACK pkt vs NULL	*	*		*	*		Y	N
	15-D		ACK pkt vs NULL	#	#	#	#	#	#	Y	N

+: correct port number required to trigger the violation

*: correct port number and SEQ number (in-window) required to trigger the violation

#: correct port number, SEQ number (in-window) and correct ACK number required to trigger the violation

- **Effectiveness of automated downscaling**: Does automated downscaling allow *SCENT* to cover more code and more importantly, find more vulnerabilities? (§4.7.3)
- **Effectiveness of model checking**: Does bounded model checking offer better scalability than bounded testing? (§4.7.4)

4.7.1 Evaluation setup

Implementation details. Our implementation of *SCENT* is built on a set of open-sourced program analysis platforms and tools. The static data-flow analysis used by Model Generator is built upon kernel-analyzer [104]. The concolic execution engine used by Model Transformer is built on top of KLEE [20] with the Z3 SMT solver [111]. The instrumentation is built on top of the LLVM compiler

infrastructure [64]. The bounded model checking is done using the SPIN model checker [52].

Stub function abstraction and crafting a standalone system. As discussed in §4.4, we abstract a few functions to facilitate scalability and make the code amenable to model checking. The details are listed below. It initially takes us longer with Linux as we were finalizing the methodology. However, it took us only 2.5 weeks to build the model for FreeBSD afterwards.

- *Mutex and Lock related functions*: Use empty function (During model checking, TCP is executed as a single thread process).
- *Memory allocation*: Pre-allocate the memory and return the corresponding memory object (because the model checker cannot track dynamic memory).
- *Memory release*: Use empty function (because we preallocated the memory).
- *Callback functions*: Use empty functions (as limited by the state explosion issue, we only consider one interleaving of concurrent events; therefore, we can focus on TCP mechanisms).
- *Out of scope functions (IP layer or User space)*: Use empty function body or craft abstractions manually, to send packets with TCP layer information.
- *Functions that include assembly code*: Either replace with `glibc` functions or abstract them based on their logic (examples include printk or __memcpy). Since we need to keep the model deterministic, we replace prandom_u32 function with a fixed but arbitrarily chosen value.
- *Timer*: Return the fixed value captured from a snapshot based on a real connection (e.g., for `tcp_current_mss`). This helps eliminate the non-determinism in the model as well as the otherwise intractable state space (time as a new dimension) that we are not interested in.

In general, manually abstracting functions results in a risk of missing vulnerabilities (lowered true positives); however, this step is necessary to ensure the feasibility of model checking. If these

manual abstractions cause false positives, they can be easily verified (in our experiments, we did not encounter such cases).

Testbed. We evaluated *SCENT* on two servers, each with a 2.6GHz (8-core) CPU and 128G memory. The secrets of interest are tabulated in Table 4.1. We consider the following scenarios: (1) the attacker has established its own TCP connection with the server; (2) the attacker sends packets to an open port at the server; and (3) the attacker sends packets to a closed port at the client. We assume that the attacker can either send packets to the client or the server, but not send to both[5]. We consider 6 different settings with regards to the victim's secret attributes and thus, with the three attacker scenarios, we run 36 experiments for each model. For the Linux model, we set 2 days as a hard limit for each experiment (given that it is more complex); while for the FreeBSD model, we set 1 day as a hard limit. We point out that these limits were imposed based on the computation capacity available, and to obtain results within a reasonable time frame. Based on the counter examples found by *SCENT*, we set up two virtual hosts (Debian OS with Linux kernel 4.8.0 and FreeBSD OS with kernel 13.0) to validate their veracity in real settings.

4.7.2 Discovered side-channels

Table 4.3 shows the violations found during our experiment. *SCENT* discovered a total of 53 distinct violations. Our manual verification confirmed that they are all true positives.

These violations relate to a total of 15 side-channels, of which 10 are found in Linux and 5 in FreeBSD. Here we define "a side-channel" as a branch that causes the violation. Since the same check over a shared variable can be applied at multiple branches, the same key constraint in Table 4.3 can be associated with multiple side-channels.

[5]The latter cases can be handled by *SCENT* but we leave such evaluations for future work.

Five of the discovered side-channels (4,6,7,8,11) are based on shared variables that are not discovered before, namely, close port reset counter, tcp memory counter, and accept queue associate with Listen socket (details to follow). Seven side-channels (1,2,3,5,9,10,12) are new ways (i.e., execution path) to exploit known shared resources [34, 11, 114]. The remaining three side-channels (13,14,15) are known ways to exploit a known shared resource [21].

Based on the shared resources, the side-channels can be categorized into 4 classes. Next, we describe the details and provide an examplar to showcase in each case.

Reset counter based side-channels (Class A). Side-channel 1, 2, and 3 in Table 4.3 are caused by what is called the "open port RST packet rate," which is used to restrict sending too many RST packets from an open port. Side-channel 4 is caused by what is called the "close port RST packet rate," which is used to restrict sending too many RST packets from a closed port at a host.

Figure 4.7 shows how side-channel 1 can be exploited to infer the port number of a victim connection. During the guess phase, the attacker sends a spoofed ACK packet with a guessed port number. As shown in the left part of the figure, if the port number is the one used in the victim connection, the server will either accept or drop the packet (depending on the SEQ and ACK numbers); the response is sent to the client if appropriate. If the port number in the packet is not used (right part of the figure), the server determines that an ACK was received before any SYN packet. It therefore drops the packet but responds to the client with an OPENPORT RST packet. Because of this, the OPENPORT RST Counter is increased by one. Subsequently, in a check phase, the attacker will send 200 SYN-ACK packets to exhaust the OPENPORT RST limit (in 1 second) and observes the number of resposnes (RSTs) received from the server. If the attacker receives 200

RSTs, it means that the victim client is using that port number to communicate with server; else, it infers that the port number that it had guessed is incorrect.

SYN-backlog-based side-channels (Class B). The SYN backlog is a buffer that stores half-opened TCP sockets from connections during the three-way-handshake. Because the SYN backlog is associated with the "Listen" socket, its state is shared by all connections to the server. In order to prevent DoS attacks, the size of the SYN backlog is constrained to a shared limit. When the number of half-opened sockets has reached this limit, the SYN backlog buffer will either remove an old element or directly drop the current one (based on the OS kernel used, i.e., FreeBSD or Linux).

Side-channels 5 and 9 in Table 4.3, are caused because of this feature, exploiting which an attacker can infer the port number of a victim connection. Side-channels 10 and 11 can be used to infer the port number, SEQ number and ACK number; however it is practically hard to do so since the attacker needs to guess all three secrets simultaneously (which leads to a prohibitive search space).

To illustrate, let us consider the side-channel 9 as an example, which can be used to infer the port number of the victim connection. To begin with, the attacker establishes a number of half-opened sockets to just leave enough space for one additional spot in the SYN backlog buffer. Next, as shown in Figure 4.8, the attacker sends a spoofed SYN packet to server pretending to be the victim. If the guessed port number is already used in an established connection (i.e., the server and the client are communicating), the server will drop the SYN or send a challenge ACK, without allocating a new half-opened socket (as shown in the left part of the figure). Otherwise, a half-open socket is allocated and this makes the buffer full (as shown in the right part of the figure). Subsequently, the attacker sends a SYN packet with its own IP address towards creating a new half-

opened socket, but more importantly to check whether the SYN backlog is full. Because Linux implements a LIFO (Last In First Out) algorithm to constrain the buffer size, if SYN backlog is full (as shown in the right part of the figure) , the server simply drops this new request for a half-opened socket without sending a response (assuming that SYN cookies are not enabled, which is common among quite a few cloud servers [13, 28]). Otherwise, the server will respond with a SYN-ACK to attacker.

Different from the Linux kernel, FreeBSD implements SYN backlog as a FIFO (First In First Out) buffer; this implies that an old half-opened socket will be dropped if the buffer is full. In this case, before sending the spoofed SYN packet, the attacker needs to *plant* its own half-opened socket first (via a legitimate SYN). After sending the spoofed SYN, it can infer whether buffer is full by checking if the previously *planted* half-opened socket still exists, by sending an ACK packet. Similar to the case with Linux, here we again assume that SYN cookies are not enabled.

TCP memory-counter-based side-channels (Class C). Side-channels 6, 7, and 8 are caused by a new shared variable discovered by *SCENT*. We refer to them as the TCP memory-counter-based side-channels. As shown in Table 4.3, all three vulnerabilities require an attacker to guess port number, SEQ number (in-window) and ACK number simultaneously, therefore they are not quite practical. Information leakage in this class are due to a global variable, viz., `tcp_memory_allocated`, which can be changed by any TCP connection. Table 4.3 depicts two key constraints associated with this variable: (a) `sysctl_tcp_mem[1]` indicates that currently the memory is under pressure, while (b) `sysctl_tcp_mem[2]` is used to indicate if the current allocated memory has reached a hard limit (thus, the server will drop data packets that need additional memory allocation). The different values of the above global variable can lead to different control flows, which in turn cause

the server to send different packets to the attacker (in response to specific sequences of inputs). To exploit this feature, the attacker will first send a spoofed packet to try to change this global variable. The changes occur only when the secret attributes of interest (i.e., SEQ number, ACK number, and port number) are guessed correctly. Subsequently, the attacker sends its own packets to try to reach the aforementioned limit; it can observe if the global variable has changed, based on the patterns of packets that are received. A change indicates that its guess of the secret attributes holds true.

To showcase this class of side channels, we sketch an exemplary case study shown in Figure 4.9. First, the attacker subsumes (pre-allocates) a large volume of memory before the attack. Next, the attacker sends a spoofed long data packet with a guessed SEQ number and ACK number. If the SEQ number is in window and the ACK number is correct (as shown on the left), the long data payload is stored in a queue that holds out-of-order packets (packets that are in window but are not equal to the next expected packet i.e., `rcv_next`) causing an increase in the `tcp_memory_allocated` counter; otherwise, the server will simply drop the packet (as shown on the right). During a subsequent probing phase, the attacker deliberately sends an out-of-order packet with a large data payload on its own connection. This is designed to significantly increase `tcp_memory_allocated`. If `tcp_memory_allocated` has increased before (in the previous step) causing the server to reach its hard memory limit, it will cause a droppage of this packet; otherwise, the attacker will receive an ACK packet from server. Therefore, attacker can infer whether the guesssed secret attributes (SEQ number and ACK number) in the spoofed packet are correct or not.

Challenge counter based side channel (Class D). Side-channel 12 is a new one that is similar to previously reported old ones (13, 14, 15). Here, we explicitly include the challenge ack mechanism

Table 4.4: Branch Coverage Information Before and After Transformation

Kernel	Before Transformations		After Transformations		Increase Rate
	Num	Rate	Num	Rate	
Linux	476	36.62%	598	46.00%	25.63%
FreeBSD	618	33.59%	781	42.45%	26.38%

in Linux 3.8.0 towards validating previously reported side channels [21]; these are based on a global variable called `challenge_count` and have been extensively described in [21]. Furthermore, this has already been patched in Linux and other OSes.

4.7.3 Effectiveness of automated downscaling

Automated downscaling is the core innovation of *SCENT* that improves the code coverage of bounded model checking. In this subsection, we evaluate the effectiveness of this technique.

Table 4.4 shows the branch coverages achieved before and after the transformation of automated downscaling. The branch coverage rate was increased by 25.63% with regards to the Linux kernel and by 26.38% with the FreeBSD kernel. Although the final branch coverage rate is seemingly low at 46.00% (as in Linux model), during our manual analysis, we found that many of the uncovered branches were due to our limited input space. Specifically, we did not explore paths related to header options, paths that involve the server actively sending packets, paths that are related to connection termination before the "Closed" state, etc. If we discard these branches (which we do not expect to cover) the branch coverage rate improves to around 70%.

Besides code coverage, a more important question is whether automated downscaling enables *SCENT* to discover more side-channels. The second last column in Table 4.3 shows the answer

to this question. In fact, *none of the side-channels can be found without automated downscaling* (all require it). We believe that this highlights the importance and effectiveness of our technique.

4.7.4 Performance of model checking

One important design choice we made when building *SCENT* is whether to use bounded testing [66], wherein we can directly test an unmodified kernel, or use bounded model checking. The benefit of model checking is that it will visit each state only once, thereby avoiding the execution of redundant steps and improving the performance of testing. In this subsection, we compare the performance of bounded model checking with bounded testing, in terms of number of iterations. Figure 4.10 shows the result. Basically, bounded model checking executes 4 orders of magnitude fewer iterations than blind enumeration (i.e., bounded testing).

The next choice we made, that is related to the performance of model checking, is imposing a limit on the number of packets to be enumerated during bounded model checking. Figure 4.11 shows how the time of one round of model checking increases as the number of packets increases. Figure 4.12 shows how the memory usage of the model checker increases as the number of packets increases. When the number of packets increase to 4, it will either take too long to test all the different configurations or exhaust all the memory on the testbed.

4.8 Case Study

When the port number is leaked, an attacker can infer whether the victim client is communicating with the server (either during the three way handshake or in ESTABLISHED state). This leaks the victim user's privacy. Side-channels 1, 3, 4, 5, 9 can leak port number information, and can

therefore be used to achieve this attack. In the previous section, we discussed how such an attack can be launched. We now construct a real attack to demonstrate the impact of the corresponding side-channels found by *SCENT*.

As an exemplar, we pick side-channel 1 (as shown in Figure 4.7), and evaluate it in terms of metrics such as success rate and the time to succeed. In our experiment, we used a Ubuntu 14.04 host on a university campus as the victim client. The victim server is a virtual machine running FreeBSD OS from a different Ubuntu 14.04 host. The attack machine is a Ubuntu 16.04 host on the same campus. The steps in the attack process are listed below:

1. Synchronize machine times between attacker and server;
2. Send spoofed and unspoofed ACK packets to linearly guess a port number range based on the number of RST packets received;
3. Given a port number range, use binary search to locate the specific port number.

The attacker can guess 200 different port numbers (via spoofed packets) in one second; otherwise, spoofed packets will always reach the reset counter limit. The attacker can guess the port number starting from the Ephemeral port range [31], and then guess the remainder of the port range. Our experiment shows that this attack of inferring a correct port number is achievable within an average time of 73 seconds with a 100% success rate.

4.9 Related Work

TCP side channel attacks. In the last decade, several TCP side channels have been manually found by researchers. These side channels can be utilized to (1) cause a TCP inference attack [41, 79, 80, 21, 27], which in turn can lead to a hijack of the connection and injection of malicious data;

(2) measure host attributes without exposing the attacker's IP address (examples include performing an idle port scan [34] or measuring the RTT between two hosts [11]). Roughly these distinct attacks can be mapped onto the exploitation of four categories of side-channel vulnerabilities: (1) Shared rate limit: these side-channels relate to a rate limit that is shared across the victim and an off-path attacker connection, such as IPID counter [16, 26, 82, 41, 73, 115], the challenge ACK rate limit [21], the reset rate limit and the shared SYN backlog queue limit [34, 11]. (2) System-wide packet counter: As the name suggests a packet counter is shared globally in these cases [79, 80]. (3) Wireless link: Wireless contention results in information leakage in these cases [27] (timing-based side channel). (4) Browser implementation's feature: A per destination port-counter and a FIFO HTTP request queue cause information leaks [43].

While most of these side-channels are discovered manual by domain experts, *SCENT* aims to automate the discovery in a principled way. Our evaluation shows that *SCENT* indeed can detected (both new and known) side-channels.

Side channel detection. Most previous side-channel vulnerabilities have been discovered manually (e.g., using domain expertise). However, a few side-channel detection tools have been proposed. [24] uses static taint analysis to discover system-wide TCP packet counter side-channel vulnerabilities. Generally, static taint analysis can be guaranteed to find all true violations, but suffers from high false positives. By relying on violation of the non-interference property, *SCENT* can avoid high false positives and can detect side-channels caused by different shared variables. There are also several efforts relating to the detection of other types of side-channels but these are orthogonal to our work [102, 19].

Program analysis and testing. There are also several efforts that use program analysis (e.g., static and/or dynamic analysis) to find bugs or other types of attacks in TCP implementations [62, 56]. These are orthogonal to our work and address significantly different problems.

Model checking and formal verification have been used to analyze the robustness of TCP implementations [69, 35]; however, their objectives are signficiantly different. More importantly, *SCENT* uses automated downscaling to improve the effectivenss of model checking.

Besides bounded module checking [30, 63, 65], one could also do bounded testing [66]. The advantage of bounded testing is that it does not require additional modification to the target program, while model checking usually requires generating a model amenable to the model checker. However, as shown in our evaluations, by avoiding redundant states, a model checker can help explore a larger input space.

Program transformation has been used to assist testing using fuzzing to patch hard-to-flip branches (like checksum checks) as a way of improving code coverage [74, 103, 58]. In contrast, *SCENT* tries to coerce both true and false path to be visited and most of the target branches (Table 4.3 column 3) have simple constraints. In addition, *SCENT* changes the internal states instead of "disabling" the branch.

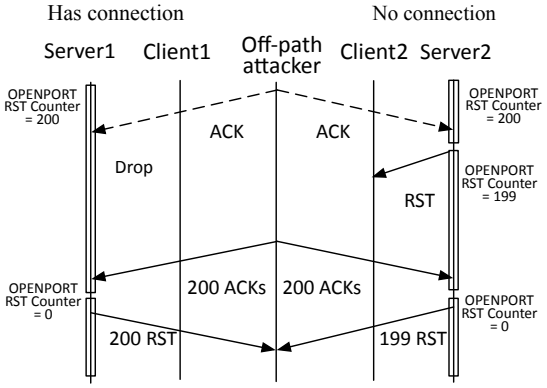


Figure 4.7: Reset counter based side channel example (Vulnerability 1, FreeBSD)

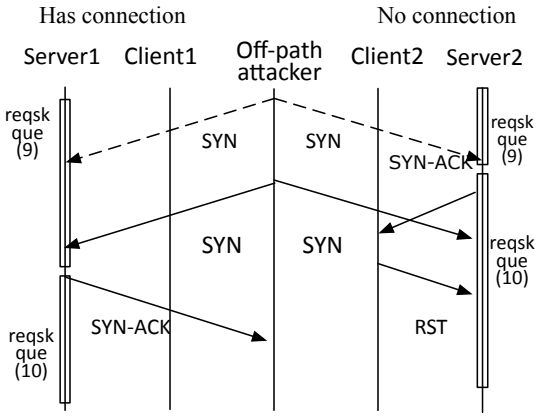


Figure 4.8: SYN-backlog based side channel example when SYN-Cookie is disabled (Vulnerability 9, Linux)

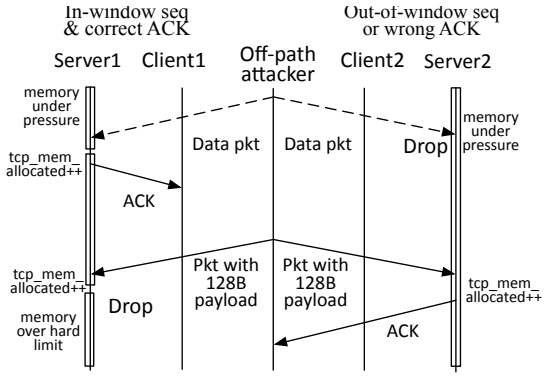


Figure 4.9: TCP memory counter based side channel example (Vulnerability 6, Linux)

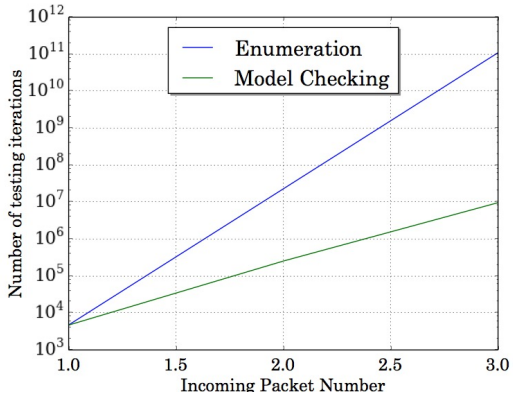


Figure 4.10: Number of testing iterations versus number of incoming packets

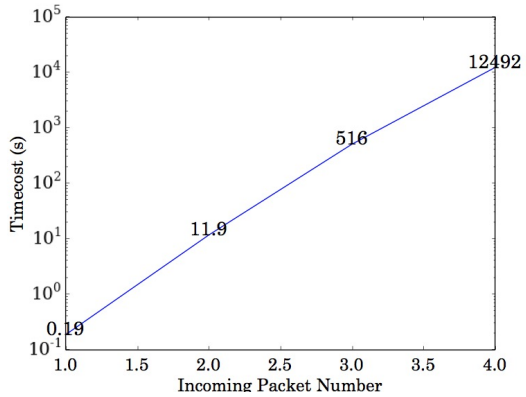


Figure 4.11: Timecost of one model checking run versus number of incoming packets

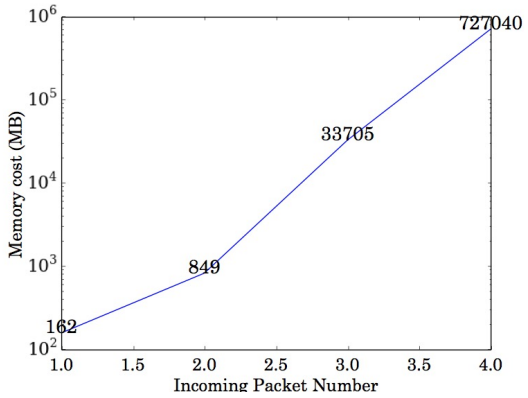


Figure 4.12: Memory cost of one model checking run versus number of incoming packets

Chapter 5

Conclusions

In this thesis, we propose a variety of techniques to identify and prevent information leakage caused by side channel vulnerabilities at different layers in the network stack.

We begin in Chapter 2, by designng and implementing *FOG*, a practical framework for obfuscation of packet headers transmitted in the open in wireless streams, using MIMO. *FOG* tracks header bits as they traverse an intricate PHY layer consisting of scrambling, application of FEC and interleaving. *FOG* provides protection against attacks that try to decipher a fixed blinding signal (as traditionally used). We show that *FOG* provides enhanced obfuscation of wireless streams compared to state of the art approaches that use MIMO to blind out entire streams; at the same time, it consumes much lower overhead.

In Chapter 3, we have discovered a subtle yet critical flaw in the design and implementation of TCP. The flaw manifests as a side channel that affects all Linux kernel versions 3.6 and beyond and may possibly be replicated in other operating systems if left unnoticed. We show that the flaw allows a variety of powerful blind off-path TCP attacks. Finally, we propose changes to the

design and implementation of TCP's global rate limit to prevent or mitigate the side channel.

In Chapter 4, we consider the challenging problem of developing a principled approach to discovering hard-to-find TCP side-channels. We use model checking as a basis for finding violations of the non-interference property between simultaneous TCP connections, which we argue is a precursor to exploitable side channels. As our main contribution, we build a tool *SCENT* that achieves our goal by addressing two hard challenges in making model checking amenable to our goal namely, (a) making a TCP code base self contained after pruning irrelevant parts and (b) systematically downscaling both the input space and the model state space by means of principled program transformations. We use the counter-examples generated by the transformed model checker in *SCENT* to discover 12 new side channels and also validate all previously discovered ones. In this work, we limit ourselves to side channels that facilitate the inference of a specific set of secret attributes (e.g., SEQ number); we will expand our threat model to find other types of vulnerabilities (e.g., idle port scans) and with more scenarios (e.g., attacker can send packets to both the client and the server and/or with different OSes) in the future.

Bibliography

[1] Advanced Encryption Standard (AES) (RFC 3826). http://www.ietf.org/rfc/rfc3826.txt.

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

[3] Cloud Messaging. https://developers.google.com/cloud-messaging/.

[4] Iphone 6 specification. https://www.apple.com/iphone-6/.

[5] [PATCH net] tcp: enable per-socket rate limiting of all 'challenge acks'. https://www.mail-archive.com/netdev@vger.kernel.org/msg119411.html.

[6] [PATCH net] tcp: make challenge acks less predictable. https://www.mail-archive.com/netdev@vger.kernel.org/msg118677.html.

[7] Rice University WARP Project. http://warp.rice.edu.

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

[9] Raz Abramov and Amir Herzberg. Tcp ack storm dos attacks. *Journal Computers and Security*, 2013.

[10] Geoffrey Alexander and Jedidiah R. Crandall. Off-Path Round Trip Time Measurement via TCP/IP Side Channels. In *INFOCOM*, 2015.

[11] Geoffrey Alexander and Jedidiah R Crandall. Off-path round trip time measurement via tcp/ip side channels. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1589–1597. IEEE, 2015.

[12] N. Anand, Sung-Ju Lee, and E.W. Knightly. Strobe: Actively securing wireless communications using zero-forcing beamforming. In *INFOCOM*, 2012.

[13] Disable tcp syn cookies.

[14] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 260–264, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

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

[16] Steven M Bellovin. A technique for counting natted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 267–272. ACM, 2002.

[17] Robert Beverly, Arthur Berger, Young Hyun, and k claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. ACM SIGCOMM IMC*, 2009.

[18] Robert Beverly, Ryan Koga, and k claffy. Initial Longitudinal Analysis of IP Source Spoofing Capability on the Internet. In *Internet Society Article*, 2013.

[19] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*, page 0. IEEE, 2019.

[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[21] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, Austin, TX, 2016. USENIX Association.

[22] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

[23] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *CCS*, 2015.

[24] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 388–400. ACM, 2015.

[25] Shuo Chen, Rui Wang, Xiaofeng Wang, and Kehuan Zhang. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[26] Weifeng Chen, Yong Huang, Bruno F Ribeiro, Kyoungwon Suh, Honggang Zhang, Edmundo de Souza e Silva, Jim Kurose, and Don Towsley. Exploiting the ipid field to infer network path and end-system characteristics. In *International Workshop on Passive and Active Network Measurement*, pages 108–120. Springer, 2005.

[27] Weiteng Chen and Zhiyun Qian. Off-path {TCP} exploit: How wireless routers can jeopardize your secrets. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1581–1598, 2018.

[28] Defenses against tcp syn flooding attacks.

[29] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[30] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. Context-bounded model checking with esbmc 1.17. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 534–537. Springer, 2012.

[31] Ephemeral source port selection strategies.

[32] Roya Ensafi, Jeffrey Knockel, Geoffrey Alexander, and Jedidiah R. Crandall. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *PAM*, 2014.

[33] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks using Model Checking. In *USENIX Security*, 2010.

[34] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[35] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[36] Tobias Flach, Ethan Katz-Bassett, and Ramesh Govindan. Quantifying Violations of Destination-based Forwarding on the Internet. In *IMC*, 2012.

[37] Jr. Forney, G.D. The Viterbi algorithm. *Proc. of the IEEE*, 1973.

[38] Matthew Gast. *802.11Ac: A Survival Guide*. O'Reilly Media, Inc., 2013.

[39] Y. Gilad, A. Herzberg, and H. Shulman. Off-Path Hacking: The Illusion of Challenge-Response Authentication. *Security Privacy, IEEE*, 2014.

[40] Yossi Gilad and Amir Herzberg. Off-Path Attacking the Web. In *USENIX WOOT*, 2012.

[41] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *WOOT*, pages 41–52, 2012.

[42] Yossi Gilad and Amir Herzberg. Spying in the Dark: TCP and Tor Traffic Analysis. In *PETS*, 2012.

[43] Yossi Gilad and Amir Herzberg. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW*, 2013.

[44] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[45] S. Goel and R. Negi. Guaranteeing secrecy using artificial noise. *IEEE Trans. on Wireless Communications*, June 2008.

[46] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.

[47] B. Greenstein, R. Gummadi, J. Pang, M. Chen, T. Kohno, S. Seshan, and D. Wetherall. Can Ferris Bueller Still Have His Day Off? Protecting Privacy in an Era of Wireless Devices. In *Proceedings of HotOS*, 2007.

[48] Ben Greenstein, Damon McCoy, Jeffrey Pang, Tadayoshi Kohno, Srinivasan Seshan, and David Wetherall. Improving wireless privacy with an identifier-free link layer protocol. In *ACM MobiSys*, 2008.

[49] Bing Han and Jonathan Billington. Termination properties of TCP's connection management procedures. In *ICATPN*, 2005.

[50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th International Conference on Model Checking Software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[51] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 597–607, May 1999.

[52] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[53] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, pages 76–91, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[54] G. Jakllari, S. V. Krishnamurthy, M. Faloutsos, P. V. Krishnamurthy, and O. Ercetin. A framework for distributed spatio-temporal communications in mobile ad hoc networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13, April 2006.

[55] Umar Javed, Italo Cunha, David Choffnes, Ethan Katz-Bassett, Thomas Anderson, and Arvind Krishnamurthy. Poiroot: Investigating the root cause of interdomain path changes. In *SIGCOMM*, 2013.

[56] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in tcp congestion control using a model-guided approach. In *Proceedings of NDSS*, 2018.

[57] M. Junaid, Muid Mufti, and M. Umar Ilyas. Vulnerabilities of IEEE 802.11i Wireless LAN CCMP Protocol. *Enformatika*, 11:228, Feb 2006.

[58] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 782–792. ACM, 2015.

[59] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas Anderson, and Arvind Krishnamurthy. Reverse Traceroute. In *NSDI*, 2010.

[60] A. Khisti, G. Wornell, A. Wiesel, and Y. Eldar. On the Gaussian MIMO wiretap channel. In *IEEE ISIT*, 2007.

[61] Jeffrey Knockel and Jedidiah R. Crandall. Counting Packets Sent Between Arbitrary Internet Hosts. In *FOCI*, 2014.

[62] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

[63] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[64] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[65] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 146–161. Springer, 2012.

[66] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 33–50, 2018.

[67] A. Mukherjee and A.L. Swindlehurst. Robust beamforming for security in mimo wiretap channels with imperfect csi. *IEEE Trans. on Signal Processing*, Jan 2011.

[68] Amitav Mukherjee and A Lee Swindlehurst. Utility of beamforming strategies for secrecy in multiuser MIMO wiretap channels. In *Allerton Conference on Communication, Control, and Computing*, 2009.

[69] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[70] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002.

[71] Angela Orebaugh, Gilbert Ramirez, Josh Burke, and Larry Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[72] Jeffrey Pang, Ben Greenstein, Ramakrishna Gummadi, Srinivasan Seshan, and David Wetherall. 802.11 user fingerprinting. In *MobiCom*, 2007.

[73] Paul Pearce, Roya Ensafi, Frank Li, Nick Feamster, and Vern Paxson. Augur: Internet-wide detection of connectivity disruptions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 427–443. IEEE, 2017.

[74] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[75] Eldad Perahia and Robert Stacey. *Next Generation Wireless LANs: 802.11 n and 802.11 ac*. Cambridge university press, 2013.

[76] Alejandro Proano and Loukas Lazos. Selective jamming attacks in wireless networks. In *IEEE ICC*, 2010.

[77] Feng Qian, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, Oliver Spatscheck, and Walter Willinger. Tcp revisited: A fresh look at tcp in the wild. In *Proc. ACM SIGCOMM IMC*, 2009.

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

[79] Zhiyun Qian and Z Morley Mao. Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361. IEEE, 2012.

[80] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *CCS*, 2012.

[81] Zhiyun Qian, Z. Morley Mao, Yinglian Xie, and Fang Yu. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy*, 2010.

[82] Zhiyun Qian, Z Morley Mao, Yinglian Xie, and Fang Yu. Investigation of triangular spamming: A stealthy and efficient spamming technique. In *2010 IEEE Symposium on Security and Privacy*, pages 207–222. IEEE, 2010.

[83] Lin Quan and John Heidemann. On the characteristics and reasons of long-lived internet flows. In *Proc. ACM SIGCOMM IMC*, 2010.

[84] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

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

[86] Hanif Rahbari and Marwan Krunz. Friendly cryptojam: A mechanism for securing physical-layer attributes. In *ACM WiSec*, 2014.

[87] Hariharan Rahul, Haitham Hassanieh, and Dina Katabi. Sourcesync: A distributed wireless architecture for exploiting sender diversity. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 171–182, New York, NY, USA, 2010. ACM.

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

[89] T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security Symposium*, 2007.

[90] Matthias Schulz, Adrian Loch, and Matthias Hollick. Practical known-plaintext attacks against physical layer security in wireless MIMO systems. In *NDSS*, 2014.

[91] Wenbo Shen, Peng Ning, Xiaofan He, and Huaiyu Dai. Ally friendly jamming: How to jam your enemy and maintain your own wireless connectivity at the same time. In *IEEE Symposium on Security and Privacy*, 2013.

[92] Chen Shuo, Wang Rui, Wang Xiaofeng, and Zhang Kehuan. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[93] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, 2001.

[94] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security*, 2001.

[95] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 133–142. ACM, 2004.

[96] Q. Sun, D.R. Simon, Yi-Min Wang, W. Russell, V.N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*, 2002.

[97] A.L. Swindlehurst. Fixed SINR solutions for the MIMO wiretap channel. In *IEEE ICASSP*, 2009.

[98] Nils Ole Tippenhauer, Luka Malisa, Aanjhan Ranganathan, and Srdjan Capkun. On limitations of friendly jamming for confidentiality. In *IEEE Symposium on Secuirty and Privacy*, 2013.

[99] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, NY, USA, 2005.

[100] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[101] Mathy Vanhoef, Célestin Matte, Mathieu Cunche, Leonardo S. Cardoso, and Frank Piessens. Why mac address randomization is not enough: An analysis of wi-fi network discovery mechanisms. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 413–424, New York, NY, USA, 2016. ACM.

[102] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 235–252, 2017.

[103] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[104] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with {KINT}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 163–177, 2012.

[105] Charles V. Wright, Lucas Ballard, Fabian Monrose, and Gerald M. Masson. Language identification of encrypted VoIP traffic: Alejandra y roberto or alice and bob? In *USENIX Security Symposium*, 2007.

[106] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *J. Mach. Learn. Res.*, 2006.

[107] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. 802.11w - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 4: Protected management frames. *IEEE Standard for Information technology*, 2009.

[108] Qiuyu Xiao, Michael K. Reiter, and Yinqian Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *CCS*, 2015.

[109] Jie Xiong and Kyle Jamieson. SecureArray: Improving Wifi security with fine-grained physical-layer information. In *MobiCom*, 2013.

[110] Taesang Yoo and Andrea Goldsmith. On the optimality of multiantenna broadcast scheduling using zero-forcing beamforming. *IEEE JSAC*, 2006.

[111] The z3 theorem prover.

[112] Kai Zeng, Daniel Wu, An Chan, and Prasant Mohapatra. Exploiting multiple-antenna diversity for shared secret key generation in wireless networks. In *IEEE INFOCOM*, 2010.

[113] Xu Zhang, Jeffrey Knockel, and Jedidiah R. Crandall. Original SYN: Finding Machines Hidden Behind Firewalls. In *INFOCOM*, 2015.

[114] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. Original syn: Finding machines hidden behind firewalls. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 720–728. IEEE, 2015.

[115] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. Onis: Inferring tcp/ip-based trust relationships completely off-path. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2069–2077. IEEE, 2018.

[116] Yao Zheng, Matthias Schulz, Wenjing Lou, Thomas Hou, and Matthias Hollick. Highly efficient known-plaintext attacks against orthogonal blinding based physical layer security. *IEEE Wireless Communications Letters*, 2015.