UC Riverside

UC Riverside Electronic Theses and Dissertations

Title

Side-Channel Isn't Sad Anymore: Towards the Leak-Free Network Stack---From DNS and Beyond

Permalink

https://escholarship.org/uc/item/7mq9r73n

Author

Man, Keyu

Publication Date

2023

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Side Channel Isn't Sad Anymore: Towards the Leak-Free Network Stack—From DNS and Beyond

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

Doctor of Philosophy

in

Computer Science

by

Keyu Man

December 2023

Dissertation Committee:

Dr. Zhiyun Qian, Chairperson
Dr. Heng Yin
Dr. Chengyu Song
Dr. Nael Abu-Ghazaleh

Copyright by
Keyu Man
2023

The Dissertation of Keyu Man is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The journey towards completing my Ph.D. was paved with invaluable contributions from numerous individuals, to whom I extend my heartfelt gratitude.

Firstly, profound gratitude goes to my Ph.D. advisor, Dr. Zhiyun Qian. Your unwavering support, insightful guidance, and expert advice played an indispensable role in my academic achievements, including the submission of three papers to top security conferences and it was also you who unleashed my potential in the cybersecurity field. Moreover, your mentorship in areas beyond academia, particularly your emphasis on aiming high and practicing patience, have left an indelible mark on me. I consider myself fortunate to have embarked on this journey. No matter how my life goes after graduation, my Ph.D. career was a wonderful and successful experience of my life.

Secondly, I extend my sincere thanks to my co-authors: Dr. Yue Cao, Dr. Haixin Duan, Dr. Zhongjie Wang, Mr. Shenghan Zheng, Mr. Xiaofeng Zheng, and Mr. Xin'an Zhou[1]. Your foundational contributions were paramount to the success of our collaborative endeavors. Your unwavering support, especially during critical submission deadlines, was critical.

Thirdly, my profound appreciation goes to my spouse, Mr. Bowen Lu. Your unwavering emotional support, especially during challenging times, has been my anchor. Your presence has brought balance and stability to my personal life, for which I am eternally grateful.

Forthly, a warm acknowledgment to my family and friends. While I choose not to list names out of privacy concerns, please know that your support and companionship

[1] Unless otherwise noted, names are ranked alphabetically by the last name.

have been invaluable. We spent a great leisure time during the Ph.D. career together and you gave me lots of support aside from research. From playing video games and traveling across the U.S. to playing tennis, you all have added joy and confidence to my life. Special thanks should be given to my parents, who gave the life and raised me up.

Fifthly, I owe a debt of gratitude to my lab mates: Dr. Weiteng Chen, Mr. Qing Deng, Mr. Xingyun Du, Mr. Yu Hao, Mr. Guoren Li, Mr. Haonan Li, Mr. Xingyu Li, Mr. Juefei Pu, Dr. Yizhuo Zhai, Mr. Zheng Zhang, Mrs. Jinmeng Zhou, Mr. Pengxiong Zhu, Dr. Shitong Zhu, and Mr. Xiaochen Zou. Special mention goes to Mr. Yu Hao, my roommate of five years. I will never forget the days we settled down at Riverside and took bus to setup utility accounts under $100^{\circ}F$ weather. Besides, despite we never collaborated on my projects, as a program analysis expert, your help made me avoid pitfalls multiple times. For the rest my beloved lab mates, the shared meals and engaging discussions on research topics have enriched my Ph.D. experience immeasurably and your influence to my life was unforgettable.

Sixthly, heartfelt thanks are extended to Dr. Qi Alfred Chen. While our endeavors in connected vehicles research did not culminate in a publication, your mentorship has been invaluable. Your guidance has illuminated my journey towards earning a Ph.D. degree, and for that, I am deeply grateful.

Seventhly, my appreciation is directed to my dissertation defense committee members: Dr. Nael Abu-Ghazaleh, Dr. Chengyu Song, and Dr. Heng Yin. Your feedback not only enriched my dissertation but also trained my skills in presenting research to a diverse

audience. Your perspectives added depth and breadth to my work, enhancing its value to the wider academic community.

Last but not least, I extend my gratitude to the National Science Foundation (NSF) for funding my research endeavors. In an era driven by commercial pursuits, securing funding for pure research has been pivotal in ensuring the continuity and integrity of my projects. I sincerely hope humanity continues to channel its energies into the quest for knowledge and enlightenment, rather than conflict and strife.

Bibliographical Notes. The thesis mainly composes of the research papers that were mostly authored by myself. Specifically, Chapter 3 is the reproduction of “DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels” [89], which is originally published on ACM Conference on Computer and Communications Security 2020 (CCS‘20). Chapter 4 is the reproduction of “DNS Cache Poisoning Attack: Resurrections with Side Channels” [90], which is originally published on ACM Conference on Computer and Communications Security 2021 (CCS‘21). Chapter 5 is the reproduction of “`SCAD`: Towards a Universal and Automated Network Side-Channel Vulnerability Detection”, which is under submission to The 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI’23), at the time when this thesis is being finalized. Minor changes may apply during the reproduction to improve the integrity of the thesis.

To my beloved dogs River & Turbo, humanity and our civilization.

ABSTRACT OF THE DISSERTATION

Side Channel Isn't Sad Anymore: Towards the Leak-Free Network Stack—From DNS and Beyond

by

Keyu Man

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2023
Dr. Zhiyun Qian, Chairperson

Network side channels have emerged as a notable threat vector in computer security, often bypassing conventional safeguards due to their elusive nature. In such attacks, the attacker leverages unintentionally leaked transformed information to derive the confidential secret on the victim. This research delves into the intricacies of these attacks, with an emphasis on DNS systems. While the vulnerabilities posed by TCP side channels have been somewhat explored, this thesis unveils the broader spectrum, especially emphasizing on UDP and ICMP side channels. The discovered flaws based on temporal and spatial shared resources lead to potent DNS cache poisoning attacks by effectively circumventing ephemeral port randomization defenses, which remains the critical defense of Dan Kaminsky's attack, rendering large portions of the Internet's open resolvers vulnerable. Specifically, 34% of these, including popular public resolvers like Quad 9, were found vulnerable.

Given these revelations, the pressing need for a robust, universal and automated detection tool became evident. Addressing this, the research introduces `SCAD`, an automated tool built upon a novel methodology termed under-constrained dynamic symbolic

execution. SCAD identifies violations of the non-interference property, recognized as the underpinning of network side channels. Without relying on comprehensive prior modeling and domain knowledge, SCAD scrutinizes multiple TCP and UDP implementations among Linux, FreeBSD and lwIP, discovering 14 network side channels, including seven previously undetected ones, at a false positive rate of 17.6%. The results reveal serious vulnerabilities, including those that can be used to compromise the previously patched Linux and FreeBSD kernels, making them susceptible to SADDNS attacks or off-path TCP exploits again.

Collectively, this thesis aims to enrich our comprehension of network side channels, paving the way for more fortified defenses in the realm of computer network security.

Contents

List of Figures **xiii**

List of Tables **xiv**

1 Introduction **1**

2 Off-Path Network Attacks **5**

2.1 Definition & Threat Model 5

2.1.1 Definition 5

2.1.2 Threat Model 7

2.2 History of Off-Path Network Attacks 8

3 SADDNS: Revive DNS Cache Poisoning Attacks With Temporal ICMP Rate Limit Counter **12**

3.1 Introduction 12

3.2 Current State of DNS Cache Poisoning Attacks 15

3.2.1 State-of-the-Art(SOTA) Defenses 16

3.2.2 New Attack Surface in the DNS Hierarchy 17

3.3 Attack Overview 18

3.3.1 Attack Workflow 19

3.4 Inferring DNS Query's Source Port 20

3.4.1 Analysis of UDP Source Port Scannability 20

3.4.2 ICMP Rate Limit Challenge 22

3.4.3 Public-Facing Source Port Scan Method 23

3.4.4 Private Source Port Scan Method 28

3.4.5 Vulnerable DNS Forwarder and Resolver Population 30

3.5 Extending the Attack Window 33

3.5.1 Extending Window in a Forwarder Attack 33

3.5.2 Extending Window in a Resolver Attack 35

3.6 Practical Attack Considerations 39

3.6.1 Bypassing the TTL of Cached Records 39

3.6.2 Timeouts and Retransmitted Queries 40

3.6.3 Handling Multiple Authoritative Nameservers 42
3.6.4 Handling Multiple Backend Servers Behind DNS Resolvers 42
3.7 End-to-End Attacks 43
3.7.1 Attacking a Forwarder (Home Router) 43
3.7.2 Attacking a Production Resolver 45
3.8 Discussion 49
3.8.1 Attack Against Unbound vs. BIND 49
3.8.2 UDP Source Port Inference on Other Operating Systems 50
3.8.3 Other Vulnerable Protocols 50
3.9 Best Practices in Configuring Response Rate Limiting (RRL) 50
3.9.1 Defenses 51
3.10 Conclusion 53

4 `SADDNS` 2.0: Resurrect DNS Cache Poisoning Attacks With Spatial Next Hop Exception Cache 54
4.1 Introduction 54
4.2 Background 56
4.3 Attack Overview 58
4.4 ICMP-Based Ephemeral Port Scans 58
4.4.1 Analysis of ICMP Error Processing Logic 60
4.4.2 Public-Facing Port Number Inference 61
4.4.3 Private-Facing Port Number Inference 62
4.4.4 Finding IPs That Cause Hash Collisions 64
4.4.5 High-Speed Scans 65
4.5 Vulnerable Population 67
4.5.1 Conditions of Successful Attacks 67
4.5.2 Open Resolvers 71
4.6 Practical Concerns 77
4.6.1 Small Attack Window 77
4.6.2 Multiple Nameservers 77
4.6.3 Multiple Backend Servers 79
4.6.4 Dual-Stack Resolvers 80
4.6.5 Noises 80
4.7 Evaluation 82
4.7.1 Resolver Attack 83
4.7.2 Other Attacks 87
4.8 Discussion 87
4.8.1 Comparison With `SADDNS` 87
4.8.2 PMTUD and DNS 89
4.8.3 New Defenses Against `SADDNS` 2.0 90
4.8.4 Ethical Concerns 91
4.9 Conclusion 92

5 SCAD: a Universal and Automated Network Side-Channel Vulnerability Detection Tool **93**
5.1 Introduction . . . 93
5.2 Insight & SCAD Overview . . . 95
5.2.1 Non-Interference Property . . . 95
5.2.2 Existing Approaches . . . 96
5.2.3 SCAD's Novel Mode of Symbolic Execution . . . 97
5.2.4 SCAD Architecture . . . 100
5.3 DSE Component . . . 100
5.3.1 Minimalistic Modeling of the Target . . . 101
5.3.2 DSE Component Workflow . . . 102
5.4 NIPVC Component . . . 103
5.4.1 Path-Level Violation Checker . . . 103
5.4.2 NIPVC Loop . . . 106
5.5 Implementation . . . 107
5.6 Evaluation . . . 108
5.6.1 Evaluation Platform & Setup . . . 108
5.6.2 Comparison with SCENT . . . 109
5.6.3 Side-Channel Detection on SOTA TCP Implementations . . . 112
5.6.4 Side-Channel Detection on UDP Implementation . . . 113
5.7 Case Study . . . 114
5.7.1 Timestamp-Based Side Channels . . . 114
5.7.2 Randomness-Based Side Channel . . . 118
5.7.3 Queue-Length-Based Side Channels . . . 120
5.7.4 Responsible Disclosure . . . 122
5.8 Mitigation . . . 122
5.9 Discussion . . . 124
5.9.1 Limitations . . . 124
5.9.2 Future Work . . . 126
5.10 Conclusion . . . 127

6 Related Work **129**

7 Conclusions **130**

Bibliography **133**

A ICMP Redirect Attacks **143**

B ICMP Rate Limit **145**

C Resetting the Exception Cache State **146**

List of Figures

3.1 DNS Infrastructure With Multiple Layers of Caching 18
3.2 SADDNS Attack Workflow 19
3.3 Fast Port Scanning of an Open Source Port 25
3.4 Fast Port Scanning of a Private Source Port 29
3.5 Example Rogue Response Acceptable by a Forwarder 34
3.6 Response Loss Rate Under Different Query Rate 37
3.7 DNS Response Used to Overwrite Cache 39

4.1 Ephemeral Port Scan 59
4.2 SADDNS 2.0 Ephemeral Port Number Inference 61

5.1 SCAD Architecture 100
5.2 An Illustrative Example of Path Tree 103
5.3 Exploits for Newly Found Side Channels by SCAD 115

List of Tables

2.1 Off-Path Network Attacks 9

3.1 DNS Forwarder Behaviors in Home Routers 30
3.2 SADDNS Vulnerable Status of Public Resolvers 31
3.3 SADDNS Production Resolver Attack Results 45

4.1 Exploitability of Different DNS Software and Kernel Versions 68
4.2 SADDNS 2.0 Vulnerable Status of Public Resolvers 76
4.3 SADDNS 2.0 Resolver Attack Results 83

5.1 Comparison Among Program Analysis Techniques 98
5.2 Side Channels Reported by SCAD 110
5.3 Statistics of SCAD on Different Targets 110

Chapter 1

Introduction

The evolution of computer networks has given rise to sophisticated attack vectors, bypassing conventional security measures. Among these, network side channels stand out for their subtle and elusive nature. Historically, the side-channel attack has always been an understated security threat. Such attacks enable adversaries to derive confidential information from victims without directly accessing it. Instead, they exploit the data unintentionally leaked through various channels. As computer systems became more intertwined, side channels began exploiting these intricate interactions, making them increasingly powerful and problematic [8, 77, 84, 46, 99, 89, 90, 22, 104, 97]. As the digital landscape expanded, the emphasis on understanding and mitigating such threats has grown significantly.

Domain Name System (DNS), which translates human-readable domain names into machine-readable IP addresses, is paramount for the modern Internet. It underpins various security services and its integrity impacts a vast range of applications and security protocols, from certificate issuance [7] to routing security [82]. Given its significance, com-

promising DNS can lead to extensive security failures, unraveling the very fabric of Internet trust. Kaminsky's discovery in 2008 [71] showcased the potential of DNS cache poisoning, where an off-path attacker could inject spoofed DNS responses, misleading resolvers.

Despite the advancements in DNS security, the system remains fragile. Measures such as DNSSEC and DNS cookies [5], though standardized, saw limited deployment due to compatibility concerns [33, 29]. This fragile nature of DNS, paired with the innovation in side channel attacks, poses severe risks to the entirety of the digital domain.

This thesis delves deep into two novel side-channel-based DNS cache poisoning attacks: `SADDNS` and `SADDNS` 2.0. Our exploration began with the analysis of the interaction between application and OS-level behaviors. We identified a shared resource—specifically, the ICMP global rate limit—that could be manipulated to infer the UDP ephemeral port number, a critical component for DNS security, when probing with UDP packets. This discovery led to the development of `SADDNS`, an attack that could poison a significant fraction of the Internet's DNS resolvers.

Building upon this discovery, our research extended into another dimension of DNS vulnerabilities with `SADDNS` 2.0. In this iteration, the emphasis was on the often-overlooked ICMP packets, which, when combined with subtle interactions across the ICMP, UDP, and application layers, revealed even more potent side channels, through the cache of the next hop exceptions (`fnhe`).

These discoveries highlight the vulnerabilities in popular DNS software as well as mainstream OSes, posing potential risks to a vast number of systems, from DNS forwarders on home routers to major DNS resolvers. The attacks were proven effective, requiring only

minutes to succeed, and affecting major components of the Internet infrastructure. From our experiment, 34% of open resovlers, including 12/14 popular public resolvers like Quad 9, were found vulnerable to either of two `SADDNS` attacks.

To address the profound vulnerabilities unveiled by `SADDNS` and `SADDNS` 2.0, the need for an automated, robust, and universal detection tool became indispensable.

This research introduces `SCAD`, a powerful analysis tool for automated side-channel detection. By harnessing symbolic execution, `SCAD` delves deep into the state space of target protocol implementations, systematically identifying potential non-interference property violations, which is the root cause of side channels, at path-level. However, symbolic execution inherently faces scalability challenges. To circumvent these, we innovated a new symbolic execution mode — under-constrained dynamic symbolic execution, enabling simultaneous exploration of diverse state spaces while managing the notorious path explosion problem.

On application, `SCAD` delivered promising results. It identified 14 network side channels, seven of which were previously unknown, with a false positive(FP) rate only of 17.6%, highlighting the tool's effectiveness and the depth of vulnerabilities within the network domain.

Contributions. This thesis presents a comprehensive exploration of network side channels, unveiling severe threats, and proposing robust solutions. The main contributions are:

1. The discovery and extensive analysis of two novel side-channel-based DNS cache poisoning attacks, `SADDNS` and `SADDNS` 2.0, emphasizing their potential risks and ramifications.

2. The introduction of `SCAD`, a groundbreaking tool for automated side-channel detection, backed by an innovative symbolic execution mode, addressing the long-standing challenges of automated side-channel detection.

3. A detailed exploration of the root causes of identified side channels, their implications, and the development of effective mitigation strategies.

This journey underscores the imperativeness of understanding and addressing network side channels, marking a significant step towards fortifying the future of computer network security.

Organization. Chapter 2 will introduce the off-path attacks and side-channel attacks. Chapter 3 & 4 will discuss the intricacies of `SADDNS` and `SADDNS` 2.0 respectively. Chapter 5 will explore the methodology and results of `SCAD`. Chapter 6 will discuss the related work. The Chapter 7 will summarize the findings and outline potential future directions in this domain.

Chapter 2

Off-Path Network Attacks

2.1 Definition & Threat Model

2.1.1 Definition

In a narrow sense, network attacks refer to attacks targeting the computer network communication, so that the integrity, confidentiality and authenticity of the network communication would be compromised. A well-known threat model for network attacks is "man-in-the-middle" model, in which the attacker sits in the path between two communication endpoints *(e.g., computers)* and is able to eavesdrop, alter and even block the communication. Such a malicious party can be an adversarial router or firewall along the communication path. However, even if the middle boxes along the path could be trusted, the communication can still be compromised by off-path attacks.

In the **off-path network attack**, the attacker is not assumed to sit in path and therefore cannot eavesdrop the traffic between two endpoints *(i.e., victims)*. Rather, they

just blindly inject packets to the communication by pretending they are one of the endpoint, in order to break the integrity and authenticity of the communication. Obviously, off-path attacks require the attacker to spoof the identify, which is enabled by the nature of IP networks — destination-based routing, meaning the sender's identity in most cases will not be checked and thus spoofing IP source address is possible.

Most network protocols like DNS and TCP carry certain kinds of identifier for multiplexing, which inadvertently increased the entropy of the protocol payload and increased difficulties for off-path attacks by introducing "secrets" to off-path attackers. For example, DNS response packets contain transaction ID to differentiate among requests sent to the same server and the off-path attacker must learn the randomly generated ID *(i.e., secret)* before the injected packet can be accepted, if they were to inject rogue DNS responses with wrong record. Blind injection carries the nature of brute forcing *(e.g., enumerate all possible IDs)* and therefore is less favorable given the large entropy *(e.g., at least 16-bit for transaction ID without considering other secrets like ephemeral port number)*. To solve this problem, off-path attacks are usually made possible by side channels.

Since the attackers cannot directly eavesdrop the traffic to obtain the secrets, to retrieve it, they must infer it from other form of available information, which leaks the secret through **side channel**. Side channel can be defined as a kind of implicit and unattended information leakage where the secret information is leaked in a form that differs from the original form. For example, the character typed on keyboard is designed to be transferred using electric signal, however, by analyzing the keystroke sound, the character typed can be leaked in the form of sound wave. Also the sound wave is the intrinsic

byproduct of the keystroke and therefore is unattended leak. Further, when the character was input to the search box with typing prediction, the character can also be leaked by analyzing network traffic shape or GPU memory usage [95], in the form of packet length or performance statistics numbers, deviating from the original form of pixels on the screen (showing the character). For off-path network attacks, side channels usually arise from the shared resources like counters [89, 22], and cache [90]. Note that side channels should leak information in an unattended manner, which serves the key difference between a side channel and a covert channel. For example, leveraging the memory bus as antenna, an unprivileged malicious agent on the victim system can extrapolate memory data by emitting electromagnetic waves [108]. However, this is considered as a covert channel, because the transformed information leakage is caused by an active and attended agent instead of the system itself.

2.1.2 Threat Model

General Off-Path Network Attack Threat Model. As exemplified by [89, 90, 22, 104, 84, 97, 27], off-path network attack can be generalized into an universal threat model. In such a model, as mentioned in Chapter 2.1.1, there is a secret that is defined according to the specific protocol. Two victim hosts communicate, using the secret that is unknown to an off-path attacker. By definition, an off-path attacker is unable to eavesdrop or tamper with the communication between the victim hosts. However, they can leverage IP spoofing (which is allowed in the majority of ASes according to a recent study [37, 86] and is offered in some bullet-proof-hosting service with around $50/month), and therefore can craft and send any packets using spoofed source IPs. The primary objective of such

attacks is to deduce secrets by sending a combination of spoofed and non-spoofed packets. The deduction can be achieved by observing in-path packets, timing information and other attacker-observables.

Off-Path DNS Cache Poisoning Threat Model. DNS cache poisoning attack aims to let the resolver cache wrong DNS records by answering the queries with forged malicious responses. If we fit the threat model into DNS cache poisoning attack, then the secret becomes the ephemeral port number and transaction ID used in DNS requests, because all other fields are known and deterministic to the off-path attacker like the domain name in the question field (as the query can be triggered by the attacker themselves), and the attack goal is to infer the ephemeral port number.[1] The victim hosts are defined as the resolver(forwarder) and nameserver(resolver) as knowing the secrets allows the attacker to inject rouge DNS responses by impersonating the nameserver(resolver). In addition, the attacker needs to control a machine that is able to trigger a request out of resolver(forwarder). In forwarder case, it can happen when an attacker can join a public wireless network in a coffee shop, a shopping mall, or an airport where the forwarder in the LAN serves them. In resolver case, it can happen when an attacker is an insider of an enterprise network or the resolver is open to the public *(e.g., Google DNS)*.

2.2 History of Off-Path Network Attacks

The first off-path TCP attack was created by Robert [93] in 1985 after TCP was invented in 1974. Robert attacks the predicable initial sequence number(ISN) generator

[1]Transaction ID may also be inferred but we leave it as future work. In this work we assume the attacker will brute-force it.

Year	Author	Finding	Target
1985	Robert T. Morris [93]	ISN is predictable in 4.2BSD.	TCP
1989	Steven M. Bellovin [15]	`netstat` service leaks SEQ. #	TCP
1995	Kevin Mitnick	Real-world exploit on predictable ISN	TCP
1999	Nalneesh Gaur [51]	First cache poisoning attack	DNS
2001	Michal Zalewski [123]	ISN is still predictable in most OSes.	TCP
2004	Paul Watson [120]	RST acceptance only requires SEQ# in window.	TCP
2008	Dan Kaminsky [71]	Real-world cache poisoning attack	DNS
2010	Zhiyun Qian [99]	IP spoofing bypasses SMTP blocking.	EMail
2010	Roya Ensafi [46]	IPID leaks packet emission event.	IP
2012	Yossi Gilad [3]	IPID leaks packet emission event, further secrets.[1]	TCP
2012	Zhiyun Qian [97]	Firewall behavior facilities off-path attacks.	TCP
2012	Zhiyun Qian [98]	OS shared states like stats & avail. ports leak secrets.	TCP
2012	Amir Herzberg [60]	Avail. ports on NAT gateway leaks port#.[2]	DNS
2013	Amir Herzberg [61]	Forging 2nd frag. to bypass port & TxID check.	DNS
2013	Amir Herzberg [62]	Time diff. from socket overloading leaks port#.	DNS
2016	Yue Cao [22]	Global challenge ACK counter leaks secrets.	TCP
2018	Markus Brandt [17]	Frag. needed ICMPs. forces NS to fragment responses.	DNS
2019	Domien Schepers [104]	Shared power save state in Linux kernel leaks TKIP key.	Wi-Fi
2019	Fatemah Alharbi [10]	Avail. ports on OS leaks port#.	DNS
2020	Xiaofeng Zheng [125]	Any response to forwarder can be fragmented w/o ICMP.	DNS
2020	Amit Klein [75]	IPID & port# predicts each other for shared generator.	DNS
2020	Keyu Man [89]	Global ICMP rate limit counter leaks port#.[3]	DNS
2021	Keyu Man [90]	Next hot exception cache leaks port#.[3]	DNS

[1] TCP secrets = SEQ# + ACK# + Client port#

[2] DNS port# = UDP ephemeral port# used in a query

[3] Detailed in this paper

Table 2.1: Off-Path Network Attacks

implemented in BSD, which is incremented by a constant amount once per second, and by half that amount each time a connection is initiated [15]. Later on, in 1989, Steven exploits the `netstat` service to acquire the TCP sequence number directly if one of the TCP end host is down [15]. In 1995, Kevin observed that ISN used in X Terminal increased predictably and launched real world attack. In 1999, Nalneesh invented the first off-path DNS cache poisoning attack [51]. Two years later, 26 years after Robert found the predictable ISN vulnerability, Michal surveyed more than 10 TCP implementations and found most of them

were still implementing the predictable ISN generator [123]. Instead of predicting the precise sequence number, Paul found in 2004 that as long as the sequence number used in TCP RST packet falls in the TCP receiving window, which is usually 64k bytes, the packet would be considered valid and the connection would be reset, if such packet was received. This reduces the sequence number entropy by 65536 times when resetting the connection off-path and practical attacks succeeded in minutes [120]. In 2008, the famous talk "It's the end of the cache as we know it" was given by Dan Kaminsky on Black OPS 2008, which presented a real-world DNS cache poisoning attacks. At that time, most DNS software used a fixed ephemeral port number to produce requests, which only left 16-bit entropy from transaction ID to the attacker and Dan simply brute forced 16-bit transaction ID to launch the attack. Two years later, in 2010, side channel attacks were applied to targets other than DNS and TCP: Zhiyun discovered IP spoofing could be used to bypass SMTP blocking and further enable the large scale email spam [99]; Roya proposed idle port scan which leveraged global IPID counter to infer the packet transmission event on an arbitrary host [46], which became the foundation of later attacks. In 2012, IPID side channel was used to launch off-path TCP attack [3]. Unlike previous attacks which rely on ISN inference, this attack managed to infer the TCP secrets after the connection was established. By observing the packet transmission event, the attacker can learn whether the previous guess on TCP SEQ#, ACK# or client port# was right. In the same year, Zhiyun found some shared states in OSes and firewall can leak TCP secrets as well [97, 98]. Similarly, at the same time, Amir found NAT can leak UDP ephemeral port#, due to the shared available port number pool [60]. In 2013, Amir proposed two novel DNS cache poisoning attacks: first he

discovered that overloading socket would produce the side channel for inferring ephemeral port number [62]; second he found the check on port# and transaction ID can be bypassed completely when the IP is fragmented [61], and therefore the attacker can inject rogue fragments to poison the cache. In 2016, Yue found global challenge ACK counter, which was implemented in mainstream OSes, can be used to leak TCP credentials [22, 23]. In 2018, as a supplementary work, Markus found nameservers can be tricked to fragment DNS responses if sent with ICMP fragment needed packets [17]. In 2019, similar to [98], Fatemah found similar shared available port pool on a host can be used to infer the ephemeral port number of DNS requests [10]. In the same year, Domien discovered the shared power save state in Linux kernel can be used to leak TKIP key of a encrypted Wi-Fi network [104]. In 2020, leveraging the unique position of DNS forwarders, Xiaofeng discovered the attacker can force fragmentation on the DNS responses sent by the upstream DNS resovler without the need for ICMP fragment needed packets [125]. Also in 2020, leveraging the cryptography properties, Amit was able to infer the next ephemeral port number selected by the OS after observing the IPID sequences of the previous sent packets.

Chapter 3

SADDNS: Revive DNS Cache Poisoning Attacks With Temporal ICMP Rate Limit Counter

3.1 Introduction

Domain name system (DNS) is an essential part of the Internet, originally designed to translate human-readable names to IP addresses. Nowadays, DNS has also been overloaded with many other security critical applications such as anti-spam defenses [70], routing security (e.g., RPKI) [20]. In addition, DNS also plays a crucial role in bootstrapping trust for TLS. TLS certificates are now commonly acquired by proving the ownership of a domain [7]. Therefore, compromising the integrity of DNS records can lead to catas-

trophic security failures, including fraudulent certificates being issued that can compromise the underpinning of public key cryptography [17].

Historically, the very first DNS cache poisoning attack was invented in 1999 [51]. The first well-known DNS cache poisoning attack was presented by Kaminsky [71] in 2008, who demonstrated that an off-path attacker can inject spoofed DNS responses and have them cached by DNS resolvers. This has led to a number of DNS defenses being deployed widely, including source port randomization [66] and "birthday protection" [59, 60]. Other defenses such as 0x20 encoding [36] and DNSSEC [11] have also gained some traction. Unfortunately, due to reasons such as incentives and compatibility, these two defenses are still far from being widely deployed as reported in recent studies [29, 67, 105, 85, 34, 33]. To summarize, source port randomization becomes the most important hurdle to overcome in launching a successful DNS cache poisoning attack. Indeed, in the past, there have been prior attacks that attempt to derandomize the source port of DNS requests [62, 60]. As of now, they are only considered nice conceptual attacks but not very practical. Specifically, [62] requires an attacker to bombard the source port and overload the socket receive buffer, which is not only slow and impractical (unlikely to succeed in time) but also can be achieved only in a local environment with stringent RTT requirement. In [60], it is assumed that a resolver sits behind a NAT which allows its external source port to be derandomized, but such a scenario is not applicable to resolvers that own public IPs.

In contrast, the vulnerabilities we find are both much more serious and generally applicable to a wide range of scenarios and conditions. Specifically, we are able to launch attacks against all layers of caches which are prevalent in modern DNS infrastruc-

ture [106, 14, 10], including application-layer DNS caches (e.g., in browsers) [10], OS-wide caches [10], DNS forwarder caches [64] (e.g., in home routers), and the most widely targeted DNS resolver caches. The vulnerabilities also affect virtually all popular DNS software stacks, including BIND [35], Unbound [80], and dnsmasq [72], running on top of Linux and potentially other OSes, with the major requirement being the victim OS allowed to generate outgoing ICMP error messages. Interestingly, these vulnerabilities result from either design flaws in UDP standards or subtle implementation details that lead to side channels based on a global rate limit of ICMP error messages, allowing derandomization of source port with great certainty.

To demonstrate the impact, we devise attack methods targeting two main scenarios, including DNS forwarders running on home routers, and DNS resolvers running BIND/Unbound. With permissions, we also tested the attack against a production DNS resolver that serves 70 million user queries per day, overcoming several practical challenges such as noises, having to wait for cache timeouts, multiple backend server IPs behind the resolver frontend, and multiple authoritative nameservers. In our stress test experiment, we also evaluate the attack in even more challenging network conditions and report positive results.

In this chapter, we make the following contributions:

1. We systematically analyze the interaction between application- and OS-level behaviors, leading to the discovery of general UDP source port derandomization strategies, the key one being a side channel vulnerability introduced by a global rate limit of outgoing ICMP error messages.

2. We research the applicability of the source port derandomization strategies against a variety of attack models. In addition, to allow sufficient time in conducting the derandomization attack, we find methods to extend the attack window significantly, one of them again leveraging the rate limiting feature (this time in the application layer).

3. We conduct extensive evaluation against a wide variety of server software, configuration, and network conditions and report positive results. We show that in most settings, an attacker needs only minutes to succeed in an end-to-end poisoning attack. We also discuss the most effective and simple mitigations.

3.2 Current State of DNS Cache Poisoning Attacks

The classic DNS cache poisoning attack in 2008 [71] targeted a DNS resolver by having an off-path attacker tricking a vulnerable DNS resolver to issue a query to an upstream authoritative nameserver. Then the attacker attempts to inject rogue responses with the spoofed IP of the nameserver. If the rogue response arrives before any legitimate ones, and if it matches the "secrets" in the query, then the resolver will accept and cache the rogue results. Specifically, the attacker needs to guess the correct source/destination IP, source/destination port, and the transaction ID (TxID) of the query. The transaction ID is 16-bit long. At the time when both the source and destination port (i.e., 53) were fixed, 16-bit is the only randomness. Thus an off-path attacker can simply brute force all possible values with 65,536 rogue responses, not to mention a few optimizations such as birthday attacks that can speed the attack even further.

3.2.1 State-of-the-Art(SOTA) Defenses

A number of defenses have since then been promoted to mitigate the threat of DNS cache poisoning. They effectively render the original attack no longer feasible. We describe below the most widely known and deployed defenses:

- Randomization of source port [66] is perhaps the most effective and widely deployed defense as it increases the randomness from 16 bits to 32 bits. As an off-path attacker has to guess both the source port and TxID at the same time.
- "Birthday protection" [59, 60] by removing duplicate queries. Kaminsky's attack optimized the attack success rate by triggering multiple outgoing requests (all using the same source and destination port), and therefore greatly improves the likelihood that an off-path attacker guesses one correct TxID out of all the outstanding ones. The defense simply disallows more than one outstanding query of the same domain name and it is also widely deployed.
- Randomization of capitalization of letters in domain names, i.e., 0x20 encoding [36]. The offered randomness depends on the number of letters and can be quite effective also, especially for long names. Unfortunately, even though it is a simple change to the protocol, in practice it has significant compatibility issues with authoritative nameservers encountered on the Internet [34, 39]. Therefore, most popular public resolvers now refrain from using 0x20 encoding by default. For example, Google DNS uses it only for a set of whitelisted nameservers [39]; Cloudflare has even recently disabled 0x20 encoding altogether [33]. As of the year 2020, we found only two (i.e.,

openNIC and Verisign) out of the 16 popular public DNS services we measured (see the other 14 in Table 3.2) use it by default to a test nameserver we setup. And the result roughly matches what was observed in a study [105].

- Randomization of the choice of nameservers (server IP addresses) [61]. The offered randomness depending on the number of nameservers. In practice, most domains employ less than 10 nameservers, translating to only 2 to 3 bits. In addition, it has been shown that an attacker can induce query failures against certain nameservers and therefore effectively "pinning" a resolver to the one remaining nameserver [60].

- DNSSEC [11]. The success of DNSSEC depends on the support of both resolvers and authoritative nameservers. However, only a small fraction of domains is signed — 0.7% for .com domains, 1% for .org domains, and 1.85% for Top Alexa 10K domains, as reported in 2017 [29]. In the same study, it is also reported that only 12% of the resolvers enabling DNSSEC actually attempt to validate the received records. As a result, the overall deployment rate of DNSSEC is far from satisfactory.

3.2.2 New Attack Surface in the DNS Hierarchy

As alluded to earlier, modern DNS infrastructure has multiple layers of caching. Figure 3.1 provides a concise view: a client application often initiates a DNS query (through an API call such `gethostbyname()`) to an OS stub resolver — typically a separate system process that maintains an OS-wide DNS cache. The stub resolver does not perform any iterative queries; instead, it always forwards the request to the next layer up, a DNS forwarder which also forwards queries to its upstream recursive resolver. DNS forwarders are com-

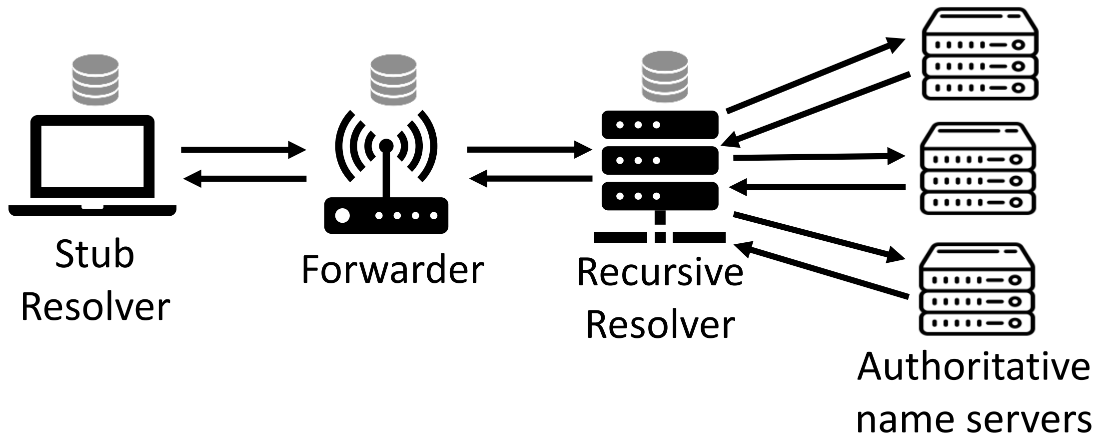


Figure 3.1: DNS Infrastructure With Multiple Layers of Caching

monly found in Wi-Fi routers (e.g., in a home) and they maintain a dedicated DNS cache also. It is the recursive resolver that does the real job to iteratively query the authoritative nameservers. The answers are then returned and cached in each layer.

All layers of caches are technically subject to the DNS cache poisoning attack. Unfortunately, most newly proposed attacks were focused on resolvers [62, 60, 59, 17], and very limited investigations have been done on stub resolvers [10] and forwarders [107].

3.3 Attack Overview

We propose a general and novel attack, under the threat model presented in Chapter 2.1.2, applicable to all modern DNS software stack, influencing all layers of DNS caching. The key characteristic is that it defeats the most effective and commonly deployed defense — randomization of source port.

3.3.1 Attack Workflow

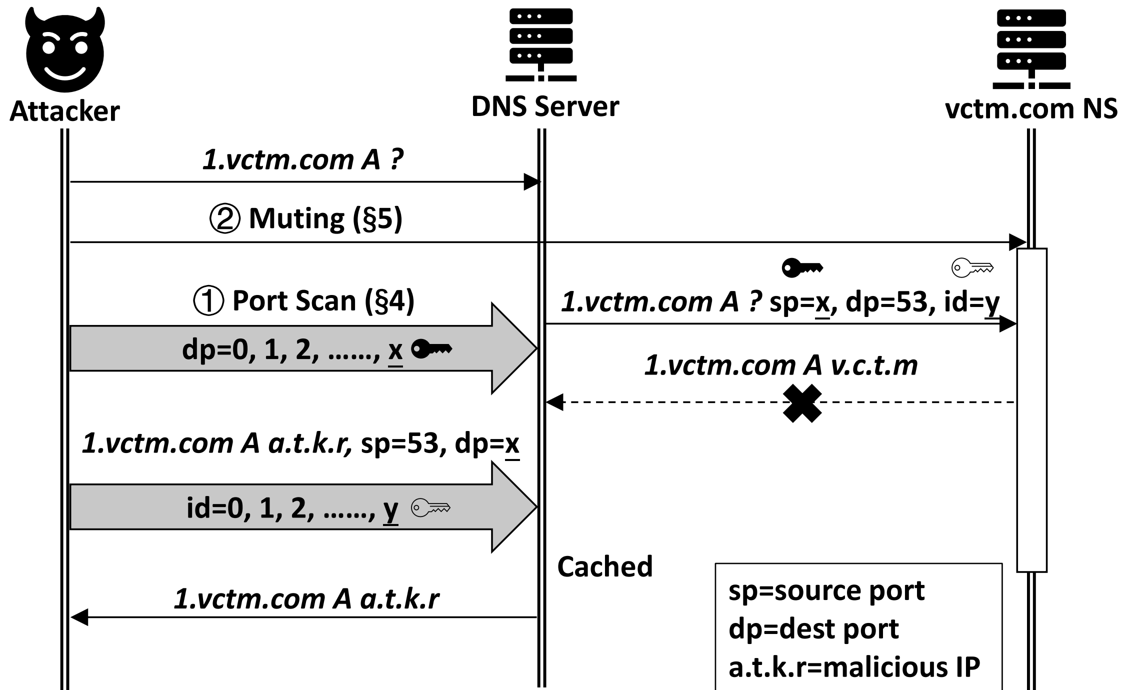


Figure 3.2: SADDNS Attack Workflow

Regardless of a forwarder or resolver, as illustrated in Figure 3.2, our newly proposed attacks always start from triggering either one to send a DNS query, followed by two key steps as outlined below:

1. *Inferring source port.* To overcome the randomization of source port, we leverage a novel and universal side channel in networking stacks to scan and discover which source ports were used to initiate a DNS query, at a speed of at most 1,000 guesses per second.

2. *Extending attack window.* Normally an outstanding query will receive a reply from the upstream server in a matter of tens or hundreds of milliseconds. This is insufficient,

given that the attacker needs time to infer the source port and to inject rogue DNS replies. We discover effective and novel strategies (different for forwarder and resolver attack) that can greatly extend the attack window to at least seconds (and even more than 10s), allowing realistic cache poisoning opportunities. We will discuss this in Chapter 3.5.

Once the source port number is known, the attacker simply injects a large number of spoofed DNS replies bruteforcing the TxIDs, which can be done in high speed, given that most servers have sufficient network bandwidth.

3.4 Inferring DNS Query's Source Port

In this section, we will describe the idea and procedure of inferring DNS source ports. We will also measure the vulnerable software and in-the-wild population when feasible.

3.4.1 Analysis of UDP Source Port Scannability

UDP is a stateless protocol and hence fundamentally different from TCP. More specifically, it is stated in the UDP programming guideline (RFC 8085 [79]) that "UDP datagrams may be directly sent and received, without any connection setup. Using the sockets API, applications can receive packets from more than one IP source address on a single UDP socket." Furthermore, to ensure that an application will receive data from only one particular source address, "these applications MUST implement corresponding checks

at the application layer or explicitly request that the operating system filter the received packets."

These are surprisingly under-scrutinized statements. On a first glance, they may be interpreted as applicable to UDP servers only, which can bind to a local port, and subsequently receive packets from "any remote IPs". Surprisingly, from our experiments, it applies to UDP clients as well — a client calling `sendto()` on a specific remote IP and subsequently `recvfrom()` on the same socket can technically receive packets from "any other IPs" as well. We have verified this behavior on all modern operating systems, including Windows, Linux, and MacOS.

This nuanced behavior has a profound impact on what an attacker can learn through a trivial UDP port scan — when a DNS server issues a query, its source port effectively becomes open to the public. This allows an attacker to simply scan the ephemeral port range with any UDP packet, which will trigger nothing upon hitting the correct port (as the probe will be accepted by the OS but discarded at the application layer), or an ICMP port unreachable message upon missing it (by design).

Next, the UDP programming guideline (RFC 8085) further states that "Many operating systems also allow a UDP socket to be connected, i.e., to bind a UDP socket to a specific pair of addresses and ports." Indeed, modern socket APIs allow `connect()` on a UDP socket but "this is only a local operation that serves to simplify the local send/receive functions and to filter the traffic". As a result, when a DNS query is issued from a source port to a particular destination IP address and port, the OS will accept incoming packets from only the same remote IP and port. Specifically, when testing the behavior on real

network stacks, we find that they will reject a packet with either a wrong IP or port, and respond with an ICMP port unreachable message (as if the packet was a port scan attempt). This effectively prevents the source port of a DNS query from being scanned directly.

In summary, the scannability of source port is dependent on the implementation of DNS software, i.e., whether a `connect()` API call is issued on the UDP socket. Interestingly, we find that out of the three most popular DNS forwarder and resolver software BIND, Unbound, and dnsmasq, only BIND uses `connect()`. Nevertheless, we develop different scan methods that can work for each (described in Chapter 3.4.3 and Chapter 3.4.4, overcoming the challenge outlined in the next section).

3.4.2 ICMP Rate Limit Challenge

A major hurdle to scan UDP source ports efficiently is the commonly deployed rate limit of outgoing ICMP error messages on endhosts. Even in the simple case where a source port is public-facing and can be scanned directly by any IP address, an attacker's scanning speed is limited by the number of allowable ICMP packets per second (a signal indicating a source port is not in use).

Historically, ICMP rate limit was first recommended to limit the resource consumption on a router (described in RFC 1812 [13]) where an attacker can force it to generate a high volume of ICMP error messages. Today, the rate limit mechanism is universally implemented by all major OSes. Here we focus on the Linux's ICMP rate limiting behavior as it is the most popular server OS, but will briefly describe the behaviors of other OSes afterwards.

For Linux, there are both a per-IP and global rate limit on how many ICMP error packets can be sent out per second. The per-IP rate limit was historically introduced in the very early versions of Linux, i.e., present in kernel 2.4.10. The global rate limit was introduced in kernel 3.18 as a way to alleviate the expensive per-IP rate limit check (e.g., red-black tree operations) [42].

By default, the per-IP rate limit is one per second (with an accrued max burst of 6) which will severely restrict the scanning speed; the global rate limit is effectively 1,000 (with periodic max allowable bursts of 50). Both are implemented in token bucket style, with the per-IP tokens recovering at a rate one per second and the global token recovering at a "nominal" rate of one per millisecond (but the actual token increment happens only after at least 20ms has elapsed since the last increment). The number of available tokens is capped at 50 at all times.

We also tested Windows Server 2019 (version 1809), macOS 10.15 and FreeBSD 12.1.0, all of which have global ICMP rate limits. Specifically, their limits are 200, 250 and 200 respectively. Besides, none of them has a per-IP rate limit.

3.4.3 Public-Facing Source Port Scan Method

Even though a source port can be directly probed by any attacker IP in this case, e.g., as in unbound and dnsmasq, it is imperative to bypass the per IP rate limit (present in Linux primarily) to achieve faster scan speed. We develop three different probing methods that can overcome the ICMP rate limit challenge:

i. If the attacker owns multiple IP addresses, either multiple bot machines or a single machine with an IPv6 address, then it is trivial to bypass the per IP limit. IPv6 address allocation states that each LAN is given a /64 prefix [63], effectively allowing any network to use 2^{64} public IP addresses. We have tested this from a machine in a residential network that supports IPv6 and picked several IPs within the /64 to send and receive traffic successfully.

ii. If an attacker owns only a single IPv4 address, it is still possible to ask for multiple addresses using DHCP. We verified that multiple private IPv4 addresses can be obtained in a home network. In addition, we have tested this in an educational network where a single physical machine is able to acquire multiple public IPv4 addresses through this method as well.

iii. If an attacker owns a single IPv4 address and the above method fails for some reason (e.g., statically assigned IPs), then the last method is to leverage IP spoofing to bypass the per IP rate limit, and the global rate limit as a side channel to infer whether the spoofed probes have hit the correct source port or not, i.e., with or without ICMP responses. As have been shown in the context of TCP recently, global rate limit can introduce serious side channels [22, 24, 46]. Here we leverage the ICMP global rate limit to facilitate UDP port scans which we describe next.

Figure 3.3 illustrates this. In observing the maximum globally allowable burst of 50 ICMP packets in Linux, the attacker first sends 50 spoofed UDP probe packets each with a different source IP (bypassing the per-IP rate limit). If the victim server does not have any

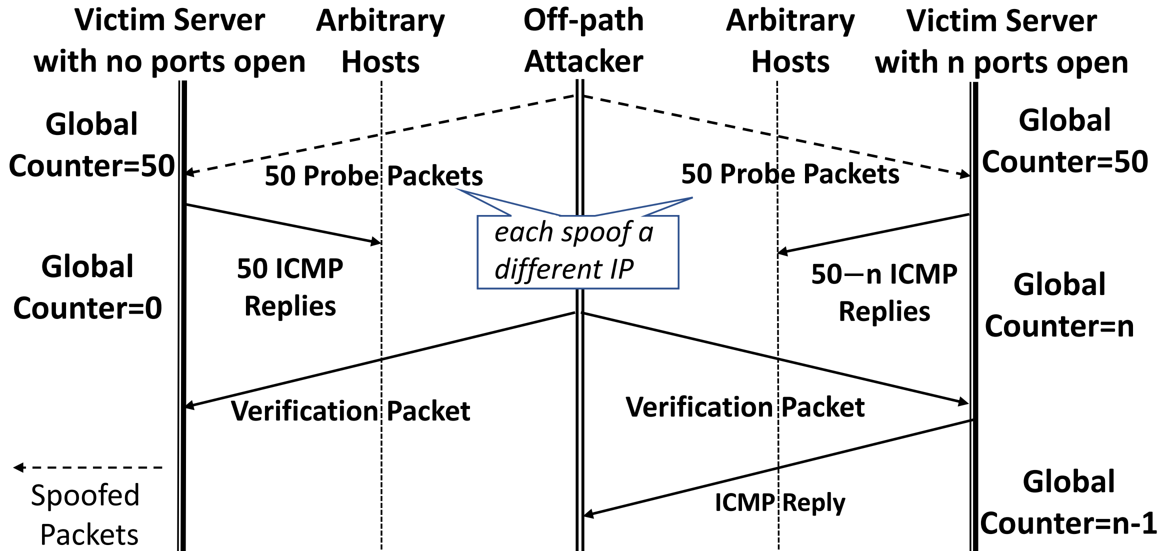


Figure 3.3: Fast Port Scanning of an Open Source Port

source port open among the 50, then 50 ICMP port unreachable messages will be triggered (but they are not directly observable to the attacker). If the victim server does have n open ports, then only $50 - n$ ICMP packets will be triggered (as the n UDP probing packets will be silently discarded at the application layer). Now, the attacker sends a verification packet using its real IP address, e.g., a UDP packet destined to a known closed port, such as 1. It will either get no response (if the global rate limit is drained), or an ICMP reply otherwise.

If no port is found in the first batch, the attacker waits for at least 50ms for the rate limit counter to recuperate, and then start the next round. Effectively, the scanning speed will be capped at 1,000 per second. It therefore takes mote than one minute to enumerate the entire port range consisting of 65536 ports. Nevertheless, it is a winning battle as the attacker can simply repeat the experiment and the probability that one experiment will succeed increases drastically (we note that this is a simple Bernoulli trial).

Time Consideration. This approach does have a strong timing requirement. The only thing the attacker has to make sure is to send 50 spoofed probing packets and the verification packet in a burst so that they are all processed within a 20ms window; otherwise, the victim may start recovering additional tokens. The other requirement is that the attacker has to wait long enough for the 50 max tokens to recover. If the network condition is not ideal, the attacker can simply wait longer than 50ms.

Binary Search to Narrowing Down to an Exact Port. Assuming there is a single open port out of the 50 in a specific probing round, we can then employ a simple binary search to quickly narrow down to the exact port. During each round of binary search, we always probe the left half of range first. If it is a match, i.e., 50 spoofed probing packets triggered 49 replies and the attacker can observe one reply to its verification packet, then we continue to search its left half. Otherwise, we assume the port lies in the right half and will conduct a binary search there. Note that we will need to send "padding packets" to ensure the global rate limit is drained when none of the 50 guesses hit a correct port. Padding packets are spoofed packets destined to known closed UDP ports, e.g., 1, that are guaranteed to trigger ICMP replies.

Handling Noises. DNS servers usually serve multiple clients at the same time, creating multiple outstanding DNS queries and source ports. As a result, the source port scan will likely discover many irrelevant ports. However, most such queries are transient, and the port scan process can quickly discover an open source port disappearing during the binary search and return to the linear search. In contrast, we assume that the attacker-

triggered DNS queries will last significantly longer, e.g., on the order of seconds instead of milliseconds (see Chapter 3.5).

Another source of noises comes from packet losses and reordering. This may lead to both FPs, e.g., loss probing packets or their replies, reordering between verification and probing packets, and false negatives(FNs), e.g., lost of the verification packet or its reply (although very rare in practice). To mitigate reordering (which may happen frequently if the jitter is large), we insert a delay, which is empirically determined to be larger than twice the jitter, between probe packets and the verification packet. When FPs do occur, they are handled automatically in the binary search process–it will detect no real port being open and return to linear search.

Even though they can be handled, excessive FPs will drain the per-IP rate limit quickly. Specifically, given the token is recovered at the slow rate of one per second, a FP rate that is higher than that will force the scan to halt until the token is recovered. Effectively, a per-IP token is a “pass to scan”. To solve this problem, the attacker may use two or more real IPs to gain more “passes”.

In addition, DNS servers themselves may be subject to random UDP port probing and therefore generate ICMP unreachable messages. This would cause FNs: we may mistakenly think there is no open port but in fact there is because the verification packet will not trigger any ICMP unreachable replies due to the noise draining the rate limit. Fortunately, not all ICMP replies are subject to rate limit. For example, the most commonly triggered ICMP echo replies are not subject to the limit.

3.4.4 Private Source Port Scan Method

As described in Chapter 3.4.1, if `connect()` is performed on a UDP socket, the port effectively becomes "private" to the remote peer, invalidating the previous method.

Our idea then is to send spoofed UDP packets with the source IP of the upstream DNS server. In the example of a DNS resolver being the victim, we can send UDP packets probing different source ports with spoofed IP of the authoritative nameserver. If it hits the correct source port, then no ICMP reply will be generated. Otherwise, there will be. We can then use the same global ICMP rate limit as a side channel to infer if such an ICMP message has been triggered. At first glance, this method can work but at a low speed of *one port per second*, due to the per-IP rate limit on ICMP messages.

Surprisingly, after we analyze the source code of the ICMP rate limit implementation, we find that the global rate limit is checked prior to the per-IP rate limit. This means that even if the per-IP rate limit may eventually determine that no ICMP reply should be sent, *a packet is still subjected to the global rate limit check and one token is deducted.* Ironically, such a decision is consciously made by Linux developers to avoid invoking the expensive check of the per-IP rate limit [42], involving a search process to locate the per-IP data structure.

This effectively means that the per-IP rate limit can be disregarded for the purpose of our side channel based scan, as it only determines if the final ICMP reply is generated but has nothing to do with the global rate limit counter decrement. As a result, we can continue to use roughly the same scan method as efficient as before, achieving 1,000 ports per second. Figure 3.4 illustrates the slightly modified scan workflow. Similar to Figure 3.3,

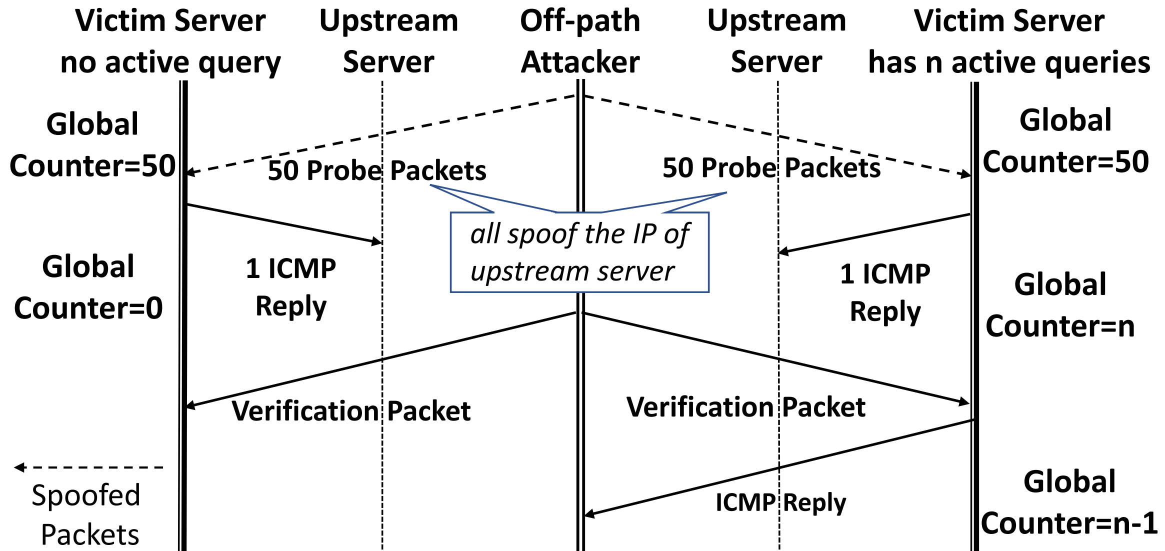


Figure 3.4: Fast Port Scanning of a Private Source Port

the attacker first sends 50 probes where this time all of which uses the spoofed IP of the upstream server. Due to per-IP rate limit, the victim server will always generate only one ICMP reply (in steady state) as long as there is at least one inactive port scanned, which is the case in both the left and right side of the figure. In the case where the 50 probes hit n private open ports (to the upstream server), the global rate limit counter still decrements to n because the victim attempted to generate $50 - n$ ICMP replies. In contrast, when all 50 probes hit inactive ports (left side of the figure), the counter decrements to 0.

The rest of the procedure is identical as before, where a binary search can be launched to narrow down to a specific port.

Influence on Public-Facing Source Port Scan. With this knowledge, we can improve method iii. in Chapter 3.4.3 as follows: instead of spoofing 50 different IPs in each round of probing, we only need to use a single spoofed IP (or a 2nd IP the attacker owns) instead of many different IPs (which sometimes can be a hurdle).

Handling Noises. We point out that there is inherently less noise in this scan compared to the one on public-facing source ports. This is because every source port is now effectively "open" to only one single remote IP which is originally specified in `connect()`. Therefore, assuming the victim is a resolver, most of its queries (i.e., noise) will be destined to a different nameserver than a specific attack target. Other noise conditions such as packet loss and reordering still apply. Similarly, noise handling techniques also apply (e.g., using more than one IP to alleviate the per-IP ICMP rate limit).

3.4.5 Vulnerable DNS Forwarder and Resolver Population

A forwarder or resolver is considered vulnerable if the UDP source port of a DNS query can be inferred successfully, or more specifically if it supports the global ICMP rate limit, and/or if it does not use `connect()` (which makes the port public).

Router	ICMP Rpl.	Global Rate Lim.	`connect()`	Pub. IP Spoof in LAN	Vuln.
Verizon Fios Gateway (G1100)	Y	N	Y	N/A	N
Xiaomi (R3)	Y	N	N	Y	Y1
Huawei A1 (WS826)	N	N/A	N/A	N/A	N
Netgear (WNDR3700v4)	Y	N	N	N	Y2
Arris Spectrum Gateway (TR4400)	Y	N	N	Y	Y1
TP-Link (Archer C59)	Y	N	N	Y	Y1

Y1: vulnerable to an insider attack. Y2: vulnerable to an attack requiring collaboration between an insider and outsider.

Table 3.1: DNS Forwarder Behaviors in Home Routers

Vulnerable Forwarders. We surveyed six home router devices, all of which act by default as a forwarder supporting DNS caching. Their behaviors are summarized in Table 3.1.

Only one router (Huawei A1) fails to respond with even the ICMP port unreachable message, which is a basic requirement of the port scan. The Verizon Gateway is not

vulnerable because it is the only one using `connect()` yet without the global rate limit. We find that all routers are running old Linux kernel versions in the range of 2.6 to 3.10, which is why global rate limit is not observed. We do believe that routers of newer generations will eventually inherit the global rate limit. Nevertheless, since most of them do not use `connect()` on the UDP socket, the source port of a DNS query can be easily probed without leveraging the side channel based on the global ICMP rate limit. In addition, we also measured the IP spoofing capability within the LAN network. Specifically, if an attacker can spoof the public IP of the resolver from within the LAN network, which often operates on a private IP range, the end-to-end attack can be conducted from a machine in the LAN alone without any external collaborator. The result shows that three routers fall under this category (Y1), and one can be attacked from an outside machine capable of spoofing the resolver's IP (Y2).

Name	Address	Example Backend Addr.	# of Backends	ICMP	Global Rate Lim.	`connect()`	Vuln.
Google	8.8.8.8	172.253.2.4	15	Y	Y	N	Y
Cloudflare	1.1.1.1	172.68.135.169	2	Y	Y	Y	Y
OpenDNS	208.67.222.222	208.67.219.11	107	Y	Y	Y	Y
Comodo	8.26.56.26	66.230.162.182	2	Y	Y	N	Y
Dyn	216.146.35.35	45.76.11.166	1	Y	Y	N	Y
Quad9	9.9.9.9	74.63.16.243	11	Y	Y	Y	Y
AdGuard	176.103.130.130	66.42.108.108	3	Y	Y	N	Y
CleanBrowsing	185.228.168.168	45.76.171.37	1	Y	Y	Y	Y
Neustar	156.154.70.1	2610:a1:300c:128::143	2	Y	Y	N	Y
Yandex	77.88.8.1	77.88.56.132	19	Y	Y	Y	Y
Baidu DNS	180.76.76.76	106.38.179.6	16	Y	Y	Y	Y
114 DNS	114.114.114.114	106.38.179.6	11	Y	N	N	Y
Tencent DNS	119.29.29.29	183.194.223.102	45	Y	N	N	N[1]
Ali DNS	223.5.5.5	210.69.48.38	160	N	N/A	N/A	N

[1] Though meeting the requirements, it is not vulnerable due to interference of fast UDP probing encountered (likely caused by firewalls).

Table 3.2: `SADDNS` Vulnerable Status of Public Resolvers

Vulnerable Resolvers. We study a list of 14 popular DNS providers shown in Table 3.2 and show that 12 of them are vulnerable which is very serious. Interestingly, we find that due to firewall policies encountered in several providers, the source port of the probing packet must be set to 53 and the destination port should be in the ephemeral port range in order to trigger ICMP responses on some servers.

Note that we also report the number of backend server IPs behind the anycasted frontend IP (e.g., 8.8.8.8). These backend IPs correspond to the reachable servers on which we can scan ports. The presence of multiple such IPs increases the attack's difficulty as we need to decide which IP(s) to scan. To discover the backend IPs, we simply send 100 queries from the same machine to the frontend and record the observed IPs at an authoritative nameserver that we own. For the cases where we encounter only a few IPs, we can simply scan all of them simultaneously. For the cases of OpenDNS and AliDNS which have over 100, we discuss possible techniques to handle them later in Chapter 3.6. Note that OpenDNS and AliDNS exhibit more than 100 IPs because our authoritative nameserver intentionally discards incoming queries and they decide to retry with potentially new IPs every time before giving up.

In addition, we also measured the general population of open resolvers. Compared with public resolvers, which are usually advertised and intended to serve the public, open resolvers, however, are generally unlisted and are intended to serve smaller numbet of clients. We obtain a list of open resolvers from Censys [44] and managed to probe a set of 138,924 live IPs, among which there are 70,503 whose backend and frontend IPs are identical, indicative of the absence of anycast. Further, 41.3% of the 138,924 cases generate ICMP

replies (following the same practice of using source port 53 in the probing packets), out of which 67.56% exhibit a global rate limit, and 53.93% use `connect()` on the socket. Overall, 34.36% of all cases are vulnerable because they either support the global rate limit or do not use `connect()`). Most of them are not vulnerable simply because of the lack of ICMP replies.

3.5 Extending the Attack Window

The longer the attack window, the more ports an attacker can scan, and also more time to inject rogue records. Therefore, our goal is to "mute" upstream servers and prevent them from being able to respond to the DNS queries triggered by the attacker. Depending on the attack target (i.e., a forwarder or resolver), we come up with two novel strategies. Ironically, one of the strategies again leverage the "rate limiting" feature commonly deployed at the application layer, which can be turned to the attacker's advantage.

3.5.1 Extending Window in a Forwarder Attack

We propose a novel strategy as follows: the attacker first sends a query of his own domain, e.g., *www.attacker.com* to the forwarder, which will eventually trigger the upstream resolver to query the attacker-controlled authoritative nameserver. The nameserver is intentionally configured to be unresponsive so that the forwarder would wait maximum amount of time possible (as the resolver is also halted) while leaving an open source port. At a first glance, this is pointless because we are not interested in poisoning an attacker's

own domain. However, due to the unique role of DNS forwarders [64], they rely completely on upstream resolvers to perform validations on responses.

More specifically, according to RFC 8499 [64], recursive resolvers' responsibility is to handle the complete resolution of a name and provide a "final answer" to its client. This includes recursively handling referrals and CNAMEs and assemble a final answer, including any CNAME redirects by design. More importantly, resolvers are required to perform integrity checks such as the bailiwick check [45], whereas forwarders are not. This means that forwarders by design trust the upstream resolvers and its response. This is not a security flaw; rather, it is a design choice to prevent forwarders from duplicating the work of resolvers. This observation is also made a in recent study dedicated to the security of DNS forwarders [125].

Answer	`www.attacker.com`	CNAME	`www.victim.com`
	`www.victim.com`	A	1.2.3.4

Figure 3.5: Example Rogue Response Acceptable by a Forwarder

As a result, a rogue response (potentially injected by an attacker from either LAN or outside) shown in Figure 3.5 will be accepted by a forwarder and both the attacker's and victim's domain records will be cached. This strategy is extremely effective because we can impose the maximum wait time on the forwarder (i.e., creating the largest possible attack window). Specifically, most forwarders have a very lenient timeout (sometimes close to a minute e.g., in dnsmasq), and will stop mostly because the upstream resolver failing first (ranging from 5 to 30 seconds) generating a `SERVFAIL` response (or `NXDOMAIN`) message. To prevent resolvers from generating such messages too early, we also employ a technique

that can sometimes keep a resolver engaged longer. The trick is to have the attacker-owned authoritative nameserver respond in a slow pace with a chain of CNAME records, creating an illusion that it is making progress. This can delay resolver's response for over a minute in some cases (e.g., Cloudflare).

3.5.2 Extending Window in a Resolver Attack

We propose to take advantage of the security feature of rate limiting in authoritative nameservers, as a way to mute nameservers and extending window in a resolver attack. Modern DNS nameserver software such as BIND, NSD, PowerDNS, all support a common security feature called response rate limiting (RRL) [116, 114], as a mitigation of the DNS amplification attack [114] where a large number of malicious DNS queries are issued to authoritative nameservers spoofing a victim's IP address. To limit the number of amplified DNS reply packets, the RRL feature allows a configurable per-IP, per-prefix, or even global limit of triggered responses. Specifically, if the limit is reached, then responses are either getting truncated or dropped. There are also dedicated DNS firewalls with similar features [31].

Ironically, this feature can be leveraged maliciously to mute a nameserver if an attacker can inject spoofed DNS queries (with the target resolver's IP) at a rate higher than the configured limit. Depending on the actual limit (some are configured to be very low), it may be trivial to create a sufficiently high "loss rate" so that the resolver's legitimate query has an extremely low probability of getting a response. To understand how likely such a strategy can succeed, we conduct an experiment to measure the response rate of nameservers used by top 10K Alexa websites.

Measurement Methodology. To trigger RRL, we send 1k queries per second(qps) for 15 seconds, followed by another around of 4kqps test of 15 seconds to each nameserver IP; the two tests are separated by a two-second gap to avoid interference. If there are multiple nameservers for a given domain, we pick the first one. In both cases, the queries are uniformly distributed (instead of sent in bursts) all attempting to ask the A record of the `www` subdomain. The rational is that 1kqps and 4kqps represent sufficiently low throughput, roughly 0.6Mbps and 2.5Mbps respectively, which is easily achievable by any attacker on the Internet.

Ethical Considerations. We consciously took a number of measures to limit the impact on the operations of these servers. First, we ask for `A` records in our queries, which generally result in smaller responses, to conserve the target network's resources; yet, a prior report [115] suggests that rate limiting behaviors are generally agnostic to the type of queries (so this would not impact the result of our measurement). Second, the domain names in the queries are always the same, resulting in minimal processing overhead on the server (the result is likely cached in memory and easy to fetch). Third, we choose to send evenly spaced queries (instead in burst) to avoid stressing the server. In general, the traffic of 4kqps is small compared to a normal load experienced by a nameserver of a Top Alexa site. Finally, we set up a web server on the IP address used to conduct the probing, serving a webpage with opt out instructions (we also configured the reverse DNS name of the IP to direct visitors to our webpage). In total, we received and honored four requests.

Results. We sort the domains by the loss rate observed in the 4kqps test in descending order and present the results in Figure 3.6. Overall, there are about 25% domains

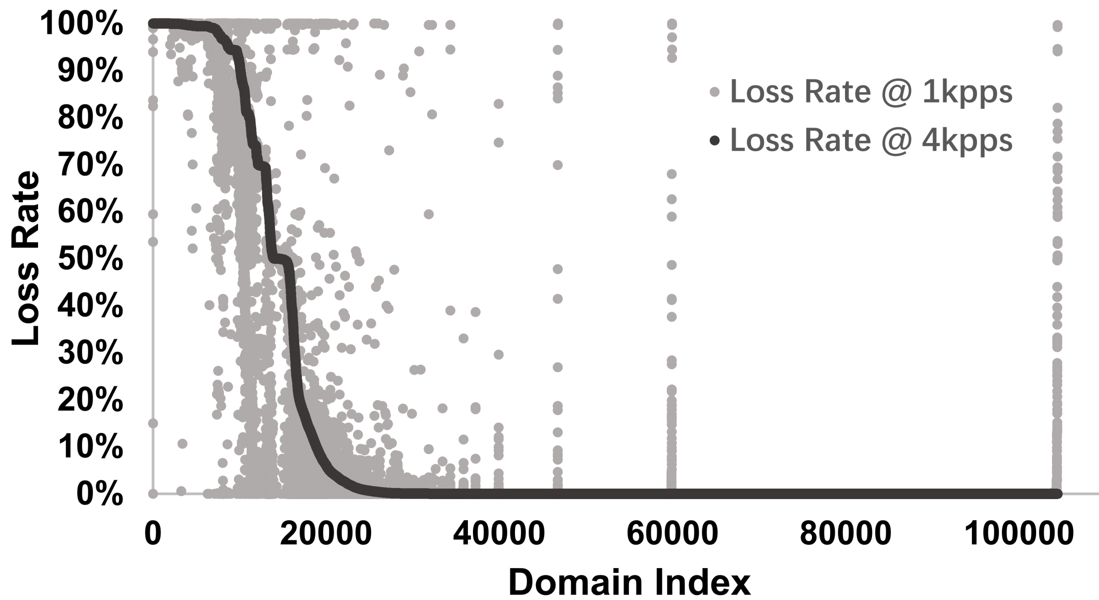


Figure 3.6: Response Loss Rate Under Different Query Rate

whose nameservers experienced higher than 1% loss rate. This is in line with a recent measurement reporting about 17% cases with loss behaviors [38]. The difference is likely due to their lower rate of queries at 500qps.

We now try to analyze what fraction of these domains are vulnerable (can be muted successfully). Here we define a domain to be vulnerable if its nameserver exhibits an induced loss rate of 66.7% or higher; the threshold is determined empirically as will be discussed in Chapter 3.7.2. Specifically, there are 13,110 domains that would already satisfy the criteria and fall victim to a simple DoS attack at a rate of 4kqps.

In addition, we also inspect the remaining cases where the loss rate increased from the 1kqps test to the 4kqps one. There are roughly 5,000 cases where the diff is 2% or higher. We believe that the majority of them can be further increased given increased probe rate,

and therefore potentially vulnerable as well. Therefore, we have a total of 18,110 (13,110 and 5,000) cases out of the 100k (18%) which we consider vulnerable.

Finally, out of the 75% cases where both 1kqps and 4kqps tests experienced no loss, we believe there may be many more vulnerable cases which we simply cannot uncover due to the relatively low probing speed. Due to ethical concerns, however, we refrain from probing at an even higher speed. To peek into those cases, we manage to obtain permission from a collaborator to test an authoritative nameserver configured for non-profit website. We are able to probe the server at a much higher rate (late at night to avoid disruption). Initially when probed at a rate 4kqps, no loss is observed. Interestingly, it started to experience loss when the probing rate is increased to 25kqps. Specifically, when the rate is increased to 50kqps, the loss rate jumps to 75%. We checked with our collaborator on whether the server is indeed configured to use such a high rate limit. To our surprise, there is no rate limit configured at all. To understand this behavior, we replicate a BIND server locally (replicating the configuration) and verified that indeed it is fairly easy to trigger high loss rate with comparable probing speeds. We find that it is because the application (i.e., BIND) not reading from the socket queue fast enough, which causes overflows. Indeed, historical DoS attacks similar to this, e.g., by flooding queries with random names, have been observed in practice [83]. To mitigate such threats, the official BIND explicitly guideline recommends rate limit [102], which would paradoxically make it vulnerable to our attack instead.

In addition, we can leverage this technique to extend the attack window against a forwarder since RRL is also deployed on resolvers to limit the rate of incoming queries. By following the same procedure and ethical standard in the previous measurements and a rate

of 4kqps probing against the resolver IPs obtained on 14, 2019 from Censys [44], we observe surprisingly 121,195 out of 136,547 exhibit a loss rate of more than 66.7%, indicating it is generally possible to mute resolvers on the Internet.

3.6 Practical Attack Considerations

3.6.1 Bypassing the TTL of Cached Records

If an attacker attempts to poison a benign domain such as `www.victim.com` by directly triggering DNS queries of `www.victim.com` on a resolver, it may cache the unwanted legitimate `A` record, for example, due to occasional failures to mute their upstream servers. This forces the attacker to wait for the cache timeouts before initiating the next attack attempt.

Field	Value
Question	`{nonce}.www.victim.com`
Answer	
Authoritative	`www.victim.com NS ns.attacker.com`
Additional	

Figure 3.7: DNS Response Used to Overwrite Cache

However, according to a recent study [76], the cached A record of `www.victim.com` could be overwritten by injecting a non-existent NS record of `www.victim.com`. Specifically, an attacker always sends queries asking for A records of domain names with random prefixes, e.g., `{nonce}.www.victim.com` where {nonce} is a random value. This forces the resolver to initiate a new query to the authoritative nameserver of `victim.com` as the

record is not cached. Then the attacker attempts to inject a rogue response as shown in Figure 3.7, claiming that `www.victim.com` is a standalone zone with its own authoritative server `ns.attacker.com`. The resolver will then query `ns.attacker.com` for all future requests asking for A record of `www.victim.com`, after the original cached record expires. This is because the attack has effectively inserted a new NS record of the `www.victim.com` zone. And resolvers are by design advised to use the most accurate delegation it has in the cache, which in this case is the NS record of `www.victim.com` instead of the one of `victim.com` [76]. We have verified that this method works against the latest versions of both BIND and Unbound.

3.6.2 Timeouts and Retransmitted Queries

When a DNS query is triggered either on a forwarder or resolver and there is no legitimate reply received from their upstream, they will not wait forever. Most of them have a timeout determining when to *close the current socket (and therefore the corresponding source port)* and retransmit. This means that some of these source ports may be short-lived and difficult to catch. Therefore, it is important to understand their behaviors in more depth.

In most DNS software such as BIND and Unbound, we conduct controlled experiments with the help of documentation and source code analysis, and summarize their behaviors (which generally match what we observe in real resolvers). Specifically, when configured as forwarders, they have a similar behavior to resolvers but typically have a longer timeout (and it is generally easier to extend their attack window using different strategies). So, we focus on resolvers' behaviors below.

In the case when there is no failure, both BIND and Unbound maintain a default retransmission timeout (RTO) — 0.8s for BIND, and a dynamically computed value based on RTT (to the authoritative nameserver) for Unbound. If timeouts occur (e.g., the nameserver is unresponsive or muted), they will contact another nameserver in a round robin manner if more than one is available. If all of them failed to respond, they will exponentially back off by doubling the RTO — BIND starts the backoff only after 3 consecutive failure whereas Unbound does it after every failure). Finally, there is another hard-stop condition — default 10s total wait time for BIND and 16 to 32 trials for Unbound (depending on the type of query). A `SERVFAIL` will be sent back to the client if the hard-stop condition is met.

Here we refer to the RTO as the "attack window" as it represents the duration where a source port remains unchanged. When the window ends, a different source port will be chosen, nullifying any previous port scan progress — a new port may happen to pop back into the range that is just scanned. It is important to note that when the attack window is too small (e.g., 1s), even if the port is correctly identified, it will still take time to inject 64k rogue DNS records (at a flooding rate of 100k packets per second(pps), it may still take a few hundred milliseconds), which may not finish before the window closes.

Generally speaking, if the authoritative nameserver is muted for an extended duration, we do expect to see larger attack windows (as RTOs double over failed attempts). With BIND being more reluctant in doubling the RTO and having a tighter hard-stop condition (default 10s), we believe it is a more difficult attack target. We describe an experiment against such a difficult case in Chapter 3.8.

3.6.3 Handling Multiple Authoritative Nameservers

Many domains in practice are configured with multiple authoritative nameserver IPs, for redundancy and security. Some consider this as a specific defense against DNS cache poisoning attacks against resolvers (called "IP randomization") [61], as it increases the randomness of a DNS query. According to a recent measurement study [94], second level domains like example.com under TLDs like .com, .net and .org have a median of only 2 NS only (and a mean of 2.3, 2.4, and 2.4), therefore this is not a strong defense by itself.

There are two ways to handle this. First, a general strategy is to simultaneously mute all the authoritative nameservers, given that on average few of them exist. This will help the RTO to grow exponentially after a resolver experiences repeated failures when contacting all the nameservers.

Second, if a resolver is Unbound, it has a unique behavior where it will stop contacting a nameserver (blacklisting the server) and switch "permanently" (i.e., minutes) to other available ones, should it repeatedly fail to hear from the originally-contacted server [61]. The authors in [61] therefore take advantage of this behavior to perform what they call "nameserver pinning". In our case, we need to allow periodic successful responses (by suspending the muting process); this is to avoid the last nameserver being blocked as well.

3.6.4 Handling Multiple Backend Servers Behind DNS Resolvers

As described in Chapter 3.4.5, many public DNS resolvers have multiple backend servers (with different IPs) that perform the actual queries. Interestingly, we find that the backend server selection is typically heavily skewed towards a few (even when we do see

100+ in total for some providers), likely determined based on location and past performance measurements. This allows us to focus on only a few IPs at the same time, which is easily achievable consider each IP only requires a scan traffic of 1kpps.

3.7 End-to-End Attacks

In this section, we evaluate our attack in realistic settings, including a forwarder used in a home, and a production resolver with a realistic configuration and network conditions.

3.7.1 Attacking a Forwarder (Home Router)

Experiment Setup. Given that most vulnerable routers have a fairly similar behavior shown in Table 3.1, we choose Xiaomi R3 (a Wi-Fi home router) as a representative case study to launch end-to-end attacks. It is used as the one and only gateway in an actual home where 10 to 15 devices are connected to the Internet through the wireless router all the time. In addition, Xiaomi R3's upstream DNS server is set to Cloudflare DNS (`1.1.1.1`). Its DHCP server is by default configured to provide 253 IPv4 addresses in a /24 network. Finally, the attack machine is a Raspberry Pi, which also connects to the router wirelessly.

Since Xiaomi R3 does not deploy global ICMP rate limit and its forwarder software does not call `connect()` on UDP sockets, we use strategy ii. in Chapter 3.4.3 (obtaining multiple IPs through DHCP) to bypass its per-IP rate limit. For extending the attack window, we use strategy described in Chapter 3.5.1 with a malicious nameserver.

Attack Process. The attack is divided into two phases, In Phase I, the attacker tries to acquire 240 IP addresses using the DHCP strategy. Afterwards, the attack goes into Phase II where the following repeats: the attacker issues a query to the forwarder asking for an arbitrary subdomain, e.g., `nonce.attacker.com`. If `SERVFAIL/NXDOMAIN` is received or if an attacker has waited for longer than one minute, indicating something is wrong, we will repeat the attack process by issuing another query. Otherwise, if a `NOERROR` response is received, it means a forged response is injected successfully. In Phase II, the attacker uses acquired IP addresses to scan open ports on the router. We rotate among the available IPs and make sure that we never go above the per-IP rate limit (which is 1pps in steady state). After a port is found open, we confirm that it stays open for at least one second by repeatedly probing the same port. If it does, we start injecting rogue responses. The experiment is repeated 20 times and we report the success rate, average time-to-succeed, and other statistics.

Results. Overall, the attack is very effective, with a success rate of 100% out of the 20 experiments (we consider it a success if the attack finishes within 30 minutes). The average time-to-succeed is 271s, with a breakdown of 103s in Phase I and 168s in Phase II. The standard deviation of Phase II is 109s with the maximum of 739s and the minimum of 83s. The variance is large because the attack time is mainly determined by the attack window size, which is the timeout before a resolver decides to give up and return `SERVFAIL/NXDOMAIN`, as mentioned in Chapter 3.5.1, and the timeout on Cloudflare's resolver varies a lot (from seconds to more than one minute for unknown reasons). Also, the attack needs to scan 36,325 ports on average to succeed; the average port scan speed

is 210 ports per second(PPS), which roughly matches the expected rate of 240PPS when using 240 IPs to scan. Besides, the attack generates 78 MB of traffic.

3.7.2 Attacking a Production Resolver

Even though the attack can work in principle against a large fraction of public DNS resolvers, due to obvious legal and ethical concerns, we refrain from targeting any of them. Fortunately, we obtained authorization to test the attack against a production resolver managed by a collaborator.

Experiment Setup. The resolver processes about 70 million queries daily with thousands of real users across multiple institutions and is configured as an *open* resolver. Because of this, it will be noisy and representing a challenging attack target. Another behavior noteworthy is that it has two backend servers, both of which appear to use `connect()` on the UDP sockets. Interestingly, we were told that they are running `Unbound`, and we suspect that the `connect()`-like behavior can be due to stateful UDP firewalls responsible for filtering out-of-state packets. We are given an attack machine in an adjacent network — 4 hops away from the resolver, which has a 1Gbps Ethernet and can perform IP spoofing.

Exp.	RTT	Probe Loss	NS Mute Lvl.	Avg. Time Used	Succ. Rate
Base(D)	0.2-1.2ms	~0%	80%	504s	20/20*
Base(M)	0.2-1.2ms	~0%	80%	410s	20/20*
Mute Lv.	0.2-1.2ms	~0%	75%	1341s	18/20*
Mute Lv.	0.2-1.2ms	~0%	66.7%	2196s	20/20#
Mute Lv.	0.2-1.2ms	~0%	50%	8985s	9/20#
Altered	37-43ms	0.20%	80%	930s	5/5*

*: 1-hour threshold. #: 3-hour threshold. D: Day. M: Midnight

Table 3.3: `SADDNS` Production Resolver Attack Results

Also, we setup a test domain and host it on an authoritative server controlled by us so that we poison only our own test domain. We configure the BIND software with a response rate limit at a low rate of 10pps to minimize the impact on the network. Once the limit is reached, we allow 1 out of 5 responses — an effective loss rate of 80%. This forms the setup of *Base* experiments, and we have conducted 20 rounds of them with one group in daytime(D) and the other group at midnight(M) local time (as shown in Table 3.3). In addition, to understand the effect of response rate limit on the authoritative nameserver, we vary the mute level by allowing a loss rate of 75%, 66.7%, to 50% — the lower the loss rate, the more difficult the attack is.

As a comparison, we also simulated more realistic network conditions by imposing additional delay, jitter, and loss on the same attack machine. The exact numbers are presented in Table 3.3 where the *Base* represents the unmodified network condition and *Altered* represents the simulated condition. We take the numbers with reference to recent Internet measurements [47][32]. We believe an attacker is likely able to find networks with even better conditions. To deal with increased FPs caused by the simulated network condition, we used two IPs to launch the attack in the *Altered* experiment; this is to avoid halting the scan too frequently due to the per-IP token being drained (see Chapter 3.4.3).

Finally, we are also interested in understanding the influence of the parameter "nameserver mute level", on the viability of the attack and will conduct a controlled experiment varying the "mute level" where all other parameters are the same as those in *Base*.

Attack Process. The process similarly starts from the attacker generating queries asking for `nonce.attacker.com`. Since the resolver has two backend server IPs, we launch the port scans on both IPs simultaneously. At the same time, we mute all authoritative nameservers with queries at a rate of 20qps so that the resolver will experience a constant loss rate of 80%. The experiment is repeated 20 times and 5 times for the *Base* and *Altered* respectively.

Results. As shown in Table 3.3, we achieved a perfect 100% success rate for the first *Base* experiment Base(D), with an average time of 504s to succeed. The standard deviation is 399s with the maximum being 1404s and the minimum being 13s (which is simply due to luck). On average, only 69 MB of attack traffic is generated, which is similar to that in the forwarder attack even though resolver attacks take much longer to succeed. This is because a forwarder attack is much more likely to enter the TxID bruteforce phase (6 times vs. twice), which generates about 10 MB of traffic every time. Specifically, strategy ii.(in Chapter 3.4.3) used in the forwarder attack does not have a binary search phase and an open port is simply confirmed twice before it enters the TxID bruteforce phase whereas the binary search phase employed in the resolver attack checks repeatedly the existence of an open port.

After inspecting the detailed log, we found that even though Base(D) experiment has a near perfect network condition, many more packets were sent compared to the forwarder attack. This is because of the frequent change of source ports caused by either resolver retries (i.e., RTOs) or new queries initiated by the attacker (if the resolver happens to receive a legitimate response), resulting in many small and fragmented attack windows.

In fact, we find more than half of these fragmented attack windows to be smaller than 1s, making them undesirable. Interestingly, we do find a decent fraction of large attack windows (10% of them with a 30s or larger). Such long attack windows match the profile of an Unbound resolver — 16 maximum allowed retransmissions, each doubling the RTO. In Chapter 3.8, we demonstrate that a BIND attack with much smaller attack windows appears to be still feasible but taking much longer time to succeed.

As shown in Table 3.3, the Base(M) experiment has the same exact setup as the Base(D) except that it is conducted after midnight where background traffic and noises will be generally lower. We observe the same 100% success rate and the average time to succeed decreasing from 504s to 410s. This is expected as our attack is sensitive to noises.

In addition, for the *mute level* experiments shown in Table 3.3, all but 50% mute level (i.e., loss rate) can still achieve a near perfect success rate and can finish generally within an hour (note the threshold of success being 3 hours for the 66.7% mute level). For 50% mute level, the attack succeeded only 9 out of 20 cases. Moreover, the average time taken is 8,985s or 2.5 hours.

Finally, for the *Altered* experiment, we also achieved a perfect 100% success rate. Specifically, the time to succeed is 2005s, 538s, 792s, 1287s and 29s respectively. On average, the attack time is 930s and 131 MB of traffic is generated. Note that the scan speed in the *Altered* experiment is higher than that in the *Base* experiment. This is because we used two IPs in the *Altered* experiment, reducing the frequency of halting during scans.

We also find that the increased loss rate and jitter causes more FPs, where we incorrectly consider a port discovered (as the verification packet successfully solicits an

ICMP). This is commonly caused by any loss of probing packets which can create two problems: (1) we waste much time filtering these FPs during the binary search stage, reducing the effective scanning speed; (2) The scan can still be halted because of frequent draining of the per-IP ICMP tokens even though we used two IPs.

3.8 Discussion

3.8.1 Attack Against Unbound vs. BIND

As mentioned previously, a BIND attack would be much tougher than Unbound as most of the fragmented attack windows will be generally smaller, as it is more reluctant in doubling the RTO and have a tighter hard-stop condition (as discussed in Chapter 3.6). To understand if is ever feasible to attack a BIND resolver, we construct an extreme experiment with 4 nameservers, and a default hard-stop condition of 10s wait time on the BIND resolver, resulting in the resolver almost always stuck in a small attack window of 0.8s, as querying 4 nameservers for 3 rounds already take 9.6s (before the RTO backoff can kick in). The experiment is conducted in a similar network environment to *Base*. Surprisingly, we run the experiment twice and both succeeded (one in 0.54 hours and the other in 1.25 hours). We find that it is indeed possible to succeed in scanning a port as well as injecting rogue records all in a 0.8s window. One attack we inspected showed that the port scan took 600ms and the record injection took 200ms.

3.8.2 UDP Source Port Inference on Other Operating Systems

In addition to Linux, we have verified that other major OS kernels are vulnerable as well, albeit with lower global rate limit — 200 in Windows and FreeBSD, and 250 in MacOS. It is concerning that not a single OS is aware of the side channel potential of global rate limits, despite the recent serious side channels specifically leverage a challenge ACK global rate limit in TCP [22]. We argue that all global rate limits in networking stacks need to be scrutinized regardless of their original design goal. We believe this work can serve as another valuable reference.

3.8.3 Other Vulnerable Protocols

Any protocols based on UDP are affected by the source port inference. A prominent example is QUIC [68] and HTTP/3 [87] which are poised to replace the traditional TCP-based web protocols with a much more efficient UDP-based protocols. They are already widely deployed in Google's web services [110]. In addition, VoIP, video streaming, and delay-sensitive online games may also use UDP, which are subject to port inference, and even off-path packet injection attacks.

3.9 Best Practices in Configuring Response Rate Limiting (RRL)

Even though response rate limit on authoritative nameservers is an important mitigation against DNS reflection/amplification attacks, if not done carefully, it can allow the extension of attack window in a DNS cache poisoning attack. We endorse the RRL

behavior (which was configurable but not always used) where a server still responds with truncated messages when a rate limit is reached [116] instead of being silent. This way, the amplification factor is no longer favorable to a DDoS attacker. Yet, it sends a strong signal to the resolver indicating something bad is going on, and the resolver should immediately react, e.g., either switching the source port and sending a new query, or falling back to TCP altogether (as recommended in [116]). This strategy can reduce the susceptibility of RRL being maliciously taken advantage of, compared to the cases where a server is completely muted (with 100% loss). Unfortunately, as we show in the resolver attack, even a 66.7% drop rate would already make a server vulnerable, not to mention that a determined attacker with more resources can simply flood the server with expensive queries (e.g., to non-existing domains [83]).

3.9.1 Defenses

The proposed attack is fundamentally an off-path attack and therefore can be mitigated by additional randomness and cryptographic solutions. Besides DNSSEC and 0x20 encoding, there is also an emerging feature called DNS cookie that is standardized in RFC 7873 [5] in 2016. At a high level, it requires both client and server to exchange some additional secrets unknown to an off-path attacker; it therefore has the potential to defeat most (if not all) off-path DNS attacks. Note that this feature requires both resolvers and authoritative nameservers to upgrade in order to see benefits. As of now, only BIND has implemented this feature and have it turned on by default in 9.11.0 forward [53] (released in 2016). We find about 5% of the open resolvers that we measured have enabled this feature by default. However, as any other unproven technology (the lesson regarding 0x20 [33]), it

remains to be seen if issues such as compatibility will prevent it from being widely adopted. Interestingly, we already found both DNSPod (operated by Tencent) and a resolver in a private company drop queries with DNS cookie options, likely for compatibility concerns.

In addition, our attack relies on the two fundamental components: (1) inferring source port of a DNS query; (2) extending attack window. Each of them can be a security threat on its own and therefore we discuss how to address both.

For (1), the simplest mitigation is to disallow outgoing ICMP replies altogether (as is done by many servers), at the potential cost of losing some network troubleshooting and diagnostic features. Otherwise, we need to address the global rate limit. As with patches on TCP global counters [100], we suggest a randomized ICMP global rate limit, including possibly randomizing the max allowable burst (currently 50), minimum number of tokens recovered each time (currently 20), minimum idle time to recover tokens (currently 20ms), and number of token recovered per time unit (currently 1 per millisecond). When the side channel is mitigated, we also recommend resolvers adopt the use of `connect()` on their UDP sockets so that their source ports will not be public-facing and directly scannable.

For (2), we have discussed best practices to use RRL to prevent an attacker from muting authoritative nameservers easily.

Other simple mitigation strategies include: (a) setting the timeout of DNS queries more aggressively (e.g., always below 1s). This way, the source port will be short-lived and disappear before the attacker can start injecting rogue responses. The downside, however, is the possibility of introducing more retransmitted queries and overall worse performance.

And (b) Employing anycast to make it harder for an attacker to DoS a specific authoritative nameserver used by a victim resolver.

Responsible Disclosure. SADDNS was assigned as CVE-2020-25705 and was largely patched at the time when this thesis is being finalized. Specifically, after our suggestion, Linux patched SADDNS by introducing the randomness to the global ICMP rate limit counter. Specifically, instead of deducting one token from the bucket, after patching, either 0, 1 or 2 tokens will be deducted from the bucket, and the attacker therefore cannot predict the number of tokens in the bucket precisely. Both FreeBSD and macOS patched SADDNS as well. Comprehensive patching details and the severe real world impact of SADDNS can be found through the link listed on SADDNS website [88]. Nevertheless, in Chapter 5, we will show how SCAD discovered unexpected side channels that would revive SADDNS attack even after patching. Note we tried our best to provide the comprehensive patches against SADDNS attack when we discovered it in 2020 and we were not even aware of the unique attack angle reported by SCAD at that time.

3.10 Conclusion

This chapter presents a novel and general side channel based on global ICMP rate limit, universally implemented by all modern operating systems. This allows efficient scans of UDP source ports in DNS queries. Combined with techniques to extend the attack window, it leads to a powerful revival of the DNS cache poisoning attack, demonstrated with real-world experiments under realistic server configuration and network conditions. Finally, we suggest practical mitigations that can be used to raise the bar against such attacks.

Chapter 4

SADDNS 2.0: Resurrect DNS Cache Poisoning Attacks With Spatial Next Hop Exception Cache

4.1 Introduction

In SADDNS, which we introduced in Chapter 3, the key insight is that a shared resource *(i.e., ICMP global rate limit)* between the off-path attacker and victim, can be leveraged to send spoofed UDP probes and infer which ephemeral port is used. Unfortunately, it is unclear how many more such side channels exist in the network stack. In this chapter, we explore a non-conventional type of port scan packets *(i.e., ICMP packets)* which are by design error messages and cannot solicit any explicit response. This is distinct from SADDNS where it has considered UDP packets which are conventional port scan pack-

ets. Even though it is known that ICMP can interact with UDP/TCP [96, 6], e.g., shutting down a socket (with an ICMP port unreachable message), it is not immediately obvious how ICMP probes can allow an off-path attacker to infer the ephemeral port number selected for a UDP socket. Surprisingly, we uncover novel side channels that have been lurking in the Linux network stack for over a decade and yet were not previously known.

The successful exploitation of these side channels in the context of DNS hinges on the subtle interactions among three different layers *(i.e., ICMP, UDP, and application)*. Interestingly, due to the lack of documentation and awareness, such interactions are often neglected and misconceived, leading to many exploitable scenarios. In addition to novel side channels, we also find that ICMP messages can be used to DoS DNS transactions, indirectly assisting the cache poisoning attack.

We have comprehensively characterized the impact of the side channels. They affect the most popular DNS software including BIND, Unbound, and dnsmasq running on top of Linux. In addition, we estimate that they affect 13.85% of open resolvers. Finally, we evaluate the end-to-end attack on the latest BIND resolver and a home router and find that it is reliable and takes only minutes to succeed. To mitigate the attack, we suggest setting proper socket options, randomizing the caching structure, and rejecting specific ICMP messages when possible.

We summarize our contributions as the followings:

- We discovered novel side channels that allow us to use ICMP probes to scan UDP ephemeral ports.

- We thoroughly analyzed the root cause of the discovered side channels and developed powerful DNS cache poisoning attacks based on that.
- We measured their impact in the real world and proposed corresponding mitigations.

4.2 Background

In this section, we will introduce the necessary background regarding the ICMP messages that interact with UDP in interesting ways.

As first introduced in RFC 792 [96], ICMP is a diagnostic protocol used to signal errors during the delivery of IP packets. This can happen, for example, when a router discards the packet and return an ICMP TTL expired message back to the source after it detects that the TTL of the forwarded packets reaches zero. To allow the source to distinguish which packets have encountered errors, a partial copy of the packet is embedded in the ICMP message, which includes the source and destination address, source and destination port. According to recent RFCs [54], the source should accept such messages only if the wrapped four-tuple matches an existing socket. Upon validating the correctness of such an ICMP message, depending on the nature of the error and the socket options set by the application, the source may ignore the error, remedy the situation by taking actions in the OS kernel *(e.g., updating routing entries)* and/or reporting the error to the application layer through the socket interface.

Below we describe a few relevant ICMP message types that have interesting interactions with UDP:

- **Fragmentation Needed.** Such messages are typically sent by a router to signal the source that the size of its packet has exceeded the MTU of the next hop [91, 6]. Specifically, they are called "fragmentation needed and DF set" or "packet too big" for IPv4 and IPv6 respectively. The desired MTU is included in the message so that the source OS can take actions *(e.g., updating its PMTU cache for the corresponding destination)* and reducing the size of all future packets with the same destination address.

- **Redirect.** Redirect messages [96, 113] are usually sent back to the source by the next-hop router *(e.g., gateway)* to signal a shorter route to a destination. After the source receives such a message, it will update its routing table and route all future packets to that destination through the new gateway, which is specified in the redirect packet. This message is only supposed to be sent by the gateway, and therefore, the OS of the source usually checks the source IP of the ICMP message before accepting the redirection [113].

- **Host/Port Unreachable.** Such messages are used to signal the source that the original packet was sent to the wrong host or port and thus cannot be delivered [96, 6]. According to RFCs [16, 6], upon receiving such messages, the OS must notify the application as long as a socket is found based on the embedded four-tuple in the ICMP message.

4.3 Attack Overview

SADDNS 2.0 takes the same threat model and attack workflow as SADDNS, which is presented in Chapter 3.3. In summary, there are 6 steps of the attack:

1. Identify the victim resolver, the domain to poison, and its nameserver.
2. Slow down nameservers and prevent them from responding to the victim resolver to give the attacker more time.
3. Start triggering the query on the resolver.
4. Infer the ephemeral port of the query using our new side channels (Chapter 4.4).
5. Once the port is known, inject 65,536 rogue responses with different TxIDs to the victim resolver by spoofing the nameserver's IP.
6. Check if the cache is poisoned. If not, go back to (3).

4.4 ICMP-Based Ephemeral Port Scans

In contrast with the traditional methods of UDP-based port scans, as mentioned in Chapter 3.4, in this chapter, we investigate the ICMP-based port scans. As mentioned in Chapter 4.2, an ICMP message embeds the header of the original packet from the source, including the source and destination port information. This opens up an opportunity to craft an ICMP message embedding a guessed port number, which is used to match a specific socket on the receiver end [96, 6]. However, the challenge is that ICMP messages are by design error messages useful for diagnostic purposes only, which do not solicit explicit

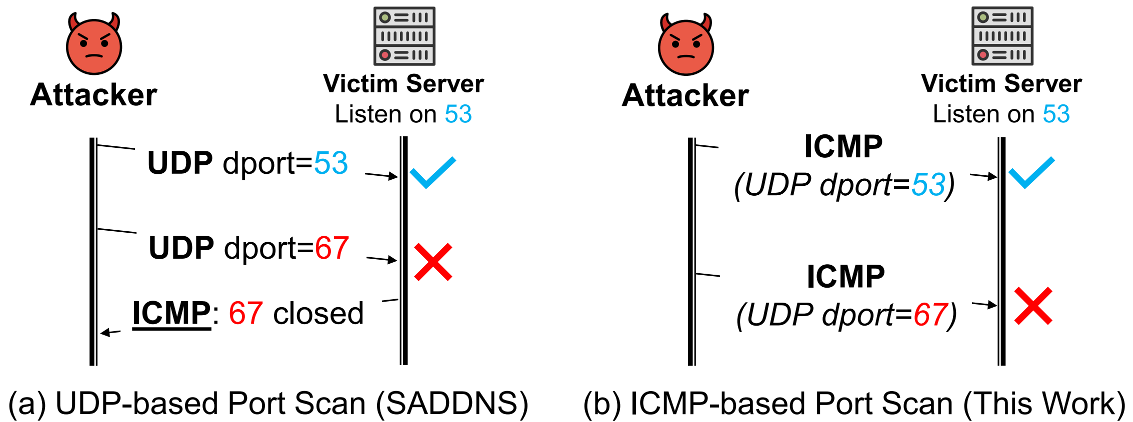


Figure 4.1: Ephemeral Port Scan

responses [16]. This means that regardless of whether a port number is guessed correctly, the receiver will not provide any response, as shown in Figure 4.1(b), making the ICMP-based port scans seem infeasible.

Surprisingly, we observe that an attacker does not necessarily have to rely on the explicit feedback from an ICMP probe. Instead, even if the processing of ICMP probes is completely silent, as long as there is some shared resource whose state is influenced, we may find ways (other probes) to observe the changed state of the shared resource. This is a generalization of the prior probing methods that rely on spoofed probes that by design can solicit responses from the victim. In addition to `SADDNS` whose probes are designed to solicit ICMP responses, it is also the case for the series of TCP side channels [22, 27, 84]. Specifically, [22] leveraged TCP probes that can solicit challenge ACKs; [27, 84] required TCP probes that can solicit any response. In summary, it requires a leap of faith to realize the potential of the ICMP-based probes to scan UDP ports.

In this chapter, we systematically investigate all types of ICMP and narrowed them down to two that are useful for port scans: *ICMP fragmentation needed (or ICMP*

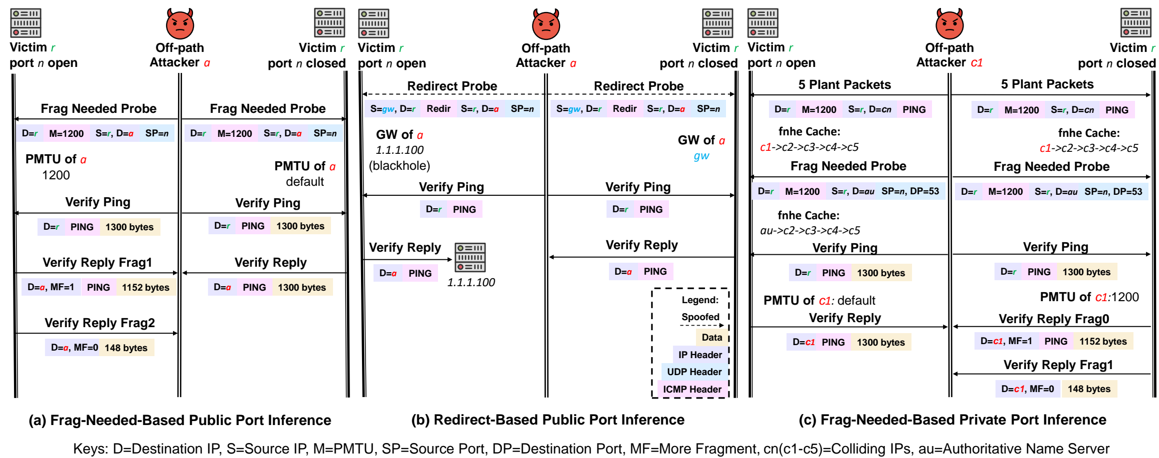
packet too big in IPv6) and *ICMP redirect*. Next, we will describe their processing logic in the Linux kernel and the corresponding shared resources that form side channels.

4.4.1 Analysis of ICMP Error Processing Logic

We use the ICMPv4 (ICMPv6 is similar) in the Linux kernel 5.11.16 as an example to illustrate this. When the OS receives an ICMPv4 message with an embedded UDP packet, it will invoke `__udp4_lib_err()` to handle the error. Here the four-tuple in the wrapped UDP packet is first checked with the socket table (`__udp4_lib_lookup()`) to verify the legitimacy of the ICMP packet *(i.e., it is indeed triggered by the packet the host sent before)*. If it passes the check, the ICMP error will be handled according to the type of error. Additionally, the ICMP error may optionally be delivered to the application if the OS has received the proper socket options (which will be described in Chapter 4.5.1).

To handle the ICMP fragmentation needed and redirect, two corresponding kernel functions are invoked respectively: `ipv4_sk_redirect()` and `ipv4_sk_update_pmtu()`. Both of them will update a global resource maintained in the routing module, called the **n**ext **h**op **e**xception (f**nhe**) cache. We refer to it as "exception cache" in short from here on. It stores various states including the non-default MTU for specific remote IPs (updated by ICMP fragmentation needed messages), and the non-default gateway IP for specific remote IPs (updated by ICMP redirect messages). These exception cache entries affect the routing decisions for all future outgoing packets destined to the remote IPs in the entries. These entries are cached for some time unless explicitly evicted due to a limit on the total number of entries (details are provided in Chapter 4.4.3).

One thing worth noting is that the OS does not check the source IP address of the ICMP fragmentation needed messages. This is by design as such messages can be generated by any router along the path. And due to the dynamic nature of the Internet, the victim resolver cannot easily verify if a given IP belongs to the routers along the path. This has an interesting implication that the attacker's probes of ICMP fragmentation needed messages, which we will describe next, do not need to spoof the source IP address at all.



(a) Frag-Needed-Based Public Port Inference (b) Redirect-Based Public Port Inference (c) Frag-Needed-Based Private Port Inference

Keys: D=Destination IP, S=Source IP, M=PMTU, SP=Source Port, DP=Destination Port, MF=More Fragment, cn(c1-c5)=Colliding IPs, au=Authoritative Name Server

Figure 4.2: SADDNS 2.0 Ephemeral Port Number Inference

4.4.2 Public-Facing Port Number Inference

We illustrate the basic idea of public-facing ephemeral port scan in Figure 4.2(a) & 4.2(b). For ICMP fragmentation needed, all we need to do is to send an ICMP fragmentation needed message with the attacker's own IP address (which is unchecked by the resolver as mentioned above). The message embeds a UDP header with a guessed source port and a destination port of 53. It is also supposed to contain the source and destination IP addresses, which should be the resolver's IP and nameserver's IP respectively. However,

some popular DNS software such as Unbound (IPv4 only) and dnsmasq produce public-facing ephemeral ports (also called wildcard sockets in the kernel terminology). It turns out that Linux (and other OSes) treat such public-facing ports much more liberally and accept any inner destination IP address in an ICMP message, as long as the inner source address matches the resolver's IP and inner source port matches the ephemeral port. This effectively means that against such public-facing ports, one can easily trick the resolver to update the MTUs for any remote IPs (even though the resolver may not have even talked to them before). Therefore, in the attacker's probe packets, we will use its own IP address to fill the destination IP of the embedded packet such that the MTU for the attacker's IP will be lowered if the guessed ephemeral port is correct.

To observe the change in the cache, the attacker can simply send a PING or any other packet (verification packet) that will trigger a reply (verification reply) from the resolver, and observe if the response will be fragmented as a result of the lowered MTU. As shown in Figure 4.2(b), if ICMP redirect is used for probing, the effect is that the victim resolver becomes unresponsive because the traffic to the attacker will now be redirected to a wrong gateway IP (potentially black hole) set in the redirect message.

4.4.3 Private-Facing Port Number Inference

Most DNS software *(e.g., BIND)* will produce private-facing ephemeral ports, rendering the previous method invalid. The first adjustment we have to make is to set the inner destination IP address to the IP of the nameserver. This is because `__udp4_lib_lookup()` will check the complete four-tuple of the embedded UDP packet to locate the socket that has previously been "connected" to a specific remote IP and port. The exception cache

state change is therefore also "private" to the nameserver and not directly observable by the attacker. For example, even if the MTU for the nameserver is reduced, an off-path attacker cannot directly observe the change because fragments will go towards the nameserver directly. Interestingly, it turns out that there is another method to indirectly observe the state change.

The key idea is to leverage the limited number of total slots in the global exception cache. By default, Linux organizes such a global exception cache as a 2048-bucket hash table which uses the destination IP address as the key and has a linked list of length 5 and 6 slots (for IPv6 and IPv4 respectively) to solve collisions for each bucket. When the linked list reaches the limit, the oldest exception will always be evicted and replaced with a newly inserted exception.

The requirement is that the attacker needs to create hash collisions with the nameserver's IP. As shown in Figure 4.2(c), the attacker first needs to find 5 IPs (in the case of IPv6) that can be hashed into the same bucket as the nameserver's IP on the victim resolver's exception cache and control at least one IP `c1` (the other 4 IPs can be spoofed). For now, we assume the attacker can find the 5 colliding IPs but will describe our tested strategy in Chapter 4.4.4.

As shown in Figure 4.2(c), once the colliding IPs are collected, the attacker first fully occupies the 5 allowed slots in the linked list using the 5 different IPs. This can be done by sending a series of ICMP fragmentation needed or ICMP redirect packets wrapping a PING reply packet [49]. The kernel blindly accepts ICMP errors caused by PING replies because they are sent by the kernel with no sockets and therefore matching the socket before

accepting is not possible. Subsequently, the attacker would proceed with the ephemeral port scan by probing different source ports with ICMP messages. If a probe happens to hit the correct ephemeral port, a new exception regarding the nameserver is to be inserted into the linked list and evict the first exception *(i.e.,* `c1`*)* prepared by the attacker. The attacker can observe this by a verification packet, in the case of MTU caches, checking the current MTU for `c1`.

4.4.4 Finding IPs That Cause Hash Collisions

Finding IP collisions has been studied before when leveraging IPID side channels [9, 48], where they needed to find a single IP address that collides within the same IPID bucket as the victim. [9] states owning 10,000 IPs would bring the colliding rate to an arbitrary IP over a 2048-entry hash table to more than 98%. Unfortunately, this naive brute force does not transfer well to our attack. Specifically, in order to observe a collision in the case of the exception cache, we know that we need 5 or 6 IP addresses to fully occupy a bucket entry. This means that we need to find at least 50,000 to 60,000 IPs to have a good chance. This is still easily achievable in IPv6 because ISPs often assign a /64 address block by default. However, for IPv4, we consider it possible but a very strict requirement. We therefore come up with an alternative strategy as follows.

Instead of finding the collision set directly, we choose to infer the secret used in the keyed hash function that computes the index into the 2048 buckets. First of all, the hash function is public (listed in the kernel source code). Secondly, since the secret is only 32-bit and persists until reboots, it is possible to crack it once and use it subsequently to check which IPs collide with a given nameserver's IP. This allows us to target a resolver and

potentially poison an arbitrary domain name after a single cracking. To infer the secret, the basic idea is to find some collision set (of 6 IPs in the case of IPv4) that allows us to test which secret can produce the collision set. The key is that in this process we no longer require a collision with a specific IP, i.e., the IP of a nameserver, and therefore we can benefit from the birthday paradox [112] — it is much more probable to observe a collision at any bucket rather than a given bucket. Based on our empirical evaluation, we only need 3,500 IPv4 addresses to reliably find one or more collision sets on some buckets. In particular, we rented 3,500 AWS EC2 instances to acquire 3,500 different random public IPs. Given that each tiny instance only costs less than one cent per hour, renting instances for sending probing packets is cheap. In practice, we found that one round of probing with 3,500 IPs is usually sufficient to find enough collision sets that allow us to uniquely pinpoint the secret — this takes only minutes computationally with 3,500 tiny CPU cores. In the rare event that we fail, we can simply re-acquire another set of 3,500 IPs and redo the probing. Finally, we also tested the same methodology with IPv6 where only 1,500 addresses were needed to achieve the same result because an IPv6 hash bucket has only 5 slots instead of 6.

4.4.5 High-Speed Scans

As one can expect, for either public-facing or private-facing ports, an attacker can probe multiple source ports simultaneously to learn if any of the guesses match the correct ephemeral port. We confirmed with small-scale experiments that both ICMP fragmentation needed and redirect messages are not rate limited on the Internet (see Appendix B). We consider two options below.

Batch Scan. We can probe many ports at once, and check whether any of them has hit the correct port. If it does, we can then re-probe a smaller sub-range (e.g., a binary search) to narrow down on the exact port. In this strategy, every round of probes will incur at least one round trip time between the attacker and victim (as mentioned in Chapter 3.4.3). Note that we will need to somehow reset the exception cache state once we hit the correct port in a batch. This is because we have already evicted one of the exceptions we planted earlier. We will describe the methods in detail in Appendix C.

Single Packet Scan. An alternative strategy is to scan only a single port in each batch (batch size equal to 1). This means that every scan will be accompanied by an additional verification packet. Even though this sounds like a sub-optimal strategy, we point out that the probes can in fact be initiated in a pipeline, without having to wait for feedback for previous probes. This is because our verification packet can encode a unique ID *(e.g., ping ID)* that can differentiate after which batch of probes, an update in the exception cache has taken place. Of course, we can also use a larger batch size. However, as mentioned, it will incur additional round trips to narrow down the search. In contrast, the single packet scan (a batch size of 1) will allow us to precisely pinpoint which port is open without the additional round trips. The tradeoff is that for every ephemeral port we scan, two packets need to be sent *(i.e., one is the probe, the other is the verification packet).*

As the attack is highly time-sensitive, we favor fewer round trips over higher bandwidth consumption. We wish to point out that this allows us to scan at a much higher speed than 1,000 per second which was the limit in `SADDNS`.

4.5 Vulnerable Population

In this section, we will first study the necessary conditions for the vulnerability to be present and exploitable. Then we study the vulnerable combination of OS and DNS software. Interestingly, the outcome is determined by both the OS and DNS software (sometimes either one). In addition, we also explored historical versions of OS and DNS software because a large fraction of resolvers on the Internet may not be running the latest software. We then conduct a measurement study to measure the vulnerable population of open resolvers on the Internet that satisfy the vulnerable conditions. Due to measurement constraints, we also conduct a small-scale experiment on ICMP redirect attack (see Appendix A).

4.5.1 Conditions of Successful Attacks

Below we summarize the key necessary conditions for a resolver to be considered exploitable:

- $C1$. Must check the port number in the embedded UDP packet of an ICMP error before processing it. [**OS**]
- $C2$. Must cache the MTU or next-hop information. [**OS**]
- $C3$. Must not ignore the ICMP fragment needed or ICMP redirect messages in the kernel. [**APP/OS**]
- $C4$. Must not shutdown or retransmit the query after receiving ICMP messages. [**APP/OS**]

For $C1.$ & $C2.$, they form the basis of side channels in the kernel. As mentioned earlier, the latest Linux kernel satisfies both conditions.

For $C3.$, interestingly the latest Linux kernel allows applications to pass special socket options (either `IP_PMTUDISC_OMIT` or `IP_PMTUDISC_INTERFACE`) which will cause the kernel to ignore the fragmentation needed messages for the corresponding sockets. However, this feature was introduced in Linux kernel 3.15. Therefore, whether or not the condition is satisfied depends on both the kernel and DNS application. Nevertheless, ICMP redirect messages are not affected by any socket option and are always processed in the kernel.

For $C4.$, it is a necessary condition because the port scan assumes the ephemeral port stays the same after it is successfully detected. If an application decides to shutdown the connection or retransmit the query after receiving an ICMP message (embedding the correct ephemeral port), then the detected ephemeral port will be effectively forfeited. Interestingly, this is again determined by the OS kernel as well as the application. First of all, the OS kernel has to expose the ICMP error messages to the application layer (again ICMP redirect never gets exposed). Secondly, an application may choose to react to such errors in different ways.

Kernel Ver.	$3.6-3.14$				$3.15-4.14$				> 4.15						
DNS Version	BIND $9.3-9.11$		BIND > 9.12		BIND $9.3-9.11$		BIND > 9.12		BIND $9.3-9.11$		BIND > 9.12		Unbound $> 1.5.2$		dnsmasq *ANY*
IP Ver.	4	6	4	6	4	6	4	6	4	6	4	6	4	6	4/6
$C1.$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$C2.$	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
Redir Vuln.	V_{priv}	✗	V_{priv}	✗	V_{priv}	✗	✗	✗	V_{priv}	V_{priv}	✗	V_{priv}	✗	V_{priv}[1]	V_{pub}
$C3.$	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓
$C4.$	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓
Frag Vuln.	V_{priv}	✗	V_{priv}	✗	V_{priv}	✗	✗	✗	V_{priv}	V_{priv}	✗	V_{priv}	✗	V_{priv}[1]	V_{pub}
Vuln. in Any	✓		✓		✓		✗		✓		✓		✓		✓

1: V_{pub} before 1.13.0. Note: V_{pub} and V_{priv} indicate vulnerable to public-facing or private-facing port scans respectively.

Table 4.1: Exploitability of Different DNS Software and Kernel Versions

In Table 4.1, we summarize the vulnerable combinations of Linux kernel and DNS software according to the above conditions. We break down the Linux kernel versions into three groups, representing three major changes that affect the above conditions. Similarly, we break down BIND into two groups because of some key changes in behaviors. As we can see, $C1$. is always satisfied in all recent kernel versions. Regarding $C2$., the Linux kernel since 3.6 is vulnerable in IPv4 because of the introduced exception cache. It took Linux some time until 4.15 to port the same exception cache to IPv6. Therefore, IPv6 redirect attacks, which only require $C1$. & $C2$. to work, are only exploitable on kernel versions newer than 4.15. Regarding $C3$., Since Linux 3.15, the socket options mentioned above become available and BIND decides to use `IP_PMTUDISC_OMIT` since 9.12 for IPv4 sockets, leaving the condition satisfied for IPv6 sockets only. For $C4$., since Linux 3.15 and BIND 9.12, `IP_PMTUDISC_OMIT` on IPv4 sockets similarly causes the kernel to notify the application regarding ICMP fragmentation needed errors for sockets that have private-facing ports (therefore does not apply to older Unbound versions and dnsmasq). Furthermore, BIND will retransmit the query (with a different ephemeral port) upon receiving such a notification. As we can see, the interactions between the kernel and application layer are very much inconsistent and evolving constantly. We will discuss the reasoning behind them in Chapter 4.8.2.

In summary, for the latest versions of BIND and Unbound on the latest kernels, their IPv6 sockets can be exploited for the ephemeral port scan. In contrast, dnsmasq is always vulnerable as it does not set any special socket option. Nevertheless, in practice, IPv6 is gaining significant traction in deployment [55]. In fact, as we will show in Chapter 4.5.2,

half of the popular public DNS resolvers support IPv6. Furthermore, our attack is fully capable of exploiting a dual-stack (IPv4/IPv6) resolver, combined with techniques such as nameserver muting (as will be discussed in Chapter 4.6.4).

Due to practical concerns, we did not show the analysis results of historic versions of dnsmasq and Unbound in Table 4.1. For dnsmasq, it is vulnerable on all kernel versions since 3.6. For Unbound, it has a similar road map as BIND and starts to use `IP_PMTUDISC_OMIT` since 1.5.2. The only difference is that it used public-facing ports in the past. This leads Unbound to be not only vulnerable in the IPv4 of kernel versions between 3.15 and 4.14, but also IPv6 in the same kernel ranges. This is because the public-facing ports can be successfully scanned (as shown in Figure 4.2) as long as the MTU or redirect information is stored somewhere in the kernel. In practice, for kernel version 3.15 to 4.14, such info is stored in a tree which can only time out as opposed to being forcefully evicted.

Other Operating Systems. We have additionally analyzed FreeBSD (whose networking stack is also used by macOS) and Windows with regard to the previously described conditions.

For FreeBSD, it is not vulnerable because both $C1$. and $C2$. are broken for ICMP fragmentation needed and redirect respectively. For ICMP fragmentation needed messages, even though the OS will check the embedded four-tuple and act accordingly, it does not store any PMTU information in any kernel-maintained data structure and thus breaking $C2$.. Instead, it simply forwards the error to the application layer. This is actually not compliant with RFC1191 [91] which explicitly states that "the IP layer should associate each PMTU value that it has learned with a specific path" and "it (a host) should be able to cache

a per-host route for every active destination". For ICMP redirect packets, surprisingly, FreeBSD will blindly accept them without checking the embedded four-tuple and therefore breaks $C1$..

For Windows, we reverse-engineered `tcpip.sys` and `ntoskrnl` of a Windows 10 copy. We found that there is a similar hash table storing the path information (including the MTU). However, we did not find any eviction algorithm and it will only stop inserting new exceptions after the kernel runs out of memory. Although the attacker can still leverage this as a side channel, due to the large and different memory configurations, it is hard to do so in practice. However, lacking a cap on memory consumption of the hash table would lead to a potential DoS attack on the entire system. Despite this, our ongoing research shows the preliminary results that it is possible to use other behaviors to observe whether the PMTU hash table was modified or not.

4.5.2 Open Resolvers

Now we move on to measure the vulnerable population in the real world. Note in this section we focus on the attack leveraging ICMP fragmentation needed messages only. This is because ICMP redirect based attacks require IP spoofing even for port scans, and we are concerned that it is invasive to conduct such a large-scale IP spoofing experiment. Instead, we defer to Appendix A for a small-scale measurement of the conditions of the redirect-based attacks.

Setup and Dataset. Open resolvers represent hosts that provide recursive DNS lookup services to the public. We obtain a list of open resolvers from Censys.io [44], which contains 1.84M IPv4 addresses, serving as the dataset used in our measurement. Unfor-

tunately, the list does not contain IPv6 open resolver addresses. Nevertheless, these IPv4 addresses only correspond to the frontend IPs. In practice, most open resolvers will go through backend servers that conduct the actual DNS query on behalf of the frontend. Therefore, we design a method to solicit queries from IPv6 backend servers. Specifically, we control two domain names whose NS records point to an IPv4 and an IPv6 address respectively. For each frontend IP, we always send two queries asking for the IPv4 and IPv6 domain names respectively. For the domain where its NS record points to an IPv6-only address, it will force a backend server to use its IPv6 address to contact our nameserver. In the end, we are able to receive 129,196 queries from IPv4 addresses and 27,541 from IPv6 addresses.

Methodology. When a backend server (either IPv4 or IPv6) contacts our nameserver, we will perform the following four tests that approximately correspond to the four conditions we discussed earlier.

$T1$. **The rejection of the ICMP error when the embedded source port is incorrect.** To verify $C1$. in Chapter 4.5.1, we first send a PING to the resolver and record the reply. Then we craft an ICMP fragment needed packet wrapping the DNS query we received to signal that the PMTU is lowered. Before we send it, we deliberately change the source port of the embedded UDP packet to a different random value to check whether the resolver will blindly accept ICMP packets without checking the port number. After sending that forged packet, we send another PING and check if the ICMP is accepted. If the PING reply is not fragmented, we consider the resolver rejects the ICMP error and thus meets $C1$..

$T2$. **The existence of the next hop exception cache.** To verify $C2$. in Chapter 4.5.1, ideally we would want to directly test the existence of an exception cache. However, as described in Chapter 4.4.4 this will require us to find 5 or 6 IPs that would be hashed into the same bucket, causing the hash collision. Although it is a one-time effort, targeting every single open resolver will require sending a large amount of traffic which can be overly invasive. Therefore, we decide to resort to `nmap` to fingerprint the OS version of the resolver and check whether the cache exists according to the OS version discussed in Chapter 4.5.1. Note that `nmap` may not be perfect, especially when considering backend servers may not always have open TCP ports, through which most of the fingerprints are extracted by `nmap`. Nevertheless, we can use the distribution obtained from resolvers that do have open ports and extrapolate to those that do not. To minimize the impact, we sampled 20 out of 8,141 backend resolver IPs that have a valid `nmap` signature and performed the collision test using 3,500 rented IPs following the methodology described in Chapter 4.4.4. Note that this is still an intrusive test (we do slow down the packet speed to about 1,000pps to minimize any disruption) and thus cannot scale. The results show 16 out of 20 servers support `nmap`'s conclusion and therefore we estimate the accuracy of `nmap` 80%.

$T3$. **The acceptance of the ICMP error.** To verify $C3$. in Chapter 4.5.1, we use a similar test to $T1$. but without modifying the port number to verify if the resolver is willing to accept the ICMP packet at all. Additionally, if there is no PING reply at all, we will send a truncated DNS response to solicit the TCP query from the resolver. If the MSS in the TCP header is decreased according to the PMTU value indicated in

our ICMP packet (which we verify to be the behavior of modern Linux kernels), it also means the resolver has accepted the PMTU value inside the ICMP packet. Besides, we will conduct another test by changing the destination IP address in the wrapped IP packet if we find the resolver accepts the original ICMP. If the resolver also accepts the modified ICMP, it means its port is open to the public, and otherwise, we consider its ephemeral port as private-facing.

$T4$. **The open-port status after receiving the ICMP error.** To verify $C4$. in Chapter 4.5.1, after the ICMP fragment needed is sent during $T3$., we follow up with a "truncated response" (if it is not sent in $T3$.) indicating the response is too big which will cause the resolver backend to switch to TCP. If we observe a TCP handshake, it indicates that the ICMP error did not cause the resolver to close the original ephemeral port, therefore supporting the attack. In the more rare cases, even if we did not observe any TCP connection attempt, it is still possible that the ephemeral port is open and it is simply due to the resolver not supporting DNS over TCP. In such cases, we will check whether the nameserver will receive a retransmitted query (with a different ephemeral port) from the resolver immediately, which potentially indicates that the ICMP has induced the DNS software to close the ephemeral port and transmit another query. To distinguish between the ICMP-induced retransmission and the timeout-induced retransmission, we record the time delay between the ICMP transmission and the time we received the retransmitted query. Specifically, if the delay is close to RTT, which we collect in $T1$. by measuring the time delay between the PING response and the request *(i.e., within a 10% margin of difference)*,

we consider the retransmission to be caused by the ICMP. Otherwise, if the delay is larger than RTT, we will consider the retransmission to be timeout-induced (and thus still supporting the attack).

Results. Overall, out of the 156,737 backend resolver IPs that reach our nameservers, 13.85% of them are estimated to be vulnerable. If we count by frontend resolver IPs, out of the 1.84M, 37.72% are estimated vulnerable. This is because a large number of frontend IPs share the same backend. To further break down the total 13.85% vulnerable population in the backend, we find that 13,914 (8.9%) are clearly vulnerable to public-facing port scans. However, when we count the vulnerable population regarding the private-facing port scans, it requires a more accurate estimate of the Linux kernel version from `nmap`. Unfortunately, as mentioned earlier, we find `nmap` has a relatively low success rate of OS fingerprinting: only 63.26% for IPv4 addresses and 1.06% for IPv6 addresses. We therefore use the distribution of kernel versions observed from the 63.26% IPv4 hosts to estimate the total vulnerable population. In particular, within these IPv4 hosts, we find that 58.66% of them have the IPv4 exception cache only or also the IPv6 exception cache. We then apply the 58.66% to the 13,277 resolver backends that are suspected to be vulnerable (passing all other tests), resulting in an estimate of 7,788 backends being vulnerable to private-facing port scans.

The results indicate that the majority of the vulnerable population is not actually running BIND. Instead, they could be running an older Unbound, dnsmasq, or other DNS resolver software that we have not explicitly tested. Among the servers that are not vulner-

able, most of them are simply because they do not accept the ICMP fragmentation needed messages (including cases that we cannot tell) and fail in $T3$..

Public Resolvers. We also highlight the results of a few well-known public DNS services and summarize the result in Table 4.2. Overall, we find 6 out of 12 to be definitely vulnerable as of 2021, 3 in IPv4 and 3 in IPv6, including famous providers such as OpenDNS and Quad9. Interestingly, although the most popular DNS software BIND is not vulnerable in IPv4 in its latest releases, there are still 3 public resolvers vulnerable in IPv4, indicating that they are either running an older BIND version or a different DNS software (we know Cloudflare runs Knot [4]). Note that currently only 6 providers support IPv6 (others are marked as N/A) and we expect more DNS services to be impacted as they start supporting IPv6.

Name	Frontend IP	IPv4 Backend					IPv6 Backend				
		$T1.$	$T2.$	$T3.$	$T4.$	Vuln.	$T1.$	$T2.$	$T3.$	$T4.$	Vuln.
Google	8.8.8.8	✓	✗	✗	✓	✗	✓	?	✗	✓	✗
Cloudflare	1.1.1.1	✓	✓	✓	✓	V_{priv}	✓	✗	✓	✓	✗
OpenDNS	208.67.222.222	✓	?	✓	✓	P_{pub}	✓	✓	✓	✓	V_{priv}
Comodo	8.26.56.26	✓	✓	✗	✓	✗	N/A				N/A
Quad9	9.9.9.9	✓	✓	✗	✓	✗	✓	?	✓	✓	V_{pub}
AdGuard	94.140.14.14	✓	✓	✗	✓	✗	✓	✓	✓	✓	V_{priv}
CleanBrowsing	185.228.168.168	✓	✓	✗	✗	✗	N/A				N/A
Neustar	156.154.70.1	✓	✓	✓	✓	V_{pub}	✗	?	✓	✓	✗
Yandex	77.88.8.1	✓	✓	✗	✓	✗	N/A				N/A
Baidu	180.76.76.76	✓	✓	✓	✓	V_{priv}	N/A				N/A
114	114.114.114.114	✓	✓	?	✓	?	N/A				N/A
Ali	223.5.5.5	✓	✓	✗	✓	✗	N/A				N/A

Table 4.2: `SADDNS` 2.0 Vulnerable Status of Public Resolvers

The most common reason for not being vulnerable is again because they failed $T3$. *(i.e., the ICMP fragment needed messages do not appear to trigger the MTU to decrease).* As shown in Table 4.2, there are still a few cases where we are unable to fingerprint the

kernel versions even after we tried testing a few custom fingerprints in addition to nmap (marked with "?" in $T2$. column). For such cases, we simply mark them as "Possibly Vulnerable" ($P_{priv/pub}$) when they pass all other tests, since it is likely their public servers are well-maintained and using a newer Linux kernel.

4.6 Practical Concerns

In this section, we will describe a few practical considerations which will influence the success and reliability of the attack.

4.6.1 Small Attack Window

In addition to the methods described in Chapter 3.5 & 3.6.1 that can be used to extend the attack window, coincidentally, one of the ICMP messages, ICMP redirect, can be also used for nameserver muting. The idea is to send the malicious ICMP redirect to either the victim resolver or the nameserver to reroute the traffic destined to each other to a black hole. Since the query/response is lost after it reaches the wrong next hop, the victim resolver would keep the ephemeral port open for responses until the query timeouts (can be several seconds) and therefore creates a huge attack window.

4.6.2 Multiple Nameservers

It is also quite common for domains to have multiple nameservers. Resolvers may choose to query these nameservers in a round-robin fashion (where the order is randomized).

In fact, this is considered a defense against DNS cache poisoning attacks [89]. However, this defense has little impact on our attacks for the following reasons.

For resolvers with private-facing ephemeral ports, we can infer the ports specific to different nameservers simultaneously by running multiple scanning instances. Since it is unlikely the nameservers' IPs will share the same hash bucket given that most second-level domains *(e.g., ucr.edu)* only have three or fewer nameservers, the side channels can be independently leveraged without self-interference.

For resolvers with public-facing ports, the attacker can just scan the port as if there was only one nameserver since the kernel does not check the destination IP address wrapped in the ICMP probe. The only difference lies in the TxID brute-forcing, where the attacker would inject multiple groups of 65,536 fake response packets, where each group uses a spoofed IP of a different nameserver. Due to the low number of nameservers typically configured, this additional load of packets is not really a fundamental hurdle.

In addition to the above, we propose two new methods for "nameserver pinning" based on ICMP messages *(i.e., either host/port unreachable or redirect)*. In the case of BIND resolvers, every time when a query is initiated, we can immediately flood 65,536 (representing the worst cases. BIND uses only 23,232 ports by default) ICMP host/port unreachable messages containing all possible ephemeral ports with a specific nameserver's IP as the destination IP address in the embedded IP header. This will cause BIND to give up a particular nameserver in the duration of a query session (up to 10 seconds by default). This is because the OS will pass the host/port unreachable messages to BIND, which will make the subsequent decision to forgo the nameserver (one of the 65536 guessed ports will

match the ephemeral port). Alternatively, we can apply targeted nameserver muting as mentioned in Chapter 4.6.1 and targeted ICMP redirect to achieve a similar effect.

In the case of Unbound, ICMP redirect can be used as described above to mute specific nameservers. This is because Unbound has special logic to "blacklist" nameservers that are non-responsive repeatedly. Therefore, the ICMP redirect will have a prolonged pinning effect beyond a single query session.

4.6.3 Multiple Backend Servers

Finally, large DNS resolvers tend to have multiple backend servers behind a single frontend IP — usually an anycast one *(e.g., 8.8.8.8)*. These backend servers are the actual workers that talk to the nameservers and they are the ones that maintain DNS caches. Therefore the backend servers should be the actual attack target. An attacker can map out the IPs of the backend servers by setting up an attacker-controlled nameserver and issuing a query of the attacker-controlled domain. This will create an additional challenge to the attacker, as a particular query may get routed to a randomly selected backend IP not known to the attacker. This will mean that the attacker needs to target $m \times n$ pairs of resolver backends and nameservers, where m is the number of backend IPs and n is the number of nameservers. Otherwise, if the attacker picks only a single backend server to attack, it will have a reduced probability of $\frac{1}{m}$ (assuming the probability of choosing backend servers is uniformly distributed) to succeed. Fortunately, when m is large, it is typically a heavily distributed system that the selection of the backend IPs is actually not random at all. Instead, it is typically based on location. In other words, backend servers that are located closer to a nameserver will be more likely to be picked for a given query (destined to the

nameserver). In such cases, the attacker only needs to target a small number of backend servers simultaneously or even a single one and is still able to achieve a decent success rate.

4.6.4 Dual-Stack Resolvers

As mentioned earlier in Chapter 4.5.1, the latest BIND and Unbound will instruct the Linux kernel to ignore ICMP fragmentation needed messages for IPv4 sockets. Therefore, the vulnerability applies to only IPv6 sockets against them. In practice, both IPv4 and IPv6 are enabled by default in recent Linux distributions *(e.g., Ubuntu 20.04 and Red Hat 7).* Therefore, we need to understand how to target their IPv6 sockets in the presence of IPv4 sockets. Specifically, BIND and Unbound by default will query different nameservers in a round-robin fashion regardless of whether the IP address is IPv4 or IPv6. As a result, we can apply the same strategy as outlined in Chapter 4.6.2 & 4.6.3 to handle them. Specifically, we can apply nameserver pinning to cause the IPv4 nameserver to become non-responsive and never (or rarely) used by a resolver.

4.6.5 Noises

Background Traffic. There are two potential sources of background traffic at the resolver that can influence the ephemeral port scan. First, the victim resolver may have multiple outstanding queries at the same time. During the port scan, it is possible that the ephemeral port we find belongs to a different query. It is not a serious concern for private-facing ports as they are "visible" to only specific nameservers, and there are typically few, if any, outstanding queries towards the same nameserver (in addition to the one triggered by the attacker). However, it can affect the public-facing ports because the ephemeral port

of *any* outstanding query to *any* nameserver can show up during a scan. Nevertheless, we point out that any of the strategies described in Chapter 4.6.1 that can extend the attack window will automatically mitigate this concern. This is because the outstanding query triggered by the attacker would then last for much longer (possibly seconds) while other ordinary queries will only last for hundreds of milliseconds at most. Therefore, we can simply confirm that the port lives long enough before deciding to brute force the TxID.

Another type of background traffic is the benign ICMP error messages a resolver may receive during a port scan. They can create additional entries in the exception cache. This has little impact on public-facing ports because the attack requires only one entry to be created in the cache and it is highly unlikely that there are many naturally-occurring ICMP errors that will hash into the same bucket as the attacker's entry and evict it, during a short time frame of an attack. For private-facing ports, the attack does require all five exception entries in the same hash bucket to be intact during the scan. However, it is still unlikely to have a hash collision from benign ICMP messages during a short time period. Even if it does occur in practice, it will just interfere with one attack attempt (triggering a FP) and the next attack attempt will follow immediately.

Packet Losses. Although unlikely, if the probing ICMP containing the correct ephemeral port happens to be lost, FNs can arise. In such cases, the attacker simply moves on to the next attempt. If the loss is on the verification or verification reply packets, it will not affect the attack since the attacker can easily notice and retransmit the verification packet. This is because a verification reply is always supposed to come back either fragmented or not (depending on whether the ephemeral port is guessed correctly).

Packet Reordering. Reordering can cause FNs on public-facing port scans and both FPs and FNs on private-facing port scans. Specifically, if the verification packet accidentally arrives before the ICMP probe containing the correct ephemeral port, it will fail to detect the exception cache change and lead to FNs. Furthermore, if the private-facing port is being scanned, such a FN would mislead the attacker into continuing the scan despite the fact that one of the planted exceptions has already been evicted. This is guaranteed to lead to a FP in the scanning of the next batch of ports, as the eviction will be detected by the next verification packet. To mitigate such problems, a small time gap can be inserted between the probing and the verification packets. To mitigate the risk of FPs and flooding the resolver with too many packets, we always double-check whether a detected port is a true positive before deciding to brute force the TxID.

4.7 Evaluation

To evaluate the efficiency of our attacks without causing real-world damage, we tested the attack in a controlled environment with different server configurations and simulated network conditions. Overall, our attacks can succeed in minutes and have a near-perfect success rate. Note that inferring private-facing ephemeral ports requires inferring the colliding IPs as described in Chapter 4.4.4. However, since it is only a one-time effort for each resolver, the time used for the attack does not include the time for inferring colliding IPs.

4.7.1 Resolver Attack

Attack Setup. In this attack, we evaluate the power of the fragment needed attack based on the private-facing port scan. There are 3 hosts involved in the attack: the attacker host, the victim resolver and the nameserver, all of which are controlled by us. The attack program is executed on the attacker host, which is a MacBook running macOS (Darwin 19.6.0) and is connected to the victim resolver via a wired router (1Gbps). The victim resolver is a PC (with a single CPU of Intel Core i7-9700) running BIND 9.16.13 on Ubuntu 20.04 (Linux 5.11.16). The nameserver, where our domain's records are kept, is hosted on AWS and also running BIND 9.16.13. The attacker's host, and the victim resolver are at home and connected to the nameserver via residential Internet and all of the traffic is sent in IPv6. The goal of the attack is to poison the cache of the victim resolver so that our own domain's A record will be altered in the cache.

Exp.	Pkt. Loss	RTT/ms	NS Mute	# of NS	Batch Size(N)	Bg. Noise	Avg. Time/s	Succ.
Base	0%	0.3-1.2	100%	1	1	0	80	20/20
Loss	0.20%	0.3-1.2	100%	1	1	0	83	20/20
RTT	0%	37-43	100%	1	1	0	149	20/20
ML	0%	0.3-1.2	50%	1	1	0	713	5/6
NS	0%	0.3-1.2	100%	3	1	0	347	20/20
Batch	0%	0.3-1.2	100%	1	1024	0	496	5/5
Real	0.20%	37-43	80%	2	1	0	410	20/20
Real1	0.20%	37-43	80%	2	1	810	659	10/10
Real2	0.20%	37-43	80%	2	1	810+10	933	10/10

Table 4.3: `SADDNS` 2.0 Resolver Attack Results

We conducted 9 groups of experiments to evaluate the impact of the different server configurations, network conditions, and levels of background query traffic on our attack as shown in Table 4.3. Specifically, we first performed a baseline (*Base*) attack, where the attacking conditions are ideal. Then we changed one configuration or network condition at

a time to check how they would influence the attack. Then, we tested the performance of our attack against a more realistic configuration and network condition to simulate a real-world scenario (*Real*). Finally, we introduced the background query traffic to the resolver and evaluate how the interfering query traffic affects our attack. Specifically, in *Real*1, we reproduced the similar workload on a production resolver, as mentioned in Chapter 3.7.2, with 70M queries per day, averaging at 810 queries per second. To simulate the worst-case scenario, the domains in these queries are randomly sampled from the Alexa top 1M to reduce the cache hit, leading to more open ports. In *Real*2, we added another 10 queries per second asking for the same domain that the attacker is trying to poison (which would cause confusion to our port scan).

To stay stealthy, we limit the rate of our packets to 7kpps (including both the probes and verification packets), which is 3.5k ports scanned per second. Note that 7kpps applies to the port scan phase only. During the TxID brute-forcing phase, we limit our brute force speed to 40kpps and 70kpps for *Real*1 and *Real*2 (to compete with the background traffic). We simulate varying degrees of packet losses, jitters, and delays according to the representative numbers reported on the Internet [32, 47]. Besides, we also evaluated how the nameserver muting level and the number of nameservers affect our attack. Although the nameserver can be completely muted *(i.e., 100% muting level)* using ICMP redirects as mentioned in Chapter 4.6.1, we also evaluate the scenario where it is difficult to completely mute a nameserver *(e.g., leveraging response rate limit).* As mentioned in Chapter 4.4.5, we also studied the impact on the attack performance when using different batch sizes *(i.e., the number of ports scanned in a batch).*

Results. Overall, we find our attacks can succeed on average in 1.3 to 15.6 minutes, depending on the setup. Note that we consider a test failed if it still does not succeed after an hour. In both baseline (*Base*) and packet loss (*Loss*) experiments, the attack succeeds in around 80s, indicating the minimal impact of moderate packet losses. This is expected as discussed in Chapter 4.6.5. In the *RTT* experiments, we found the delay and jitter do affect our attack. Under such unstable networks, the attack may experience FPs as the verification packet may be received before the probe. Fortunately, our attack can still succeed because we have inserted time gaps to minimize reordering (see Chapter 4.6.5).

For nameserver muting levels, we find they do have a significant impact on our attack but are much smaller compared to the impact on `SADDNS`, as indicated in Chapter 3.7.2. Under the same muting level (50%), our attack (*ML*) is 10x faster than `SADDNS`. This should be attributed to the substantially faster scan speed and the fact that we do not need to perform iterative probes to narrow down the search space. As a result, this allows our attack to fare better under smaller attack windows. Experiment *Batch* further confirms this. With N=1024, the average success time increased by five times compared to the baseline where N=1. Note in *ML*, there is one attack attempt that failed (after an hour) likely due to a link-layer issue that we are unable to reproduce.

We also notice it would take about 4x the amount of time to poison a domain with 3 nameservers (*NS*). This is due to the limit of 7kpps packet sending rate, which forces us to scan for each nameserver at 1/3 of the total rate. However, if an attacker scans with 3 times the bandwidth, the result would have been close to the baseline.

In the real world scenario experiments (*Real*), we succeeded in 410s on average, which is 2x the speed of `SADDNS` with the same setting, despite the fact that the test is against BIND which is known to have a much smaller attack window (about only 2s as experienced in our experiments) than Unbound (more than 30s).

Finally, for the background query traffic experiment *Real*1, we found random domain queries do not significantly impact the attack performance. As expected, we do not find our scan being confused by the additional open ephemeral ports because they are all private ports and not visible to the nameserver which hosts the target domain name (see Chapter 4.6.5). Instead, we find that the increase of time-to-succeed is mostly attributed to the machine being slowed down in processing these query packets. Compared to *Real*1, *Real*2 experienced worse results because the additional 10 queries per second can generate ephemeral ports that are visible to the target nameserver, therefore creating confusion to our scan. Looking into the detailed logs, we see that *Real*2 experiences 22 failed TxID brute force attempts on average whereas *Real*1 experiences only 11. The majority of the additional failed brute force attempts are due to the failure in inferring the correct port number.

In general, we make two additional general observations on the results. First, the overall attack time is spent predominantly on repeated port scans (starting from the smallest port to the largest), accounting for 96% to 98% of the time. The remaining time is spent on brute-forcing the TxIDs. Second, the time-to-succeed varies significantly depending on how close the correct port is to the beginning of the port scan. In many cases, we see the

time-to-succeed being a few seconds, whereas in the worse case (especially when noise is introduced), it can take 30 minutes to find the port and succeed in brute-forcing the TxIDs.

4.7.2 Other Attacks

Forwarder Attack. To evaluate the performance of the public-facing port scan, we launched the attack against an ASUS AX6600 Wi-Fi router which has a built-in DNS forwarder. We used a similar setup as the *Base* experiment in the resolver attack where the attacker is a LAN machine that can trigger DNS queries on the forwarder. In this attack, we used the IPv4 network and set the upstream resolver as 8.8.8.8, which the attacker needs to spoof when brute-forcing the TxIDs. Finally, the attack succeeded in 13s.

Redirect Attack. Similar to *Base*, we launched the redirect-based attack under the same settings, with the only change of replacing IPv6 with IPv4, to demonstrate the private port scan under different IP versions. Finally, the attack succeeded in around 150s.

Public Resolver Attack. We obtained authorization to test our attack against an anonymized popular public resolver listed in Table 4.2 and it took about 30s and 180s to succeed in two attack attempts respectively, with the NS mute level set to 75%.

4.8 Discussion

4.8.1 Comparison With `SADDNS`

Ephemeral Port Inference Method. As mentioned in Chapter 4.4, the first and foremost difference is the use of ICMP probes in `SADDNS` 2.0. By design, ICMP messages are considered errors that should not solicit any responses [16]. This makes them an unlikely

avenue to probe any secret. Nevertheless, we demonstrate a superior understanding of the nature of side channels, making ICMP probes a successful entry point in UDP ephemeral port scans.

Side-Channel Type. `SADDNS` 2.0 leverages the space resource limit (*(i.e., t)*he space for storing the next hop exception cache is limited) while `SADDNS`' side channel leverages the time resource limit *(i.e., ICMP error generating rate is limited).* Moreover, `SADDNS` 2.0 arises when processing incoming ICMP packets (and this is why we can still infer the ephemeral port despite no reply to the ICMP probing packet is sent) while `SADDNS`' side channel arises when processing outgoing ICMP packets.

Port Scan Speed. Thanks to the novel space-constraint side channel arising in the packet receiving path, the ICMP-based ephemeral port scan rate can be theoretically unlimited. In practice, the attacker can also adjust the scan rate and strategy flexibly to achieve a higher success rate according to different network conditions. `SADDNS`, however, only allows the fixed 1000 pps slow port scan due to the nature of the time-constraint side channel it uses. The slow scan rate leads directly to a lower success rate when racing against legitimate DNS responses.

Resistance to the Noise. Unlike the global counter used in saddns, which is shared across all remote IPs, the exception cache used in our side channel is a hash-based structure and is only shared with a smaller range of IPs, which reduces the noise level of our side channel — it is less likely to be interfered with by background traffic associated with random IPs. Besides, `SADDNS` requires a strong 50-ms time block synchronization,

which can be hard to achieve with noise. In contrast, our attack does not have such a strict synchronization requirement.

Preparation of the Attack. Compared to `SADDNS`, our attack requires an additional step of inferring colliding IPs that hash into the same bucket. Nevertheless, as described in Chapter 4.4.4, it is only a one-time effort for each resolver we target.

4.8.2 PMTUD and DNS

It has been a controversial decision to enable Path MTU Discovery (PMTUD) on DNS packets. Historically, [12] indicates ICMPv6 packet too big messages could benefit the responsiveness of DNS queries while [57] argues the opposites claiming that it could lead to fragmentation-based DNS cache poisoning attacks. As a result, we see DNS software (especially BIND) changing back and forth regarding its socket options related to PMTUD.

Recently, there appears to be a convergence as both BIND and Unbound start to set the socket option of `IP_PMTUDISC_OMIT`, which instructs the kernel to never reduce the MTU. This is mostly in fear of the fragmentation-based DNS cache poisoning attacks that rely on tricking the nameserver to fragment its responses [57]. Interestingly, this option is now enabled for the sockets on both the nameservers and resolvers (even though the concern was mostly on nameservers). In addition, both BIND and Unbound decide to enable this option for IPv4 sockets only and leave IPv6 unchanged.

The reason for leaving IPv6 sockets unchanged is likely that fragmentation can be avoided most of the time as the minimum MTU is increased to 1280. This means that any link carrying IPv6 datagrams must be able to handle at least 1280 bytes of payload. This

is large enough to transmit most DNS packets and makes the fragmentation-based attacks unlikely to succeed.

4.8.3 New Defenses Against `SADDNS` 2.0

In addition to the existing defenses mentioned in Chapter 3.2.1, we also propose a set of orthogonal and near-term solutions to mitigate `SADDNS` 2.0. We will further discuss the generalized defense against the network side channels in Chapter 5.8.

Set Proper Socket Options. The most direct way is to use the socket option `IP_PMTUDISC_OMIT`, which instructs the OS not to accept the ICMP fragmentation needed messages and therefore eliminates the side-channel related processing in the kernel. However, legitimate ICMP fragmentation needed messages can be sent by a router which will be ignored also. In such cases, we recommend that the application can retransmit the query using TCP to avoid failing to transmit a UDP query due to real problems with the MTU.

Randomize the Caching Structure. Similar to the solutions to other network side channel attacks [89, 22, 23, 24, 100], sufficiently randomizing the shared resource would make the side channel practically unusable. With regard to the exception cache, we recommend a few places where randomization can take place: (1) the max length of the linked list used for solving hash collisions, (2) the eviction policy (currently the oldest will always be evicted), and (3) the secret of the hash function *(i.e., re-key periodically (every few seconds or tens of seconds)).*

Reject ICMP Redirects. Redirects are originally designed for a network with multiple gateways (similar to a router with multiple next-hop options). If a DNS server has only one default gateway, the administrator should consider ignoring ICMP redirect

messages to prevent redirect-based attacks, which can be configured via `sysctl` (see Appendix A).

4.8.4 Ethical Concerns

We conduct our experiments with ethics as a top concern. During the measurement of the vulnerable population in the wild, we attempt to minimize the impact of our probes by (1) querying our own domain and (2) at a mild speed for each resolver (under 1,000 packets per second). Also, we avoid sending suspicious-looking packets, e.g., an excessive number of ICMP packets or packets with spoofed IPs that can potentially trigger firewall alerts.

In the evaluation section, since it requires flooding fake DNS responses to finish the end-to-end attack, we refrain from attacking any real resolver and performed the attack in the local setup instead.

Responsible Disclosure. We reported our findings to the key stakeholders in the DNS community, including BIND, Cloudflare, and Linux. `SADDNS` 2.0 was assigned as CVE-2021-20322. To patch it, Linux applied two fixes on both IPv4 and IPv6 stacks to randomize the depth of the linked list storing the exceptions. Besides, Linux also increased the hashing secret from 32-bit to 64-bit to reduce the likelihood to be brute-forced. BIND also began to set `IP_PMTUDISC_OMIT` on IPv6 sockets from 9.16.20. More details can also be found on `SADDNS` website [88].

4.9 Conclusion

This chapter presents novel side channels *(i.e., SADDNS 2.0)* during the process of handling ICMP errors, a previously overlooked attack surface. We find that side channels can be exploited to perform high-speed off-path UDP ephemeral port scans. By leveraging this, the attacker could effectively poison the cache of a DNS server in minutes. We show that side channels affect many open resolvers and thus have serious impacts. Finally, we present mitigations against the discovered side channels.

Chapter 5

SCAD: a Universal and Automated Network Side-Channel Vulnerability Detection Tool

5.1 Introduction

While the threat to network security is undeniable given the presence of SADDNS and SADDNS 2.0, the discovery of most network side channels remains largely a manual endeavor. Despite the proliferation of tools for automated microarchitectural side channel detection [19, 40, 41, 117], only a handful cater to network side channels[1] [24, 26].

The existence of side channels can be fundamentally formulated as violations of the non-interference property [52], where some shared resources between the attacker and

[1] For the rest of the chapter, the term "side channel" will specifically refer to "network side channel", unless otherwise specified.

victim causes some sensitive data to leak through such resources to the attacker. However, automation attempts, such as `PacketGuardian` [26], instead of looking for violations of the non-interference property, looked for less distinctive patterns through program analysis, which resulted in numerous FPs. Conversely, `SCENT` [24] adopted a more principled strategy, leveraging model checking, to detect non-interference property violations and, by extension, side channels. Yet, the significant reliance on manual interventions *(e.g., extraction of relevant functions, abstraction of external functions,)* and heuristics bound to protocol implementations *(e.g., downscaling)* limits its generality, usability, and even completeness. As will be demonstrated in our evaluation, it misses important side channels that are found by our solution.

Given the current landscape, we see a gap in addressing the issue of side-channel attacks. Specifically, we see a lack of reusable tools that can be easily applied to a variety of protocol implementations (including future ones). To fill this gap, we introduce `SCAD` (Side-ChAnnel Detector)—an analysis tool for automated side-channel detection, based on the threat model presented in Chapter 2.1.2. Since timing information carries with noises (and thus difficult to measure) and only few previous network side-channel attacks leveraged timing variable, `SCAD` only counts the in-path packet transmission *(i.e., destined to the attacker)* event and its content as attacker-observable. Other attacker-obervables like timing variance can also be detected with some modeling. At a high level, `SCAD` employs symbolic execution to explore the state space of a target protocol implementation. For each path, it summarizes the associated data flows regarding (1) how a secret propagates to various shared variables in the protocol along each execution path, and (2) how the value of

shared variables may influence attacker-observables (e.g., presence or absence of a response, or differences in a response). Then, it pairs different paths and their associated data flow behaviors to look for any non-interference property violations.

This solution has several challenges. First, network side channels may be revealed only after multiple packets are processed by the victim [89, 90, 22], making the desirable part of the state space difficult to reach. Second, symbolic execution by design is not scalable, and may not be able to finish exploring all paths and the state space in time (also known as the path explosion challenge). To overcome these challenges, we develop a novel mode of symbolic execution that allows for the exploration of multiple parts of the state space simultaneously and at the same time manages the path explosion.

To evaluate `SCAD`, we applied it to diverse targets, including the TCP implementations of Linux, FreeBSD, and lwIP, and the UDP implementation of Linux, while also incorporating other protocols that interact with these protocols, e.g., ICMP. With a minimal effort of specifying the secrets and attacker-observables for each target, `SCAD` seamlessly runs on these targets. With a 48-hour symbolic execution run for each target, `SCAD` reports 17 side channels, of which 14 were true positives(TPs), and unveiling 7 previously unknown side channels.

5.2 Insight & `SCAD` Overview

5.2.1 Non-Interference Property

Network side channels can be modeled as violations of the non-interference property [24]. Formally, consider a protocol implementation P with a memory state M, which

can be divided into a low (sensitivity) part M_L *(e.g., storing input packets from attackers)* and a high (sensitivity) M_H *(e.g., storing the ephemeral port number of sockets)* (note both the input and output part of M). P adheres to non-interference property if and only if, for any two initial memory states M_1 and M_2 with the same low-sensitivity memory *(i.e.,* $M_{L1} = M_{L2}$*)*, after executing P, the resulting low-sensitivity memory snapshots remain identical *(i.e.,* $(P(M_1))_L = (P(M_2))_L$*)* [52, 111, 2]. In essence, the processing of low-sensitivity memory snapshots should remain unaffected by variations in high-sensitivity memory snapshots. Conversely, a violation of non-interference property allows an attacker to deduce differences between M_{H1} and M_{H2} by observing disparities between $P(M_1)_L$ and $P(M_2)_L$, forming the basis for side channel attacks. Note non-interference property violation is orthogonal to the semantics of the memory *(e.g., randomness)*. In fact randomness only reduces the entropy of the useful information deduced by the attacker but it does not govern the violation of non-interference property (and therefore the existence of side channels, as shown in Chapter 5.7.2 & 5.8).

5.2.2 Existing Approaches

To identify non-interference violations, we can in principle apply a variety of automated formal methods and testing techniques, e.g., static analysis, fuzzing, model checking, and symbolic execution. Each of these methods offers unique strengths and weaknesses when tailored to the specific problem. In the ensuing discussion, we compare them to motivate the design choice of our solution.

Static analysis. While `PacketGuardian` [26] employed static taint analysis to detect "implicit information leakage", which is an approximation of the violations of non-

interference property. However, since it is much less precise, it incurs a high FP rate, primarily attributed to the absence of path sensitivity [74].

Fuzzing. As a dynamic analysis technique that feeds random inputs to test program behaviors by executing them concretely. The advantage of fuzzing is that whatever bugs or violations of non-interference must be true positives, as they can be proven with concrete inputs and executions. Although [81] utilized fuzzing to test non-interference property violations, it is by design probabilistic and its sporadic exploration raises concerns regarding comprehensive coverage.

Model checking. In contrast to fuzzing, model checking assures exhaustive coverage of the test target. `SCENT` [24] leveraged a model checker to formally systematic search for the existence of non-interference property violations in TCP implementations. A key practical challenge lies in the creation of a self-contained model, which demands substantial manual effort, domain expertise, and consequently, limits the generality, usability, and completeness of the tool. Specifically, `SCENT` needs to extract relevant code from actual TCP implementations and turn them into models, where researchers need to specify which functions are in scope and which ones should be pruned for simplification, and abstract away certain details. Due to these issues, we find [24] missed some side channels that are discovered by `SCAD` (as will be shown in our evaluation).

5.2.3 `SCAD`'s Novel Mode of Symbolic Execution

Dynamic Symbolic Execution. The root cause of the need for modeling is the lack of a concrete execution environment, and dynamic symbolic execution (DSE) emerges

Technique	Precision	Coverage	Versatility
Static Analysis	✗	✓	✓
Fuzzing	✓	✗	✓
Model Checking	✓	✓	✗
Symbolic Execution	✓	✓	✓

Table 5.1: Comparison Among Program Analysis Techniques

as a solution to the challenge. Unlike traditional static symbolic execution, DSE operates on a system with concrete memory and under each state, every symbol maintains a possible concrete value. Under DSE, one can selectively symbolize a subset of memory *(e.g., within a predefined range)* and concretely execute any "external functions" *(e.g., `memcpy()`)* outside the range. In addition to alleviating the modeling requirements, this approach still achieves a precise (path-sensitive) analysis of the critical part of the program, while simultaneously improving performance by concretely executing functions that are external. The concreteness nature and the more precise nature makes DSE superior than static analysis approaches where significant FPs can arise. When compared with fuzzing, DSE offers superior coverage due to its systematic path exploration strategy. As summarized in Table 5.1, DSE stands out as the most appropriate solution for a universal side-channel detector.

Under-Constrained State Variables Over DSE. While the original dynamic mode of symbolic execution offers a compelling advantage over other techniques, it still faces the well-known scalability challenge when processing high-order inputs (sequences of packets). `S2E`, the SOTA DSE engine, is designed to support only symbolic inputs (and everything else is concrete, e.g., protocol states). Notably, prior side channels necessitated multiple (dozens to several hundreds) packet inputs to trigger [89, 90, 22], and in some scenarios, the number of input packets could be theoretically limitless. Handling a large

number of symbolic packets can be a significant challenge for symbolic execution (including DSE mode) [118, 119].

Nevertheless, our key insight is that most input packets primarily steer the system towards a specific state where the non-interference property violation becomes possible; once this state is achieved, a single packet suffices to trigger the violation of non-interference. Therefore, we invented a novel mode of symbolic execution based on DSE that is more suitable for side channel detection: instead of relying on multiple packets to reach this state, one can assume the system to already be in arbitrary states using the notion of "under-constrained state variables". In other words, in addition to symbolizing the input packets, we will also symbolize state variables (we will describe what part of the memory is considered state variables in Section 5.3.2). This ensures that the symbolic execution engine can explore execution paths for any packet under arbitrary protocol states.

Taking `SADDNS` vulnerbility as an example, if the global ICMP rate limit counter *(i.e., `icmp_global.credit`)* is symbolized, the path summary will directly show a reply will or will not be solicited when `icmp_global.credit` is not or is zero when the input is a UDP packet, without the need to send 49 repeated packets in advance to reduce the concrete value of `icmp_global.credit` from 50 to 1.

Compared with `SCENT`, `SCAD` obviates the need for heuristic-based downscaling [24] that also aims to solve the high-order inputs problem, but with a more principled approach, based our novel mode of symbolic execution.

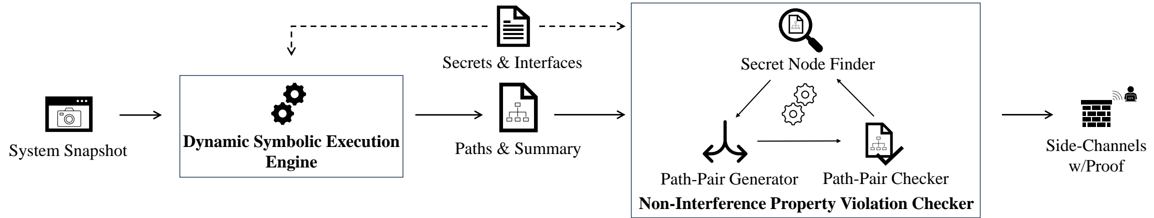


Figure 5.1: SCAD Architecture

5.2.4 SCAD Architecture

The SCAD framework, as depicted in Figure 5.1, consists two components : the DSE component and the Non-Interference Property Violation Checker (NIPVC) component. The DSE component conducts symbolic execution of the target and generates "path summaries" regarding the data flows concerning the high input (e.g., presence of an active ephemeral port), and low output (i.e., attacker-observables). The NIPVC component ingests the path summaries, orchestrates paths to form path-pairs, and subsequently conducts checks for non-interference property violations across these pairs. The outcome is a list of detected side channels, represented as execution traces and corresponding path summaries. In essence, SCAD is a white/grey-box symbolic detector for side channels based on the symbolic summary, while SCENT is a black-box concrete detector for side channels based on concrete packet outputs.

5.3 DSE Component

In this section, we will present the detail of the novel mode of symbolic execution as mentioned in Chapter 5.2.3.

5.3.1 Minimalistic Modeling of the Target

While `SCAD` aspires to be a versatile tool, adaptable across diverse targets, a modicum of modeling remains indispensable. side-channel attacks, being data-oriented [65], derive significance within the framework of target-specific logic. For example, there will be no non-interference property violation without a clearly-defined high-sensitivity memory *(e.g., secret variable).* Unlike tools like `SCENT`, which necessitate extensive manual modeling over implementation details, `SCAD` mandates a target-specific threat model complemented by a few interfaces for symbolic execution. The following enumerates the essential models required by `SCAD`:

System Memory Snapshot. As alluded to in Chapter 5.2.3, a snapshot, furnishes the concrete memory essential for DSE. This snapshot is instrumental in dynamically establishing the target-specific threat model. Taking TCP as an example target, according to the threat model of [22, 24], the snapshot should contain two established connections with one for the attacker and one for the victim.

Secrets. Integral to the threat model, secrets, define the parameters that `SCAD` seeks to protect. These secrets, delineated based on protocol specifications, typically encompass parameters like UDP ephemeral port numbers or TCP sequence numbers [22, 24, 89, 90].

Interfaces of the Target. Given the nature of DSE, it's imperative for the symbolic execution engine to discern the commencement and termination points of execution. Specifically, `SCAD` starts symbolic execution when the execution passes the start point, terminates the current path when the execution reaches termination point and switches

to concrete execution when the execution is beyond the symbolic execution address range. Therefore, as a part of model, start point, termination point, and symbolic execution address range must be explicitly provided to guide the engine.

5.3.2 DSE Component Workflow

Upon encountering the start point, the DSE component initiates its operation by substituting the input packet with symbols, transitioning to symbolic execution. Typically, the start point corresponds to the entry point of the function processing the protocol payload. Taking Linux TCP implementation as an example, DSE will hook `tcp_v4_rcv()` [50] as start point and replace the TCP header in `skb` with a symbol when reached.

State Variable Symbolization. During the symbolic execution, state variables need to be symbolized to implement the novel mode of symbolic execution. The symbolization, executed in a "lazy initialization" manner [73], treats all non-stack variables as state variables, which includes heap and global variables that have a lifetime longer than processing a single packet. Specifically, when a variable is accessed for the first time, if the access is a read, then DSE engine will symbolize it and the read returns a new symbol, as there is no existing constraint over it and thus DSE assumes it can take any value. But if it is write, DSE will not symbolize it because the write essentially adds the constraint that the variable must be equal to the value written. In other words, at the current state of the path, the variable written is totally determined by other variables, and thus symbolizing it (assuming it can take any value) is unnecessary and can potentially introduce infeasible paths.

Path Summary Computation. Throughout its execution, the DSE component seamlessly toggles between concrete and symbolic modes, contingent on the symbolic execution address range. We will present the range selection in Chapter 5.6.1. During the symbolic execution mode, it computes the path summary by logging data flows relating to symbols (input or state variables) for subsequent use by the NIPVC.

Upon reaching the termination point, the engine concludes the current path, outputting both path constraints and the corresponding path summary. These termination points typically signify the end *(e.g., the return of the packet processing function)* of input packet processing or the detection of a fatal error *(e.g., kernel panic).*

5.4 NIPVC Component

5.4.1 Path-Level Violation Checker

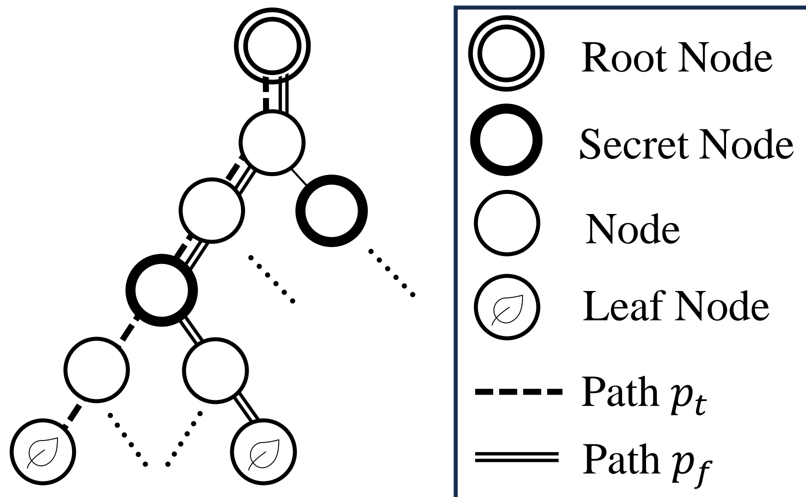


Figure 5.2: An Illustrative Example of Path Tree

The NIPVC component processes the paths and summaries generated by the DSE component, as outlined in Section 5.2. Figure 5.2 provides a visual representation of a typical path tree generated by symbolic execution, where nodes represent forking events and edges signify sequential execution without branching. The root and leaf nodes demarcate the beginning and end of a path execution, respectively.

Given the definitions of non-interference property and the associated threat model, a violation implies that, 1) given the same sequence of probing packets crated by an off-path attacker (same low input), 2) depending on the different values of the secret (different high memory), 3) the resulting attacker-observable output will be different (different low output). Since the non-interference property is a hyperproperty [30] that requires two traces to verify, in the context of symbolic execution, this translates to the existence of two paths that satisfy the following:

I. the two paths can assign the same value to symbolic inputs (i.e., attacker sending the same crafted packet and two paths refer to the same system state other than the unknown secret).

II. one of them takes the true branch and the other takes the false branch after forking from a node representing the secret (secret node in Figure 5.2) – this implies the path summary will likely differ depending on the value of the secret (high memory); and

III. the two paths write to at least one state variable differently (e.g., one updates the state variable and the other does not).

I.- III. corresponds to 1) - 3). The NIPVC's primary objective is to ascertain the existence of such path pairs. The most straightforward way is to pair paths that go through the same

secret node together and check if they meet I.- III., but it is not hard to find that two paths must intersect at the secret node in order to not break I., because otherwise it will cause contradiction: if two paths intersect at any other node (note that two paths must intersect, at least at the root node, as shown in Figure 5.2), and I. requires them to assign the same value to the non-secret symbols in the forking condition, then two paths must take the same branch, which contradicts with the intersecting at the node. Note that the secret symbol is allowed to take different values on either path as it belongs to high-sensitivity memory.

To find non-interference property violation for a secret, based on the above observation, NIPVC first traverses the path tree to gather all secret nodes, which can be easily differentiated from the forking conditions. For each secret node, it arranges paths, that pass through the secret node, in pairs, and perform the above check for them, and report the found violation.

To formally express the conditions for non-interference property violation, denote the secret symbol as s, non-secret symbols as set N, and the path constraints of the path-pair (p_t, p_f), as shown in Figure 5.2, as assertions $C_t(s, N)$ and $C_f(s, N)$ (is true) respectively. To satisfy II., two paths need to take different secret values, therefore a shadow secret symbol s' is used for the false (or true) branch to replace s and then its constraint becomes $C_f(s', N)$. Satisfying I. then becomes satisfying $C_t(s, N) \wedge C_f(s', N)$. Note that additionally satisfying $s \neq s'$ is unnecessary as $s = s'$ will dissatisfy I. by introducing the similar contradiction as mentioned above. As stated in Section 5.3.2, DSE component records the memory write summary of variable v on path p_q as an expression $W_q^v(s, N)$. Therefore, III. can be written as assertion $\exists n \in N, W_t^n(s, N) \neq W_f^n(s', N)$. Putting everything together, a pair of path

violates non-interference property is equal to $(C_t(s, N) \wedge C_f(s', N)) \wedge (\exists n \in N, W_t^n(s, N) \neq W_f^n(s', N)))$ can be satisfied. This problem can be solved by SAT solver like `Z3`. In the real implementation, the checker will enumerate all possible n and return every n that satisfies the assertion.

5.4.2 NIPVC Loop

Using `SADDNS` as an example, the overarching non-interference property violation is a composite of multiple path-level violations: port match result interferes with the rate limit counter and the counter interferes with the ICMP reply generation, which is visible to the attacker. And there is no path that takes a single packet (note only one packet is symbolized in DSE component) to make port match result directly interfere with the ICMP reply generation. To uncover end-to-end side-channel attacks, it is imperative to chain multiple non-interference property violations.

NIPVC operates iteratively, akin to taint analysis, executing the path-level check in a graph traversal fashion. The overarching goal is to trace a chain of violations from the secret to an output observable by the attacker. To achieve this, NIPVC maintains a variable set P that $\forall v_p \in P$, v_p can be interfered by the secret s through the chain of interference and initially only $s \in P$. During the loop, NIPVC will take a variable $v_p \in P$, that has not been checked, as the secret, to run the path-level checker. If a possible $n \in N$, where N is the set of non-secret symbols, is identified, then a propagation $v_p \Rightarrow n$, along with the proving path-pair, will be recorded and the intermediate variable n will be added to

P.[2], unless v_o is the packet output buffer prepared for the attacker (v_a)[3], which is already visible to the attacker and no further propagation is needed. Arguably, the "propagation" gets its name because it reads the secret implicitly embedded in v_p and encode it to n. This iterative process continues until all variables in P have been examined. Upon completion, a directed acyclic graph(DAG) is generated, mapping the propagation relationships among variables, with s as the root node *(i.e., 0 in-degrees)*. NIPVC then traverses this graph to ascertain if a path exists from s to v_a, subsequently outputting the identified propagation chain and the details of each propagation. Note this DAG should not be confused with path tree as shown in Figure 5.2. In fact, each edge of DAG represents a propagation associated with a path-pair.

5.5 Implementation

DSE Component. DSE Component is constructed atop the `S2E` framework [28], which itself is built upon the foundations of `KLEE` [21] and `Z3`. This choice was motivated by the success of similar systems such as those presented in [118, 119]. The DSE component of `SCAD` is realized as a plugin for `S2E`. This plugin, crafted in C++, spans approximately 2,300 lines of code (LoC) and is responsible for under-constrained state variable symbolization and the collection of memory operation summaries. In addition to the S2E plugin, we had to make non-trivial modifications, encompassing hundreds of lines of code, to the `S2E` engine itself, in order to support the new mode of symbolic execution we proposed. The changes

[2] If n depends on v_p in data-flow, a propagation will also be produced.
[3] This can be separated by checking path constraints.

include its `KLEE` and `libcpu` components. During this process, we identified and rectified three critical logic bugs in `S2E`, one of which only manifested under high workloads.

NIPVC Component. The output from the DSE component, presented in a human-readable log format, serves as the input for the NIPVC component. This component, written in C++ for enhanced concurrency, spans approximately 5000 LoC. Its architecture comprises three asynchronous services, depicted in Figure 5.1. These services operate concurrently, optimized to utilize all available CPU cores. The Secret Node Finder service processes the log to identify all secret nodes for a given secret, subsequently passing this data to the Path-Pair Generator. This generator, after eliminating duplicates, pairs paths and forwards these pairs to the Path-Pair Checker. This checker, leveraging `Z3` as its SAT solver, implements the algorithm detailed in Chapter 5.4.1. It outputs path-level non-interference property violations for the current secret and also feeds interfered variables back to the Secret Node Finder, establishing a feedback loop.

5.6 Evaluation

5.6.1 Evaluation Platform & Setup

Our evaluation platform comprises a server equipped with an AMD EPYC 7542 26-Core processor[4] and 2.0TB RAM, running Ubuntu 20.04 with the Linux kernel 5.4.0.

Given that most side-channel attacks [89, 90, 22, 104, 84, 97] typically involve a single intermediate variable with two steps of non-interference property violations (propagations) *(i.e., $s \Rightarrow v_p \Rightarrow v_a$ with the notations defined in Chapter 5.4.2)*, we configured `SCAD`

[4]Only 26 of the 32 cores are utilized to ensure server availability.

to detect side channels that only involves one intermediate variable as well. We therefore refer to the intermediate variable accessed by both the attacker and victim in the threat model (as discussed in Chapter 2.1.2) as the "shared variable". To classify side channels, we adopt the shared variable as the distinguishing factor, in line with previous works [26, 24].

To manage the inherent complexities of symbolic execution, we set a loop limit of 2 to prevent path explosion. We also empirically determined the timeout for the SAT solver (Z3) and restricted the DSE component's runtime to 48 hours. For each side channel report generated by SCAD, we manually verified its validity. For both TCP & UDP targets, besides the protocol implementation was included in the symbolic execution address range, ICMP implementation was also included, as it interacts with TCP & UDP closely.

For each target, as delineated in Chapter 5.3.1, individual snapshot construction and identification of secrets and interfaces are requisite. However, the inherent generality of SCAD ensures that the modeling process for any given target does not exceed a single person-day. This efficiency stands in contrast to [24], which necessitated 2.5 weeks to construct a self-contained model.

5.6.2 Comparison with SCENT

Setup. To benchmark against SCENT, we applied SCAD to the TCP stack of Linux kernel 4.8 (L4 in Table 5.2 & 5.3), the same version used in the evaluation of SCENT [24]. To replicate the threat model, we emulated a victim server scenario by creating a listening socket and establishing two connections, representing the attacker's and victim's connections. These connections were differentiated using distinct client IPs and port numbers.

#	Target	Shared Variable	TP/FP	Secret[1]	New?	Variable Type
1	L4	inet_csk(sk)->icsk_accept_queue->young	TP	CP	Y	Queue Length
2	L4	tcp_memory_allocated	TP	SN&RN	N	Memory Limit
3	L4&L6	challenge_timestamp	TP	CP&SN&RN	Y	Timestamp
4	L4&L6	inet_csk(sk)->icsk_accept_queue->qlen	TP	CP	N	Queue Length
5	L4&L6	challenge_count	TP	CP&SN&RN	N	Rate Limit
6	L4&L6	req->rsk_rcv_wnd	FP	CP		
7	L6	skb (output)	FP	CP&SN&RN		
8	L6U	icmp_global.stamp	TP	CP	Y	Timestamp
9	L6U	icmp_global.credit	TP	CP	N	Rate Limit
10	F	V_icmplim_curr_jitter	TP	CP	Y	Randomness
11	F	cr->cr_ticks	TP	CP	Y	Timestamp
12	F	cr->cr_rate	TP	CP	N	Rate Limit
13	LW	tcp_pcb_listen->accepts_pending	TP	CP	Y	Queue Length

[1] CP=Client Port # RN=rcv_nxt SN=snd_nxt

Table 5.2: Side Channels Reported by SCAD

Target	Paths/k	# of Symbols	CPU Hours
L4	2,763	527	3,089
L6	2,762	596	1,348
L6U	774	107	1,306
F	3,856	538	1,867
LW	33	103	27

Table 5.3: Statistics of SCAD on Different Targets

A packet with a symbolized IP header[5] and TCP header served as the input, simulating an off-path attacker with IP spoofing capabilities. We also designated the foreign client port number, expected sequence number (rcv_nxt), and expected acknowledgment number (snd_nxt) as secrets from the perspective of the victim server.

Performance. As shown in Table 5.3, SCAD required 3,089 CPU hours to evaluate the Linux 4.8 kernel. The DSE component consumed 1,248 of these hours, with the remainder attributed to the NIPVC component. Despite the longer runtime compared to SCENT, beyond all side channels discovered by SCENT (#2, #4 and #5 in Table 5.2, corresponding to class B, C and D in [24]), SCAD also identified two new side channels (#1

[5] 4-tuple matching logic is inside symbolic execution address range

and #3). Besides, after examining the log of NIPVC, we found it only took 4.5 CPU hours to find all 6 side channels (including one FP) listed in Table 5.2 and the remaining 1,836.5h are spent on exploring side-channel possibilities over other variables. This is due to short-circuit evaluation—if two propagations of a shared variable are found, then there is no need to continue checking on this variable. Furthermore, for DSE component, we found the paths explored in first 234 CPU hours are sufficient for NIPVC component to find all 6 side channels, which means all side channels had already been found in 238.5 CPU hours. In total, DSE component explored 2.76M paths after identified and symbolized 527 state vairables, as a result of under-constrained state variables. Note the number of explored paths, rather than branch coverage, is widely used to evaluate symbolic execution-based tools [118, 119, 124], as symbolic execution is a path-sensitive analysis. Therefore, the branch coverage of `SCENT` is not directly comparable.

Results. Among the side channels reported by `SCAD` *(i.e., #1-#6 in Table 5.2)*, #1 and #3 were previously undiscovered. Specifically, similar to #4, #1 creates a "SYN-backlog-based side-channel" [24]. #3 (`challenge_timestamp`) is used to implement the replenishing logic of challenge ack counter. Even after challenge ack counter side channel [22] has been fixed [100], `SCAD` found `challenge_timestamp` can still be leaky when paired with `challenge_count`(#5).

Nevertheless `SCENT` did not manage to discover them, because [24] did not model time and limited the TCP state to `ESTABLISHED` or `SYN_RECV` only, for scalability concerns, according the authors of `SCENT`, after we present #1 and #3 to them. This also proves the

advantage of dynamic symbolic execution and under-constrained state variables of `SCAD`, respectively.

There is one FP identified due to an implementation oversight regarding under-constrained state variables. This FP arose from the binary-level emulation of `SCAD`, which uses memory addresses as unique variable identifiers. However, this approach can lead to ambiguities when two paths dynamically allocate memory after snapshot creation. For #6, even though the allocated addresses of the request sockets *(i.e.,* ***`req`****)* are the same in two paths, they actually refer to different sockets and therefore becomes an FP, because two paths execute separately and the allocator is unaware of each other. The root cause is that the variable information is not available in the binary and `SCAD` runs binary-level emulation as a result of using S2E and also for better performance with native concrete execution. If DSE component runs over IR *(e.g., LLVM)*, then this problem could be better addressed. Despite this limitation, the overall FP rate remained relatively low at 17.6%(3/17)[6] They could also be easily filtered out with a little bit of domain knowledge.

5.6.3 Side-Channel Detection on SOTA TCP Implementations

Setup. To show the effectiveness of `SCAD` for finding new side channels, we applied `SCAD` to three distinct SOTA TCP implementations: Linux 6.1.32 (L6), FreeBSD 13.2 (F), and lwIP 2.2.0 RC1 (LW), using the same setup as in Chapter 5.6.2.

Performance. Table 5.3 indicates that the DSE component explored a similar number of paths for both Linux 4.8 and 6.1.32 within the 48-hour limit. This consistency is expected given the stability of the Linux TCP stack across versions. It is also noticed

[6]Side channels of L4 & L6 are counted twice.

that L6 takes shorter time to finish, and again this is due to short-circuit evaluation used in NIPVC component as the path-pairs are randomly chosen to check for non-interference property violation. For lwIP, DSE component finished exploring all possible paths in 25.76 CPU hours. This is expected as lwIP is a relatively light weight user-space network stack designed for embedded devices [1] and the TCP implementation of lwIP only consists 8,000 lines of C code. This also proves `SCAD` is an universal tool that can be applied to user-space targets as well.

Results. For Linux 6.1.32, `SCAD` identified five side channels, including one new side channel `challenge_timestamp` and two FPs due to semantic discrepancies between variables and memory addresses. Compared with L4, #1 and #2 is no longer found by `SCAD` in L6. After investigation, we found #1 is implicitly patched to fix the logic flaw when accepting a SYN packet [43] in Linux 4.10, but #2 is not. Finally we found this is because we increased solver timeout from 5s to 20s for L6, which enables DSE component to explore the deeper part of a path, like the output TCP header generation logic, and causes it to identify `skb`, instead of `tcp_memory_allocated` on another path, given the limited execution time. For FreeBSD, three side channels related to rate limits were reported, with two being novel. Ironically, `V_icmplim_curr_jitter`, which was used to introduce randomness to the rate limit counter to patch `SADDNS`, creates another side channel that forfeits the randomization effort. In the case of lwIP, `SCAD` detected a "SYN-backlog-based side-channel" #13.

5.6.4 Side-Channel Detection on UDP Implementation

Setup. To showcase the versatility of `SCAD`, we evaluated the UDP stack of Linux 6.1.32 (L6U). Similar to the threat model of `SADDNS` [89, 90], the snapshot models a victim

DNS resolver talking to the nameserver. Specifically, we create a listen socket on UDP port 53 to resemble the server side of a DNS resolver. We also created two connected sockets to resemble the state where the client side of the resolver sent queries to the upstream nameserver and is waiting for the response. Two sockets represent the naive nameserver which hosts the domain to be poisoned, and the attacker-controlled nameserver, respectively. The ephemeral port number of the naive socket was set as the secret, as its exposure could lead to cache poisoning attacks [89, 90]. A packet with symbolized IP header and UDP header is served as the input.

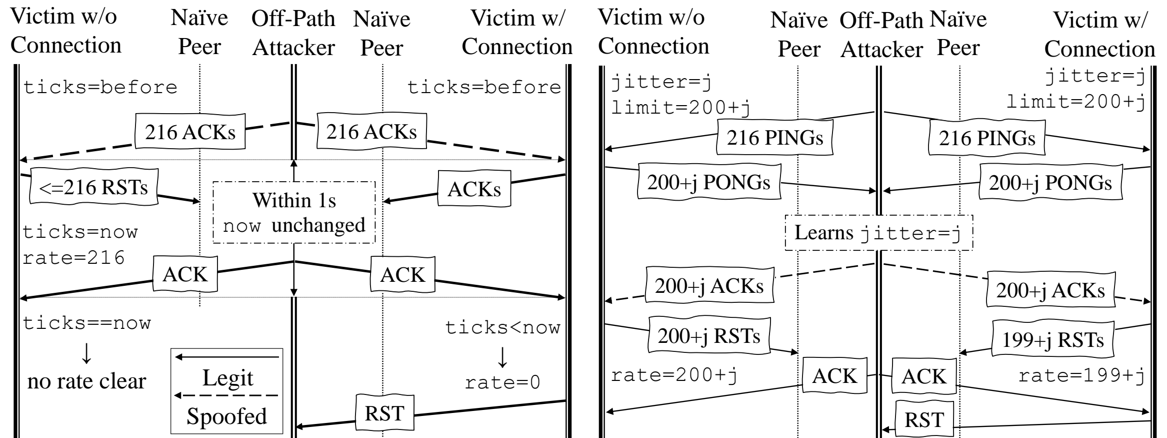
Results. `SCAD` identified two side channels related to the ICMP global rate limit counte, the foundation of the `SADDNS` attack. The discovered `icmp_global.stamp` can be used to revive the `SADDNS` attack.

5.7 Case Study

5.7.1 Timestamp-Based Side Channels

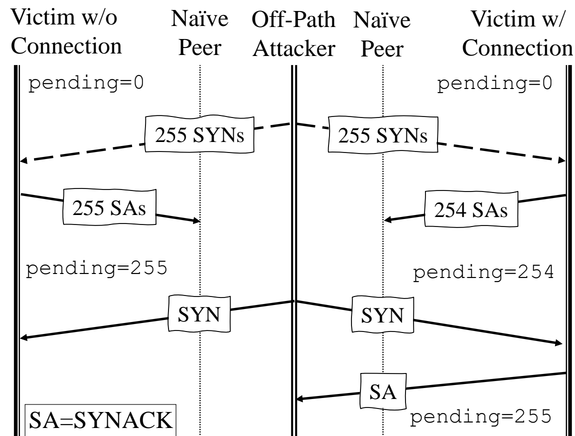
#3, #8 and #11 in Table 5.2 are timestamp-based side channels. The shared variables of them all serve the purpose of resetting rate limit counters *(i.e., #5, #9 and #12 respectively)* and is an indivisible part of the rate limit implementation.

Taking FreeBSD as an example (#11), it limits the rate of outgoing RST to 200 packets per second (pps). Therefore, aside from the counter (`cr_rate`) counting how many RSTs have been sent within 1s, it also records the last time when the counter was reset as a timestamp (`cr_ticks`). Every time the counter is accessed *(i.e., a RST is solicited)*, the



(a) Timestamp Side Channel #11

(b) Randomness Side Channel #10



(c) Queue Length Side Channel #13

Figure 5.3: Exploits for Newly Found Side Channels by SCAD

current time will be compared with the timestamp. If the timestamp is more than 1s old, the counter will be reset to 0 and the timestamp itself will be updated to the current time. Previous side-channel attacks [24] leveraged the (`cr_rate`) as the shared variable to infer the source port of an established TCP connection, as the counter value is interfered by port matching result.

Later on the side channel was patched by introducing a random jitter (`V_icmplim_curr_jitter`) to the hard limit 200pps, to patch `SADDNS`, and the now every time the the counter needs to be reset, the real limit will be calculated by adding the hard limit with the jitter. By default, with the jitter rangeing [-16, 16], the real limit ranges between [184, 216]. This destroyed counter-based side channel as the attacker will not predict counter value precisely and thus cannot effectively correlate their observance with the probing result.

Nevertheless, `SCAD` provides us a new perspective to the same logic by indicating `cr_ticks` can also be used to infer the port match result. According to the output of `SCAD`, the generation of a RST packet can set `cr_ticks` to the current time (when 1s has elapsed since the last reset) which can leak the port inference result to `cr_ticks`. This is because `cr_ticks` can decide whether the RST transmission will occur or not.

Figure 5.3a depicts the end-to-end exploit. Since the attacker does not know the exact value of the limit, to make the `cr_rate` meet or exceed the limit, they first send the max possible amount *(i.e., 216)* of spoofed ACKs to a guessed port. If there is no connection, meaning a port guess was wrong, then RST packets will be solicited and the following events will happen in order: 1) upon receiving the very first ACK packet, `cr_ticks` will be set to now, assuming 1s has elapsed since the last reset of `cr_rate`. `cr_rate` will

first be reset to 0 and then incremented to 1 immediately upon generating a RST packet in response; 2) up to an additional 215 RST packets will be generated in response to the attacker's ACK probes, and 3) `cr_rate` will increment to 216 (its value can go beyond the rate limit).

If there is a connection, challenge ACK packets will be returned instead, leaving anything related to RST rate limit unchanged. As we can see now, `cr_ticks` is updated when the guess of a port is incorrect, and remains if the guess is correct. To observe such a difference in value of `cr_ticks`, within 1s of sending the previous 216 spoofed ACK probes, the attacker sends one single non-spoofed ACK packet to a known closed port to solicit a RST. If `cr_ticks` was just updated to now, then `cr_rate` will not be reset to 0, and no RST will be sent to the attacker; this is because `cr_rate` is 216 which already exceeds the limit. If `cr_ticks` has not been recently updated, the counter will be reset, and a RST will be sent. By observing the presence or absence of the RST packet, the attacker will learn the result of the guess, indirectly through `cr_ticks`. Note that even if `cr_rate` is patched to be per-IP as opposed to global, `cr_ticks` can still be leveraged to perform this attack. In other words, the shared variable of `cr_ticks` is the culprit here.

The exploit allows an off-path attacker to scan client ports at the speed of 1 port/s, as `cr_ticks` needs to be reset before the next trial. We implemented the PoC exploit by limiting the port scan range to 100 ports and it can figure out the correct port number for 10 out of 10 times.

The exploit for Linux UDP (#8) and TCP (#3) side channels are similar. For UDP port scan, the only difference is that we would target the ICMP error rate limit

(instead of FreeBSD's RST rate limit) which is 20 tokens per 20ms, where each packet consumes 0, 1, or 2 tokens for randomness. To ensure the limit is reached, an attacker would empirically send 150 spoofed packets per 20ms. Note that since the tokens are reset every 20ms (as opposed to 1s), the effective port scan speed increases to 50 ports per second, which essentially revives `SADDNS`. For TCP port scan, we would target the challenge ACK rate limit [22]. However, to exhaust the global challenge ACK rate limit using spoofed packets, the attacker must pass the per-socket challenge ACK rate limit, which is much smaller than the global one, making the attack infeasible (unless such values are adjusted by admins through configuration changes).

To demonstrate the revival of `SADDNS`, we implemented the end-to-end attack based on timestamp side channel found in Linux UDP stack. Similar to the evaluation setup in `SADDNS` [89], we setup attacker, victim resovler and nameserver host on AWS with Linux kernel 6.2.0. The nameserver drops 80% of replies with response rate limit feature and the resolver runs `unbound 1.13.1`. We repeated the experiment for 5 times and on average it took 4586s to successfully poison the resolver's cache, which is about 10x slower compared with `SADDNS`. Given the slower probing speed, this result is of expectation. Note that despite it took 76 minutes to poison the cache, we argue this is an one time effort as the attacker can set a large TTL to make the poisoning persist longer.

5.7.2 Randomness-Based Side Channel

One of the most unexpected side channel is #10, as the jitter (`V_icmplim_curr_jitter`) itself was introduced to mitigate the rate limit counter (`cr->cr_rate`) side channel, as mentioned in Chapter 5.7.1.

In FreeBSD, the jitter is only reset *(i.e., pick another value from [-16, 16])* after the counter (`cr_rate`) exceeds the previous real limit, as otherwise, by counting the number of received RST packets, an attacker could reveal the jitter value and thus void the randomization effort. However, jitter is not immediately reset after reaching the limit, instead, it is reset along with the counter, as resetting the jitter (and therefore real limit) without resetting counter will cause instability and increase the complexity for the counter design. Since the rate limit logic is completely synchronous, both the counter and the jitter will only be reset next time the RST is sent, after exceeding the previous real limit, which means jitter will remain unchanged if no RST is solicited.

This seems to be a flawless scheme not leaking the jitter, as when probing starts, the previously learnt jitter will be immediately reset. Nevertheless, since jitter is shared across protocols but the counter and timestamp (`cr->cr_ticks`) are not, it is possible to use one protocol to retrieve the jitter value without resetting it, and then use it to de-randomize another protocol's rate limit.

Figure 5.3b shows an exploit. Since both ICMP echo reply (`PONG`) and TCP RST packet are rate limited using the same `V_icmplim_curr_jitter`, the attacker first sends 216 (the max possible real rate limit) ping packets to reveal jitter j by counting received ping replies. With the knowledge of j, the victim system is degraded to become vulnerable to the counter-based attacks for inferring client port number. Similar to [22, 24, 89], the attacker then sends $200 + j$ probing ACK packets with each destined to a different port. If one of the probed port has a connection, $199 + j$ RSTs and 1 ACK will be sent, and the `cr_rate` will become to $199 + j$. If there is no connection, the `cr_rate` will become

to $200 + j$ after sending $200 + j$ RSTs. To differentiate between these two possible values of `cr_rate`, similar to previous exploits, the attacker sends a legitimate ACK to a closed port and check if they can get the RST back, which depends on the value of `cr_rate`, and thus reveals the probing result. Note the probing result only reveals whether there is a port open among probed ports without specifying the open port number, and therefore binary search should follow [89, 90]. Since `V_icmplim_curr_jitter` is added to FreeBSD to patch `SADDNS` after 2020, `SCENT`, which was published in 2019, would not have a chance to discover this side channel. But similar to the timestamps, which were explicitly excluded in [24], randomness were also excluded in order to obtain a self-contained model. Therefore it would be impossible to `SCENT` to uncover this randomness-based side channel.

The exploit enables an off-path attacker to scan the TCP client port for around 200 ports per second and similarly the limitation lies on the 1s `cr_rate` reset interval. The exploit can also be used for UDP and SCTP ephemeral port inference, as they both share the same `V_icmplim_curr_jitter`. We implemented the PoC exploit without binary search. By limiting the port scan range to 5000 ports, it can correctly figure out the correct port number range for 10 out of 10 times.

5.7.3 Queue-Length-Based Side Channels

Unlike temporal side channels discussed in Chapter 5.7.1 & 5.7.2, #1 and #13 are spatial side channels that leverage the limited queue size for half-open TCP connections.

Taking #13 as an example, in lwIP, `accepts_pending` represents the length of the backlog queue that stores the new TCP connection requests that have not been finished *(i.e., half-open)*. The queue belongs to a listen socket that can be accessed by any host,

and the length will increase by one when the socket received a SYN packet and decrease by one when 3-way handshake is finished. If the queue length exceeds the maximum backlog limit, then new SYN packet will be dropped to prevent DoS attacks. Similarly, by observing whether the queue is full, the attacker can learn whether the client port guessed is correct or not.

Figure 5.3c shows the exploit of #13. Since in FreeBSD, the limit of the backlog queue is 255 by default, the attacker sends 255 probing SYN packets with each destined to a different port. If none of the probed port has a connection, the backlog queue will be saturated with half-open connections *(i.e., `accepts_pending` becomes 255)*, as the SYN will be treated as new connection requests, and the attacker never sends ACKs to finish the 3-way handshake. If one of the probed port has a connection, there will be one remaining slot in the queue *(i.e., `accepts_pending` becomes 254)*, as one of the SYN packet will solicit an challenge ACK of the existing connection. To detect the difference in `accepts_pending`, the attacker tries to inject another half-open connection by sending a legitimate SYN packet, and if they can receive the SYNACK reply, meaning the queue is not full and thus a connection is found and otherwise it is not found. Similarly, a binary search should follow to pinpoint the exact open port.

This side channel allows an attacker to scan 255 client ports in 20s, which is 12.75 ports per second. Since continuous port scanning requires resetting the queue, the scan speed is thus constrained by the 20-second purging interval of backlog queue of lwIP. There is no way for an off-path attacker to purge the queue earlier by either finishing the 3-way handshake or by force aborting *(i.e., sending RST packet)* half-open connections, because

both require the valid ACK number of the SYNACK packet, which, however, was not delivered to the attacker.

We implemented the PoC exploit without binary search. By limiting the port scan range to 1,000 ports, it can correctly figure out the correct port number range for 10 out of 10 times.

Similarly, #1 represents the number of half-open connections of a listen socket plus the number of orphaned half-open connection that the listen socket has received (which is unlike #4). Therefore the exploit largely remains the same. The differences are the default limit of backlog queue is 128 in Linux 4.4 and it requires a full accept queue, which stores TCP connection requests that have finished 3-way handshake but have not been accepted by the user-space application yet, to trigger. The default size of accept queue is also 128 in Linux 4.4. Given most application will accept connections using best effort, unless the system is under load, it is unlikely to make accept queue saturated in order to trigger the side channel.

5.7.4 Responsible Disclosure

We are in the process of reporting the discovered side channels to the Linux/FreeBSD/lwIP maintainers when this thesis is being finalized.

5.8 Mitigation

A primary approach to mitigate side channels is to infuse randomness into shared variables. From the perspective of the non-interference property, this introduces an addi-

tional interference source to the variable, alongside the secret interference. Consequently, this renders the alterations to shared variable unpredictable, effectively increasing entropy, or in simpler terms, making its change independent of the secret. For instance, randomizing the interval between counter resets can counteract timestamp-based side channels.

However, it's crucial to understand that randomization merely increases the entropy of the side channel without truly sealing the channel. Even if randomness is integrated into the shared variable, the interference between the variable and the secret persists. This only elevates the challenge of exploiting the side channel. To illustrate, in the Linux UDP stack, prior to introducing randomness to `icmp_global.credit`, `SADDNS` could deduce the ephemeral port at a rate of 1000pps. Post-randomization, as identified by `SCAD`, by utilizing `icmp_global.stamp`, the ephemeral port can still be scanned, albeit at a reduced rate of 50pps.

To eradicate this interference, it's imperative to isolate the variable. For queue-length-based side channels (#1, #4, and #13), narrowing the sharing scope by substituting per-socket SYN backlog queues with per-IP queues can effectively dismantle the side channel. This is because the port match result will no longer influence the attacker's queue. While retaining the per-socket queue, a global control can be introduced. However, akin to `icmp_global.credit`, it should be: 1) randomized in size, 2) inserted only after per-IP queue is successfully inserted, and 3) larger than per-IP queue in terms of capacity. Given that #2 behaves like a global queue length variable, with each addition of a half-open connection to the queue also increasing memory usage, a similar patching strategy can be employed. For timestamp-based side channels (#3, #8, and #11) and rate-limit-

counter-based side channels (#5, #9, and #12), their sharing scope can be minimized by transitioning to a per-IP equivalent. A global counterpart can be retained, provided it adheres to the aforementioned three criteria. For randomness-based side channel #10, to curtail the sharing scope, `V_icmplim_curr_jitter` can be made a distinct member of `cr`, aligning its scope with that of the counter and timestamp in the `cr`. This ensures that attackers cannot deduce the jitter using other protocol counters.

In essence, when developing network software, it's pivotal to account for off-path attackers in the security threat model [92]. With this consideration, variables should have the narrowest possible sharing scope. If feasible, individual variables should be preferred over global ones. While manual tracking and analysis of potential side-channel inducers can be difficult, `SCAD`, when provided with the threat model (including secrets, inputs, and outputs), can automate the identification of side-channel vulnerabilities for developers.

5.9 Discussion

5.9.1 Limitations

Path Explosion. Symbolic execution inherently grapples with the path explosion challenge, where the number of potential paths grows exponentially with the branching points involving symbols [25]. This exponential growth can hinder the engine's ability to explore all paths within a feasible timeframe, leading to FNs in `SCAD`. Notably, both [118, 119] could only symbolize three input TCP packets when executing symbolic execution on the Linux TCP stack to circumvent this issue. The incorporation of symbolic state variables in `SCAD` exacerbates this problem due to the increased symbol count. However, `SCAD` can

complete symbolic execution for lighter targets like lwIP. Potential solutions for path explosion such as path merging [78, 122] and function modeling [109] have been proposed. However, to maintain `SCAD`'s generality and versatility, these manual interventions may not be optimal.

Over-Approximation. Other than path explosion, symbolizing state variables can lead to the exploration of system states that may never be encountered in practice. For instance, in Table 5.2, the 32-bit variable `icmp_global.credit` of #9 will never surpass 50 according to the rate limit logic. Yet, `SCAD` assumes it can adopt any valid 32-bit integer value. If a side channel is contingent on this counter exceeding 50, it results in a FP. The underlying issue is that a program will not process inputs based on unreachable states.

Erroneous Concretization. Merging the under-constrained mode with the dynamic mode in symbolic execution introduces a unique erroneous concretization challenge. Specifically, erroneous concretization pertains to situations where an under-constrained variable, due to relaxed constraints, might be concretized to a value that is only locally or statically feasible, but globally infeasible *(e.g., concretize `icmp_global.credit` to 51).* This not only risks identifying side channels in unareachable states but can also cause system crashes, for instance, by concretizing a pointer that was just symbolized, to 0, leading to FNs. In theory, this can be circumvented by forking and concretizing the variable for each feasible value under the current constraint. However, this approach might negate the benefits of concrete execution. To address this, `SCAD` employs heuristics, such as avoiding the symbolization of pointers.

High-Sensitivity Over Execution Range Selection. While SCAD offers flexibility for easy execution range adjustment, determining the optimal range remains crucial. An overly broad range might cause SCAD to waste time on irrelevant logic, given the limited time budget, leading to FNs. Conversely, a narrow range might overlook certain side channels if the non-interference property violation code executes concretely, also resulting in FNs. This challenge is intrinsically domain-specific and contingent on the threat model, lacking a universal or formal solution. For instance, available memory can serve as a shared variable to leak secrets, such as port matching results, with exploits akin to those discussed in Chapter 5.7.3. Given the difficulty for an attacker to deplete memory, we excluded memory management logic from the symbolic execution range.

Despite these limitations, as highlighted in Chapter 5.2, DSE with symbolized state variables remains the most suitable approach for SCAD.

5.9.2 Future Work

Automatic Exploit Generation (AEG). Currently the roadblock for SCAD to perform AEG is that despite SCAD can generate the concrete input and the value of state variables to trigger the side channel discovered, it does not know how to drive the victim system to that specific state. Given that SCAD has the capability to summarize the memory operations of any input over any system state (see Chapter 5.2.3), it is possible to autonomously engineer a weird machine [18] over the victim system using integer programming, to drive the system from initial state to the target state, as a part of AEG. This can also mitigate the FPs caused by over-approximation as if a state cannot be driven into from the initial state, then the side channel discovered in that state should be filtered out.

In practice, similar to SAT problem, integer programming is also NP-complete, which can be hard to solve. Also without finishing the symbolic execution, the state driving will be fundamentally unsound.

Detection of Side Channels Involving Multiple Shared Variables In this work, we used `SCAD` to detect side channels that only involve one intermediate variable, but `SCAD` is also capable of detecting a chain of non-interference property violations *(i.e., multiple intermediate variables)* as mentioned in Chapter 5.4.2. Nevertheless, such chain of side channels have not been widely studied and theoretically the exploit of them may be difficult due to the accumulation of the noises arising from each individual propagation, which may reduce the entropy leaked from the attacker-observable. Besides, FP rate will also increase exponentially due to the accumulation of FPs of individual propagations. AEG is necessary to perform at each individual propagation to increase the report precision. Finally, despite `SCAD` is already capable to detect multiple intermediate variables, the performance remains a problem, and some optimization over the algorithm will alleviate it.

5.10 Conclusion

In this chapter, we introduced `SCAD`, a generic and automated tool designed for the detection of network side channels. Utilizing the power of under-constrained dynamic symbolic execution, `SCAD` explores the state space of protocol implementations. Through the pairing of identified paths, `SCAD` pinpoints path-level non-interference property violations, which subsequently lead to interferences between individual variables. By constructing and analyzing the interference (propagation) graph, `SCAD` systematically identifies chains

of non-interference property violations, culminating in the interference between secrets and attacker-observable outputs.

When compared with SCENT, the versatility of SCAD becomes evident, offering a plug-and-play approach across diverse protocol implementations. Our evaluations spanned multiple protocol implementations, encompassing Linux, FreeBSD, and lwIP. Impressively, across five distinct targets, SCAD identified 17 side channels, with 14 being true positives. Notably, 7 of these vulnerabilities hwere never discovered before. For each newly discovered side channel, we demonstrated its exploitability, specifically in inferring the client port number. Our proof-of-concept exploits consistently achieved a 100% success rate. The adaptability of SCAD unveiled novel attack vectors, even on previously patched side channels, enabling us to rejuvenate attacks such as SADDNS [89, 90] and off-path TCP exploits [22, 24].

In our pursuit of a safer networking landscape, we also delineated variable isolation as a robust strategy to counteract side-channel attacks, proposing potential mitigations for all 14 identified vulnerabilities. With SCAD as a universal tool in their arsenal, we anticipate developers will be better equipped to preemptively address side-channel vulnerabilities prior to software releases.

Chapter 6

Related Work

Other than the previous off-path side channel attack and DNS cache poisoning attack mentioned in Chapter 2, the realm of automated side-channel detection has seen various methodologies. `PacketGuardian` [26] employed static analysis to identify "implicit information leakage", essentially a form of side channel. However, it yielded a significant number of FPs and was limited to detecting leaks to statistical counters, which remain inaccessible to off-path attackers. An enhancement over this was presented in [103], which improved upon [26] by focusing on leaks observable to attackers, *(i.e., packet outputs)*. Despite the advancements, it still grappled with a high FP rate. The study in [24] adopted model checking to pinpoint side channels within the TCP stack. However, as delineated in Chapter 5.2.2, its versatility remains limited. On the preventive front, [121] introduced a novel approach to ensure non-interference property at the programming language level by integrating new notations for Java. While promising, this method demands rewriting existing applications under the new framework, posing challenges for its widespread adoption.

Chapter 7

Conclusions

The research embodied in this thesis has undertaken a significant exploration into the domain of network side channels, an area that has emerged as a major threat to the cybersecurity. This journey, spanning the discovery of potent attack vectors to the development of pioneering detection tools, underscores the multifaceted nature of the challenge posed by network side channels. By comprehensively examining the intricacies of these side channels and proposing robust defenses, this thesis makes substantial strides towards fostering a safer and more resilient network.

The investigations commenced with the discovery and subsequent analysis of `SADDNS`, a novel side-channel attack rooted in the global ICMP rate limit counter. By derandomizing the ephemeral UDP port numbers in DNS queries using the side channel found in the counter, `SADDNS` revived the traditional DNS cache poisoning attacks that were first introduced in 1990s. The global counter is prevalent in most modern OSes and the real-

world experiments under realistic server configurations confirmed the revival of DNS cache poisoning attacks, reinforcing the gravity of the vulnerabilities inherent in the DNS system.

Building upon the foundations established by `SADDNS`, our research ventured further into the realm of network side channels with the introduction of `SADDNS` 2.0. This iteration elucidated the side channels present during the handling of ICMP errors—an attack surface previously overlooked. Similar to `SADDNS`, the alarming revelations from our findings showed that these side channels could be leveraged to execute high-speed off-path UDP ephemeral port scans, jeopardizing the integrity of numerous open resolvers. However, `SADDNS` 2.0 showed side channels can also arise in spatial constraints, in addition to the temporal constraints, which were exploited in the original `SADDNS`.

Recognizing the urgent need for robust defenses in light of these groundbreaking discoveries, this thesis introduced `SCAD`, a universal and automated tool tailored for the detection of network side channels. Designed to capitalize on the strengths of under-constrained dynamic symbolic execution, `SCAD` delves deep into the state space of protocol implementations. Through this approach, `SCAD` systematically identifies non-interference property violations, offering insights into the complex interplay of individual variables that give rise to side channels. Our empirical evaluations of `SCAD` elucidated its power, revealing novel vulnerabilities across SOTA protocol implementations and even rejuvenating previous patched attacks by showing the unusual and ignored angle of side channel exploitation.

Given the vast implications of these findings, the thesis also explored potential countermeasures to bolster defenses against these side-channel attacks. The emphasis was placed on variable isolation as a cornerstone strategy to thwart potential adversaries. With

tools like SCAD at the disposal of developers, the path to preemptively addressing these vulnerabilities becomes considerably more navigable.

This thesis, through its multifaceted exploration of network side channels, has undoubtedly enriched our understanding of the domain. The discoveries of SADDNS and SADDNS 2.0 have magnified the urgency of addressing the vulnerabilities in DNS systems. In parallel, the advent of SCAD has brought forth a new dawn in the realm of automated side-channel detection.

In reflection, this research underscores the pursuit in the domain of network security. As the digital landscape continually evolves, so too must our defenses. Armed with the insights and tools unveiled through this thesis, we are poised to make informed strides towards securing the future of computer networks. The hope is that subsequent endeavors in this domain will continue to build upon the foundations established here, steering us closer to an era where the sanctity of our digital ecosystems is uncompromised.

Bibliography

[1] lwip. https://en.wikipedia.org/wiki/LwIP).

[2] Non-interference (security). https://en.wikipedia.org/wiki/Non-interference_(security).

[3] Off-Path attacking the web. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, August 2012. USENIX Association.

[4] Introducing dns resolver, 1.1.1.1 (not a joke). https://blog.cloudflare.com/dns-resolver-1-1-1-1/, 2018.

[5] D. Eastlake 3rd and M. Andrews. RFC 7873: Domain Name System (DNS) Cookies. Technical report, May 2016.

[6] S. Deering A. Conta and Ed. M. Gupta. RFC 4443: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. Technical report, March 2006.

[7] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, et al. Let's encrypt: An automated certificate authority to encrypt the entire web. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2473–2487, 2019.

[8] G. Alexander and J. R. Crandall. Off-path round trip time measurement via tcp/ip side channels. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[9] Geoffrey Alexander, Antonio M Espinoza, and Jedidiah R Crandall. Detecting tcp/ip connections via ipid hash collisions. *Proceedings on Privacy Enhancing Technologies*, 2019(4), 2019.

[10] Fatemah Alharbi, Jie Chang, Yuchen Zhou, Feng Qian, Zhiyun Qian, and Nael Abu-Ghazaleh. Collaborative client-side dns cache poisoning attack. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1153–1161. IEEE, 2019.

[11] D. Atkins and R. Austein. RFC 3833: Threat Analysis of the Domain Name System (DNS). Technical report, August 2004.

[12] Hanieh Bagheri, Victor Boteanu, Willem Toorop, and Benno Overeinder. Making do with what we've got: Using pmtud for a higher dns responsiveness, 2013.

[13] F. Baker. RFC 1812: Requirements for IP Version 4 Routers. Technical report, June 1995.

[14] Adib Behjat. Dns forwarders. `https://www.isc.org/blogs/dns-forwarders/`, 2011.

[15] S. M. Bellovin. Security problems in the tcp/ip protocol suite. *SIGCOMM Comput. Commun. Rev.*, 19(2):32–48, apr 1989.

[16] R. Braden. RFC 1122: Requirements for Internet Hosts – Communication Layers. Technical report, October 1989.

[17] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain validation++ for mitm-resilient pki. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076. ACM, 2018.

[18] SERGEY Bratus, MICHAEL E Locasto, MEREDITH L Patterson, L Sassaman, and ANNA Shubina. Exploit programming. *From buffer overflows to "Weird Machines" and theory of computation. USENIX: login*, 2011.

[19] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019.

[20] R. Bush and R. Austein. RFC 8210: The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1. Technical report, September 2017.

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[22] 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, August 2016. USENIX Association.

[23] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path tcp exploits of the challenge ack global rate limit. *IEEE/ACM Transactions on Networking*, 26(2):765–778, 2018.

[24] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 211–224, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[26] 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*, CCS '15, page 388–400, New York, NY, USA, 2015. Association for Computing Machinery.

[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] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278. ACM, 2011.

[29] Taejoong Chung, Roland van Rijswijk-Deij, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. A longitudinal, end-to-end view of the DNSSEC ecosystem. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1307–1322, Vancouver, BC, August 2017. USENIX Association.

[30] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[31] CloudFlare. Shield your dns infrastructure from ddos attacks with cloudflare's dns firewall. `https://www.cloudflare.com/dns/dns-firewall/`.

[32] European Commision. Quality of broadband services in the eu. `http://ec.europa.eu/newsroom/dae/document.cfm?action=display&doc_id=10816`, 2014.

[33] Cloudflare community. Case randomization recently disabled? `https://community.cloudflare.com/t/case-randomization-recently-disabled/61376`, 2018.

[34] Cloudflare community. Incorrect resolution for my domain. `https://community.cloudflare.com/t/incorrect-resolution-for-my-domain/17966`, 2018.

[35] Internet Systems Consortium. Bind 9. `https://www.isc.org/bind/`, 2020.

[36] David Dagon, Manos Antonakakis, Paul Vixie, Tatuya Jinmei, and Wenke Lee. Increased dns forgery resistance through 0x20-bit encoding: Security via leet queries. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[37] Tianxiang Dai and Haya Shulman. Smap: Internet-wide scanning for spoofing. In *Annual Computer Security Applications Conference*, ACSAC '21, 2021.

[38] Casey Deccio, Derek Argueta, and Jonathan Demke. A quantitative study of the deployment of dns rate limiting. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 442–447. IEEE, 2019.

[39] Google Public DNS. Introduction: Dns security threats and mitigations. `https://developers.google.com/speed/public-dns/docs/security`, 2019.

[40] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C., August 2013. USENIX Association.

[41] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 406–421, 2017.

[42] Eric Dumazet. icmp: add a global rate limitation. `https://github.com/torvalds/linux/commit/4cdf507d54525842dfd9f6313fdafba039084046`, 2014.

[43] Eric Dumazet. tcp/dccp: drop syn packets if accept queue is full. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5ea8ea2cb7f1d0db15762c9b0bb9e7330425a071`, 2022.

[44] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[45] R Elz and R Bush. Rfc 2181: Clarifications to the dns specification. `https://tools.ietf.org/html/rfc2181`, 1997.

[46] 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, page 17, USA, 2010. USENIX Association.

[47] FCC. Eighth measuring broadband america fixed broadband report. `https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eighth-report`, 2018.

[48] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. Off-path tcp exploits of the mixed ipid assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1323–1335, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Linux Foundation. net/ipv4/icmp/c. `https://github.com/torvalds/linux/blob/v5.9/net/ipv4/icmp.c\#L268`, 2020.

[50] Linux Foundation. net/ipv4/tcp_ipv4/c. `https://github.com/torvalds/linux/blob/v6.1/net/ipv4/tcp_ipv4.c#L1924`, 2022.

[51] Nalneesh Gaur. Securing name servers on unix. *Linux J.*, 1999(68es):5–es, dec 1999.

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

[53] Suzanne Goldlust, Cathy Almond, and Mark Andrews. Dns cookies in bind 9. `https://kb.isc.org/docs/aa-01387`, 2017.

[54] F. Gont. RFC 5927: ICMP Attacks against TCP. Technical report, Jult 2010.

[55] Google. Ipv6 adoption statistics. `https://www.google.com/intl/en/ipv6/statistics.html`, 2021.

[56] Hang Guo and John Heidemann. Detecting icmp rate limiting in the internet. In *International Conference on Passive and Active Network Measurement*, pages 3–17. Springer, 2018.

[57] Matthias Göhring, Haya Shulman, and Michael Waidner. Path mtu discovery considered harmful. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 866–874, 2018.

[58] Brendon Harris and Ray Hunt. Tcp/ip security threats and attack methods. *Computer communications*, 22(10):885–897, 1999.

[59] Amir Herzberg and Haya Shulman. Unilateral antidotes to dns poisoning. In *International Conference on Security and Privacy in Communication Systems*, pages 319–336. Springer, 2011.

[60] Amir Herzberg and Haya Shulman. Security of patched dns. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS 2012*, 2012.

[61] Amir Herzberg and Haya Shulman. Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org. In *2013 IEEE Conference on Communications and Network Security (CNS)*, pages 224–232. IEEE, 2013.

[62] Amir Herzberg and Haya Shulman. Socket overloading for fun and cache-poisoning. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, 2013.

[63] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. Technical report, February 2006.

[64] P. Hoffman, A. Sullivan, and K. Fujiwara. RFC 8499: DNS Terminology. Technical report, January 2019.

[65] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[66] A. Hubert and R. van Mook. RFC 5452: Measures for Making DNS More Resilient against Forged Answers. Technical report, January 2009.

[67] Geoff Huston. The state of dnssec validation. `https://blog.apnic.net/2019/03/14/the-state-of-dnssec-validation/`, 2019.

[68] Ed. J. Iyengar, Ed. andM. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Technical report, February 2020.

[69] J. Mogul J. McCann, S. Deering and Ed. R. Hinden. RFC 8201: Path MTU Discovery for IP version 6. Technical report, July 2017.

[70] A. J. Kalafut, C. A. Shue, and M. Gupta. Touring dns open houses for trends and configurations. *IEEE/ACM Transactions on Networking*, 19(6):1666–1675, 2011.

[71] Dan Kaminsky. Black ops 2008: It's the end of the cache as we know it. *Black Hat USA*, 2008.

[72] Simon Kelley. Dnsmasq - network services for small networks. `http://www.thekelleys.org.uk/dnsmasq/doc.html`, 2020.

[73] Sarfraz Khurshid, Corina S. PĂsĂreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[74] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, page 56–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[75] Amit Klein. Cross layer attacks and how to use them (for dns cache poisoning, device tracking and more), 2020.

[76] Amit Klein, Haya Shulman, and Michael Waidner. Internet-wide study of dns cache injections. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[77] Jeffrey Knockel and Jedidiah R. Crandall. Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, San Diego, CA, August 2014. USENIX Association.

[78] Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33:645–666, 2005.

[79] R. G. Fairhurst L. Eggert and G. Shepherd. RFC 8085: UDP Usage Guidelines. Technical report, March 2017.

[80] NLnet Labs. Unbound dns resolver. `https://nlnetlabs.nl/projects/unbound/about/`, 2020.

[81] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2019.

[82] M. Lepinski and S. Kent. RFC 6480: An Infrastructure to Support Secure Internet Routing. Technical report, Feburary 2012.

[83] Cricket Liu. A new kind of ddos threat: The "nonsense name" attack. `https://www.networkworld.com/article/2875970/a-new-kind-of-ddos-threat-the-nonsense-name-attack.html`, 2015.

[84] lkm. Blind tcp/ip hijacking is still alive. `http://phrack.org/issues/64/13.html`, 2007.

[85] Chaoyi Lu, Baojun Liu, Zhou Li, Shuang Hao, Haixin Duan, Mingming Zhang, Chunying Leng, Ying Liu, Zaifeng Zhang, and Jianping Wu. An end-to-end, large-scale measurement of dns-over-encryption: How far have we come? In *Proceedings of the Internet Measurement Conference*, IMC '19, page 22–35, New York, NY, USA, 2019. Association for Computing Machinery.

[86] Matthew Luckie, Robert Beverly, Ryan Koga, Ken Keys, Joshua A. Kroll, and k claffy. Network hygiene, incentives, and regulation: Deployment of source address validation in the internet. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 465–480, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Ed. M. Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3). Technical report, June 2020.

[88] Keyu Man. Saddns website. `https://www.saddns.net/`.

[89] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1337–1350, New York, NY, USA, 2020. Association for Computing Machinery.

[90] Keyu Man, Xin'an Zhou, and Zhiyun Qian. Dns cache poisoning attack: Resurrections with side channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3400–3414, New York, NY, USA, 2021. Association for Computing Machinery.

[91] J. Mogul and S. Deering. RFC 1191: Path MTU Discovery. Technical report, November 1990.

[92] J. Mogul and S. Deering. RFC 3552: Guidelines for Writing RFC Text on Security Considerations. Technical report, July 2003.

[93] Robert Tappan Morris. A weakness in the 4.2 bsd unix tcp/ip software, 1985.

[94] Moritz Müller, Giovane C. M. Moura, Ricardo de O. Schmidt, and John Heidemann. Recursives in the wild: Engineering authoritative dns servers. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 489–495, New York, NY, USA, 2017. Association for Computing Machinery.

[95] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery.

[96] J. Postel. RFC 792: INTERNET CONTROL MESSAGE PROTOCOL. Technical report, September 1981.

[97] 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.

[98] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative tcp sequence number inference attack: How to crack sequence number under a second. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 593–604, New York, NY, USA, 2012. Association for Computing Machinery.

[99] 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, 2010.

[100] Alan Quach, Zhongjie Wang, and Zhiyun Qian. Investigation of the 2016 linux tcp stack vulnerability at scale. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), jun 2017.

[101] Riccardo Ravaioli, Guillaume Urvoy-Keller, and Chadi Barakat. Characterizing icmp rate limitation on routers. In *2015 IEEE International Conference on Communications (ICC)*, pages 6043–6049, 2015.

[102] Vicky Ris, Suzanne Goldlust, and Alan Clegg. Bind best practices - authoritative. `https://kb.isc.org/docs/bind-best-practices-authoritative`, 2020.

[103] Kaiqi Ru, Yaning Zheng, Xuewei Feng, and Dongxia Wang. The side-channel vulnerability in network protocol. In *Proceedings of the 2021 11th International Conference on Communication and Network Security*, ICCNS '21, page 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[104] Domien Schepers, Aanjhan Ranganathan, and Mathy Vanhoef. Practical side-channel attacks against wpa-tkip. Asia CCS '19, page 415–426, New York, NY, USA, 2019. Association for Computing Machinery.

[105] Paul Schmitt, Anne Edmundson, Allison Mankin, and Nick Feamster. Oblivious dns: Practical privacy for dns queries. In *PoPETS*, 2019.

[106] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 77–90. ACM, 2013.

[107] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. Dns record injectino vulnerabilities in home routers. `http://www.icir.org/mallman/talks/schomp-dns-security-nanog61.pdf`, 2014. Nanog 61.

[108] Cheng Shen, Tian Liu, Jun Huang, and Rui Tan. When lora meets emr: Electromagnetic covert channels can be super resilient. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1304–1317, 2021.

[109] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[110] Sergio De Simone. The status of http/3. `https://www.infoq.com/news/2020/01/http-3-status//`.

[111] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, pages 291–307, Boston, MA, 2007. Springer US.

[112] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In *International Conference on Information Security and Cryptology*, pages 29–40. Springer, 2006.

[113] W. Simpson T. Narten, E. Nordmark and H. Soliman. RFC 4861: Neighbor Discovery for IP version 6 (IPv6). Technical report, September 2007.

[114] US-Cert. Alert (ta13-088a) - dns amplification attacks. `https://www.us-cert.gov/ncas/alerts/TA13-088A`, 2019.

[115] Paul Vixie. On the time value of security features in dns. `http://www.circleid.com/posts/20130913_on_the_time_value_of_security_features_in_dns/`, 2019.

[116] Paul Vixie and Vernon Schryver. Dns response rate limiting (dns rrl). https://ftp.isc.org/isc/pubs/tn/isc-tn-2012-1.txt, 2012.

[117] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *NDSS*, 2019.

[118] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[119] Zhongjie Wang, Shitong Zhu, Keyu Man, Pengxiong Zhu, Yu Hao, Zhiyun Qian, Srikanth V. Krishnamurthy, Tom La Porta, and Michael J. De Lucia. Themis: Ambiguity-aware network intrusion detection based on symbolic model comparison. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3384–3399, New York, NY, USA, 2021. Association for Computing Machinery.

[120] Paul Watson. Slipping in the window: Tcp reset attacks. *Presentation at*, 2004.

[121] Jian Xiang and Stephen Chong. Co-inflow: Coarse-grained information flow control for java-like languages. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 18–35, 2021.

[122] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–363, 2005.

[123] Michal Zalewski. Strange attractors and tcp/ip sequence number analysis. https://lcamtuf.coredump.cx/oldtcp/tcpseq.html, 2001.

[124] Xiaodong Zhang, Zijiang Yang, Qinghua Zheng, Yu Hao, Pei Liu, and Ting Liu. Tell you a definite answer: Whether your data is tainted during thread scheduling. *IEEE Transactions on Software Engineering*, 46(9):916–931, 2020.

[125] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 577–593. USENIX Association, August 2020.

[126] Pengxiong Zhu, Keyu Man, Zhongjie Wang, Zhiyun Qian, Roya Ensafi, J. Alex Halderman, and Haixin Duan. Characterizing transnational internet performance and the great bottleneck of china. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.

Appendix A

ICMP Redirect Attacks

We performed the following small-scale experiments to measure the four conditions (outlined in Chapter 4.5.1) for redirect-based attacks.

University Network Experiment. We verified the conditions of successful attacks against resolvers in a university network. Since we are able to craft ICMP redirect messages with the spoofed IPs inside the university network, we target 9 resolvers by redirecting the packets destined to our test machine to an IP that is considered nearby of the resolver. The result shows 3 out of 9 resolvers are vulnerable, *(i.e., meeting all four conditions)*. Most resolvers are not vulnerable because they do not accept ICMP redirect packets at all, which breaks $C3$.. In practice, the acceptance of redirects can be configured via `sysctl` on Linux and the default value varies on different Linux distributions. Two resolvers are not vulnerable because they run FreeBSD which blindly accepts redirects and invalidates $C1$..

Delivery of ICMP Redirect on Internet. Since ICMP redirects are potentially dangerous [58], one concern is that such messages may be dropped on the Internet and only work

in local networks. We therefore performed a small-scale experiment by having 8 vantage points (corresponding to 8 ASes) distributed across the world *(i.e., in five continents)* to send ICMP redirect messages to each other. Specifically, our vantage points reside in AWS (multiple continents), Google Cloud Platform, China educational network, US university campus network, and China residential network. The result shows ICMP redirects can successfully traverse the Internet in all pairs of experiments.

Appendix B

ICMP Rate Limit

ICMP traffic is generally considered as control-plane traffic and it has been proposed that the source should rate-limit the generation of such packets [13, 101]. If such traffic is rate limited not only at the source but also during transit (for ICMP PING [56]), the port scan speed can be significantly hampered. As a result, we conduct a small-scale experiment using the same setup as mentioned in Appendix A and send ICMP fragmentation needed or redirect messages to each other at a rate of 10kpps. We find that none except one Chinese residential host showed packet losses, which confirms rate-limiting in the transit network is not a popular policy. Even for the Chinese residential host, we find that the losses seem to be affected by the nationwide slowdown effect as reported recently [126]. We had the suspicion because UDP packets destined to the same residential host experienced similar losses also.

Appendix C

Resetting the Exception Cache State

Since the search of the ephemeral port we conduct requires multiple rounds of probes, the attacker has to reset the cache state after getting a positive response *(i.e., a probing packet in a batch hitting the correct open ephemeral port or the false positive caused by noises)*. Generally speaking, this can be done similarly to the cache planting phase in the private-facing port scan where the attacker finds 5 hash-collision IPs (note these can be done via IP spoofing instead of direct ownership) to evict the cache entry containing his primary scanning IP. Note that an easier method exists specifically for the public-facing port scans using ICMP fragmentation needed messages. This is because when a correct port is hit, the resolver will reduce the MTU for the attacker's host to that specified in the ICMP fragmentation needed message. The attacker can continue to lower the MTU in future rounds of probes. Each time the MTU is decreased, an attacker can simply send

a PING verification packet to infer if the new MTU is now in effect. Note that it is not possible to raise the MTU using this method according to the specification [69, 91]. As a result, if the minimum MTU is reached, the attacker would have to fall back to the general method *(i.e., replanting the cache)*.